ВПЕРВЫЕ ВВЕДЕНИЕ В ГЕЙМДИЗАЙН, ПРОТОТИПИРОВАНИЕ И ГЕЙМДЕВ ОБЪЕДИНЕНЫ В ОДНУ КНИГУ

UNITY и C# ГЕЙМДЕВ от ИДЕИ до РЕАЛИЗАЦИИ

второе издание

Джереми Гибсон **БОНД**

Предисловие Ричарда Лемарчанда

Introduction to Game Design, Prototyping, and Development

From Concept to Playable Game with Unity and C#

Jeremy Gibson Bond

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Джереми Гибсон **БОНД**

Предисловие Ричарда Лемарчанда

UNITY и C# ГЕЙМДЕВ от ИДЕИ до РЕАЛИЗАЦИИ

второе издание

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2019

ББК 32.973.23-018.9
УДК 004.388.4
Б81

Бонд Джереми Гибсон

Б81 Unity и C#. Геймдев от идеи до реализации. 2-е изд. — СПб.: Питер, 2019. — 928 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-0715-5

Впервые введение в геймдизайн, прототипирование и геймдев объединены в одну книгу. Если вы собираетесь заняться разработкой игр, то в первую очередь вам необходима информация о современных методах и профессиональных инструментах. Эти незаменимые знания можно получить в книге Джереми Гибсона Бонда. Кросс-платформенная разработка Unity позволяет создать игру, а затем с легкостью портировать куда угодно — от Windows и Linux до популярных мобильных платформ.

Начните путешествие в мир игровой индустрии прямо сейчас! Заявите гордо: «Я — геймдизайнер» Ведь если вас услышат другие, то вы будете стараться соответствовать своим словам. А что дальше? Как стать геймдизайнером? Ответы на эти вопросы дает книга Джереми Гибсона Бонда — геймдизайнера и профессора, который больше 10 лет учит других создавать великолепные игры и делает это сам.

Вы погрузитесь в увлекательный мир игровой индустрии, построите 8 реальных прототипов и овладеете всеми необходимыми инструментами.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.23-018.9
УДК 004.388.4

Права на издание получены по соглашению с Pearson Education Inc. Все права защищены Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими

ISBN 978-0134659862 англ
ISBN 978-5-4461-0715-5

© 2018 Pearson Education, Inc
© Перевод на русский язык ООО Издательство «Питер», 2019
© Издание на русском языке, оформление ООО Издательство «Питер», 2019
© Серия «Для профессионалов», 2019

Краткое содержание

Часть I. Проектирование игры и прототипирование на бумаге 39

Глава 1. Думать как дизайнер 40

Глава 2. Методы анализа игр 57

Глава 3. Многоуровневая тетрада 68

Глава 4. Фиксированный уровень 76

Глава 5. Динамический уровень 100

Глава 6. Культурный уровень 119

Глава 7. Действовать как дизайнер 131

Глава 8. Цели проектирования 148

Глава 9. Прототипирование на бумаге 170

Глава 10. Тестирование игры 183

Глава 11. Математика и баланс игры 198

Глава 12. Руководство игроком 238

Глава 13. Проектирование головоломок 254

Глава 14. Agile-мышление 267

Глава 15. Индустрия цифровых игр 283

Часть II. Цифровое прототипирование 299

Глава 16. Цифровое мышление 300

Глава 17. Введение в среду разработки Unity 311

Глава 18. Знакомство с нашим языком: C# 327

Глава 19. Hello World: ваша первая программа 337

Глава 20. Переменные и компоненты 355

Глава 21. Логические операции и условия 374

Глава 22. Циклы 388

Глава 23. Коллекции в C# 399

Глава 24. Функции и параметры 428

Глава 25. Отладка 443

Глава 26. Классы 458

Глава 27. Объектно-ориентированное мышление 470

Часть III. Прототипы игр и примеры 495

Глава 28. Прототип 1: Apple Picker 496

Глава 29. Прототип 2: Mission Demolition 535

Глава 30. Прототип 3: SPACE SHMUP 586

Глава 31. Прототип 3.5: SPACE SHMUP PLUS 625

Глава 32. Прототип 4: PROSPECTOR SOLITAIRE 682

Глава 33. Прототип 5: BARTOK 750

Глава 34. Прототип 6: Word Game 796

Глава 35. Прототип 7: DUNGEON DELVER 836

Часть IV. Приложения ..... www.piter.com

Приложение А. Стандартная процедура настройки проекта ..... www.piter.com

Приложение Б. Полезные идеи ..... www.piter.com

Приложение В. Ссылки на интернет-ресурсы ..... www.piter.com

Оглавление

Предисловие **22**

Вступление **26**

Цель этой книги 26

Другие книги 28

Наше окружение для цифрового прототипирования: Unity и C# 30

Кому адресована эта книга 31

Типографские соглашения 32

Веб-сайт книги 34

Благодарности **35**

Об авторе 37

От издательства 38

ЧАСТЬ I. ПРОЕКТИРОВАНИЕ ИГРЫ И ПРОТОТИПИРОВАНИЕ НА БУМАГЕ **39**

Глава 1. Думать как дизайнер 40

Вы — геймдизайнер 40

Bartok: пример игры 40

Определение игры 46

Итоги 54

Глава 2. Методы анализа игр 57

Методы анализа для игрологии 57

МДЭ: механика, динамика и эстетика 58

Формальные, драматические и динамические элементы ... 62
Простая тетрада ... 65
Итоги ... 66

Глава 3. Многоуровневая тетрада ... 68
Фиксированный уровень ... 68
Динамический уровень ... 70
Культурный уровень ... 71
Ответственность дизайнера ... 73
Итоги ... 74

Глава 4. Фиксированный уровень ... 76
Фиксированная механика ... 76
Фиксированная эстетика ... 84
Фиксированный сюжет ... 87
Фиксированная технология ... 98
Итоги ... 99

Глава 5. Динамический уровень ... 100
Роль игрока ... 100
Непредсказуемость ... 101
Динамическая механика ... 102
Динамическая эстетика ... 109
Динамический сюжет ... 115
Динамическая технология ... 118
Итоги ... 118

Глава 6. Культурный уровень ... 119
За рамками игры ... 119
Культурная механика ... 121
Культурная эстетика ... 122
Культурный сюжет ... 122
Культурная технология ... 124
Авторизованный перенос не является частью культурного уровня ... 125
Культурное влияние игры ... 126
Итоги ... 130

Глава 7. Действовать как дизайнер ... 131
Итеративное проектирование ... 131
Новаторство ... 139

Мозговой штурм и формирование идей ... 140
Изменение мнения ... 143
Ограничение объемов работ ... 146
Итоги ... 147

Глава 8. Цели проектирования ... 148
Цели проектирования: неполный список ... 148
Цели для дизайнера ... 149
Цели для игрока ... 152
Итоги ... 169

Глава 9. Прототипирование на бумаге ... 170
Преимущества прототипирования на бумаге ... 170
Прототипирование интерфейсов на бумаге ... 174
Пример бумажного прототипа ... 175
Лучшие примеры использования прототипирования на бумаге ... 180
Неудачные примеры использования прототипирования на бумаге ... 181
Итоги ... 182

Глава 10. Тестирование игры ... 183
Зачем тестировать игры? ... 183
Воспитайте в себе хорошего тестировщика ... 184
Круги тестировщиков ... 185
Методы тестирования игр ... 188
Другие важные типы тестирования ... 195
Итоги ... 197

Глава 11. Математика и баланс игры ... 198
Значение баланса игры ... 198
Важность электронных таблиц ... 199
Выбор Google Sheets для этой книги ... 199
Исследование вероятности с игральной костью в Sheets ... 201
Математика вероятности ... 213
Технологии внесения элемента случайности в настольные игры ... 219
Взвешенные распределения ... 223
Перестановки ... 225
Использование Google Sheets для балансировки оружия ... 227
Положительная и отрицательная обратная связь ... 236
Итоги ... 237

Глава 12. Руководство игроком ... 238
Прямое руководство ... 238
Четыре метода прямого руководства ... 239
Косвенное руководство ... 241
Семь методов косвенного руководства ... 241
Обучение новым навыкам и понятиям ... 249
Итоги ... 252

Глава 13. Проектирование головоломок ... 254
Скотт Ким о проектировании головоломок ... 254
Примеры головоломок в активных играх ... 263
Итоги ... 265

Глава 14. Agile-мышление ... 267
Манифест гибкой разработки программного обеспечения ... 267
Методология Скрам ... 269
Пример диаграммы сгорания задач ... 272
Создание собственной диаграммы сгорания задач ... 282
Итоги ... 282

Глава 15. Индустрия цифровых игр ... 283
Об индустрии игр ... 283
Обучение созданию игр ... 287
Вхождение в индустрию ... 290
Не ждите, чтобы начать делать игры! ... 295
Итоги ... 298

ЧАСТЬ II. ЦИФРОВОЕ ПРОТОТИПИРОВАНИЕ ... 299

Глава 16. Цифровое мышление ... 300
Системный подход к настольным играм ... 300
Упражнение в простых инструкциях ... 301
Анализ игры: Apple Picker ... 304
Итоги ... 310

Глава 17. Введение в среду разработки Unity ... 311
Загрузка Unity ... 311
Введение в нашу среду разработки ... 315
Первый запуск Unity ... 320
Пример проекта ... 321

Настройка размещения окон Unity ... 322
Устройство Unity ... 326
Итоги ... 326

Глава 18. Знакомство с нашим языком: C# ... 327
Знакомство с особенностями C# ... 327
Чтение и понимание синтаксиса C# ... 333
Итоги ... 336

Глава 19. Hello World: ваша первая программа ... 337
Создание нового проекта ... 337
Создание нового сценария на C# ... 339
Пример поинтереснее ... 345
Итоги ... 354

Глава 20. Переменные и компоненты ... 355
Введение в переменные ... 355
Строго типизированные переменные в C# ... 356
Важные типы переменных в C# ... 357
Область видимости переменных ... 360
Соглашения об именах ... 361
Важные типы переменных в Unity ... 362
Игровые объекты и компоненты в Unity ... 369
Итоги ... 373

Глава 21. Логические операции и условия ... 374
Булева математика ... 374
Операторы сравнения ... 378
Условные инструкции ... 382
Итоги ... 387

Глава 22. Циклы ... 388
Виды циклов ... 388
Подготовка проекта ... 389
Цикл while ... 389
Цикл do...while ... 392
Цикл for ... 393
Цикл foreach ... 394
Инструкции перехода в циклах ... 395
Итоги ... 398

Глава 23. Коллекции в C# 399
Коллекции в C# 399
Обобщенные коллекции 402
List 403
Dictionary 408
Массивы 411
Многомерные массивы 416
Ступенчатые массивы 419
Когда использовать массивы или списки 423
Итоги 424

Глава 24. Функции и параметры 428
Настройка проекта Function Examples 428
Определение функции 428
Параметры и аргументы функций 432
Возвращаемые значения 434
Выбор правильных имен для функций 435
Зачем нужны функции? 436
Перегрузка функций 437
Необязательные параметры 438
Ключевое слово params 439
Рекурсивные функции 441
Итоги 442

Глава 25. Отладка 443
Знакомство с отладкой 443
Пошаговое выполнение кода в отладчике 450
Итоги 457

Глава 26. Классы 458
Основы классов 458
Наследование классов 466
Итоги 469

Глава 27. Объектно-ориентированное мышление 470
Объектно-ориентированная метафора 470
Объектно-ориентированная реализация стаи 472
Итоги 493

ЧАСТЬ III. ПРОТОТИПЫ ИГР И ПРИМЕРЫ 495

Глава 28. Прототип 1: Apple Picker 496
Цель создания цифрового прототипа 496
Подготовка 497
Программирование прототипа Apple Picker 507
ГИП и управление игрой 523
Итоги 533

Глава 29. Прототип 2: Mission Demolition 535
Начало: прототип 2 535
Идея прототипа игры 535
Художественные ресурсы 536
Программный код прототипа 543
Итоги 584

Глава 30. Прототип 3: SPACE SHMUP 586
Начало: прототип 3 586
Настройка сцены 589
Создание космического корабля игрока 590
Добавление вражеских кораблей 599
Случайное создание вражеских кораблей 608
Настройка тегов, слоев и физики 611
Повреждение корабля игрока вражескими кораблями 614
Перезапуск игры 618
Стрельба (наконец) 620
Итоги 624

Глава 31. Прототип 3.5: SPACE SHMUP PLUS 625
Начало: прототип 3.5 625
Добавление других вражеских кораблей 625
Снова стрельба 634
Отображение попаданий 651
Добавление бонусов для увеличения мощности оружия 654
Сбрасывание бонусов с вражеских кораблей 665
Enemy_4 — самый стойкий враг 668
Скроллинг фона с изображением звездного поля 677
Итоги 679

Глава 32. Прототип 4: PROSPECTOR SOLITAIRE 682
Начало: прототип 4 682
Настройка сборки 683
Импортирование изображений в виде спрайтов 685
Конструирование карт из спрайтов 687
Игра Prospector 705
Программная реализация Prospector 707
Реализация логики игры 720
Подсчет очков в Prospector 728
Улучшение оформления игры 742
Итоги 748

Глава 33. Прототип 5: BARTOK 750
Начало: прототип 5 750
Настройка сборки 752
Программный код игры Bartok 754
Сборка для WebGL 793
Итоги 795

Глава 34. Прототип 6: Word Game 796
Начало: прототип 6 796
Об игре Word Game 797
Парсинг списка слов 798
Настройка игры 806
Компоновка экрана 812
Добавление интерактивности 822
Отображение очков 825
Добавление анимации в плитки 828
Добавление цвета 831
Итоги 834

Глава 35. Прототип 7: DUNGEON DELVER 836
Dungeon Delver — обзор игры 836
Начало: прототип 7 838
Настройка камер 839
Данные для игры Dungeon Delver 841
Добавление героя 853
Соглашение об именовании спрайтов с изображением Дрея 853

Добавление анимации атаки ... 863
Меч Дрея ... 866
Враги: скелеты ... 868
Сценарий InRoom ... 871
Столкновения с плитками ... 873
Добавления коллайдера в объект Dray ... 877
Выравнивание по сетке ... 877
Переход из комнаты в комнату ... 884
Перемещение камеры вслед за Дреем ... 887
Отпирание дверей ... 889
Пользовательский интерфейс для отображения количества ключей и уровня здоровья ... 894
Реализация нанесения ущерба Дрею врагами ... 898
Реализация нанесения урона врагам ... 902
Сбор предметов ... 905
Бросание предметов врагами на месте гибели ... 907
Реализация крюка ... 910
Реализация нового подземелья — Hat ... 919
Редактор уровней для игры Dungeon Delver ... 924
Итоги ... 924

Часть IV. Приложения ... www.piter.com

Приложение А. Стандартная процедура настройки проекта ... www.piter.com

Приложение Б. Полезные идеи ... www.piter.com

Приложение В. Ссылки на интернет-ресурсы ... www.piter.com

Отзывы на книгу «Unity и C#. Геймдев от идеи до реализации»

Книга *«Unity и C#. Геймдев от идеи до реализации»* по-настоящему помогла мне включиться в разработку игр и открыла множество полезных приемов и методов. Эта книга не только содержит полное введение в язык C#, но и включает информацию о тестировании игр, игровых фреймворках и игровой индустрии в целом. Джереми удалось объяснить сложные понятия простыми и доступными словами. Я также увидел, насколько эффективно прием прототипирования помогает развивать полезные методы разработки. Я настоятельно рекомендую эту книгу всем, кто хочет научиться разработке игр с нуля или просто расширить свои навыки. Я с нетерпением жду возможности использовать ее в качестве руководства и справочника в будущих проектах.

— **Логан Сандберг (Logan Sandberg)**, Pinwheel Games & Animation

Широта знаний Джереми Гибсона Бонда и его проницательный аналитический взгляд на проектирование игр видны на каждой странице этого чрезвычайно интересного руководства по разработке игр. Сочетание его практического и академического опыта дает читателям возможность приступить к созданию игр, предлагая невероятно ценный и редкий сплав теории и практики. Начинающие разработчики игр найдут в этой книге обширный аналитический инструментарий и глубокое понимание ценности итераций, а также опыт создания нескольких игр. Эта книга идеально подойдет и для новичков, и для уже опытных разработчиков.

— **Эйлин Холлингер (Eileen Hollinger)**, продюсер, инструктор, независимый разработчик и энтузиаст. Бывший ведущий продюсер в студии Funomena

Одна из самых больших проблем в изучении (и обучении) разработки игр состоит в том, что лишь несколько идей и приемов действительно применимы ко всем играм — у космического шутера мало общего с карточной игрой, которая, в свою очередь, не похожа на игры-лабиринты. Нет ничего удивительного, когда начинающий разработчик, изучив десяток руководств, все еще не понимает, с чего начать создание своей игры.

Подход, предлагаемый Джереми в этой книге, отражает многолетний опыт разработки игр и обучения этому ремеслу. Он последовательно знакомит с ключевым набором идей и навыков и затем дает буквально целую серию руководств по разработке космических шутеров, карточных игр, игр-лабиринтов и многих других, демонстрируя приемы и инструменты, необходимые каждому.

Никакая другая книга по Unity не охватывает такого разнообразия игр. Я редко встречала столь хорошо подготовленные учебные пособия. Каждый шаг подробно объясняется; каждая игра является отдельным, самостоятельным проектом и тщательно спроектирована для демонстрации конкретных понятий, от самых простых, таких как игровые объекты `GameObject`, до мощных объектно-ориентированных приемов, таких как создание *роев* (boids).

Я использовала эту книгу как основу в своем вступительном цикле лекций о Unity и рекомендовала многим студентам для самостоятельного изучения. Они по достоинству оценили четкое и полное описание языка C# и движка Unity и неизменно удивлялись, как многому удается научиться, создавая прототипы. Я использую эту книгу как основной справочник по архитектуре кода в Unity.

Второе издание обещает быть еще более ценным благодаря обновлению всех глав и использованию совершенно нового руководства, основанного на «The Legend of Zelda» и включающего передовые приемы, например движение по плиткам координат или даже простой редактор уровней. Так же как предыдущее издание, это станет важной частью моего настольного набора преподавателя и моей личной библиотеки Unity.

— **Маргарет Мозер (Margaret Moser)**, ассистент кафедры «Игры и интерактивная среда», Южно-Калифорнийский университет

Если вы хотите подняться на следующий уровень в разработке игр, прочитайте эту книгу! Она не только познакомит вас с множеством примеров реализации игр от идеи до выпуска, но также — что самое важное — научит вас думать как геймдизайнера. Что делает игру веселой и увлекательной? Что заставляет игрока снова и снова возвращаться к игре? Все ответы вы найдете здесь. Эта книга даст вам много больше, чем онлайн-руководства. Она даст вам целостное понимание!

— **Дэвид Линдског (David Lindskog)**, основатель студии Monster Grog Games

Прототипирование и тестирование игр часто остаются недооцененными и недостаточно широко используемыми этапами в процессе проектирования и разработки. Итеративные циклы тестирования и доработки на ранних этапах являются важным условием создания хороших игр. Начинающие разработчики часто полагают, что достаточно освоить все тонкости языка программирования для успеха в создании игр. Второе издание потрясающей книги Джереми делает еще больший упор на аспекты проектирования и еще лучше готовит читателей к погружению в процесс проектирования и прототипирования. Изменения и дополнения в части, касающейся прототипирования, помогут увеличить шансы на успех и вновь прибывшим читателям, и поклонникам первого издания.

— **Стивен Джейкобс (Stephen Jacobs)**, профессор школы интерактивных игр и цифровой среды, глава FOSS@MAGIC, Рочестерский технологический институт, консультант национального музея игры имени Стронга

Я использовал книгу профессора Бонда, чтобы научиться программированию на C# и поближе познакомиться с Unity. С тех пор я использую эту книгу как основу для своего курса «Проектирование цифровых игр». Книга дает отличные уроки, прототипы прекрасно демонстрируют самые разные аспекты программирования и способы их использование для реализации механики игр, к тому же легко адаптируются студентами под личные предпочтения. Не могу дождаться момента, когда получу второе издание и смогу использовать его на занятиях.

— **Уэсли Джеффрис (Wesley Jeffries)**, преподаватель курса «Проектирование игр», объединенный школьный округ Риверсайд

Второе издание книги «*Unity и C#. Геймдев от идеи до реализации*» Бонда основано на прочном фундаменте первого. Улучшенное издание с новыми примерами и темами во всех главах. Эта книга представляет тщательное исследование процесса создания игр.

— **Дрю Дэвидсон (Drew Davidson)**, директор центра развлекательных технологий в университете Карнеги-Меллон

«*Unity и C#. Геймдев от идеи до реализации*» сочетает надежный фундамент развивающейся теории проектирования игр с богатством подробных примеров цифровых прототипов. Вместе они образуют отличное введение в проектирование и разработку, кульминацией которого является создание действующих игр с помощью Unity. Эта книга одинаково полезна и как вводный курс для начинающих, и как справочное руководство для опытных дизайнеров. Я использую ее как основу для моего курса по проектированию игр, и она одна из немногих, к которым я часто обращаюсь.

— **Майкл Селлерс (Michael Sellers)**, практикующий профессор и руководитель программы проектирования игр в университете штата Индиана. В прошлом художественный руководитель и генеральный директор компании Kabam

Обучающие проектированию и разработке игр часто слышат страшный вопрос: «Где можно узнать все это?» Книга «*Unity и C#. Геймдев от идеи до реализации*» стала спасением. Благодаря ей у меня появились решение и ответ. Эта книга совершенно уникальна и в своей глубине охвата разработки игр, и в иллюстрации идеи сбалансированного объединения дизайна проектирования, прототипирования в итеративный процесс. Книга показывает, что создание игр — одновременно сложная и решаемая задача и является, как мне кажется, — отличным инструментом. Новое издание с более подробными примерами выглядит еще лучше.

— **Пьетро Полсинелли (Pietro Polsinelli)**, прикладной разработчик игр в Open Lab

Подход Джереми к проектированию игр демонстрирует важность прототипирования игровых правил и учит читателей проверять их собственные идеи. Возможность создавать собственные прототипы позволяет совершать быстрые и короткие итерации и экспериментировать, а также способствует воспитанию опытных разработчиков игр.

—**Хуан Гриль (Juan Gril)**, исполнительный продюсер, Flowplay. Бывший президент Joju Games

«*Unity и C#. Геймдев от идеи до реализации*» сочетает в себе важные философские и практические понятия, необходимые всем, кто хочет стать настоящим разработчиком игр. Эта книга познакомит вас с высокоуровневыми теориями проектирования, понятиями из мира разработки игр и основами программирования. Я рекомендую эту книгу всем начинающим разработчикам, желающим приобрести новые или расширить имеющиеся навыки в создании игр. Джереми использовал весь свой многолетний опыт работы в должности профессора, чтобы научить вас мыслить категориями, важными для разработки игр, и создавать игры, используя все доступные инструменты. Неважно, как долго вы работаете в игровой индустрии, в этой книге вы обязательно найдете вдохновляющие идеи, которые помогут вам усовершенствовать процесс разработки. Лично я буду рад погрузиться в это новое издание и получить новые знания по некоторым приемам создания потрясающих игр!

— **Мишель Пун (Michelle Pun)**, игровой продюсер в Osmo. Бывший ведущий дизайнер игр в Disney и Zynga

Одним из наиболее популярных примеров для обучения новых разработчиков является приключенческая игра «Zelda» эпохи 1980-х, события в которой разворачиваются в подземных лабиринтах. В последней главе этого обновленного издания «*Unity и C#. Геймдев от идеи до реализации*» Джереми подробно объясняет, что должен знать программист, чтобы использовать возможности Unity для воссоздания игры такого типа. Предлагаемый им подход включает разумное использование встроенных структур и механизмов Unity и расширяет их хорошо организованными собственными функциями. Он абсолютно естественно охватывает технические и дизайнерские понятия и показывает их практическое применение к этому конкретному примеру. Такой прагматичный подход поможет новым разработчикам усвоить полученные знания и применить их в своих будущих проектах.

— **Крис Де Леон (Chris DeLeon)**, основатель Gamkedo Game Development Training, сопредседатель IndieCade Workshops, входит в тридцатку самых успешных бизнесменов в индустрии развлечений моложе тридцати по версии журнала Forbes

Для всех, кто просто интересуется созданием игр или собирается профессионально заниматься их разработкой, эта книга послужит доступным введением в проектирование игр. Опыт Джереми как разработчика игр и профессора проявляется в итеративной организации, которую он использует для объяснения основ разработки игр. Каждая глава и каждый урок сочетают базовый технический подход с теоретическими основами и пробуждают в вас творческие наклонности, помогая вам расти как разработчику. Как бывший его студент, могу сказать, что эта книга — лучшее, что может быть, сразу после личного обучения в его классе.

— **Хуан Вака (Juan Vaca)**, ассоциированный дизайнер, Telltale Games

«Unity и C#. Геймдев от идеи до реализации» Джереми Гибсона Бонда — важная книга, которая знакомит учащихся с решающими аспектами теории игр и итеративным процессом быстрого прототипирования в Unity. Примеры игр, разработанных Джереми, показывают, как на одном и том же движке можно реализовать разные игровые жанры, и знакомят студентов с практическими приемами использования возможностей базовых объектов Unity и других библиотек C#, помогая читателю-новичку стать довольно опытным программистом.

— **Билл Кросби (Bill Crosbie)**, ассистент кафедры информационных технологий, колледж Raritan Valley Community

Эта книга посвящается:

Моей супруге Мелани, любви всей моей жизни,
за ее любовь, ум и поддержку.

Моим родителям и сестрам.

Моим профессорам, коллегам и студентам,
вдохновившим меня на создание этой книги.

И всем читателям первого издания, приславшим мне свои отзывы
и пожелания; все вы помогли мне сделать
второе издание еще лучше.

Предисловие

У меня есть своя теория о геймдизайнерах и преподавателях. Думаю, что, несмотря на внешние различия, все мы втайне схожи; многие навыки хорошего геймдизайнера совпадают с таковыми у хорошего учителя. Доводилось ли вам встретить учителя, очаровывающего свою аудиторию головоломками и историями? Дающего ясные приемы, которые вы легко могли понять и повторить, но с трудом освоили бы самостоятельно? Того, кто постепенно, мудро помогал вам собрать в своем уме в единую картину все кусочки информации, и вы даже не осознавали этого, пока однажды он не отходил в сторону, наблюдая, как вы делаете что-то удивительное, что прежде посчитали бы для себя невозможным?

Мы, разработчики видеоигр, тратим уйму времени на поиск способов обучить людей тому, как играть в наши игры без ущерба для развлечения. Но порой людям совсем не нужно знать, что их учат, — лучший эффект, как показали видеоигры, достижим, когда предчувствуешь начало нового захватывающего приключения. Мне посчастливилось восемь лет работать в отмеченной наградами студии Naughty Dog, где в роли ведущего или соведущего дизайнера я участвовал в создании всех трех игр из серии *Uncharted* для PlayStation 3. Все в студии были очень довольны последовательностью, открывавшей нашу игру *Uncharted 2: Among Thieves*. Она эффективно обучала игроков всем основным движениям, которые могут пригодиться в игре, поддерживая при этом у них неподдельный интерес затруднительным положением главного героя, Натана Дрейка, вывалившегося из зависшего над пропастью поезда.

Дизайнеры видеоигр делают это снова и снова, создавая для нас цифровые приключения. Работая над последовательностями передачи игрового опыта, подобными тем, что есть в играх *Uncharted*, я должен акцентировать внимание на том, чему игрок обучился совсем недавно. Я должен предложить своим игрокам интересные ситуации для использования вновь обретенных навыков, достаточно простые, чтобы не расстроить их, и достаточно сложные, чтобы удержать интерес. Сделать это с совершенно незнакомыми людьми, используя только каналы взаимодействия, доступные игре, — графическое окружение с персонажами и объектами в нем, звуковое сопровождение и интерактивное управление игрой — очень сложно. В то же время [illegible]

Теперь, когда я стал профессором и занялся обучением проектированию игр в университете, я обнаружил, что многие навыки, которые я приобрел, занимаясь проектированием игр, пригодились мне как учителю. Я также обнаружил, что обучение — не менее благодарный труд, чем создание игр. Поэтому для меня не стало неожиданностью, когда я узнал, что Джереми Гибсон Бонд, автор этой книги, одинаково талантлив и как дизайнер игр, и как преподаватель, в чем вы очень скоро убедитесь.

Первый раз я встретил Джереми около 15 лет назад на ежегодной конференции разработчиков игр в Северной Калифорнии, и мы сразу же подружились. У него за плечами уже была успешно складывающаяся карьера разработчика игр, и его энтузиазм в отношении проектирования игр вызвал отклик в моей душе. Как вы увидите сами, когда будете читать эту книгу, он любит говорить о разработке игр как о ремесле, практике дизайна и нарождающейся форме искусства. Мы с Джереми продолжали общаться, когда он вернулся в аспирантуру в замечательном центре развлекательных технологий в университете Карнеги-Меллон, чтобы продолжить учебу у таких визионеров, как доктор Рэнди Пауш (Randy Pausch) и Джесси Шелл (Jesse Schell). Наконец, я узнал Джереми как профессора и коллегу на кафедре «Игры и интерактивная среда» школы кинематографического искусства в Южно-Калифорнийском университете — в рамках программы USC Games, которую сейчас преподаю.

Во время пребывания Джереми в школе я еще ближе познакомился с ним... став его учеником! Чтобы обрести опыт, необходимый для разработки экспериментальных игр в рамках исследований, проводившихся в лаборатории USC Game Innovation Lab, я решил прослушать один из курсов Джереми. В результате я превратился из полного нуба в Unity, обладающего некоторыми базовыми навыками программирования, в опытного программиста на C#, в полной мере владеющего Unity — одним из самых мощных, удобных и быстро развивающихся игровых движков. Все курсы, которые вел Джереми, не только наполнены информацией о Unity и C#, но и щедро приправлены вдохновляющими объяснениями особенностей проектирования игр и практическими советами по разработке — от приемов «лерпинга[1]» до советов по тайм-менеджменту и использованию разработчиками электронных таблиц для улучшения их игр. Я окончил курс Джереми с огромным желанием повторить его, понимая, сколь многому могу у него поучиться.

Поэтому я обрадовался, узнав, что Джереми написал книгу, и испытал еще большую радость, читая том, который вы сейчас держите в руках. Спешу сообщить, что Джереми наполнил эту книгу всем тем, что я еще хотел бы узнать. Работая в игровой индустрии, я узнал много практических приемов проектирования, разработки и производства игр и рад поведать вам, что в этой книге Джереми замечательно обобщил некоторые подходы к созданию игр, которые я сам считаю лучшими. На страницах

[1] Термин «лерпинг» происходит от англ. «lerping» (сокращение от Linear Interpolation — линейная интерполяция) и обозначает применение механизма линейной интерполяции для сглаживания движений игровых объектов. — *Примеч. пер.*

этого издания вы найдете пошаговые руководства и примеры кода, которые помогут вам стать хорошим дизайнером и разработчиком игр. Упражнения, которые приводятся в книге, могут показаться сложными, но имейте в виду, что Джереми не просит сделать что-то сложное, прежде не объяснив это четко и доходчиво.

В этой книге вы также познакомитесь с историей и теорией. Джереми давно размышляет над вопросами проектирования игр и глубоко знает эту тему. В первой части этого тома вы найдете необычайно широкий и глубокий обзор новейшей теории проектирования игр, а также уникальный синтез лучших идей Джереми. Он поддерживает интерес к дискуссии, разбавляя ее короткими забавными историями и увлекательными фактами из долгой традиции игр в человеческой культуре. Он постоянно заставляет вас сомневаться в ваших предположениях об играх и учит мыслить шире рамок консоли, контроллера, экрана и динамиков, способствуя таким образом воспитанию целого поколения новаторов игр.

Проектирование игр — итеративный процесс, в ходе которого мы тестируем созданное нами, получаем обратную связь, пересматриваем наш проект и создаем новую, улучшенную версию. Если автору посчастливилось опубликовать новое издание своей книги, он также совершает итеративный процесс, и это именно то, что делает Джереми. Он больше года работал над этим, вторым, изданием, которое вы сейчас держите в руках, и в течение этого времени пересмотрел и переделал каждую отдельную главу. Он дополнил теорию проектирования новыми положениями, для лучшей читаемости в примеры кода добавлено выделение синтаксиса цветом, и он значительно улучшил все углубленные примеры в третьей части книги. Каждый пример теперь включает пошаговые инструкции, и все они обновлены для поддержки последней версии Unity. Глава «Космический фиксированный шутер» — одна из самых полезных и одновременно самых длинных в первом издании — была разбита на две отдельных, простых в изучении главы, а два устаревших примера игр заменены новым — игрой *Dungeon Delver* («Исследователь подземелий»), реализованной в стиле *Legend of Zelda* («Легенда о Зельде»), наглядно демонстрирующей мощь компонентной архитектуры и прототипирования. Как человек, влюбившийся в первое издание и широко рекомендовавший его студентам, преподавателям и разработчикам, я очень рад этим новым изменениям и весьма счастлив видеть жизнь и развитие этой книги.

В 2013 году Джереми Гибсон Бонд (Jeremy Gibson Bond) ушел из школы кинематографического искусства (USC) и теперь преподает в фантастической программе gamedev.msu.edu в Университете штата Мичиган. Я очень рад за новые поколения студентов этого университета, которых он поведет в ближайшие годы к пониманию профессии проектирования игр. В первую весну после своего ухода из USC Джереми вошел в ресторан, где проходил ежегодный ужин выпускников конференции разработчиков игр, организованный программой USC Games, и зал, заполненный нынешними и прошлыми студентами, огласился приветствиями, переросшими в бурные аплодисменты. Это ярче всего свидетельствует о качествах Джереми как учителя. Вам повезло, что теперь, благодаря этой книге, он станет и вашим учителем.

Мир дизайна и разработки игр меняется очень быстро. Вы можете стать частью этого прекрасного мира — мира, не похожего ни на какой другой и который я люблю всем сердцем. Навыки, полученные в процессе чтения этой книги, вы сможете использовать для создания прототипов новых видов игр и других типов интерактивных сред и даже для создания целых новых жанров с выразительным новым оформлением, которые привлекут новые рынки. Вы сможете помочь людям расслабиться и развлечь их, вы сможете войти в жизни людей, как это делает великое искусство, и даже оказать какую-то помощь в решении сложных проблем, просвещая и объясняя вселенную так, как вы ее видите. Некоторые из завтрашних звезд игрового дизайна сейчас учатся проектировать и программировать у себя дома, в школе, в офисах по всему миру, и многие из них читают эту книгу. Если вы последуете моему совету и будете выполнять все упражнения, предложенные здесь, это повысит ваши шансы создать свой современный игровой дизайн.

Удачи и приятного времяпрепровождения!

Ричард Лемарчанд (Richard Lemarchand),
доцент, USC Games,
руководитель кафедры «Игры и интерактивная среда»

Вступление

Добро пожаловать во второе издание книги «*Unity и C#. Геймдев от идеи до реализации*»! В основе этой книги мой многолетний опыт и в качестве профессионального геймдизайнера, и как профессора, читающего лекции по дизайну игр в нескольких университетах, включая кафедру «Медиа и информация» в Университете штата Мичиган и кафедру «Игры и интерактивная среда» в Южно-Калифорнийском университете.

Это вступление познакомит вас с целью, темой и подходами в этой книге.

Цель этой книги

Задумывая эту книгу, я ставил перед собой простую цель: познакомить вас со всеми инструментами и передать начальные знания, необходимые для достижения успеха в разработке игр и прототипов. Я постарался втиснуть в эту книгу максимальный объем знаний. В отличие от многих других книг, эта сочетает обе дисциплины — проектирование игр и цифровую разработку (то есть компьютерное программирование) — и обертывает их практикой итеративного прототипирования. Появление продвинутых и относительно простых в использовании игровых движков, таких как Unity, еще больше упростило создание прототипов, выражающих концепции игрового дизайна, и увеличило ваши шансы стать квалифицированным (и востребованным) дизайнером игр.

Книга делится на четыре части.

Часть I. Проектирование игры и прототипирование на бумаге

Первая часть книги начинается с исследования разных теорий проектирования игр и аналитических основ игрового дизайна, предлагавшихся в некоторых более ранних книгах. В этой части описывается *многуровневая тетрада* (Layered

Tetrad) — способ объединения и расширения лучших черт этих более ранних теорий. Исследованию многоуровневой тетрады уделено много внимания, потому что это касается многих решений, которые вам придется принимать как проектировщику интерактивных взаимодействий. В этой части также содержится информация об интересных проблемах разных дисциплин проектирования игр; описывается процесс прототипирования на бумаге, тестирование и выполнение итераций; дается конкретная информация, способная помочь вам стать одним из лучших проектировщиков; представляются эффективные стратегии управления проектами и временем, помогающие обеспечить соблюдение графика разработки проекта.

Часть II. Цифровое прототипирование

Вторая часть книги учит программированию. Эта часть опирается на мой многолетний опыт обучения студентов нетехнических специальностей приемам выражения идей проектирования игр в цифровом коде. Если у вас нет опыта разработки или программирования, эта часть для вас. Но даже имеющие некоторый опыт могут просмотреть эту часть, чтобы познакомиться с новыми приемами или освежить в памяти некоторые подходы. Вторая часть охватывает C# — наш язык программирования — от самых основ до наследования классов и приемов объектно-ориентированного программирования.

Часть III. Прототипы игр и примеры

Третья часть книги включает несколько примеров, каждый из которых демонстрирует процесс разработки прототипа игры определенного жанра. Эта часть преследует две цели: раскрывает некоторые передовые приемы быстрого прототипирования игр на примере моего собственного подхода к прототипированию и помогает сформировать фундамент, опираясь на который вы сможете создавать свои игры. Многие книги, рассказывающие о Unity (наше окружение для разработки игр) используют похожий подход, но ведут читателя через единственный монолитный пример на протяжении нескольких сотен страниц. Эта книга, напротив, демонстрирует несколько намного более коротких примеров. Конечные результаты этих примеров неизбежно менее надежны, чем те, что можно найти в других книгах, но я убежден, что разнообразие поможет вам лучше подготовиться к созданию собственных проектов в будущем.

Часть IV. Приложения[1]

В этой книге есть несколько важных приложений, заслуживающих упоминания здесь. Чтобы не повторять одну и ту же информацию и не заставлять вас искать ее по разным главам, вся информация, которая многократно упоминается в книге или

[1] Часть IV вы можете скачать с нашего сайта по адресу: https://goo.gl/VDxhzE

к которой, как мне кажется, вы захотите обратиться позже (когда закончите чтение книги в первый раз), помещена в приложения. Приложение А содержит краткое пошаговое введение в процедуру создания проекта игры в Unity. Самое длинное приложение — приложение Б «Полезные идеи». Несмотря на непритязательное название, вы, я уверен, чаще всего будете обращаться именно к этому приложению, после того как прочитаете книгу в первый раз. «Полезные идеи» — это сборник передовых приемов и стратегий, которыми я сам постоянно пользуюсь, выполняя прототипирование игр, и думаю, что здесь вы найдете много интересного для себя. Третье, и заключительное, приложение — список ссылок на ресурсы в интернете, где вы найдете ответы на вопросы, не охваченные этой книгой. Часто бывает трудно определить правильное место для поиска помощи; в этом приложении собраны ссылки, которыми я сам пользуюсь чаще всего.

Другие книги

Как дизайнер и создатель, я уверен, что совершенно недопустимо забывать о тех, чей опыт и знания способствовали вашему становлению. Об играх и их проектировании было написано много книг, и я хочу перечислить некоторые из них, оказавшие существенное влияние на меня и на мои представления о проектировании игр. Ссылки на эти книги неоднократно будут встречаться в этом тексте, и я рекомендую прочитать их, как только представится возможность.

«Game Design Workshop» Трейси Фуллертон

Первоначально написанная Трейси Фуллертон (Tracy Fullerton), Крисом Свейном (Chris Swain) и Стивеном С. Хоффманом (Steven S. Hoffman), книга «Game Design Workshop» претерпела уже три издания. Я обращаюсь к ней за советами по проектированию игр чаще всего. Эта книга первоначально была основана на учебном курсе «Game Design Workshop», который Трейси и Крис преподавали в Южно-Калифорнийском университете, легшем в основу всей программы игровых технологий в USC (и моего собственного курса, который я преподавал в 2009–2013 годах). Программа «Игры и интерактивная среда» для аспирантов USC почти каждый год признавалась лучшей школой обучения проектированию игр в Северной Америке компанией Princeton Review, пока она занималась оценкой подобных программ, и основы этого успеха были заложены книгой и учебным курсом «Game Design Workshop».

В отличие от многих других книг, рассказывающих о теории игр, книга Трейси с точностью лазерного луча сосредоточивает внимание на информации, которая помогает начинающим дизайнерам повышать свое мастерство. Я долго использовал эту книгу в своей преподавательской работе (даже задолго до того, как устроился в USC) и считаю, что если вы честно выполните все упражнения из этой книги, то в итоге несомненно получите замечательный игровой проект на бумаге.

Fullerton, Tracy, Christopher Swain, and Steven Hoffman, *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*, 2nd ed. (Boca Raton, FL: Elsevier Morgan Kaufmann, 2008)

«Art of Game Design» Джесси Шелла

Джесси Шелл (Jesse Schell) — один из моих профессоров в Университете Карнеги-Меллон и фантастический геймдизайнер с огромным опытом проектирования, приобретенным за годы работы в компании Walt Disney Imagineering. Книга Джесси любима многими действующими дизайнерами, потому что рассматривает проектирование игр как дисциплину через 100 разных линз, которые раскрываются в ней. Книга Джесси — чрезвычайно занимательна и рассматривает несколько тем, не затрагиваемых в этой книге.

Jesse Schell, *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008)

«The Grasshopper» Бернарда Сьютса

На самом деле эта книга вообще не о проектировании игр, «The Grasshopper» («Кузнечик») — превосходное исследование определения слова «игра». Написанная в стиле, напоминающем метод Сократа, книга почти в самом начале представляет свое определение игры, когда Кузнечик (из басни Эзопа «Муравей и Цикада») дает определение на смертном одре, а его ученики на протяжении оставшейся части книги пытаются осмыслить и понять его. Эта книга также исследует вопрос о месте и роли игры в обществе.

Bernard Suits, *The Grasshopper: Games, Life and Utopia* (Peterborough, Ontario: Broadview Press, 2005)

«Level Up!» Скотта Роджерса

Роджерс много лет накапливал свои знания, занимаясь разработкой игр, пока наконец не оформил их в виде книги, очень интересной, доступной и практичной. Когда мы вместе преподавали курс многоуровневого проектирования, мы использовали эту книгу. Роджерс также обладает талантом художника комиксов, и его книга наполнена забавными и полезными иллюстрациями, которые втолковывают понятия проектирования.

Scott Rogers, *Level Up!: The Guide to Great Video Game Design* (Chichester, UK: Wiley, 2010)

«Imaginary Games» Криса Бейтмана

В этой своей книге Крис Бейтман (Chris Bateman) утверждает, что игры являются оправданным средством для научных исследований. Он опирается на разные научные, практические и философские источники, а его обсуждения книг, таких как

«Homo Ludens» Йохана Хейзинга[1], «Man, Play, and Games» Роже Кайуа[2] и статьи «The Game Game»[3] Мэри Миджли, отличаются глубиной и доступностью.

Chris Bateman, *Imaginary Games* (Washington, USA: Zero Books, 2011)

«Game Programming Patterns» Роберта Найстрома

Эта книга — отличный ресурс для программистов среднего уровня. В ней Роберт Найстром (Robert Nystrom) исследует шаблоны разработки программного обеспечения (первоначально классифицированные в книге «Design Patterns: Elements of Reusable Object-Oriented Software»[4]), которые, по его мнению, наиболее востребованы в разработке игр. Это действительно очень хорошая книга, и ее определенно стоит прочитать всем, кто имеет некоторый опыт программирования игр. Все примеры в книге написаны на псевдокоде, напоминающем язык C++, но в них легко смогут разобраться программисты на C#. Кроме того, книга совершенно бесплатно доступна в электронном виде по адресу http://gameprogrammingpatterns.com[5].

«Game Design Theory» Кита Бургуна

В этой книге Кит Бургун (Keith Burgun) исследует недостатки текущего состояния теории проектирования и разработки игр и предлагает более узкое определение игры, чем дает Бернард Сьютс (Bernard Suits). Работая над этой книгой, Бургун ставил перед собой цель спровоцировать обсуждение и продвинуть вперед теорию проектирования игр. Несмотря на общий негативный тон, книга Бургуна, поднимая ряд интересных вопросов, помогла мне полнее понять суть проектирования игр.

Keith Burgun, *Game Design Theory: A New Philosophy for Understanding Games* (Boca Raton, FL: A K Peters/CRC Press, 2013)

Наше окружение для цифрового прототипирования: Unity и C#

Все примеры цифровых игр в этой книге основаны на игровом движке Unity и языке программирования C#. Я уже более десяти лет занимаюсь обучением

---

[1] Йохан Хейзинга, «Homo ludens. Человек играющий». — *Примеч. пер.*

[2] *Кайуа Р.* Игры и люди: Статьи и эссе по социологии культуры / Сост., пер. с фр. и вступ. ст. С. Н. Зенкина. М.: ОГИ, 2007. — *Примеч. пер.*

[3] Перевод на русский язык можно найти по адресу https://iphras.ru/uplfile/root/biblio/phan/2017_2/30-56.pdf. — *Примеч. пер.*

[4] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2018. (Серия «Библиотека программиста»). — *Примеч. пер.*

[5] Перевод на русский язык доступен по адресу https://www.gitbook.com/book/likerrr/gameprogrammingpatterns/details. — *Примеч. пер.*

студентов приемам разработки игр и интерактивных проектов, и на мой взгляд, Unity — лучшая среда для обучения созданию игр. Я также считаю, что C# — лучший язык для обучения прототипированию игр. Конечно, существуют другие инструменты, которые проще в изучении и не требуют программирования (например, Game Maker и Construct 2), но Unity обладает большей гибкостью и производительностью уже в бесплатном пакете (бесплатная версия Unity обладает почти всеми возможностями платной версии и используется в этой книге). Еще один движок, используемый многими студиями, — Unreal. Но в нем очень тонкий промежуточный слой между упрощенной системой графического программирования Blueprint и очень сложным кодом на C++, на котором основан движок. Если вы действительно хотите научиться программировать игры и добиться успеха, Unity — это то, что вам нужно.

Аналогично, некоторые языки программирования изначально немного проще, чем C#. В прошлом я преподавал своим студентам языки ActionScript и JavaScript. Однако C# остается для меня единственным языком, который не перестает удивлять своей гибкостью и богатством возможностей. Обучение языку C# предполагает обучение не только программированию, но также хорошему стилю программирования. Такие языки, как JavaScript, допускают небрежное отношение, что в конечном итоге замедляет разработку. C# воспитывает сдержанность (благодаря строго типизированным переменным), и эта сдержанность поможет вам не только стать прекрасными программистами, но также ускорит разработку (например, строгая типизация повышает надежность кода и позволяет пользоваться всплывающими подсказками и функцией автодополнения, что помогает быстро писать аккуратный код).

Кому адресована эта книга

Существует много книг о проектировании игр и много книг о программировании. Моя книга призвана заполнить пробел между ними. С развитием и распространением технологий разработки игр, таких как Unity, все большую важность для дизайнеров игр приобретает возможность не только набросать свои идеи на бумаге, но и воплотить их в действующие цифровые прототипы. Цель этой книги — помочь вам научиться делать именно это:

- Если у вас есть желание заняться созданием игр, но прежде вы никогда не занимались программированием, эта книга идеально подойдет для вас. **Часть I** знакомит с некоторыми практическими теориями дизайна игр и приемами, которые помогут вам развить и усовершенствовать свои идеи. **Часть II** обучит вас программированию, с самых азов до понимания объектно-ориентированных иерархий классов. В колледже, где я занимаю должность профессора, большинство студентов не изучали программирование прежде, поэтому я обучаю их в основном через программирование игр. Я вложил весь свой опыт во вторую часть этой книги. **Часть III** проведет вас через процесс разработки прототипов

игр нескольких разных жанров. Каждый пример демонстрирует быстрые методы перехода от идеи к действующему цифровому прототипу. Наконец, в приложениях подробно описываются конкретные идеи разработки игр и программирования, а также перечисляются ресурсы, к которым можно обратиться после чтения этой книги. Все это сосредоточено в основном в приложении Б «Полезные идеи» чтобы вы могли долгие годы использовать этот раздел книги как справочник.

- Если вы программист, интересующийся созданием игр, части I и III этой книги будут для вас наиболее ценными. **Часть I** знакомит с некоторыми практическими теориями дизайна игр и приемами, которые помогут вам развить и усовершенствовать свои идеи. Вы можете бегло пролистать **часть II**, где рассказывается о языке программирования C# и особенностях его использования в Unity. Для тех, кто знаком с другими языками программирования, замечу, что C# напоминает C++, но обладает некоторыми замечательными особенностями Java. **Часть III** проведет вас через процесс разработки прототипов игр нескольких разных жанров. Разработка игр в Unity сильно отличается от разработки с использованием других игровых движков. Управление многими аспектами разработки осуществляется за пределами программного кода. Каждый прототип демонстрирует стиль разработки, лучше всего подходящий для Unity, и методы быстрого перехода от идеи к действующему цифровому прототипу. Вам также стоит внимательно прочитать приложение Б «Полезные идеи», наполненное подробной информацией о различных подходах к программированию в Unity и структурированное так, что его можно использовать как справочник.

Типографские соглашения

В этой книге используется несколько соглашений по оформлению текста, цель которых — помочь вам лучше понять написанное.

В любом месте книги надписи на кнопках, пункты меню и любой другой текст, который отображается на экране, будут оформляться так: Main Camera. Новые ключевые понятия будут оформляться *курсивом*. Команды меню, такие как Edit > Project Settings > Physics, показывают, что сначала нужно выбрать меню Edit в полосе главного меню, затем в открывшемся меню выбрать подменю Project Settings и затем пункт Physics.

Элементы книги

В тексте книги встречаются несколько видов врезок, сообщающих важную или полезную информацию, которая выходит за рамки основного текста.

В таких врезках приводится полезная, но не особенно важная информация. Информация в примечаниях часто будет иметь отношение к основному тексту, добавляя чуть больше сведений о рассматриваемой теме.

В советах приводятся дополнительные сведения, имеющие отношение к основному тексту, которые могут вам помочь в изучении описываемых понятий.

БУДЬТЕ ВНИМАТЕЛЬНЫ. В предупреждениях приводится важная информация, которую вы должны знать, чтобы избежать ошибок и ловушек.

ВРЕЗКА

Во врезках приводятся более длинные обсуждения тем, важных для основного текста, но которые должны рассматриваться отдельно.

Код

Несколько соглашений по оформлению применяется к программному коду. Элементы программного кода в основном тексте книги будут оформляться моноширинным шрифтом. Примером может служить имя переменной `variableOnExistingLine` из следующего листинга.

Листинги кода также будут оформляться моноширинным шрифтом, как показано ниже:

```
1 public class SampleClass {
2     public GameObject     variableOnExistingLine;          // a
3     public GameObject     variableOnNewLine;               // b
      ...                                                    // c
7     void Update() { ... }                                  // d
8 }
```

a. Листинги кода часто будут сопровождаться комментариями; в данном случае дополнительная информация о строке с меткой `// a` приводится в этом первом комментарии.

b. Многие листинги расширяют или дополняют код, который мы написали перед этим или уже имеющийся в файлах сценариев на C# по каким-то другим причинам. В таком случае старые строки будут оформлены обычным моноширинным шрифтом, а новые — **жирным**. Во всех таких случаях я старался включить достаточное количество строк существующего кода, чтобы вы без труда могли понять, куда вставлять новые строки.

c. Всякий раз, когда код в листингах будет опускаться (для экономии места в книге), я заменю его многоточием (`...`). В данном примере листинга я опустил строки с 4-й по 6-ю.

d. В местах, где я полностью опускаю тело существующей функции или метода, вы увидите многоточие, заключенное в фигурные скобки (`{ ... }`).

Большинство листингов в первых двух частях включают номера строк (как показано в примере выше). Эти номера не нужно вводить в код (среда разработки MonoDevelop автоматически пронумерует строки). В заключительной части книги номера строк не указываются из-за больших размеров листингов и их сложности.

Наконец, везде, где строки кода не умещаются по ширине книжной страницы, вы увидите символ продолжения (➥) в начале следующей строки. Он подсказывает, что на компьютере такие строки должны вводиться в одну строку. Сам символ продолжения вводить не нужно.

Веб-сайт книги

На веб-сайте книги вы найдете все файлы, упоминаемые в главах, примечания автора, начальные пакеты, действующие примеры некоторых игр, обновления и многое другое! Он доступен по адресу

http://book.prototools.net

Благодарности

Огромное число людей заслуживает благодарности. Прежде всего, я хочу поблагодарить мою супругу Мелани, чье мнение и помощь, оказываемая на протяжении всего процесса, помогли значительно улучшить эту книгу. Я также хочу сказать спасибо моей семье за многолетнюю поддержку, и особое спасибо моему отцу, который с детства учил меня программированию.

В этот второй раз мне помогали несколько сотрудников издательства Pearson, они вновь провели меня через весь процесс. Главными среди них были Крис Зан (Chris Zahn), Лаура Левин (Laura Lewin), Паула Ловелл (Paula Lowell), Лори Лайонс (Lori Lyons), Оливия Баседжио (Olivia Basegio) и Даянидхи Карунанидхи (Dhayanidhi Karunanidhi). Все они продемонстрировали необычайное терпение, работая со мной. Также фантастическую поддержку оказали мне технические рецензенты: Марк Дестефано (Marc Destefano), Чарльз Дуба (Charles Duba) и Маргарет Мозер (Margaret Moser) — при подготовке первого издания и Грейс Кендалл (Grace Kendall), Стивен Биман (Stephen Beeman) и Рид Коук (Reed Coke) — при подготовке второго. Их зоркие глаза и острые умы нашли много мест в рукописи, которые можно было уточнить и улучшить.

Я также хочу поблагодарить всех преподавателей, учивших меня и работавших со мной как коллеги. Сердечное спасибо доктору Рэнди Паушу (Randy Pausch) и Джесси Шеллу (Jesse Schell). Хотя я уже был профессором и дизайнером игр до встречи с ними, оба они сильно повлияли на мое понимание разработки и обучения. Я также очень благодарен Трейси Фуллертон (Tracy Fullerton), Марку Боласу (Mark Bolas) и Скотту Фишеру (Scott Fisher) — моим друзьям и наставникам в годы, когда я преподавал на кафедре «Игры и интерактивная среда» Южно-Калифорнийского университета. Мои коллеги по новой работе в Университете штата Мичиган также оказали мне большую помощь, в том числе: Эндрю Деннис (Andrew Dennis), взявший на себя художественное оформление для главы 35, Брайан Винн (Brian Winn), Элизабет Ла Пенсе (Elizabeth LaPensée), Рикардо Гуимараес (Ricardo Guimaraes) и другие. Многие блестящие преподаватели и друзья из USC и Мичиганского университета также помогли мне воплотить мои идеи в этой книге, среди них: Адам Лижкевич (Adam Liszkiewicz), Уильям Хубер (William Huber), Ричард Лемарчанд (Richard Lemarchand), Скотт Роджерс (Scott Rogers), Винцент Диамант (Vincent

Diamante), Сэм Робертс (Sam Roberts), Логан Вер Хоф (Logan Ver Hoef) и Маркус Дарден (Marcus Darden).

Многие мои друзья из отрасли компьютерных игр тоже помогали мне, выдвигая свои предложения для книги и высказывая свое мнение об идеях, представленных в ней. Среди них: Майкл Селлерс (Michael Sellers), Николас Фортуньо (Nicholas Fortugno), Дженова Чен (Jenova Chen), Зак Павлов (Zac Pavlov), Джозеф Стивенс (Joseph Stevens) и многие другие.

Спасибо также моим замечательным студентам, которым я читал лекции последние два десятка лет. Именно вы вдохновили меня написать эту книгу и убедили меня, что в моем стиле преподавания разработки игр есть что-то важное и особенное. Каждый день, когда я прихожу в аудиторию, меня вдохновляют ваши творчество, интеллект и страсть.

Наконец, я хочу поблагодарить вас, читатель. Спасибо за покупку этой книги и за интерес к разработке игр. Я надеюсь, что эта книга поможет вам заняться разработкой, и мне было бы приятно видеть, что вы делаете это благодаря знаниям, почерпнутым здесь.

Об авторе

Джереми Гибсон Бонд (Jeremy Gibson Bond) — профессор практики, преподает проектирование и разработку игр на кафедре «Медиа и информация» в Университете штата Мичиган (http://gamedev.msu.edu), которая последние несколько лет входит в десятку лучших программ обучения проектированию игр. Начиная с 2013 года участвует в международном фестивале и конференции независимых видеоигр IndieCade в качестве председателя по образованию и развитию, где каждый год возглавляет саммит IndieXchange. В 2013 году Джереми основал компанию ExNinja Interactive, благодаря которой разрабатывает свои независимые проекты видеоигр. Кроме того, Джереми несколько раз выступал на конференции Game Developers Conference.

До прихода на факультет Университета штата Мичиган Джереми три года работал на кафедре «Проектирование электрических систем и информатика» Мичиганского университета в Анн-Арборе, где преподавал проектирование игр и разработку программного обеспечения. С 2009 по 2013 год Джереми занимал должность ассистента профессора практики на кафедре «Игры и интерактивная среда» школы кинематографического искусства в Южно-Калифорнийском университете, которая удерживала звание школы номер один в Северной Америке по обучению проектированию игр в течение всего времени его работы там.

Джереми получил степень магистра развлекательных технологий в Центре развлекательных технологий при Университете Карнеги-Меллона в 2007 году и степень бакалавра в области радио, телевидения и кинематографа в Техасском университете в Остине в 1999-м. Свою карьеру он начинал как программист и разработчик прототипов в таких компаниях, как Human Code и Frog Design; преподавал в кампусе «Великий северный путь» (Ванкувер, провинция Британская Колумбия, Канада), в Университете штата Техас, в Институте искусств в городе Питтсбург, в муниципальном колледже Остина и в Техасском университете в Остине; а также работал для многих компаний, включая Walt Disney Imagineering, Maxis и Electronic Arts/Pogo.com. Когда он обучался в магистратуре, его команда создала онлайн-игру Skyrates, выигравшую премию Silver Gleemax среди стратегий на фестивале независимых игр в 2008 году. Кроме того, Джереми, вероятно, первый человек, преподававший проектирование игр в Коста-Рике.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ЧАСТЬ I

Проектирование игры и прототипирование на бумаге

Глава 1. Думать как дизайнер
Глава 2. Методы анализа игр
Глава 3. Многоуровневая тетрада
Глава 4. Фиксированный уровень
Глава 5. Динамический уровень
Глава 6. Культурный уровень
Глава 7. Действовать как дизайнер
Глава 8. Цели проектирования
Глава 9. Прототипирование на бумаге
Глава 10. Тестирование игры
Глава 11. Математика и баланс игры
Глава 12. Руководство игроком
Глава 13. Проектирование головоломок
Глава 14. Agile-мышление
Глава 15. Индустрия цифровых игр

1 Думать как дизайнер

Здесь начинается наше путешествие. В этой главе представлены основные теории дизайна, на которых построена вся книга. Здесь вы также встретите свое первое упражнение по проектированию игры и узнаете больше об основополагающей философии книги.

Вы — геймдизайнер

С этого момента вы — геймдизайнер, и я хочу, чтобы вы сказали вслух[1]:

«Я — геймдизайнер».

Это нормально. Вы можете сказать и погромче, даже если вас услышат. В сущности, как рассказывает психолог Роберт Чалдини в своей книге «Психология влияния»[2], если ваше заявление о чем-то слышат другие, вы с большей вероятностью будете этому следовать. Поэтому отправляйтесь и заявите в Facebook, своим друзьям, своей семье:

«Я — геймдизайнер».

Но что значит быть геймдизайнером? Эта книга поможет вам ответить на вопрос и даст вам инструменты для создания своих игр. Начнем с упражнения.

Bartok: пример игры

Впервые я увидел это упражнение на встрече с дизайнером игр Малкольмом Райаном (Malcolm Ryan), автором книги «Game Design Workshop», на конференции «Foundations of Digital Gaming». Цель этого упражнения — показать, как даже самое малое изменение правил игры может оказать огромное влияние на характер игрового процесса.

---

[1] Я благодарю своего бывшего профессора Джесси Шелла за то, что попросил меня сделать то же самое в аудитории, полной людей. Он также просил об этом читателей своей книги «The Art of Game Design: A Book of Lenses» (Boca Raton, FL: CRC Press, 2008), 1.

[2] Robert B. Cialdini. «Influence: The Psychology of Persuasion», New York: Morrow, 1993. (*Роберт Б. Чалдини. Психология влияния. СПб.: Питер, 2012. Прим. пер.*)

Bartok — простая игра с одной колодой карт, которая очень напоминает коммерческую игру *Uno*. В эту игру лучше играть с тремя друзьями, которые также хотят заняться проектированием игр; однако я написал цифровую версию игры, чтобы вы могли сыграть в нее сами. Нам прекрасно подойдет любая версия игры, с настоящей колодой карт или цифровая.[1]

ЦИФРОВАЯ ВЕРСИЯ ИГРЫ BARTOK

Цифровая версия игры Bartok доступна в разделе Chapter 1 на веб-сайте книги:

http://book.prototools.net

Цель

Первым избавиться от всех своих карт.

Игра

Вот основные правила игры *Bartok*:

1. Для игры используется обычная колода игральных карт, из которой нужно убрать джокеров, чтобы осталось ровно 52 листа (13 карт каждой масти, от двойки до туза).
2. Перетасуйте колоду и раздайте игрокам по 7 карт.
3. Положите оставшуюся часть колоды на стол рубашкой вверх. Это будет ничейная колода.
4. Снимите верхнюю карту с ничейной колоды и положите ее рядом лицом вверх. Это будет колода сброса.
5. Далее, по часовой стрелке, начиная с игрока слева от раздающего, каждый должен побить карту, лежащую сверху на колоде сброса. Если у игрока в руках нет такой карты, он должен взять одну карту из ничейной колоды (рис. 1.1).
6. Побить карту в колоде сброса можно картой:
   - Той же масти. (Например, если верхняя карта в колоде сброса — двойка треф, ее можно побить любой картой трефовой масти.)
   - Той же ценности. (Например, если верхняя карта в колоде сброса — двойка треф, ее можно побить любой другой «двойкой».)

[1] Изображения карт для рисунков и цифровых карточных игр, представленных в этой книге, основаны на наборе Vectorized Playing Cards 1.3, Copyright 2011, Chris Aguilar. Набор доступен на условиях лицензии LGPL 3 (http://www.gnu.org/copyleft/lesser.html) по адресу http://sourceforge.net/projects/vector-cards/.

7. Победителем считается тот, кто первым благополучно избавится от карт на руках.

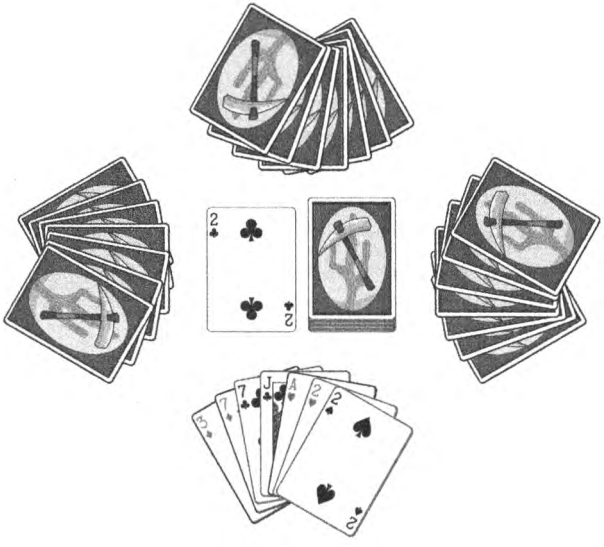


Рис. 1.1. Начальный расклад в игре *Bartok*. В данной ситуации игрок может пустить в игру любую из карт (семерка треф, валет треф, двойка черв, двойка пик)

Пробная игра

Попробуйте пару раз сыграть в эту игру, чтобы почувствовать, что это такое. Не ленитесь тщательно перетасовать колоду перед каждым раундом. Игра часто будет завершаться с колодой сброса, отсортированной так или иначе, и без тщательного перемешивания карт следующий раунд будет иметь результаты, зависящие от неслучайного распределения карт.

Устранение блочности. Этим термином называются стратегии разбиения групп (блоков) похожих карт. В случае успеха каждый раунд в игре *Bartok* завершается с колодой сброса, в которой все карты объединены в группы (блоки) по масти или по значению. Если не устранить эту блочность (не разбить группы), последующий раунд игры закончится намного быстрее, потому что на руки игрокам будут розданы карты, соответствующие друг другу.

Как утверждает математик и иллюзионист Перси Диаконис, семи хороших перетасовок достаточно для большинства игр[1]; но если вы испытываете проблемы, воспользуйтесь следующими стратегиями устранения блочности.

[1] Persi Diaconis, «Mathematical Developments from the Analysis of Riffle Shuffling», *Groups, Combinatorics and Geometry*, edited by Ivanov, Liebeck и Saxl. World Scientific (2003): 73–97. Доступна в электронном виде по адресу http://statweb.stanford.edu/~cgates/PERSI/papers/Riffle.pdf.

Вот несколько советов по устранению блочности в колоде карт, если обычная перетасовка не помогает:

- сдайте карты в несколько разных кучек, затем перемешайте эти кучки вместе;
- разбросайте карты на столе рубашкой вверх как получится, затем перемешайте их двумя руками, как воду или как костяшки домино, — это поможет разбить образовавшиеся блоки, затем соберите карты в одну колоду;
- сыграйте в игру *52 карты*: разбросайте карты по полу и соберите их.

Анализ: постановка правильных вопросов

После каждой пробной игры важно уметь задавать правильные вопросы. Конечно, для каждой игры вопросы немного отличаются, но вы можете основываться на следующих общих принципах:

- Сложность игры соответствует целевой аудитории? Игра слишком сложная, слишком простая или сложная ровно настолько, насколько нужно?
- Результат игры зависит от избранной стратегии или от везения? Не слишком ли сильно исход игры зависит от случайностей или, может быть, игра детерминирована настолько, что после захвата лидерства одним игроком другие лишаются шанса нагнать его?
- Допускает ли игра осмысленные, интересные решения? Когда вы получаете право сделать ход, есть ли у вас выбор между несколькими альтернативами и насколько интересным может быть выбор между ними?
- Вызывает ли игра интерес, когда право хода получают другие игроки? В состоянии ли вы оказать влияние на ходы, выбираемые другими игроками, или, может быть, другие игроки способны повлиять на ваш выбор?

Мы могли бы задать еще массу вопросов, но перечисленные выше — наиболее типичные.

Остановитесь ненадолго и подумайте, какие ответы можно дать в отношении игры Bartok, в которую вы только что сыграли, и запишите их. Если вы попробовали сыграть в эту игру с живыми соперниками, используя физическую колоду карт, попросите их также записать свои ответы на вопросы и затем коллективно обсудите их. Это устранит взаимное влияние на ответы других игроков.

Изменение правил

Как вы увидите далее в этой книге, проектирование игры — это прежде всего процесс.

1. Постепенное изменение правил небольшими шагами после каждой пробной игры.

2. Пробная игра с новыми правилами.
3. Анализ восприятия игры по новым правилам.
4. Проектирование новых правил, которые, по вашему мнению, могут сместить восприятие игры в нужном вам направлении.
5. Повторение процесса до появления ощущения удовлетворенности.

Повторяющийся процесс обдумывания изменений в правилах игры, реализация пробной игры по новым правилам, анализ влияния изменений на восприятие игры и переход в начало процесса для внесения следующих небольших изменений, называется *итеративным проектированием*. Более подробно об итеративном проектировании рассказывается в главе 7 «Действовать как дизайнер».

Например, применительно к игре *Bartok*, можно для начала выбрать одно из следующих трех изменений в правилах и опробовать его:

- **Правило 1:** если игрок играет двойкой, следующий игрок пропускает ход и берет из ничейной колоды две карты.
- **Правило 2:** если у игрока на руках есть карта, соответствующая по значению и цвету масти (черная или красная) верхней карте в колоде сброса, он может объявить: «Парная карта!» — и сыграть вне очереди. После этого игра продолжается с игрока, находящегося по левую руку от игрока, сделавшего внеочередной ход. Таким способом игроки могут заставлять друг друга пропускать ход.

  Например: первый игрок сыграл тройкой треф. У третьего игрока на руках имеется тройка пик, поэтому он объявил: «Парная карта!» — и побил тройку треф своей тройкой пик, сделав ход вне очереди, а второй игрок пропустил ход. После этого игра продолжилась с четвертого игрока.
- **Правило 3:** игрок должен объявить: «Последняя карта!», — если у него на руках осталась только одна карта. Если кто-то другой сказал эти слова первым, опоздавший игрок должен взять из ничейной колоды две карты (чтобы общее число карт у него на руках достигло трех).

Выберите одно из предложенных изменений и сыграйте в игру пару раз по новым правилам. Затем пусть каждый игрок запишет свои ответы на четыре вопроса, предложенных выше. Попробуйте также сыграть, добавив другое изменение (хотя я рекомендовал бы вносить изменения по одному и пробовать их по отдельности).

Если вы играете в цифровую версию игры, используйте флажки в экране меню для выбора разных вариантов.

Наблюдайте за неожиданностями в игре. Из-за недостаточно тщательного перемешивания карт или по каким-то другим причинам раунды игры могут отличаться друг от друга. Это называется *случайной неожиданностью*. Вы не должны принимать решений о дизайне игры, исходя из таких неожиданностей. Если что-то повлияло на игру неожиданным образом, обязательно несколько раз сыграйте пробную игру с этим изменением в правилах, чтобы убедиться, что неожиданность не случайна.

Анализ: сравнение раундов

Сыграв в игру с несколькими вариантами правил, проанализируйте результаты разных раундов. Загляните в свои записи и посмотрите, как каждый набор правил отразился на восприятии игры. Как вы уже понимаете, даже простое изменение правил может значительно изменить восприятие игры. Вот некоторые общие реакции на правила, описанные выше:

- **Начальные правила.**

  Многие игроки считают, что исходная версия игры довольно скучна. Отсутствуют интересные варианты действий, и по мере уменьшения количества карт на руках количество возможных вариантов развития игры сокращается, из-за чего игрок часто может сделать только один верный ход, не имея вариантов на выбор. Игра в значительной степени основана на случайности, и игроки не имеют веских причин следить за ходами других игроков, потому что не имеют никакой возможности повлиять друг на друга.

- **Правило 1:** *если игрок ходит двойкой, следующий игрок пропускает ход и берет из ничейной колоды две карты.*

  Это правило дает игрокам возможность влиять друг на друга, что увеличивает интерес к игре. Однако наличие двоек у игрока полностью зависит от удачи, а кроме того, каждый игрок может повлиять только на игрока слева, что часто кажется несправедливым. Тем не менее ходы других игроков вызывают больше интереса, потому что они (по крайней мере игрок справа) могут повлиять на вас.

- **Правило 2:** *если у игрока на руках есть карта, соответствующая по значению и цвету масти (черная или красная) верхней карте в колоде сброса, он может объявить: «Парная карта!» — и сыграть вне очереди. После этого игра продолжается с игрока, находящегося по левую руку от игрока, сделавшего внеочередной ход.*

  Это правило часто больше всего влияет на внимание игроков. Любой игрок может отобрать ход у другого игрока, поэтому все игроки уделяют больше внимания ходам других. Игра с этим правилом часто вызывает больше эмоций, чем игра с другими правилами.

- **Правило 3:** *игрок должен объявить: «Последняя карта!», если у него на руках осталась только одна карта. Если кто-то другой сказал эти слова первым, опоздавший игрок должен взять из ничейной колоды две карты.*

  Это правило начинает сказываться уже ближе к концу, поэтому не оказывает влияния на большую часть процесса, однако оно влияет на поведение игроков в конце игры. Это может придать игре интересную напряженность, так как игрок может вскочить и объявить: «Последняя карта!» — до того, как это сделает другой игрок, у которого тоже осталась одна карта. Это обычное правило в домино и в карточных играх, где цель игрока — сбросить все, что у него есть на руках, и помогает другим игрокам нагнать лидера, забывшего о правиле.

Проектирование желаемого восприятия игры

Теперь, увидев влияние разных правил на игру *Bartok*, можно приступить к выполнению обязанностей дизайнера и улучшить игру. Прежде всего следует решить, какой вы хотите сделать игру: захватывающей и драматичной, медленной и неторопливой — или придать ей более стратегический характер, уменьшив влияние случайностей?

Определив, как должна восприниматься ваша игра, поразмыслите об опробованных правилах и попытайтесь придумать дополнительные, которые подтолкнут восприятие игры в нужном вам направлении. Вот некоторые советы, которые желательно помнить, разрабатывая новые правила для игры.

- После каждой пробной игры изменяйте только что-то одно. Если вы измените (или даже немного скорректируете) сразу несколько правил, вам будет сложно определить, как повлияло каждое правило. Вносите изменения постепенно, и вам проще будет понять влияние каждого из них.
- Чем более крупные изменения вы вносите, тем больше пробных игр нужно сыграть, чтобы оценить их влияние на характер игры. Если вносятся только очень тонкие изменения, одного-двух раундов может оказаться достаточно, чтобы понять, как они повлияли на восприятие игры. Но при существенных изменениях в правилах понадобится больше раундов, чтобы исключить появление суждений, основанных на случайностях.
- Измените число — и вы измените восприятие. Даже, казалось бы, небольшое изменение может оказать большое влияние на процесс игры. Например, представьте, насколько динамичнее развивалась бы игра при наличии двух колод сброса, из которых игроки могли бы выбирать любую и у каждого на руках первоначально имелось бы только пять карт.

Конечно, изменять правила в карточных играх намного проще при игре вживую, с друзьями, чем при игре с цифровым прототипом. Это одна из причин, почему бумажные прототипы играют важную роль даже при разработке цифровых игр. В первой части этой книги обсуждается и бумажный, и цифровой дизайн, но большинство примеров и упражнений выполняется с бумажными играми, потому что развивать и тестировать их намного проще, чем цифровые аналоги.

Определение игры

Прежде чем углубиться в проектирование и итерации, следует уточнить, что подразумевается под такими терминами, как *игра* и *проектирование игры*. Многие очень умные люди пытались дать точное определение слову *игра*. Вот некоторые из них в хронологическом порядке.

- В 1978 году, в своей книге «The Grasshopper», Бернард Сьютс (бывший в ту пору профессором философии в университете Ватерлоо) заявил, что «игра — это добровольная попытка преодоления ненужных препятствий»[1].
- Легенда игровой индустрии Сид Мейер сказал, что «игра — это серия интересных решений».
- В книге «Game Design Workshop» Трейси Фуллертон определила игру как «закрытую формальную систему, которая вовлекает игроков в организованный конфликт и разрешает его неопределенность в неравноценности результатов»[2].
- В книге «The Art of Game Design» Джесси Шелл исследует несколько определений и, наконец, решает, что «игра — это работа по решению проблем с игривым настроем»[3].
- В книге «Game Design Theory» Кит Бургун представил намного более узкое определение игры: «Система правил, руководствуясь которыми агенты конкурируют друг с другом, делая неоднозначные, эндогенно значимые решения»[4,5].

Каждое из этих определений убедительно и по-своему правильно. Но гораздо важнее понимать, какой смысл вкладывал автор в свое определение.

Определение Бернарда Сьютса

Кроме краткого определения: игра — это добровольная попытка преодоления ненужных препятствий», Сьютс предложил более многословную и точную версию:

> Игра — это попытка добиться определенного положения, используя только средства, разрешенные правилами, где правила запрещают использовать более эффективные средства в пользу менее эффективных и принимаются только потому, что они делают возможной такую деятельность.

---

[1] Bernard Suits, *The Grasshopper* (Toronto: Toronto University Press, 1978), 56.

[2] Tracy Fullerton, Christopher Swain и Steven Hoffman. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games, 2nd ed.* (Boca Raton, FL: Elsevier Morgan Kaufmann, 2008), 43.

[3] Jesse Schell, *Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), 37.

[4] Keith Burgun. *Game Design Theory: A New Philosophy for Understanding Games* (Boca Raton, FL: A K Peters/CRC Press, 2013), 10, 19.

[5] Эндогенность присуща или возникает из внутренней системы вещей, поэтому под «эндогенно значимыми» подразумеваются решения, которые оказывают влияние на состояние игры и изменяют результат. Выбор цвета одежды для персонажа в игре *Farmville* не является эндогенно значимым, тогда как в *Metal Gear Solid 4* является, потому что в этой игре цвет одежды влияет на видимость персонажа врагам.

На протяжении всей своей книги Сьютс предпринимает и отражает нападки на это определение. Прочитав его книгу, я хочу отметить, что он нашел определение слову «игра», наиболее точно соответствующее использованию этого слова в обыденной жизни.

Однако важно также помнить, что это определение было дано в 1978 году. И хотя в то время уже существовали цифровые и ролевые игры, Сьютс либо не знал о них, либо намеренно их игнорировал. Фактически в главе 9 (в книге «The Grasshopper») Сьютс заявляет об отсутствии драматических игр, играя в которые игроки могли бы растрачивать свою драматическую энергию (как дети растрачивают физическую энергию, играя в подвижные и спортивные игры), даже при том, что именно к этой категории относятся ролевые игры, такие как *Dungeons & Dragons*[1].

Однако здесь не хватает сущей мелочи: несмотря на точность определения слова «игра», Сьютс ничего не предлагает дизайнерам, стремящимся создавать хорошие игры для других.

Чтобы лучше понять, что я имею в виду, улучите момент и сыграйте в фантастическую игру *Passage* Джейсона Рорера (Jason Rohrer): http://hcsoftware.sourceforge.net/passage/ (рис. 1.2)[2]. Чтобы пройти ее, потребуется всего пять минут, но она демонстрирует, насколько увлекательными могут быть даже короткие игры. Попробуйте сыграть в нее пару раз.

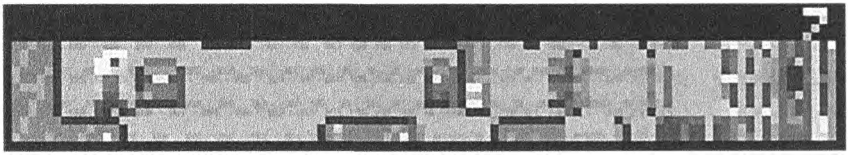


Рис. 1.2. Игра Passage Джейсона Рорера (выпущена 13 декабря 2007 года)

Определение Сьютса скажет вам, что да, это игра. В частности, это «открытая игра», которая, согласно его определению, имеет единственную цель — продолжать играть[3]. В игре *Passage* цель — продолжать играть максимально долго... Впрочем, не совсем так. Теоретически в *Passage* есть несколько целей, и только игрок может решить, к какой из них он будет стремиться, в их числе:

- забраться как можно дальше вправо, прежде чем персонаж погибнет (исследование);

---

[1] Suits, *The Grasshopper*, 95.

[2] Игра *Passage* была создана более десяти лет тому назад, и Рорер долго не добавлял поддержку новейших систем. Однако в прошлом я написал ему об этом, и он обновил свою сборку.

[3] Сьютс противопоставляет открытые игры закрытым, которые имеют определенную цель (например, пересечь финишную линию в гонке или избавиться от карт на руках в игре *Bartok*). Примером открытых игр, по определению Сьютса, могут служить «игры понарошку», в которые играют дети.

- заработать как можно больше очков, отыскивая сундуки с сокровищами (достижение);
- поиск жены (социализация).

Фактически каждая из этих целей в игре *Passage* может быть целью в жизни, и, так же как в жизни, эти цели являются взаимоисключающими до определенной степени. Если вы нашли жену в начале игры, вам сложнее будет собирать сундуки с сокровищами, потому что вдвоем невозможно входить в некоторые области, куда можно входить поодиночке (зато каждый шаг вправо будет давать теперь два очка вместо одного). Если вы решите собирать сокровища, вы больше времени будете тратить на перемещения по вертикали и не сможете исследовать пейзаж далеко справа.

Решив зайти как можно дальше вправо, вы не сможете собрать много сокровищ. В этой невероятно простой игре Ропер дает возможность принять несколько фундаментальных решений, которые каждый из нас должен приять в реальной жизни, и демонстрирует, как даже самые ранние решения могут влиять на всю оставшуюся жизнь. Здесь важно отметить, что он дает игрокам свободу выбора и показывает, что их выбор имеет значение.

Это пример одной из множества целей дизайнера, о которых я рассказываю в этой книге: *обретение понимания на практике*. В отличие от художественных произведений, которые могут стимулировать появление сочувствия к герою, показывая читателю отрезок его жизненного пути и принимаемые им решения, игры позволяют игрокам не только понять, к чему ведут принимаемые решения, но и получить чувство причастности к результатам, давая игроку возможность и ответственность принимать решения и затем показывая результат этих решений. В главе 8 «Цели проектирования» мы подробнее исследуем обретение понимания на практике и другие цели дизайнера.

Определение Сида Мейера

Утверждая, что «игра — это серия интересных решений», Мейер очень мало говорит об определении слова *игра* (многое, очень многое можно охарактеризовать как серию интересных решений, что фактически не относится к играм) и еще меньше о том, что, по его мнению, делает игру хорошей игрой. Как дизайнер игр *Pirates*, *Civilization*, *Alpha Centauri* и многих других, Сид Мейер считается одним из лучших дизайнеров, и он постоянно создает игры, в которых игроки могут принимать интересные решения. Возникает естественный вопрос: что делает решения *интересными*? Обычно решение представляет интерес, когда:

- у игрока на выбор есть несколько допустимых вариантов;
- каждый вариант имеет свои положительные и отрицательные последствия;
- результат каждого варианта предсказуем, но не гарантируется.

Это ведет нас ко второй из целей дизайнера: *создание интересных решений*. Если игрок имеет на выбор несколько вариантов, но один из них очевидно перевешивает другие, тогда фактически никакого выбора нет. В хорошо спроектированной игре игроки часто оказываются перед выбором из множества вариантов и сделать выбор очень сложно.

Определение Трейси Фуллертон

В своей книге Трейси заявляет, что ее больше интересуют инструменты дизайнера, позволяющие создавать лучшие игры, чем философское определение слова *игра*. Соответственно, ее слова: «закрытая формальная система, вовлекающая игроков в организованный конфликт и разрешающая его неопределенность в неравноценности результатов» — не только являются хорошим определением игры, но и перечисляют элементы, которые дизайнеры могут изменять в своих играх:

- **Формальные элементы:** элементы, отличающие игру от других видов интерактивной среды — правила, процедуры, игроки, ресурсы, цели, границы, конфликт и результат.
- **(Динамические) системы:** методы взаимодействий, развивающиеся в процессе игры.
- **Организация конфликта:** способы взаимодействий игроков друг с другом.
- **Неопределенность:** взаимосвязь между случайностью, детерминизмом и стратегией игрока.
- **Неравноценность результатов:** как завершается игра? Победой игрока, проигрышем или как-то иначе?

В книге Фуллертон упоминается еще один важный элемент — *постоянная практика в создании игр*. Лучшим дизайнером игр можно стать, только создавая игры. Какие-то игры, которые вы создадите, будут ужасными, — я тоже создал несколько таких игр, но даже создавая плохие игры, вы продолжаете учиться, и с каждой созданной вами игрой вы будете совершенствовать свои навыки и лучше понимать, как создавать хорошие игры.

Определение Джесси Шелла

Шелл определяет игру как «работу по решению проблем с игривым настроем». Это определение перекликается с определением Сьютса и точно так же определяет игру с точки зрения игрока. Согласно обоим определениям, именно игривый настрой игрока делает игру игрой. Фактически в своей книге Сьютс приводит пример, когда два человека вовлекаются в одну и ту же деятельность, но один воспринимает ее как игру, а другой — нет. В качестве примера Сьютс приводит состязание в беге, когда один бегун бежит, просто потому что желает поучаствовать в состязании, а другой — потому что знает, что в конце дистанции находится

бомба и ее нужно обезвредить. Согласно Сьютсу, участвуя в одном и том же состязании, первый бегун, принимающий участие ради участия, будет следовать установленным правилам, и такой настрой Сьютс называет *игровитым, развлекательным*. Другой бегун, который спешит обезвредить бомбу, напротив, будет стремиться нарушить правила, потому что относится к состязанию *серьезно* (он должен обезвредить бомбу), а не как к игре.

Сьютс предлагает использовать термин *игровитый настрой* для описания отношения того, кто охотно принимает участие в игре. Именно благодаря игровитому настрою игроки без возражений следуют правилам игры, даже если есть более простой способ достижения цели (которую Сьютс называет *игровой целью*). Например, цель игры в гольф — загнать мяч в лунку, однако существует много более простых способов сделать это, чем бить по мячу клюшкой, находясь в сотнях ярдов от лунки. Когда люди имеют игровитый настрой, они придумывают себе сложности только ради радости их преодоления.

Итак, еще одна цель дизайнера — *поощрять игровитое отношение*. Ваши игры должны проектироваться так, чтобы игроки с удовольствием преодолевали ограничения, устанавливаемые правилами. Поразмышляйте, для чего существует каждое правило и как оно влияет на восприятие игры. Если игра хорошо сбалансирована и устанавливает разумные правила, игроки будут получать удовольствие от накладываемых ограничений, а не испытывать раздражение.

Определение Кита Бургуна

Определяя игру как «систему правил, руководствуясь которыми агенты конкурируют друг с другом, делая неоднозначные, эндогенно значимые решения», Бургун попытался вытолкнуть обсуждение игр из колеи, куда он попал, по его мнению, сузив смысл игры до чего-то более понятного и легче поддающегося изучению. Суть этого определения состоит в том, что игрок делает выбор, и этот его выбор одновременно является неоднозначным (игрок не знает точно, что получится в результате его выбора) и эндогенно значимым (выбор имеет смысл, потому что оказывает заметный эффект на игровую систему).

Определение Бургуна намеренно ограниченно и целенаправленно исключает виды деятельности, которые многие считают играми (в том числе бег и другие состязания, основанные на физических навыках), а также игры-размышления, такие как *The Graveyard*, выпущенная студией Tale of Tales, в которой игрок управляет старушкой, блуждающей по кладбищу. Они были исключены, потому что принимаемые в них решения лишены неоднозначности и эндогенной значимости.

Бургун выбрал такое ограниченное определение, потому что хотел добраться до сути игр и понять, что делает их уникальными. При этом он делает несколько ценных замечаний, в том числе заявляет, что положительное восприятие мало зависит от того, является ли некоторое занятие игрой. Даже самая скучная игра остается игрой, просто это плохая игра.

Беседуя с другими дизайнерами, я обнаружил отсутствие единого мнения о том, что можно считать *игрой*. Игры — это сфера, пережившая бурный рост и распространение за последнюю пару десятков лет, и нынешний взрыв независимой разработки игр только ускорил темпы. Ныне еще больше людей, чем когда-либо, с разными представлениями и навыками, вовлекается в игровую индустрию; как результат, определение сферы расширяется, что по понятным причинам беспокоит некоторых людей, потому что это выглядит как размывание границ того, что можно назвать игрой. Реакция Бургуна — это выражение его беспокойства, что неукоснительное развитие сферы усложнится из-за отсутствия четкого определения границ, окружающих ее.

Почему важно иметь определение игры?

В 1953 году в своей книге «Philosophical Investigations»[1] Людвиг Витгенштейн предложил использовать термин *игра* в том смысле, в каком он используется в разговорной речи, для обозначения широкого круга самых разных занятий, обладающих некоторыми общими чертами (он уподоблял их семейному сходству), несоединимыми, между тем, в одном определении. В 1978 году Бернард Сьютс раскритиковал эту идею в своей книге «The Grasshopper», настаивая на конкретном определении слова *игра*, которое вы видели выше в этой главе. Однако, как отметил Крис Бейтман в своей книге «Imaginary Games», несмотря на то что Витгенштейн использовал слово *игра* в качестве примера, в действительности он пытался донести более важную мысль: это слова создаются для определения чего-то, а не наоборот.

В 1974 году (между публикациями книг «Philosophical Investigations» и «The Grasshopper») философ Мэри Миджли опубликовала статью «The Game Game»[2], в которой она исследовала и опровергла заявление Витгенштейна о «семейном сходстве», не давая строгого определения слову *игра*, но исследуя причины его существования. В своей статье она согласилась с Витгенштейном, что слово *игра* возникло задолго до появления игр, но заявила, что слова, такие как *игра*, определяются не вещами, которые они подразумевают, а потребностями, которым они соответствуют. Она отмечает:

> Мы можем принять нечто за стул при условии, что оно надлежащим образом сделано для сидения на нем, независимо от того, имеет ли оно форму пластикового баллона, большого куска пеноматериала или подвешенной к потолку корзины. Если вы осознаете потребность, то понимаете, имеет ли она надлежащие характеристики, и пригодность для удовлетворения этой потребности и есть то, что у стульев имеется общего[3].

---

[1] Перевод на русский язык доступен в электронном виде по адресу http://filosof.historic.ru/books/item/f00/s00/z0000273/st000.shtml. — *Примеч. пер.*

[2] Перевод на русский язык можно найти по адресу https://iphras.ru/uplfile/root/biblio/phan/2017_2/30-56.pdf. — *Примеч. пер.*

[3] Mary Midgley. «The Game Game», Philosophy 49, no. 189 (1974): 231–53.

В своей статье Миджли стремится понять те потребности, которые удовлетворяются играми. Она полностью отвергает идею, что игры являются закрытыми системами, ссылаясь на множество примеров, когда игры находят продолжение в реальной жизни, и отмечая, что игры не могут быть закрытыми, потому что у людей есть причины для вступления в игру. Для нее эта причина имеет первостепенное значение. Вот лишь несколько причин для самого процесса игры.

- **Люди стремятся к организованному конфликту:** Миджли отмечает: «Желание шахматиста — это не желание абстрактной интеллектуальной деятельности вообще, которое сдерживается и встречает препятствия в виде определенного набора правил. Это желание определенного вида интеллектуальной деятельности, которая направляется правилами шахматной игры». Как указывает Сьютс в своем определении, ограничивающие правила существуют именно для того, чтобы сложностями преодоления этих ограничений вызвать интерес у игроков.
- **Люди желают попробовать себя в какой-то другой роли:** мы все прекрасно осознаем, что у нас только одна жизнь (по крайней мере, мы не можем проживать две жизни сразу), и игра дает нам возможность попробовать прожить ее как-то иначе. Например, игра *Call of Duty* позволяет игроку прожить жизнь солдата, а игра *The Graveyard* — почувствовать себя старой женщиной, так же как роль Гамлета позволяет актеру почувствовать себя беспокойным принцем датским.
- **Люди стремятся пережить волнение:** душевным волнениям посвящены многие популярные произведения, будь то боевики, драматичные репортажи из зала суда или любовные романы. В этом отношении игры отличаются лишь тем, что игрок активно участвует в волнительных перипетиях, а не впитывает их, как это имеет место для большинства линейных произведений. Как игрок, вы не смотрите, как кого-то преследуют зомби, потому что в игре преследуют вас.

Миджли считает крайне важным учитывать потребности, удовлетворяемые играми, чтобы осознать их важность для общества, а также положительное и отрицательное влияние на людей, играющих в них. Оба — и Сьютс, и Миджли — говорили о потенциально захватывающих качествах игр еще в 1970-х, задолго до повсеместного распространения видеоигр и появления беспокойств в обществе по поводу зависимости игроков от игр. Понимание этих потребностей и осознание их движущей силы может быть невероятно полезным для разработчиков игр.

Туманная природа определений

Как заметила Миджли, полезно поразмышлять над определением слова *игра* с точки зрения тех потребностей, что она удовлетворяет. Она также отметила, что шахматист хочет играть не в любую игру, а именно в шахматы. Придумать всеобъемлющее определение слову *игра* сложно еще и потому, что одно и то же слово несет разный смысл для разных людей в разные времена. Когда я говорю, что пошел играть в игру, я обычно подразумеваю консоль или видеоигру; однако когда моя супруга говорит

то же самое, она обычно имеет в виду *Scrabble* или другую игру в слова. Когда мои родители говорят, что хотят сыграть в игру, под этим подразумевается что-то вроде *Ticket to Ride* Алана Р. Муна (настольная игра, интересная, но не требующая от игроков чрезмерной конкуренции друг с другом), а мои родственники под игрой обычно понимают игру в карты или в домино. Даже в рамках нашей семьи слово *игра* имеет очень широкое толкование.

Кроме того, значение слова *игра* постоянно изменяется. Когда создавались первые компьютерные игры, никто не мог представить себе индустрию с многомиллиардным оборотом, которую мы видим теперь, или фантастический взлет инди-ренессанса, который мы наблюдаем последние несколько лет. Все знали лишь то, что компьютерные игры очень похожи на настольные (я имею в виду *Space War*), и называли их «компьютерными», чтобы обозначить отличие от прежнего значения слова *игра*.

Цифровые игры эволюционировали постепенно — каждый новый жанр чем-то напоминал предшествующие, и попутно расширялось значение слова *игра*, охватывавшее их.

Теперь, когда художественная форма достигла зрелости, в эту сферу стали приходить дизайнеры из других дисциплин и привносить свои собственные идеи о применении методов и технологий, разработанных для создания цифровых игр. (Возможно даже, что вы один из них.) Когда новые художники и разработчики приходят в эту сферу, некоторые из них создают вещи, сильно отличающиеся от того, что мы привыкли считать игрой. И это хорошо; я бы даже сказал, что это фантастически хорошо! И так думаю не я один. Международный фестиваль независимых игр, IndieCade, каждый год ищет игры, раздвигающие границы значения слова *игра*. По словам председателя фестиваля Селии Пирс и директора фестиваля Сэма Робертса, если независимый разработчик назовет игрой нечто интерактивное, созданное им, IndieCade примет его разработку как игру[1].

Итоги

После всех этих переплетающихся и иногда противоречивых определений вам может быть интересно узнать, почему в этой книге так много внимания уделяется исследованию определения слова *игра*. Должен признать, что в своей повседневной практике педагога и дизайнера игр я не трачу много времени на борьбу с определениями слов. Как говорил Шекспир, как ни назови розу, она все равно будет пахнуть розой, иметь шипы и отличаться хрупкой красотой. Тем не менее я считаю, что понимание этих определений может быть важным для вас как дизайнера.

- Определения помогают понять, чего люди ждут от ваших игр. Это особенно верно, если вы работаете в определенном жанре или для определенной ауди-

[1] Эти слова были сказаны Селией Пирс (Celia Pearce) и Сэмом Робертсом (Sam Roberts) во время обсуждения заявок на участие в фестивале IndieCade East 2014 и опубликованы на веб-сайте IndieCade: http://www.indiecade.com/submissions/faq/.

тории. Понимание, как ваша аудитория определяет этот термин, поможет вам создавать для нее лучшие игры.

- Определения помогут вам понять не только ядро определяемой идеи, но и периферию. Читая эту главу, вы встретили несколько разных определений, данных разными людьми, и у каждого было свое ядро и своя периферия (например, игры, четко укладывающиеся в определение [ядро], и игры, соответствующие ему лишь с большой натяжкой [периферия]). Места, где определения вступают в противоречия, могут служить признаком интересных областей для исследований через создание новых игр. Например, определения Фуллертон и Миджли в части классификации игр как закрытых систем подсвечивают ранее неисследованную область, которая в 2000-х превратилась в игры альтернативной реальности (Alternate Reality Games, ARG), жанр, разрывающий закрытый магический круг игры[1].
- Определения помогают яснее выражать свои мысли при общении с другими людьми, действующими в этой же сфере. В этой главе больше ссылок и сносок, чем в любой другой в этой книге, потому что я хочу дать вам возможность исследовать философские аспекты игры, далеко выходящие за рамки этой книги (тем более что в этой книге основное внимание уделяется практике создания цифровых игр). Следуя по этим ссылкам и читая первоисточники, вы сможете улучшить свое понимание игр.

Основные уроки этой книги

В этой книге вы узнаете, как разрабатывать не только игры. Фактически она учит создавать интерактивные пространства любого вида. Вот как я их определяю:

> **Интерактивное пространство — это любое пространство, созданное дизайнером, ограниченное правилами, средой или технологиями и интерпретируемое людьми через игру.**

Такое определение преднамеренно делает термин *интерактивное пространство* очень емким. Фактически всякий раз, пытаясь создать подобное пространство для людей — будь то игра, сюрприз на день рождения или даже сценарий свадьбы, — вы будете использовать те же инструменты, с которыми познакомитесь как дизайнер игр. Процессы, описываемые в книге, — это не просто правильный подход к про-

[1] Первой масштабной игрой в жанре ARG стала *Majestic* (Electronic Arts, 2001). После регистрации игроки в этой игре начинали получать по ночам таинственные сообщения по телефону, электронной почте и даже по факсу. Среди менее масштабных игр в жанре ARG можно назвать игру *Assassin*, в которую играют во многих студенческих городках. Игроки в этой игре могут совершать «политические убийства» («assassinate») друг друга (обычно с помощью игрушечного бластера Nerf или водяного пистолета или даже просто щелкнув по фотографии) в любой момент, когда они находятся за пределами учебных аудиторий. Один из любопытных аспектов таких игр — они всегда вмешиваются в нормальную жизнь.

ектированию игр. Они с успехом могут использоваться для решения любых задач проектирования, а *итеративный процесс проектирования*, представленный в главе 7 «Действовать как дизайнер», является важнейшим методом улучшения качества любого проекта.

Блестящими дизайнерами игр не рождаются, ими становятся. Мой друг Крис Свейн[1] любит говорить: «Проектирование игр — это 1 % вдохновения и 99 % итераций», — перефразируя известную цитату Томаса Эдисона. Он абсолютно прав, и одна из замечательных особенностей проектирования игр (в отличие от прежде упомянутых примеров с сюрпризом на вечеринке и сценарием свадьбы) — вы получаете шанс создать свой дизайн, сыграть пробную игру, внести мелкие изменения и снова сыграть. С каждым прототипом — и с каждой итерацией по улучшению прототипа — вы улучшаете свои навыки дизайнера. Точно так же, достигнув разделов этой книги, где рассказывается о разработке цифровых игр, продолжайте экспериментировать, совершая итерации. Каждый фрагмент кода, каждый учебный пример в этой книге приводится с целью показать вам, как создавать действующие прототипы игр, но каждый пример когда-то заканчивается, и с этого места должна начинаться ваша работа как дизайнера. Каждый из представленных прототипов можно встроить в другую, более крупную, более надежную и более сбалансированную игру, и я призываю вас поступать именно так.

Что дальше

Теперь, когда вы узнали чуть больше о проектировании игр и исследовали разные определения слова *игра*, пришло время заняться более глубоким изучением методов анализа, которые дизайнеры игр используют для понимания игр и приемов их создания. В следующей главе исследуются некоторые такие методы, применяющиеся последние несколько лет, а глава, следующая за ней, объединяет их в *многоуровневую тетраду* (Layered Tetrad), которая будет использоваться на протяжении оставшейся части книги.

[1] Крис Свейн участвовал в создании первого издания *Game Design Workshop* вместе с Трейси Фуллертон и много лет преподавал курс с тем же названием в Южно-Калифорнийском университете. Теперь он предприниматель и независимый геймдизайнер.

2 Методы анализа игр

Игрология, или *людология*, — так называется наука, изучающая игры и проектирование игр. За последнее десятилетие игрологи предложили разные способы анализа игр, чтобы упростить понимание и обсуждение структуры, основных элементов игр и их влияния на игроков и общество в целом.

В этой главе представлено несколько наиболее часто используемых методов анализа, которые вы, как дизайнер, должны знать. Следующая глава (глава 3 «Многоуровневая тетрада») объединяет идеи, заложенные в этих методах, в многоуровневую тетраду (layered tetrad), которая будет использоваться на протяжении оставшейся части книги.

Методы анализа для игрологии

В этой главе рассказывается о следующих методах:

- **MDA:** впервые представлен Робин Ханике (Robin Hunicke), Марком Ле Бланом (Marc LeBlanc) и Робертом Зубеком (Robert Zubek). Название MDA — это аббревиатура от «Mechanics, Dynamics, and Aesthetics» (механика, динамика и эстетика — МДЭ). Этот метод наиболее известен профессиональным дизайнерам игр и поднимает важные вопросы о различиях в подходах к играм дизайнеров и игроков.
- **Формальные, драматические и динамические элементы:** представлен Трейси Фуллертон (Tracy Fullerton) и Крисом Свейном (Chris Swain) в книге «Game Design Workshop». Этот метод сосредоточен на применении конкретных аналитических инструментов, помогающих дизайнерам создавать более качественные игры и продвигать их идеи. Тесно связан с историей киноискусства.
- **Простая тетрада:** представлен Джесси Шеллом в его книге «The Art of Game Design». Метод простой тетрады (elemental tetrad) предполагает деление игры на четыре основных элемента: механика, эстетика, сценарий и технология.

Каждый из этих методов имеет свои достоинства и недостатки, и каждый внес свой вклад в разработку метода многоуровневой тетрады, представленного в этой книге. Далее они рассматриваются в порядке их публикации.

МДЭ: механика, динамика и эстетика

Впервые предложенный на конференции Game Developers Conference в 2001 году и формализованный в 2004 году, в статье «MDA: A Formal Approach to Game Design and Game Research»[1], МДЭ является наиболее широко используемым методом анализа в игрологии. Ключевыми элементами МДЭ являются определения механики, динамики и эстетики; понимание разных точек зрения, с которых дизайнеры и игроки рассматривают игру; и предположение, что дизайнеры должны сначала подходить к созданию игры с позиции эстетики, а затем обращаться к динамике и механике, генерирующим эту эстетику.

Определения механики, динамики и эстетики

Один из аспектов, который может вызвать путаницу при обсуждении трех подходов, рассматриваемых в этой главе, — это использование одинаковых слов в их названиях, потому что каждый подход определяет их по-своему. Вот как они определяются в МДЭ[2]:

- **Механика:** конкретные компоненты игры на уровне представления данных и алгоритмов.
- **Динамика:** поведение компонентов механики во время выполнения в ответ на ввод игрока и вывод друг друга.
- **Эстетика:** желательные эмоциональные реакции, возникающие у игрока, когда он взаимодействует с игровой системой[3].

Игра с точки зрения игрока и дизайнера

Согласно методике МДЭ, дизайнер сначала должен рассмотреть игру с точки зрения *эстетики* — эмоций, которые, по его мнению, должны испытывать игроки во время игры. Определившись с эстетикой, он может вернуться к динамике, поддерживающей эти эмоции, и, наконец, заняться механикой, создающей эту динамику. Игроки, в свою очередь, смотрят на игру с противоположного конца этой цепочки: сначала экспериментируют с механикой (обычно читая правила игры), затем пробуют динамику, пытаясь сыграть в игру, и, наконец, испытывают эстетику (как мы надеемся), которую задумывал дизайнер (рис. 2.1).

---

[1] Robin Hunicke, Marc LeBlanc, and Robert Zubek, «MDA: A Formal Approach to Game Design and Game Research», in *Proceedings of the AAAI workshop on Challenges in Game AI Workshop* (San Jose, CA: AAAI Press, 2004), http://www.cs.northwestern.edu/~hunicke/MDA.pdf.

[2] Там же, с. 2.

[3] Обратите внимание на это очень необычное определение эстетики. Ни один другой подход не определяет эстетику подобным образом. Обычно под эстетикой подразумеваются философские понятия красоты, уродства и т. д. Выражаясь простым языком, эстетика дизайна — это основная цель дизайна.

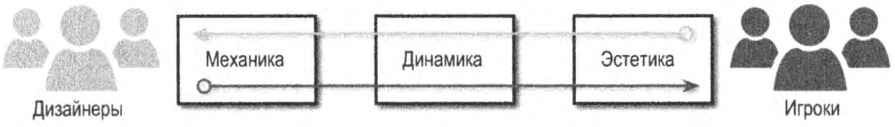


Рис. 2.1. Согласно методике МДЭ, дизайнеры и игроки рассматривают игру с разных направлений[1]

Проектирование от эстетики к динамике и механике

Исходя из этих разных точек зрения, метод МДЭ предполагает, что дизайнеры должны сначала добиться у игрока желаемого эмоционального отклика (эстетики), а затем вернуться к динамике и механике, соответствующим этой эстетике.

Например, детские игры часто проектируются так, чтобы до самого конца игры у игрока оставалось впечатление, что он все делает правильно и имеет все шансы на победу. Чтобы породить это впечатление, игрок должен чувствовать, что конец игры не является неизбежностью и он может надеяться на удачу на протяжении всей игры. Вспомните об этом, когда будете рассматривать макет игры *змейки-лесенки* (*Snakes and Ladders*).

Змейки-лесенки

Змейки-лесенки — это настольная игра для детей, появившаяся в Древней Индии, где она была известна под названием *Мокша патам*[2]. Игра не требует никаких навыков, и выигрыш в ней полностью зависит от удачи. Выполняя ход, игрок бросает один кубик и перемещает свою фишку на выпавшее число клеток. Первоначально фишки находятся вне пределов игрового поля, поэтому если на первом ходе у игрока выпадает 1, он ставит свою фишку на первое поле. Цель состоит в том, чтобы первым достичь конца игрового поля (клетки с номером 100). Если ход заканчивается на клетке, откуда начинается зеленая стрелка (лесенка), игрок должен передвинуть фишку на поле, где она заканчивается (например, если ход закончился на клетке 1, то он должен передвинуть фишку на поле с номером 38). Если ход заканчивается на клетке, откуда начинается красная стрелка (змейка), игрок должен передвинуть фишку на поле, где она заканчивается (например, если ход закончился на клетке 87, то он должен передвинуть фишку на поле с номером 24).

Расположение змеек и лесенок на макете игрового поля, изображенного на рис. 2.2, имеет значение. Вот несколько примеров, почему:

- Лесенка из клетки 1 в клетку 38. Она дает возможность игроку, у которого выпало число 1 в первом ходе (что обычно создает ощущение неудачи), перепрыгнуть сразу на клетку 38 и захватить лидерство.

---

[1] С небольшими изменениями взят из статьи: Hunicke, LeBlanc, and Zubek, «MDA: A Formal Approach to Game Design and Game Research».

[2] Jack Botermans, *The Book of Games: Strategy, Tactics, & History* (New York / London: Sterling, 2008), 19.

- В последнем ряду игрового поля есть три змейки (из клетки 93 в клетку 73, из 95 в 75 и из 98 в 79). Они замедляют движение игроков, оказавшихся ближе к концу игры.
- Змейка из 87 в 24 и лесенка из 28 в 84 образуют любопытную пару. Если игрок окажется в клетке 28 и перескочит в клетку 84, есть вероятность, что на следующем ходе он окажется в клетке 87 и скатится назад, в клетку 24. Напротив, если игрок окажется в клетке 87 и вернется назад в клетку 24, у него будет шанс попасть в клетку 28 и перескочить в клетку 84.

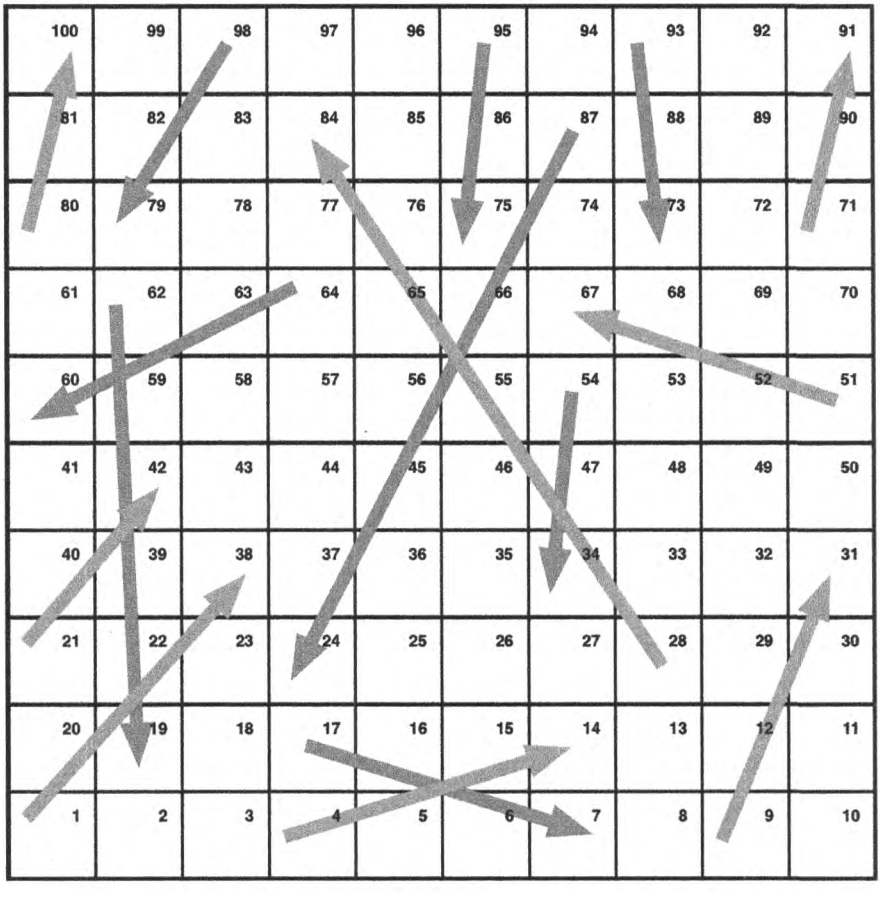


Рис. 2.2. Макет классической игры *змейки-лесенки*

Каждый из этих примеров размещения змеек и лесенок наглядно показывает, как можно заронить надежду и помочь игрокам поверить, что положение в игре может претерпевать драматические изменения. В отсутствие змеек и лесенок у значительно отставшего игрока не было бы никакой надежды догнать других.

В этой оригинальной версии игры желаемая эстетика заключается в создании ощущения надежды, поворота удачи и волнения, не ставя при этом игроков перед выбором. *Механика* — это змейки и лесенки, а *динамика* — пересечение *эстетики* и механики, когда столкновение с механикой приводит к переживаниям чувств надежды и волнения.

Изменение игры змейки-лесенки для придания ей более стратегического характера

Более взрослые игроки часто ищут большей сложности в играх и хотят чувствовать, что выигрывают не случайно, а путем использования стратегии выбора. Чтобы дать игрокам ощущение участия в стратегической игре, мы, как дизайнеры, можем изменить правила (элемент механики), не меняя игровое поле. Например, можно было бы внести следующие изменения в правила:

1. Дать игрокам по две фишки вместо одной.
2. Выполняя ход, игрок бросает два кубика.
3. Игрок может сложить очки, выпавшие на двух кубиках, и двинуть одну фишку — или двинуть две фишки, каждую на число очков, выпавшее на одном из кубиков.
4. Игрок может пожертвовать очками, выпавшими на одном из кубиков, и двинуть одну из фишек соперника назад на число очков, выпавшее на другом кубике.
5. Если фишка игрока окажется в одной клетке с фишкой любого соперника, фишка соперника спускается на один ряд вниз. (Например, из клетки 48 она переместится в клетку 33, а из клетки 33 — в клетку 28 и тут же взлетит по лесенке в клетку 84!)
6. Если обе фишки игрока оказываются в одной клетке, одна из них поднимается на ряд выше. (Например, если фишки встретились в клетке 61, одна из них должна подняться в клетку 80 и затем по лесенке взлететь в клетку 100!)

Эти изменения позволят игрокам принимать стратегические решения (и изменят динамику игры). Правила 4 и 5, например, дают возможность напрямую помешать или помочь другим игрокам[1], что может способствовать созданию альянсов или обострению соперничества. Правила с 1-го по 3-е тоже позволяют принимать стратегические решения и уменьшить влияние фактора случайности на ее ход. Выбирая, как двигать фишки, и имея возможность не двигать их вообще, умный игрок может избежать попадания на змейки.

Это лишь один из примеров, как дизайнеры могут менять механику для изменения динамики игры и достижения эстетических целей.

[1] Примером помощи другому игроку может служить спуск его фишки на один ряд, в клетку, где начинается лесенка, ведущая вверх.

Формальные, драматические и динамические элементы

В отличие от методики МДЭ, помогающей дизайнерам и критикам лучше понимать и обсуждать игры, подход, получивший название «Формальные, драматические и динамические элементы»[1], был создан Трейси Фуллертон и Крисом Свейном, чтобы помочь слушателям их курса «Game Design Workshop» в Южно-Калифорнийском университете эффективнее проектировать игры.

Эта методика предполагает деление игры на элементы трех типов.

- **Формальные элементы:** элементы, отличающие игры от других форм интерактивных пространств и образующие структуру игры. К формальным элементам относятся правила, ресурсы и границы.
- **Драматические элементы:** история и сюжет игры, включая замысел. Драматические элементы связывают игру, помогают игрокам понять правила и поощряют эмоциональную заинтересованность игрока в исходе игры.
- **Динамические элементы:** игра в движении. Когда игрок превратит правила в фактический игровой процесс, игра перейдет в динамические элементы. К динамическим элементам относятся: стратегия, поведение и взаимоотношения между игровыми сущностями. Важно отметить, что понятие динамических элементов в этом подходе схоже с понятием динамики в МДЭ, но шире, потому что включает больше, чем поведение механики во время выполнения.

Формальные элементы

В книге «Game Design Workshop» определяется семь формальных элементов, отличающих игры от других интерактивных форм:

- **Схема взаимодействия с игроком:** как происходит взаимодействие с игроком? Это одиночная игра, индивидуальная, групповая, многосторонняя (несколько игроков, состязающихся друг с другом, как в большинстве настольных игр), односторонняя (один игрок против всех других, как в некоторых мини-играх *Mario Party* или в настольной игре *Scotland Yard*), совместная или даже для нескольких отдельных игроков, играющих против одной и той же системы?
- **Цели:** чего пытаются добиться игроки в игре? В каком случае игрок считается выигравшим?
- **Правила:** правила ограничивают действия игроков, определяя, что они могут, а чего не могут в игре. Многие правила записаны явно и включены в игру, но некоторые неявно понимаются всеми игроками. (Например, в игре *Monopoly* нет правила, явно указывающего, что нельзя украсть деньги из банка, но все следуют ему.)

[1] Tracy Fullerton, Christopher Swain, and Steven Hoffman. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games, 2nd ed.* (Boca Raton, FL: Elsevier Morgan Kaufmann, 2008.)

- **Процедуры:** виды действий, выполняемых игроками в игре. Правило в игре *змейки-лесенки* сообщает игроку, что тот должен бросить кубик и двинуть фишку на выпавшее число клеток. Процедура, определяемая этим правилом, — это комплекс фактических действий, связанных с броском кубика и перемещением фишки. Процедуры часто определяются комплексом из нескольких правил. Некоторые не зависят от правил: например, блеф является важной составляющей игры в покер, хотя он никак не оговаривается в правилах.
- **Ресурсы:** ресурсы — это элементы, имеющие ценность в игре. К ним относятся, например, деньги, здоровье, предметы и имущество.
- **Границы:** где заканчивается игра и начинается реальность? В своей книге «Homo Ludens»[1] Йохан Хейзинга описывает, как игры создают временный мир, в котором действуют правила игры, а не правила обычного мира, что стало известно как «магический круг». В спортивных играх, таких как футбол или хоккей, магический круг определяется границами игрового поля; но в играх альтернативной реальности, таких как *I Love Bees* (альтернативная реальность для *Halo 2*), границы оказываются более расплывчатыми.
- **Исход:** как заканчивается игра? Игры имеют не только конечные, но и дополнительные результаты. В играх с нулевой суммой, таких как шахматы, конечным результатом является победа одного игрока и поражение другого. В ролевой игре на бумаге, такой как *Dungeons & Dragons*, имеются дополнительные результаты, когда игрок побеждает противника или переходит на новый уровень, и даже гибель персонажа часто не является конечным результатом, потому что есть возможность воскресить его.

По словам Фуллертон, формальные элементы имеют еще один характерный признак — когда они удаляются, игра перестает существовать. Если убрать из игры правила, исход и т. д., она перестанет быть игрой.

Драматические элементы

Драматические элементы помогают сделать правила и ресурсы понятнее и увеличить эмоциональную отдачу.

Фуллертон определяет три драматических элемента:

- **Замысел:** предыстория игрового мира. По замыслу в игре *Monopoly* каждый игрок является владельцем фирмы, оперирующей недвижимостью, и пытается монополизировать рынок корпоративной недвижимости в Атлантик-Сити, Нью-Джерси. В *Donkey Kong* игрок пытается в одиночку спасти Паулину от похитившей ее гориллы. Замысел создает основу, на которой строится остальная часть сюжета игры.
- **Персонаж:** персонажи — это индивидуумы, вокруг которых вращается сюжет игры, будь то безымянный, почти неопределенный и молчаливый главный герой

[1] *Йохан Хейзинга.* Homo ludens. Человек играющий. М.: Азбука-классика, 2007. — *Примеч. пер.*

в играх от первого лица, таких как *Quake*, или такой герой, как Натан Дрейк из серии игр *Uncharted*, столь же глубокий и многоплановый, как главные персонажи в кинофильмах. В отличие от кинофильмов, в которых режиссеры стараются вызвать у зрителей симпатию к главному герою, в играх игрок сам является главным персонажем, и дизайнеры должны решить, будет ли главный герой играть роль *аватара* для игрока (транслировать эмоции, желания и намерения игрока в игровой мир и следовать требованиям игрока) или роль будет играть сам игрок (исполняя желания персонажа). Последнее более распространено и проще в реализации.

- **История:** интрига игры. История включает фактический сценарий, описывающий происходящее в игре. Замысел определяет этап, начиная с которого разворачивается сюжет.

Одна из главных целей драматических элементов, которые не попадают в перечисленные выше три типа, — помочь игроку лучше понять правила. В настольной игре *змейки-лесенки*, называя зеленые стрелки на рис. 2.2 «лесенками», мы подчеркиваем подъем вверх по ним. В 1943-м, начиная выпуск игры в США, компания Milton Bradley изменила ее название на *Chutes and Ladders* (горки-лесенки)[1]. Как предполагалось, это название должно было помочь американским детям лучше понять правила игры, потому что горки (chutes), какие обычно устанавливают на детских игровых площадках, лучше ассоциируются со скатыванием вниз, чем змейки, так же как лесенки достаточно хорошо ассоциируются с движением вверх.

В дополнение к этому многие версии игры включали изображения детей, делающих добрые дела, в начале лесенки и изображения с наградой — в конце. Напротив, в начале горки, ведущей вниз, изображались дети, совершающие плохие поступки, а в конце, внизу, изображалось наказание за это. То есть сюжет игры также поощрял следование моральным стандартам Америки 1940-х. Драматические элементы охватывают сюжет, помогающий игрокам помнить правила (как в случае с горками, заменившими змеек) и передающий смысл, сохраняющийся за пределами игры (как в случае с рисунками, изображающими добрые и плохие дела и их последствия).

Динамические элементы

Динамические элементы проявляют себя только в процессе игры. Ниже перечислены базовые понятия динамических элементов игры, предложенные Фуллертон.

- **Непредсказуемость:** простые, казалось бы, правила могут порой приводить к неожиданным результатам. Даже такая невероятно простая игра, как *змейки-лесенки*, может вызывать неожиданные динамические переживания. Если один игрок, по стечению обстоятельств, будет все время попадать на лесенки, а другой только на змейки, они получат совершенно разные впечатления от игры. Если добавить шесть «стратегических» правил, предложенных выше в этой главе,

[1] Статья «*Chutes and Ladders* versus *Snakes and Ladders*»: http://boardgames.about.com/od/gamehistories/p/chutes_ladders.htm. Была доступна в марте 2018 года.

нетрудно вообразить, насколько расширится диапазон эмоций, переживаемых игроками, благодаря этим новым правилам. (Например, теперь кроме судьбы игроку А может противостоять еще и игрок Б, что может привести к переживанию острых отрицательных эмоций игроком А.) Даже простые правила могут приводить к сложным и непредсказуемым переживаниям. Одна из главных задач дизайнера игр — попытаться понять возможные последствия от введения тех или иных правил.

- **Сюжетная непредсказуемость:** кроме динамического поведения механики, как определено в модели МДЭ, модель Фуллертон допускает возможность динамического изменения сюжета, что дает фантастическую широту вариантов развития событий, вытекающих из самого игрового процесса. Игры сами по себе ставят игроков в необычные условия и в результате могут приводить к интересному развитию сюжета. Это одна из самых привлекательных сторон настольных ролевых игр, таких как *Dungeons & Dragons*, в которых один игрок играет роль Мастера Подземелий и определяет сценарий действий для других игроков и персонажей, с которыми им предстоит взаимодействовать. Это отличается от встроенной сюжетной линии, охватываемой драматическими элементами модели Фуллертон, и является одной из развлекательных особенностей, уникальных для интерактивных переживаний.
- **Пробная игра — единственный способ понять динамику:** опытные дизайнеры часто точнее прогнозируют динамическое поведение игры, чем начинающие, но никто не способен понять, как будет развиваться динамика, без ее опробования. На первый взгляд, шесть дополнительных правил, предложенных для игры *змейки-лесенки*, способны придать ей стратегический характер, но оценить фактический эффект от их внедрения можно, только пройдя несколько раундов пробной игры. Пробная игра раскрывает информацию о разных аспектах динамического поведения игры и помогает дизайнерам понять, какой диапазон эмоций способна вызывать игра.

Простая тетрада

В книге «The Art of Game Design: a Book of Lenses»[1] Джесси Шелл описал простую тетраду (elemental tetrad), посредством которой представил свои четыре основных элемента игр.

- **Механика:** правила взаимодействия игрока с игрой. Механика — это элемент в тетраде, который отличает игры от всех неинтерактивных форм передачи информации (таких, как фильмы или книги). Механика определяет правила, цели и другие формальные элементы, описанные Фуллертон. Этот элемент отличается от механики в модели МДЭ, потому что Шелл отличает механику игры и технологии, ее обеспечивающие.

[1] Jesse Schell, *The Art of Game Design: a Book of Lenses* (Boca Raton, FL: CRC Press, 2008).

- **Эстетика:** описывает, как игра воспринимается пятью органами чувств: зрение, слух, обоняние, вкус и осязание. Эстетика охватывает все, от звукового сопровождения в игре до моделей персонажей, упаковки и оформления обложки. Такое понимание эстетики отличается от принятого в МДЭ, где слово «эстетика» обозначает эмоциональный отклик, вызываемый игрой, потому что в понимании Шелла это слово обозначает активы, созданные разработчиками, такие как фактическое художественное и звуковое оформление игры.
- **Технология:** этот элемент охватывает все технологии, заставляющие игру работать. Хотя это наиболее очевидно относится к таким составляющим, как аппаратное и программное обеспечение, конвейеры отображения и т. д., сюда также относятся технологические элементы настольных игр. Технология настольных игр может включать, например, типы и количество игровых кубиков, что используется в качестве источника случайности — кубики или колода карт, а также различные статистики и таблицы для определения результата действий. Фактически награда игровой конференции IndieCade 2012 за использование технологий была отдана Заку С. (Zac S.) за *Vornheim*, коллекцию инструментов — в форме печатной книги — для использования разработчиками настольных ролевых игр, в которых действие происходит в городе[1].
- **История:** Шелл использует термин *история*, чтобы передать все то же самое, что передают драматические элементы Фуллертон, а не только то, что подразумевает слово «история». История — это сюжет игры, включающий также замысел и персонажей.

Шелл включил эти элементы в тетраду, изображенную на рис. 2.3.

Тетрада показывает, как связаны друг с другом все четыре элемента. Кроме того, в ней Шелл отмечает, что эстетика игры всегда хорошо видна игроку (напомню, что в данном случае значение слова «эстетика» отличается от его значения в МДЭ), а технология является наименее заметной, и обычно механика игры (например, как змейки и лесенки влияют на позицию игрока) более понятна игроку, чем технология (например, распределение вероятностей выпадения очков для пары кубиков). Тетрада Шелла не касается динамики и в большей степени относится к статическим элементам игры, продаваемой в коробке (если это настольная игра) или на диске. Простая тетрада Шелла обсуждается и значительно расширяется в следующей главе как образующая упрощенный аспект многоуровневой тетрады.

Итоги

Каждая из этих методик анализа игр и интерактивных произведений подходит к пониманию игр со своей точки зрения.

[1] http://www.indiecade.com

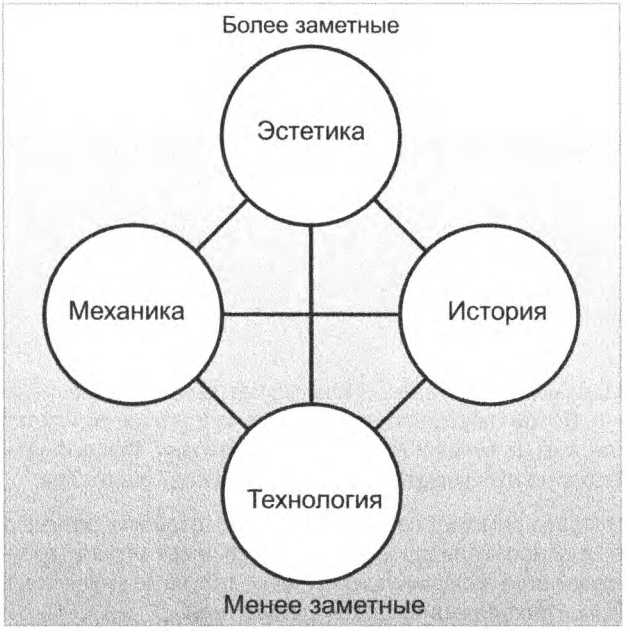


Рис. 2.3. Простая тетрада Джесси Шелла[1]

- МДЭ стремится продемонстрировать и конкретизировать идею, что игроки и дизайнеры видят игры с разных направлений, и предлагает дизайнерам учиться видеть свои игры с позиции игроков.
- Методика на основе деления на формальные, драматические и динамические элементы (ФДД) разбивает дизайн игры на конкретные компоненты, которые можно рассматривать и совершенствовать по отдельности. Цель этой методики состоит в том, чтобы дать дизайнерам инструменты и позволить им изолировать, исследовать и улучшать все части своих игр. ФДД также утверждает первичность сюжета для восприятия со стороны игрока.
- Простая тетрада в большей степени представляет взгляд разработчика на игры. Она выделяет простые элементы игры в разделы, которые обычно разрабатываются разными группами: дизайнеры игр занимаются механикой, художники заботятся об эстетике, писатели формируют сюжет, а программисты воплощают технологию.

Следующая глава представляет многоуровневую тетраду, объединяющую и расширяющую идеи, заложенные в перечисленных здесь методиках. Важно воспринимать их как основу теории, ведущей к многоуровневой тетраде, и я настоятельно рекомендую прочитать оригинальные статьи и книги, в которых они были представлены.

[1] С небольшими изменениями взят из книги Шелла *The Art of Game Design*, 42.

3 Многоуровневая тетрада

В предыдущей главе были представлены разные методики анализа для исследования игр и игрового дизайна. В этой главе рассказывается о многоуровневой тетраде, сочетающей и расширяющей многие из лучших аспектов этих методик. Каждый уровень многоуровневой тетрады подробно обсуждается в одной из последующих глав.

Многоуровневая тетрада поможет вам исследовать и создавать разные аспекты игр. С ее помощью вы сможете проанализировать понравившиеся игры и составить целостное представление о своей игре, получив понимание не только ее механики, но и последствий с точки зрения игры, социализации, смысла и культуры.

Многоуровневая тетрада — это сочетание и расширение идей, выражаемых тремя методиками анализа игр, представленными в предыдущей главе. Многоуровневая тетрада не определяет игру как таковую. Скорее, это инструмент, помогающий понять все элементы, которые требуется создать, чтобы получить игру, и то, что происходит с этими элементами во время игры и за ее пределами по мере того, как игра становится частью культуры.

Многоуровневая тетрада включает четыре элемента — как простая тетрада Шелла, но эти четыре элемента размещены на трех уровнях. Первые два уровня — *фиксированный* и *динамический* — основаны на делении между формальными и динамическими элементами, предложенными Фуллертон. Их дополняет третий уровень — *культурный*, охватывающий влияние игры на жизнь за ее пределами и обеспечивающий связь между игрой и культурой, что должны понимать все ответственные дизайнеры игр и создатели выразительного искусства.

Каждый уровень кратко описывается в этой главе, и каждому будет посвящена отдельная глава далее в книге.

Фиксированный уровень

Фиксированный уровень тетрады (рис. 3.1) очень похож на простую тетраду Шелла. Определения четырех элементов на этом уровне схожи с определениями Шелла, но они ограничены аспектами, которые существуют вне игры.

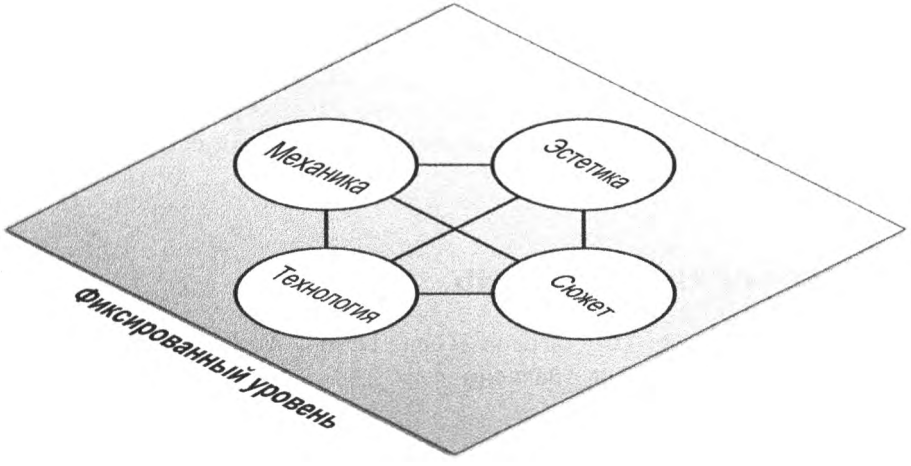


Рис. 3.1. Фиксированный уровень в многоуровневой тетраде[1]

- **Механика:** система, определяющая взаимодействия между игроком и игрой. Включает правила игры и другие сопутствующие формальные элементы, перечисленные в книге Фуллертон: схемы взаимодействия с игроком, цели, ресурсы и границы.
- **Эстетика** описывает внешний вид игры, звуковое сопровождение, запахи, вкус и тактильные ощущения. Эстетика охватывает все — от звукового сопровождения в игре до моделей персонажей, упаковки и оформления обложки. Это определение отличается от принятого в методике МДЭ (Механика, Динамика, Эстетика), где под словом «эстетика» понимается эмоциональный отклик, вызываемый игрой, тогда как в определении Шелла это слово обозначает элементы, воспринимаемые игроком, такие как художественное и звуковое оформление.
- **Технология:** так же, как элемент «технология» в простой тетраде Шелла, этот элемент охватывает все технологии, заставляющие игру — настольную или цифровую — работать. Технологии для цифровых игр разрабатываются в первую очередь программистами, но дизайнеры должны понимать этот элемент, потому что технология, написанная программистом, формирует пространство возможных решений, которые могут приниматься дизайнерами игр. Это понимание имеет решающее значение также потому, что, казалось бы, простое конструктивное решение (например, перенос игрового уровня с твердой почвы на качающуюся палубу корабля во время шторма) может потребовать нескольких тысяч человеко-часов для реализации.

[1] Взято из книги: Jesse Schell, *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), 42.

- **Сюжет:** для обозначения этого элемента в своей тетраде Шелл использует термин *«история»*, но я выбрал более широкий термин *«сюжет»*, чтобы дополнительно охватить замысел и персонажей и точнее соответствовать использованию этих терминов у Фуллертон. Фиксированный сюжет включает предварительно написанную историю и заранее созданных персонажей, участвующих в игре.

Динамический уровень

Как описывается в книге Фуллертон «Game Design Workshop», динамический уровень (рис. 3.2) появляется с началом игры.

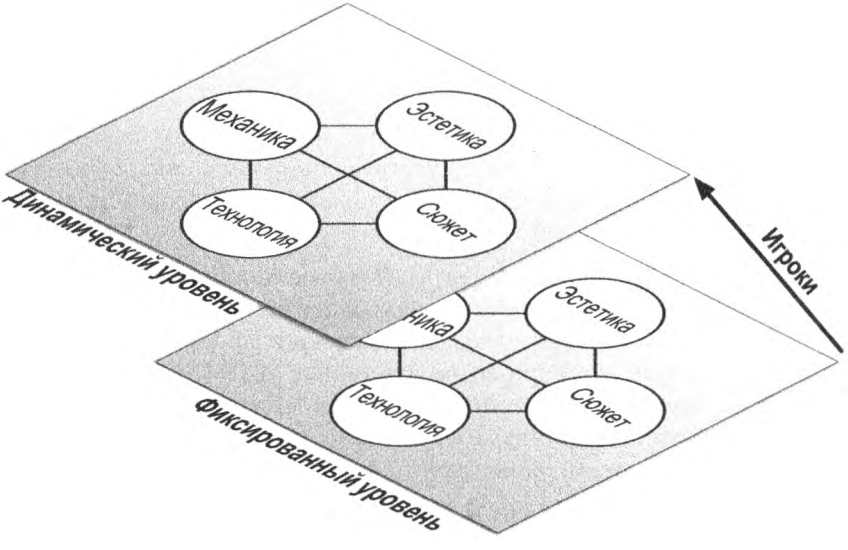


Рис. 3.2. Динамический уровень находится над фиксированным

Как можно видеть на рис. 3.2, именно игроки конструируют динамический уровень игры на основе статического фиксированного уровня. Все имеющееся на динамическом уровне возникает в процессе игры, и динамический уровень состоит из элементов, непосредственно управляемых игроком, и из результатов их взаимодействий с фиксированными элементами. Динамический уровень — это область *неожиданностей*, результат сложного поведения, возникающего из, казалось бы, простых правил. Поведение игры часто трудно предсказать, но одно из умений, которое вы обретете с течением времени, — способность делать такие предсказания или, по крайней мере, оказываться недалеко от истины. К четырем элементам динамического уровня относятся:

- **Механика:** в отличие от механики фиксированного уровня, охватывающей правила, цели и т. д., динамическая механика определяет порядок взаимодействия игрока с этими фиксированными элементами. Динамическая механика включает процедуры, стратегии, поведение игры и, наконец, ее исход.
- **Эстетика:** динамическая эстетика определяет способы создания эстетических элементов в процессе игры. Сюда входит все, от процедурного искусства (изображения или звуки, генерируемые компьютерным кодом «на лету») до физического напряжения, которое может возникнуть из-за необходимости долгое время снова и снова жать на кнопку.
- **Технология:** динамическая технология описывает поведение технологических компонентов в процессе игры. Она охватывает, например, тот факт, что пара кубиков никогда не генерирует гладкую колоколообразную кривую результатов, предопределяемую математической вероятностью. Она также охватывает почти все, что делает компьютерный код в цифровой игре. Одним из конкретных примеров может служить реализация искусственного интеллекта врагов, но в более широком смысле динамическая технология охватывает абсолютно все, что делает цифровой код после запуска игры.
- **Сюжет:** под динамическим сюжетом понимается история, которая создается процедурно игровой системой. Под этим может пониматься путь конкретного игрока через ветвящийся сценарий, как в игре *L.A. Noire or Heavy Rain*, семейная история в игре *The Sims* или истории, создаваемые в командной игре с другими игроками. В 2013 году бейсбольная команда Boston Red Sox прошла путь от худшей до наилучшей, отражая восстановление Бостона после теракта на Бостонском марафоне 2013 года. Такая история, подкрепленная правилами профессионального бейсбола, также подходит под динамический сюжет.

Культурный уровень

Третьим и заключительным в многоуровневой тетраде является культурный уровень, и он описывает игру за рамками игры (рис. 3.3). Культурный уровень охватывает влияние культуры на игру и игры на культуру. Сообщество игроков перемещает игру в культурный уровень, и в этот момент у игроков больше прав и власти над игрой, чем у дизайнеров; именно благодаря этому слою становится очевидна наша ответственность, как дизайнеров, перед обществом.

Границы между четырьмя элементами в культурном уровне более размыты, но все же стоит рассмотреть этот уровень через призму четырех элементов:

- **Механика:** простейшая форма культурной механики может быть представлена, например, *игровыми модулями* — «модами» (создаваемыми игроками для изменения встроенного поведения игры). Также она охватывает сложные процессы, такие как влияние игры на общество. Например, часто критикуемая возможность в игре *Grand Theft Auto 3* переспать с проституткой и затем убить ее, чтобы

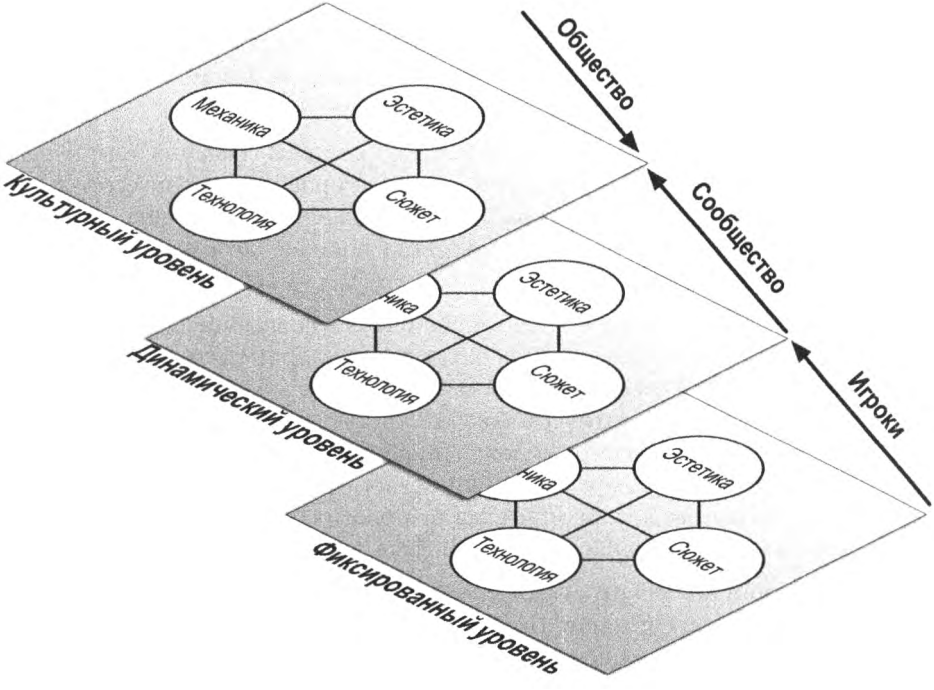


Рис. 3.3. Культурный уровень существует на стыке игры и общества

вернуть деньги, стала результатом появления динамической механики в игре, но она оказала огромное влияние на восприятие игры в обществе (которое является частью культурного уровня).

- **Эстетика:** по аналогии с механикой, культурная эстетика может охватывать художественное творчество фанатов, ремиксы на музыку в игре или другие эстетические результаты творчества фанатов, такие как *костюмированные представления* (когда фанаты одеваются в костюмы, напоминающие одежду игровых персонажей). Здесь важно отметить, что *авторизованный* перенос (например, перенос прав интеллектуальной собственности на другое произведение владельцами этих прав, как, например, в случае с фильмом по мотивам игры *Tomb Raider*, сумочкой для школьного завтрака с *покемонами* и т. д.) не является частью культурного уровня. Это объясняется тем, что авторизованный перенос собственности контролируется владельцами интеллектуальных прав на игру, тогда как культурная эстетика контролируется и создается сообществом любителей игры.
- **Технология:** к культурным технологиям относится использование технологий из игры для неигровых целей (например, алгоритмы группового поведения игровых персонажей также могут использоваться в робототехнике) и возможность влияния технологий на игровой процесс. Еще во времена NES (Nintendo

Entertainment System) контроллеры Advantage или Max имели турбокнопки (имитировавшие быстрые повторяющиеся нажатия обычных кнопок A и B). Это давало огромное преимущество в некоторых играх и оказывало влияние на восприятие игры. Культурные технологии также приумножают значения слова *«игра»*, постоянно расширяя пространство возможностей игр, и технологические аспекты модов, создаваемых игроками для изменения встроенных элементов игры.

- **Сюжет:** культурный сюжет включает повествовательные аспекты произведений, созданных фанатами на основе игры (например, фанфики — произведения по мотивам, сценарии фильмов или персонажи и обстановка, являющиеся частью некоторых игровых модов). Он также охватывает истории, рассказывающие об игре в культуре и обществе, в том числе истории, очерняющие такие игры, как *Grand Theft Auto*, и превозносящие добродетели и художественные достоинства таких игр, как *Journey* и *Ico*.

Ответственность дизайнера

Все дизайнеры осознают свою ответственность за фиксированный уровень игры. Совершенно понятно, что разработчики игры должны определить четкие правила, обеспечить привлекательное художественное оформление и т. д., чтобы вызвать у игроков желание сыграть в их игру.

Ответственность за динамический уровень некоторыми дизайнерами ощущается не так явно. Некоторые удивляются поведению, которое проявляется в их играх, и возлагают ответственность за это на игроков. Например, несколько лет тому назад компания Valve решила подарить головные уборы игрокам в *Team Fortress 2*. Для этого они выбрали способ случайного распределения подарков между игроками, вошедшими в игру. Поскольку распределение подарков было приурочено к определенному отрезку времени, в течение которого должен быть выполнен вход, появилось очень много игроков, которые выполняли вход, желая получить подарок, но не играли в игру. В компании Valve обнаружили это и решили наказывать игроков, отбирая подарки у тех, кто подозревался во входе в игру без цели играть.

Такое поведение игроков можно было бы интерпретировать как попытку обмануть игру. Но если взглянуть с другой стороны, игроки просто выбрали наиболее эффективный способ получения подарка, соответствующий правилам, которые определила сама компания Valve. Поскольку система предусматривала розыгрыш подарков между игроками, находящимися онлайн, независимо от того, что они фактически делали в это время, игроки выбрали самый простой путь. Возможно, действия игроков не соответствовали намерениям дизайнеров, создавших систему розыгрыша подарков, но они точно не обманывали саму систему. Их динамическое поведение в точности соответствовало правилам системы, установленным компанией Valve. Как показывает этот пример, дизайнер также несет ответственность за происходящее на динамическом уровне и должен осознавать последствия спроекти-

рованной им системы. Фактически прогнозирование и создание динамики игрового процесса является одним из важнейших аспектов проектирования игр. Конечно, это очень сложная задача, но отчасти это и делает игру интересной.

А несет ли дизайнер ответственность за культурный уровень? Из-за того, что мало кто из дизайнеров уделяет внимание культурному уровню, видеоигры рассматриваются обществом как незрелые и вульгарные — пропагандирующие среди подростков насилие и женоненавистничество. Мы с вами знаем, что это далеко не так, и несправедливо давать такую оценку многим или даже большинству игр, но таково восприятие общества. Игры могут учить, игры могут поддерживать и игры могут лечить. Игры могут поощрять гуманное поведение и помогать игрокам обретать новые навыки. Игривый настрой и простые правила могут сделать приятным даже самое скучное занятие. Как дизайнер, вы несете ответственность за то, что ваша игра говорит обществу об играх, и за влияние, оказываемое на игроков. Мы слишком преуспели в создании игр, которые вызывают нездоровое пристрастие к ним. Некоторые дизайнеры даже создавали игры, обманывающие детей, в расчете заработать на этом сотни или тысячи долларов (в конце концов это привело к массовым судебным разбирательствам по крайней мере в одном таком случае). Такое отношение дизайнеров наносит ущерб репутации игр в обществе, из-за чего многие люди считают, что игры не могут быть художественным произведением и на них не стоит тратить свое время, и это очень грустно.

Я считаю, что мы, как дизайнеры, должны пропагандировать гуманное, взвешенное отношение в наших играх и уважать наших игроков и время, которое они уделяют плодам нашего труда.

Итоги

Как показано в этой главе, многоуровневая тетрада включает три уровня, представляющих переход собственности из рук разработчиков в руки игроков. Все, что находится на фиксированном уровне, придумывается и реализуется дизайнерами и разработчиками. Фиксированный уровень находится в полной власти разработчика.

Динамический уровень определяет этап, на котором фактически протекает игровой процесс, и дизайнеры передают игрокам право предпринимать какие-то действия и принимать решения в рамках, предусмотренных дизайнером. Принимая решения и влияя на игровую систему, игроки обретают некоторую власть над игрой, но эта власть все еще зависит от решений, реализованных разработчиками. То есть власть над динамическим уровнем делится между разработчиками и игроками.

На культурном уровне игра выходит из-под власти разработчиков. Именно поэтому такие вещи, как игровые моды, находятся на культурном уровне; с помощью модов игрок берет под контроль и изменяет встроенные аспекты игры. Конечно, большая часть реализации игры остается прежней, но теперь уже игрок (как раз-

работчик мода) определяет, какие встроенные элементы останутся, а какие будут заменены; игрок получает власть. Вот почему я исключил авторизованный перенос из культурного уровня. Разработчики и владельцы оригинальной игры сохраняют право на авторизованный перенос, тогда как культурный уровень характеризуется передачей власти игрокам и сообществам, окружающим игру. Кроме того, аспект культурного уровня, который определяет отношение к игре со стороны не игроков в обществе, также в значительной степени контролируется представлениями сообщества игроков об игровом процессе. Люди, не игравшие в игру, составляют свое мнение о ней по отзывам и публикациям, написанным (надеюсь) теми, кто действительно играл в нее. Однако даже при том, что культурный уровень в значительной степени контролируется игроками, разработчики и дизайнеры все еще способны оказывать существенное влияние и несут ответственность за игру и ее влияние на общество.

В следующих трех главах мы подробнее рассмотрим все три уровня, составляющие многоуровневую тетраду.

4 Фиксированный уровень

Это первая из трех глав, детально исследующих уровни многоуровневой тетрады.

Как вы узнали в главе 3 «Многоуровневая тетрада», фиксированный уровень охватывает все элементы, спроектированные и реализованные непосредственно разработчиками игры.

В этой главе мы рассмотрим предопределенные аспекты всех четырех элементов: механики, эстетики, сюжета и технологии.

Фиксированная механика

Фиксированная механика составляет большую часть традиционной работы дизайнера игр. В настольных играх она включает проектирование игрового поля, правил, разных карточек, необходимых для игры, и любых таблиц, которые можно использовать для справки. Большая часть фиксированной механики хорошо описана в книге Трейси Фуллертон «Game Design Workshop», в главе о формальных элементах, и ради лексической солидарности (и из-за моего неприятия книг о проектировании игр, использующих другую терминологию) на протяжении всего этого раздела я буду пользоваться ее терминологией, насколько это позволит метод многоуровневой тетрады.

В главе 2 «Методы анализа игр» я перечислил семь формальных элементов игр, представленных в книге «Game Design Workshop»: схемы взаимодействия с игроком, цели, правила, процедуры, ресурсы, границы и исход. В методе формальных, драматических и динамических элементов эти семь элементов составляют аспекты, которые отличают игры от других интерактивных форм.

Фиксированная механика отличается от формальных элементов, но между ними также немало общего, потому что механика — это элемент тетрады, уникальный для игр. Тем не менее основой фиксированного уровня является все, что преднамеренно создано разработчиком игры, и механика не является исключением. Как результат, фиксированная механика не включает процедуры и исход (хотя в классификации Фуллертон они относятся к формальным элементам), потому что и то

и другое контролируется игроком и поэтому принадлежит динамическому уровню. Мы также добавим несколько новых элементов и получим в результате следующий список элементов фиксированной механики:

- **Цели** — это результат, к которому стремится игрок. К чему стремятся игроки?
- **Отношения между игроками** определяют способы противоборства и сотрудничества игроков друг с другом и с игрой. Как пересекаются цели игроков и как это заставляет их сотрудничать или противоборствовать?
- **Правила** определяют и ограничивают действия игрока. Что могут и чего не могут делать игроки для достижения своих целей?
- **Границы** определяют рамки игры и напрямую относятся к магическому кругу. Где пролегает граница игры? Где существует магический круг?
- **Ресурсы** включают активы или ценности, имеющие значение в границах игры. Чем игрок обладает в игре, что позволяет ему действовать?
- **Пространства** определяют форму игрового мира и возможности взаимодействий в его пределах. Это наиболее очевидно проявляется в настольных играх, где сама игровая доска является пространством игры.
- **Таблицы** определяют статистическую форму игры. Как растут возможности игрока по мере накопления опыта? Какие действия доступны игроку в данный момент?

Все эти элементы фиксированной механики взаимодействуют и перекрываются друг другом (например, технологическое дерево в игре *Civilization* — это таблица, которая управляет навигацией, как в пространстве). Цель деления на семь категорий в этой книге — помочь вам, как дизайнеру, подумать о разных возможностях проектирования ваших игр. Не все игры имеют полный набор перечисленных элементов, но, так же как в случае с «линзами» в книге Джесси Шелла «The Art of Game Design: A Book of Lenses», эти элементы фиксированной механики представляют семь разных способов взглянуть на аспекты, которые вы можете создать для игры.

Цели

Хотя многие игры имеют простую и очевидную цель — выиграть, в действительности каждый игрок в течение игры постоянно преследует несколько целей. Их можно классифицировать по актуальности и значимости для игрока, и некоторые можно считать очень важными для одного игрока и неважными для другого.

Актуальность целей

Как показано на рис. 4.1, где изображена сцена из замечательной игры *Journey*, созданной в студии thatgamecompany (TGC), почти каждый экран современной игры ставит перед игроком кратко-, средне- и долгосрочные цели.

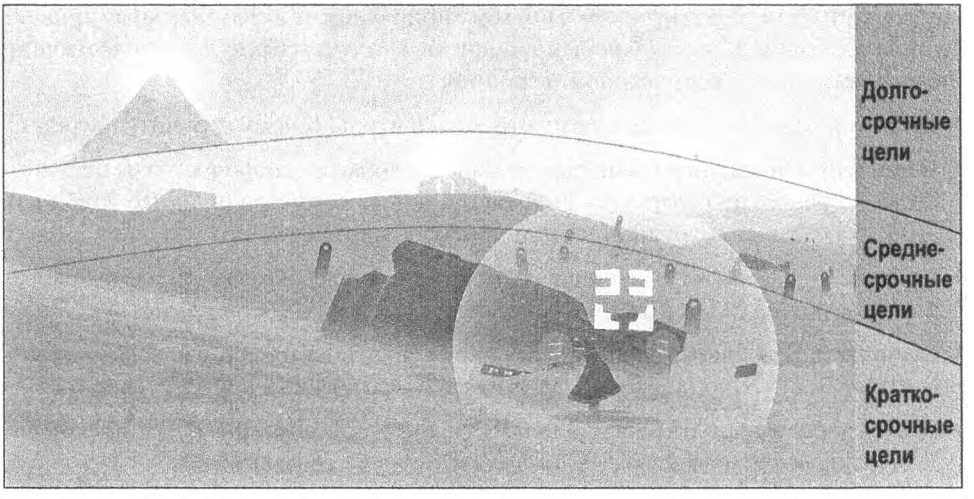


Рис. 4.1. Кратко-, средне- и долгосрочные цели на первом уровне *Journey*

- **Краткосрочные цели:** игроку нужно зарядить свой шарф (который в игре *Journey* позволяет летать), поэтому персонаж должен петь (белая сфера вокруг него), чтобы приманить дополнительные фрагменты шарфа. Также внимание игрока привлекается к исследованию близлежащего строения.
- **Среднесрочные цели:** ближе к горизонту можно видеть еще три строения. Остальная часть пустыни практически безжизненна, поэтому внимание игрока, вне всяких сомнений, будет привлечено к этим руинам, и он почти наверняка отправится к ним (такая стратегия косвенного управления несколько раз используется в игре *Journey* и анализируется в главе 12 «Руководство игроком»).
- **Долгосрочные цели:** в первые несколько минут игры игроку демонстрируется гора, из которой вверх бьет луч света (видна слева вверху на рис. 4.1), и долгосрочная цель, цель всей игры, — добраться до вершины этой горы.

Важность целей

Цели различаются не только по актуальности, но также по важности для игрока. Игры с открытым миром, например *Skyrim*, созданная студией Bethesda Game Studios, имеют главные и дополнительные цели. Некоторые игроки могут выбирать исключительно главные цели и пройти игру *Skyrim* всего за 10–20 часов, тогда как другие, желающие исследовать побочные квесты и достичь дополнительных целей, могут потратить более 400 часов, но так и не исчерпать игру (и даже не достигнуть ни одной главной цели). Дополнительные цели часто привязаны к игровым процессам определенных типов; в *Skyrim* существует целая серия заданий для игроков, желающих вступить в гильдию воров и специализироваться на обмане и воровстве. Существуют также серии заданий для тех, кто желает сосредоточиться на стрельбе

из лука или рукопашном бое. Благодаря этому игра может адаптироваться под предпочтения разных игроков.

Противоречивые цели

Цели, к которым стремится игрок, часто противоречат друг другу или конкурируют за одни и те же ресурсы. Например, в игре *Monopoly* общая цель состоит в том, чтобы завершить игру с как можно большим количеством денег, но вы должны тратить деньги на приобретение таких активов, как недвижимость, дома и гостиницы, которые позднее принесут дополнительные деньги. С точки зрения проектирования целей наибольший интерес для игрока представляют *обоюдоострые* варианты выбора, когда ради достижения одной цели приходится жертвовать другой.

Если посмотреть с более прагматичной точки зрения, игрок не может посвятить все свое время достижению какой-либо цели. Например, в игре *Skyrim* многие люди (в том числе и я) никогда не доходили до конца главного квеста, потому что все время, уделяемое игре, тратят на прохождение дополнительных квестов и забывают о главной сюжетной линии. Вероятно, цель дизайнеров *Skyrim* состояла в том, чтобы дать каждому игроку возможность сформировать свой сюжет, и, возможно, дизайнерам было безразлично, что я не закончил главный квест, пока получал удовольствие от игры, но, как игрок, я чувствовал, что игра заканчивается не красивой концовкой, а пшиком, поскольку наслоение квестов друг на друга приносило мне все меньше удовольствия. Если вам, как дизайнеру, важно, чтобы ваши игроки доходили до конца главного квеста игры, вы должны постоянно напоминать игроку о важности основной задачи, и (в отличие от многих игр с открытым миром) вам может потребоваться предусмотреть последствия для игрока, если он не завершил главный квест к определенному сроку. Например, в классической игре *Star Control*, если игрок не успевает спасти иноземную расу за определенное время с момента начала игры, планета расы исчезает из Вселенной.

Отношения между игроками

Подобно тому как отдельный игрок имеет несколько целей в каждый конкретный момент времени, цели, которые преследуют игроки, также определяют отношения между ними.

Схемы взаимодействий игроков

В книге «Game Design Workshop» Фуллертон перечислила семь разных схем взаимодействий игроков.

- **Единственный игрок против игры:** перед игроком стоит цель победить игру.
- **Несколько независимых игроков против игры:** несколько независимых игроков нацелены победить игру, но они практически не взаимодействуют друг с другом.

Эту схему часто можно наблюдать в MMO-играх (массовых многопользовательских ролевых онлайн-играх (иногда их также называют «MMORPG»)), таких как *World of Warcraft*, когда все игроки стремятся добиться успеха в одном и том же игровом мире, но не обязаны взаимодействовать друг с другом.

- **Командная игра:** несколько игроков нацелены вместе победить игру.
- **Игрок против игрока:** каждый из двух игроков нацелен победить другого.
- **Все против всех:** то же, что и в схеме «игрок против игрока», но на этот раз в игре участвует больше двух игроков и каждый нацелен победить всех остальных.
- **Игрок против команды:** один игрок играет против целой команды. Примером может служить настольная игра *Scotland Yard* (также известна под названием *Mr. X*), в которой один игрок — преступник, пытающийся ускользнуть от полиции, и от двух до четырех других игроков — полицейские, которые совместными усилиями пытаются поймать преступника.
- **Команда против команды:** в игре участвуют две команды, каждая из которых нацелена победить другую.

Некоторые игры, такие как *Mass Effect* студии BioWare, предоставляют игрокам союзников, управляемых компьютером. С точки зрения схем взаимодействия с игроками таких союзников можно рассматривать как дополнительные возможности единственного игрока или как представителей других игроков, участвующих в игре, то есть дизайнер может рассматривать такую игру с единственным игроком и союзниками, управляемыми компьютером, как игру с единственным игроком, играющим против игры, или как командную игру.

Цели определяют роли и отношения между игроками

Кроме схем взаимодействий, перечисленных в предыдущем разделе, существуют также разные их комбинации, и в некоторых играх игрок может менять свою роль, становясь то союзником другого игрока, то противником. Например, меняя деньги на недвижимость в такой игре, как *Monopoly*, два игрока могут создать краткосрочный союз, хотя в целом игра реализует схему «все против всех».

В любой момент отношение каждого игрока к игре и к другим игрокам определяется комбинацией всех многоуровневых целей игроков. Эти отношения вынуждают каждого игрока играть одну из нескольких разных ролей:

- **Главный герой:** роль главного героя игрок играет, когда пытается победить игру.
- **Противник:** игрок пытается победить других игроков. Обычно главной целью противника является победа над игрой, но в редких случаях игрок может играть на стороне игры (как, например, в настольной игре *Betrayal at House on the Hill*, выпущенной в 2004 году, где один из игроков переходит на сторону зла и пытается убить других игроков).
- **Соратник:** игрок помогает другим игрокам.

- **Участник:** игрок действует в одном игровом мире с другими игроками, но не сотрудничает и не соперничает с ними.

Во многих многопользовательских играх все игроки примеряют все эти роли в разные моменты времени, и, как вы узнаете, когда мы будем рассматривать динамический уровень, разные типы игроков предпочитают разные роли.

Правила

Правила ограничивают действия игроков. Правила также являются самым непосредственным отражением представлений дизайнера о том, как вы должны играть в игру. В письменных правилах к настольной игре дизайнер стремится определить и зашифровать восприятие, которое должны получить игроки, играя в игру. Игроки расшифровывают эти правила посредством игры и, возможно, испытывают чувства, похожие на те, что планировал дизайнер.

В отличие от игр на бумаге, в цифровых играх обычно очень немного вписанных правил, которые читаются непосредственно игроком; однако программный код, написанный разработчиками игры, дает возможность зашифровать дополнительные правила, которые расшифровываются посредством игры. Поскольку правила — это самый прямой способ общения дизайнера с игроком, они используются для определения многих других элементов. Деньги в игре *Monopoly* имеют ценность только потому, что правила объявляют возможность их использования для приобретения активов и ресурсов.

Явно писаные правила являются наиболее очевидной формой правил, но кроме них имеются также неписаные правила. Например, когда вы играете в покер, действует неписаное правило, запрещающее прятать карты в рукаве. Это явно не отражено в правилах, но любой игрок понимает, что подобное действие считается обманом[1].

Границы

Границы определяют рамки пространства и времени, в которых протекает игра. Внутри границ действуют правила и другие аспекты игры: покерные фишки имеют какую-то ценность, считается нормальным шумно выражать поддержку хоккеистам на льду и очень важно, какой автомобиль первым пересечет финишную черту. Иногда у границ есть физическое воплощение, как стены вокруг хоккейного поля. Иногда они не так очевидны. Когда кто-то играет в игру альтернативной реальности (Alternate Reality Game, ARG), она окружает и пронизывает обычную жизнь игрока.

[1] Это хороший пример одного из отличий в проектировании однопользовательских и многопользовательских игр. В многопользовательской игре в покер сокрытие карты считается обманом и может испортить игру. Но в игре *Red Dead Redemption* студии Rockstar Studios турниры по покеру становятся намного интереснее и увлекательнее, когда игрок приобретает костюм, который позволяет скрывать и подменивать карты, хотя и с риском быть пойманным персонажами, которыми управляет компьютер.

В одной из первых таких игр, *Majestic* (выпущена в 2001 году студией Electronic Arts), игроки передавали EA номера своих телефонов и факсов, адреса электронной почты и домашние адреса и затем могли получать телефонные звонки, факсы и т. д. в любое время суток от персонажей в игре. Цель игры состояла в том, чтобы размыть границы между игрой и обычной жизнью.

Ресурсы

Ресурсы — это все, что имеет ценность в игре. Это могут быть *активы* (объекты внутри игры) или нематериальные *атрибуты*. К числу активов в играх можно отнести, например, экипировку, собранную в игре *Legend of Zelda*; карточки ресурсов, которые игроки собирают в настольной игре *Settlers of Catan*; или дома, отели и ценные бумаги, которые игроки приобретают в игре *Monopoly*. К числу атрибутов обычно относятся здоровье, объем воздуха, оставшегося в аквалангах при подводном плавании, и очки опыта. Деньги, благодаря своей универсальности и распространенности, находятся где-то посередине. Игра может иметь физические денежные активы (как наличность в *Monopoly*) или нефизический атрибут денег (как сумма денег на счету игрока в игре *Grand Theft Auto*).

Пространства

Дизайнерам часто приходится создавать пространства, в которых можно перемещаться в разных направлениях. К ним можно отнести игровые доски в настольных играх и виртуальные уровни в цифровых играх. В обоих случаях нужно подумать, как будут протекать действия в этом пространстве и как сделать области этого пространства уникальными и интересными. Вот о чем следует помнить при проектировании пространств:

- **Назначение пространства:** архитектор Кристофер Александер в течение многих лет изучал вопрос, почему одни пространства особенно хорошо подходят для строительства, а другие нет. Свои знания он выразил в идее *шаблонов проектирования* в книге «A Pattern Language»[1], где исследовал разные шаблоны организации архитектурных пространств. Цель этой книги — рассказать о шаблонах, которые другие могли бы использовать для организации пространств, изначально хорошо подходящих для подразумеваемого назначения.
- **Динамика движения:** позволяет ли пространство легко перемещаться в нем? Если оно ограничивает перемещения, есть ли тому веские причины? В настольной игре *Clue* игроки бросают единственный кубик перед каждым ходом, чтобы

[1] Christopher Alexander, Sara Ishikawa и Murray Silverstein, *A Pattern Language: Towns, Buildings, Construction* (New York: Oxford University Press, 1977). (*Кристофер Александер, Сара Исикава, Мюррей Сильверстайн.* Язык шаблонов. Города. Здания. Строительство. М.: Издательство Студии Артемия Лебедева, 2014. — *Примеч. пер.*)

определить, как далеко они могут переместиться. Движение по игровому полю может быть очень медленным. (Поле имеет размер 24 × 25 клеток, то есть если принять, что при каждом броске в среднем выпадает 3,5 очка, для пересечения поля потребуется 7 ходов.) Понимая это, дизайнеры добавили секретные проходы, позволяющие игрокам телепортироваться из одного угла в противоположный, что помогает перемещаться довольно быстро.

- **Ориентиры:** создание карты виртуального трехмерного пространства в памяти игрокам дается сложнее, чем физического пространства, в котором они пребывают в реальной жизни. Поэтому важно создавать в виртуальном пространстве геоточки, с помощью которых игрокам проще было бы ориентироваться. В Гонолулу, на Гавайях, люди не указывают направления по сторонам света (север, юг, запад, восток), потому что они неочевидны, за исключением периодов восхода или заката солнца. Вместо этого люди в Гонолулу пользуются ясными ориентирами: *маука* (горы на юго-востоке), *макаи* (океан на юго-западе), *Бриллиантовая голова* (приметная гора на юго-востоке) и *Ева* (район на северо-западе). В других частях Гавайских островов *маука* означает направление вглубь острова, а *макаи* — направление к океану, независимо от сторон света (острова, по сути, круглые). Создание ориентиров избавляет игроков от необходимости часто сверяться с картой, чтобы узнать, где они находятся.
- **Впечатления:** игра в целом — это комплекс впечатлений, поэтому карта или пространство игры также должны давать игрокам интересные впечатления. В игре *Assassin's Creed 4: Black Flag* карта мира — это сильно усеченная версия Карибского моря. Даже при том, что между островами в Карибском море лежат огромные пространства пустынного океана, для пересечения которых на парусном судне может потребоваться целый день, а то и больше, в Карибском море игры *AC4* каждую минуту происходит множество событий, дающих игрокам новые впечатления. Это может быть небольшое событие, такое как находка сундука с сокровищами на крошечном атолле, или большое, такое как встреча с флотом вражеских кораблей.
- **Кратко-, средне- и долгосрочные цели:** как показано на скриншоте игры *Journey* (рис. 4.1), пространство может содержать несколько целей разных уровней. В играх с открытым миром игроку часто показывают врага верхнего уровня в самом начале, чтобы он знал, кого он должен победить. Многие игры также ясно обозначают на карте области с низкой, средней и высокой сложностью преодоления.

Таблицы

Таблицы играют важную роль в поддержании баланса игры, особенно в современных цифровых играх. Таблицы — это самые обычные таблицы с данными, их еще часто называют электронными таблицами, но таблицы также можно использовать в дизайне и для иллюстрации самых разных аспектов.

- **Вероятность:** в некоторых ситуациях таблицы можно использовать для определения вероятности. В настольной игре *Tales of the Arabian Nights* игрок выбирает соответствующую таблицу для встретившегося существа и определяет по ней список возможных реакций на эту встречу и результат для каждой реакции.
- **Развитие:** в настольных ролевых играх, таких как *Dungeons & Dragons*, в таблицах показывается, как увеличиваются и изменяются способности игрока с увеличением уровня.
- **Данные пробной игры:** кроме таблиц, используемых в игре, вы, как дизайнер, также будете создавать таблицы для хранения данных пробной игры и информации о восприятии игроков. Подробнее об этом рассказывается в главе 10 «Тестирование игры».

Конечно, таблицы также являются формой технологий в играх, поэтому они пересекают грань между механикой и технологией. Таблицы, как технология, включают хранение информации и любых преобразований этой информации, которые могут иметь место в таблице (например, формул в электронных таблицах). Таблицы, как механика, включают проектные решения, которые дизайнеры игр принимают и вписывают в таблицу.

Фиксированная эстетика

Фиксированная эстетика — это элементы эстетики, которые создаются разработчиками игры. Они охватывают все пять органов чувств, и вы, как дизайнер, должны понимать, что в течение всего процесса игры все пять органов чувств у игрока будут продолжать работать.

Пять органов чувств

Занимаясь созданием игры, дизайнер должен помнить обо всех пяти органах чувств человека:

- **Зрение:** из пяти чувств зрению уделяется больше всего внимания со стороны разработчиков игр. В результате качество визуального восприятия, которое мы можем дать игрокам, за последние десятилетия стало заметно лучше по сравнению с другими аспектами восприятия. Размышляя о видимых элементах своей игры, не ограничивайтесь трехмерным представлением в цифровой игре или оформлением игровой доски или карточек в настольной игре. Не забывайте: все, что видят игроки (или потенциальные игроки), так или иначе имеющее отношение к вашей игре, влияет на их впечатление. Некоторые разработчики игр в прошлом потратили колоссальное количество времени на внутреннее оформление игры, только чтобы упаковать ее (и спрятать красоту) в ужасную обертку.
- **Слух:** звуки в играх занимают второе место по качеству передачи, уступая только визуальному представлению. Все современные игровые консоли под-

держивают вывод 5.1-канального звука, а некоторые обеспечивают еще более высокое качество. Звуковое сопровождение игр включает звуковые эффекты, музыку и диалоги. Все элементы звукового сопровождения требуют разного количества времени для интерпретации игроком, и для каждого имеются свои оптимальные варианты использования. Кроме того, в средних и крупных командах разработчиков имеются свои специалисты для работы со следующими тремя типами звукового сопровождения.

Тип звукового сопровождения	Актуальность	Оптимальный вариант использования
Звуковые эффекты	Немедленная	Уведомление игрока; передача простой информации
Музыка	Средняя	Создание настроения
Диалоги	Средняя/Долгая	Передача сложной информации

Другой аспект звукового сопровождения, который следует учитывать, — фоновый шум. В отношении мобильных игр почти всегда можно предполагать, что игрок находится не в самых лучших условиях для прослушивания звукового сопровождения. Хотя звук всегда можно добавить в игру, неразумно делать его жизненно важным аспектом мобильной игры, если только это не основная особенность игры (например, такой как *Papa Sangre* студии Somethin' Else или *Freeq* студии Psychic Bunny). Фоновый шум также желательно иметь в виду при создании компьютерных игр и игр для консолей. Некоторые охлаждающие кулеры работают очень шумно, и вы должны учитывать это при создании пауз тишины в цифровых играх.

- **Осязание:** значимость осязания существенно отличается для настольных и цифровых игр, но в обоих случаях это чувство дает самый прямой контакт с игроком. В настольной игре осязание участвует в ощущении игровых предметов, карточек, доски и т. д. Создают ли эти элементы ощущение предметов высокого качества или, наоборот, — дешевых поделок? Часто хотелось бы первое, но и последнее не особенно страшно. Джеймс Эрнст, вот уже несколько лет самый известный, пожалуй, дизайнер настольных игр в мире, основал компанию с названием Cheap Ass Games, целью которой стало создание отличных игр по минимальной цене. Чтобы максимально снизить себестоимость, в изготовлении игр использовались самые дешевые материалы, но это даже нравилось игрокам, потому что игры этой компании стоили меньше 10 долларов вместо 40–50, обычных для многих настольных игр. Любые решения возможны; просто убедитесь, что узнали все варианты.

Одним из самых захватывающих технологических достижений для прототипирования настольных игр стала трехмерная печать, и многие дизайнеры начали печатать детали для прототипов своих игр. В настоящее время существует даже несколько онлайн-компаний, готовых напечатать вам игровую доску, карточки и фишки.

Цифровые игры также могут задействовать осязание. Насколько удобно пульт управления лежит в руках игрока и как быстро он вызывает ощущение усталости — вот главные аспекты, которые дизайнер должен принять во внимание. Занимаясь переносом фантастической игры *Okami* для PlayStation 2 на Nintendo Wii, дизайнеры решили изменить команду атаки и вместо нажатия кнопки (X на пульте PlayStation) использовать движение пультом Wiimote (напоминающее жест атаки в *The Legend of Zelda: Twilight Princess*, хорошо зарекомендовавший себя в Wii). Однако в игре *Twilight Princess* даже в самый разгар боя атаки происходили не чаще одного раза в несколько секунд, а в *Okami* они могли следовать с частотой до нескольких раз в секунду, поэтому жест атаки, хорошо показавший себя в *Twilight Princess*, быстро вызывал усталость игрока в *Okami*. С распространением планшетов и смартфонов касания и жесты превратились в элементы, которые должен принимать во внимание каждый дизайнер цифровых игр.

Еще один интересный аспект использования осязания в цифровых играх — обратная связь с игроком посредством вибрации. Большинство современных пультов игровых консолей позволяют выбрать интенсивность и стиль вибрации, а некоторые, такие как Nintendo Switch, предлагают очень широкие возможности организации обратной связи через вибрацию.

- **Обоняние** не часто рассматривается как аспект фиксированной эстетики, но все же ему тоже уделяют внимание. Так же как разные процессы печати книг дают разные запахи, так же по-разному пахнут игровые доски и карточки, напечатанные с применением разных процессов. Обязательно попросите образец у своего производителя, прежде чем заказывать печать 1000 экземпляров чего-то, что может иметь странный запах.
- **Вкус** еще реже имеет значение для игр, чем запах, и все же в некоторых играх он имеет значение, в том числе в играх, связанных с употреблением напитков или с поцелуями.

Цели эстетики

Человечество занималось искусством и музыкой задолго до появления письменной истории. Поэтому, проектируя и разрабатывая встроенные эстетические элементы игры, мы, как разработчики игр, пользуемся многовековыми культурными традициями. Интерактивные возможности позволяют максимально эффективно использовать эти традиции и внедрять все приемы и знания эстетического искусства в создаваемые нами игры. Но при этом мы должны руководствоваться здравым смыслом, и эстетические элементы должны естественно сочетаться с другими элементами игры. Эстетические элементы в играх служат двум важным целям: созданию настроения и передаче информации.

- **Настроение:** эстетика оказывает фантастическую помощь в создании эмоционального настроения в игре. Конечно, настроение можно передавать через механику игры, но изображения и музыка способны влиять на настроение игрока намного быстрее и сильнее, чем механика.

- **Информация:** некоторые цвета и их сочетания особо интерпретируются нашей психикой так же, как у других млекопитающих. Например, многие животные воспринимают красный и чередование черного и желтого цвета как признак опасности[1]. Напротив, холодные цвета, такие как синий и зеленый, обычно воспринимаются как умиротворяющие.

  Кроме того, дизайнеры могут приучать игроков понимать различные эстетические аспекты как имеющие конкретный смысл. Игра *X-Wing* студии LucasArts стала одной из первых, в которой звуковое сопровождение формировалось процедурно, в зависимости от игровой ситуации[2]. В этой игре напряженность музыки усиливалась, предупреждая игрока об атаке со стороны противника. Аналогично, как описывается в главе 12 «Руководство игроком», игра *Uncharted 3* студии Naughty Dog использует яркий синий и желтый цвет, чтобы помочь игроку заметить опоры для рук и ног при восхождении.

Фиксированный сюжет

Так же как все другие формы восприятия, драматизм и сюжет являются важной частью многих интерактивных пространств. Однако сюжеты игр сталкиваются с проблемами, неведомыми линейным повествованиям, и поэтому писатели все еще учатся правильно создавать и преподносить интерактивные сюжеты. В этом разделе рассматриваются компоненты встроенной драматургии, цели их использования, методы повествования в играх и различия между игровым и линейным сюжетами.

Компоненты фиксированного сюжета

И в линейном, и в интерактивном повествовании используются одинаковые драматические компоненты: замысел, окружение, герой и сюжет.

- **Замысел** — это основа повествования, из которого рождается история[3]:

  Давным-давно в далекой галактике межгалактическая война докатилась до порога дома молодого фермера, который еще не осознавал важности своего происхождения и самого себя.

---

[1] Предупреждающая окраска, как эта, называется также *апосематизмом*. Это название «предупреждающей окраски» предложено в 1867 году Альфредом Расселом Уоллесом (Alfred Russel Wallace) и опубликовано в 1877-м. Wallace, Alfred Russel (1877). «The Colours of Animals and Plants. I.—The Colours of Animals». Macmillan's Magazine. 36 (215): 384–408.

[2] В числе других первых игр, где звуковое сопровождение генерировалось процедурно, можно назвать *Wing Commander* (1990) студии Origin Systems и *Monkey Island 2: Le Chuck's Revenge* (1991) студии LucasArts.

[3] Далее приводятся примеры замыслов для игр *Star Wars: A New Hope*, *Half-Life* и *Assassin's Creed 4: Black Flag*.

Гордон Фримен не подозревал, какие сюрпризы поджидают его в первый день работы в сверхсекретном исследовательском центре Black Mesa.

Эдвард Кенуэй должен сражаться и заниматься пиратством в Карибском море на пути к своей фортуне. Его цель — найти таинственную обсерваторию, которую также ищут тамплиеры и ассасины.

- **Окружение** дополняет замысел деталями и создает подробный мир, в котором разворачивается сюжет. Окружение может быть большим, как галактика, или маленьким, как крошечная комната под лестницей, важен не размер, а правдоподобность в границах замысла и внутренняя непротиворечивость: если ваши герои должны сражаться на мечах в мире, полном разного оружия, тому должна быть веская причина.

  В «Звездных войнах», когда Оби-Ван Кеноби дает Люку световой меч, он объясняет Люку и зрителям, что такие мечи используются во вселенной «Звездных войн» потому, что это «не такое грубое и беспорядочное оружие, как бластер; это изящное оружие для более благородной эпохи».

- **Герой:** истории рассказывают о героях, а хорошие истории — о героях, вызывающих сопереживание. Обычно образ героя создается из предыстории и одной или нескольких целей. Они объединяются, чтобы придать герою определенную роль в повествовании: главный герой, противник, соратник, слуга, наставник и т. д.

- **Сюжет** — это последовательность событий, происходящих в повествовании. Обычно заключается в описании того, как главный герой с трудом достигает желаемых целей из-за противодействия со стороны противника или сложившихся обстоятельств. Сюжет превращается в рассказ, как главный герой пытается преодолеть возникающие перед ним трудности или препятствия.

Традиционные драматические элементы

Несмотря на то что интерактивное повествование предлагает писателям и разработчикам много новых возможностей, оно часто следует традиционным структурам драматургии.

Пятиактная структура

Немецкий писатель Густав Фрейтаг описал пятиактную структуру в 1863 году в своей книге «Die Technik des Dramas» (Техника драмы). Он описал назначение пяти актов, часто используемых Шекспиром и многими его современниками (а также древнеримскими драматургами), и предложил то, что позднее стало называться пирамидой Фрейтага (рис. 4.2). Вертикальные оси на рис. 4.2 и 4.3 обозначают уровень взволнованности аудитории в разных точках повествования.

Далее перечислены характеристики актов по Фрейтагу:

- **Акт I. Завязка:** описывает сюжетный замысел, окружение и знакомит с основными героями. В первом акте трагедии «Ромео и Джульетта» Уильям Шекспир

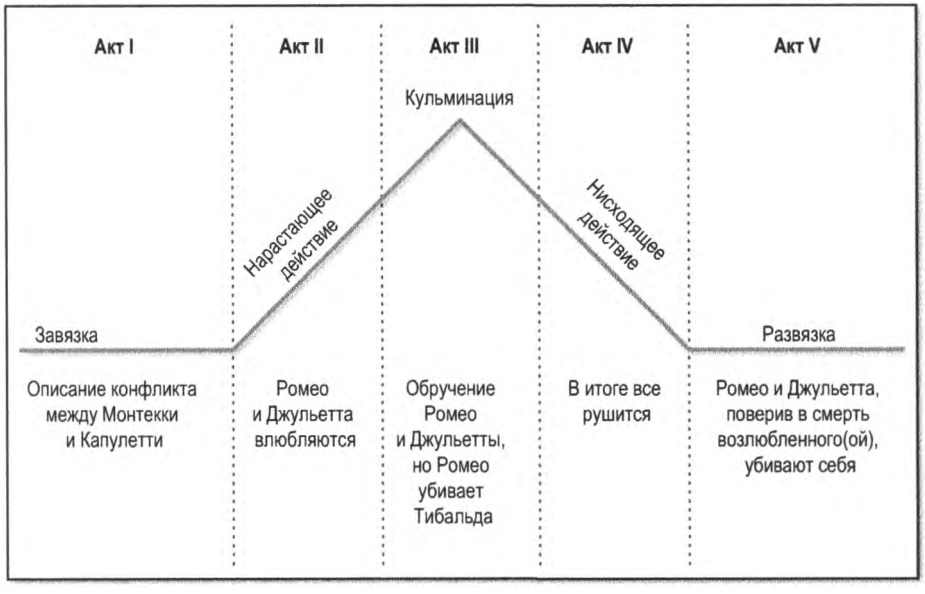


Рис. 4.2. Пирамида Фрейтага пятиактной структуры, показывающей развитие сюжета в трагедии «Ромео и Джульетта» Уильяма Шекспира

знакомит нас с Вероной, городом в Италии, и описывает вражду между могущественными семьями Монтекки и Капулетти. Ромео представляется как сын Монтекки, увлеченный Розалиной.

- **Акт II. Нарастающее действие:** некоторые события, вызывающие новое волнение о судьбе основных героев, и драматическое напряжение резко возрастает. Ромео пробирается на бал в дом Капулетти и оказывается мгновенно сражен красотой Джульетты, дочери Капулетти.
- **Акт III. Кульминация:** наступает апогей и решается исход пьесы. Ромео и Джульетта втайне обручаются, и местный священник надеется, что это приведет к миру между семьями. Но следующим утром Тибальд, двоюродный брат Джульетты, провоцирует Ромео. Ромео отказывается драться, поэтому вместо него в драку ввязывается Меркуцио, и Тибальд убивает его из-под руки Ромео, когда последний пытался остановить драку. Ромео в ярости преследует Тибальда и убивает его. Убийство Тибальда — это кульминация пьесы, потому что до этого момента казалось, что все обойдется и закончится хорошо для влюбленных, но после него зрители понимают, что все закончится ужасно.
- **Акт IV. Нисходящее действие:** пьеса неуклонно движется к неизбежному завершению. Если это комедия, ситуация улучшается, если трагедия — может показаться, что ситуация становится лучше, но в действительности неизбежно ухудшается еще больше. Зрители наблюдают последствия кульминационного момента. Ромео изгоняется из Вероны. Священник придумывает план, как Ромео

и Джульетта могли бы бежать вместе. Он предлагает Джульетте сымитировать ее смерть и послать известие Ромео, но посланник не смог донести письмо до Ромео.

- **Акт V. Развязка:** исход пьесы разрешается. Ромео приходит в склеп, думая, что Джульетта действительно умерла, и выпивает яд. В это время пробуждается Джульетта, обнаруживает рядом мертвого Ромео и тоже убивает себя. Семьи осознают весь ужас трагедии и плачут, обещая прекратить вражду.

Трехактная структура

В своих лекциях и книгах американский сценарист Сид Филд предлагал другой способ традиционного повествования — в трех актах[1]. Между актами сюжет меняет направление и побуждает героев к действиям. На рис. 4.3 изображен пример, который описывается ниже.

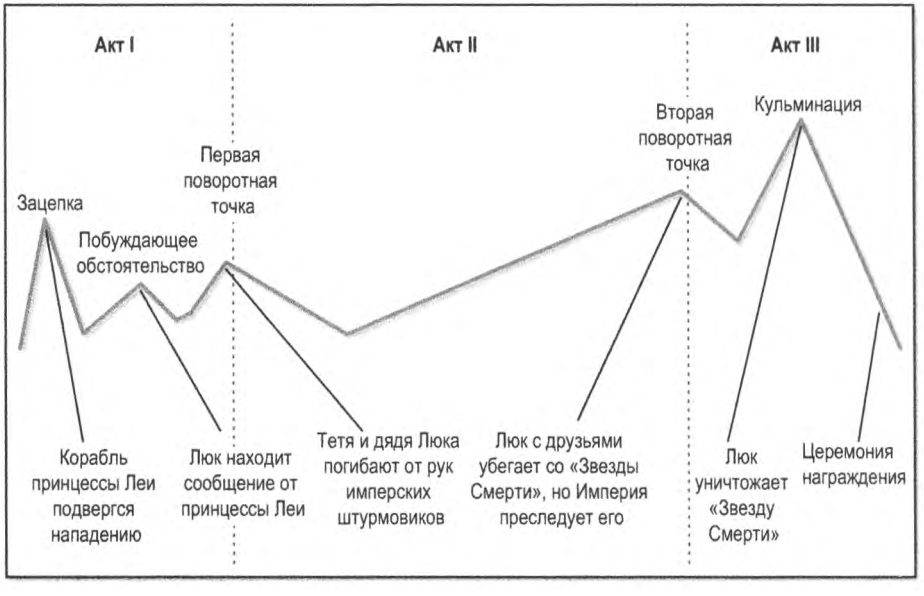


Рис. 4.3. Трехактная структура Сида Филда с примерами из *Star Wars: A New Hope*

Вот основные элементы трехактной структуры Филда:

- **Акт I. Завязка:** описывает мир, о котором рассказывается в повествовании, окружение и знакомит с основными героями. В первом акте «Звездных войн» Люк — юный идеалист, работающий на влагодобывающей ферме дяди. В галактике происходит восстание против фашистской Империи, но он — всего лишь деревенский мальчишка, мечтающий стать пилотом истребителя.

[1] Syd Field, *Screenplay: The Foundations of Screenwriting*, New York: Delta Trade Paperbacks, 2005. — *Примеч. пер.*

- **Зацепка:** быстро привлекает внимание аудитории. По словам Филда, первые несколько минут решают, будут ли зрители смотреть фильм, поэтому первые минуты должны быть по-настоящему захватывающими, даже если действие в них не имеет прямого отношения к оставшейся части фильма (как, например, начало любого фильма о Джеймсе Бонде). В «Звездных войнах» начальная сцена с нападением звездного разрушителя на корабль принцессы Леи насыщена самыми потрясающими визуальными эффектами, которые когда-либо видели зрители в 1977 году, и сопровождается фантастической музыкой Джона Уильямса. И то и другое создали отличную зацепку.
- **Побуждающее обстоятельство:** нечто новое, появляющееся в жизни главного героя, заставляющее его начать приключение. Люк ведет обычную жизнь, пока не находит секретное сообщение внутри R2-D2. Эта находка заставляет его пуститься на поиски «старого Бена» Кеноби, который изменит его жизнь.
- **Первая поворотная точка** в сюжете заканчивает первый акт и подталкивает главного героя встать на путь ко второй поворотной точке. Люк решает остаться дома и не помогать Оби-Ван Кеноби, но когда обнаруживает дядю и тетю, убитыми штурмовиками Империи, он передумывает и решает присоединиться к Оби-Вану, чтобы стать джедаем.

- **Акт II. Противоборство:** главный герой начинает свое путешествие, но сталкивается с чередой препятствий. Люк и Оби-Ван нанимают Хана Соло и Чубакку, чтобы они помогли им доставить на Альдераан секретные планы, хранящиеся в R2-D2; но когда они прибывают на место, то обнаруживают, что Альдераан разрушен и их корабль захватывает «Звезда Смерти».
  - **Вторая поворотная точка** в сюжете заканчивает второй акт и вынуждает главного героя решить, что он будет делать в третьем акте. Ценой неимоверных усилий Люку с друзьями удается бежать со «Звезды Смерти», захватив с собой принцессу и планы, но его наставник, Оби-Ван Кеноби, погибает. «Звезда Смерти» следует за ними к секретной базе мятежников, и Люк должен выбирать — помогать в нападении на «Звезду Смерти» или уйти с Ханом Соло.
- **Акт III. Развязка:** завершение истории, где главный герой либо добивается успеха, либо терпит неудачу. Так или иначе он выходит из истории с новым пониманием, кто он есть на самом деле. Люк решает помочь атаковать «Звезду Смерти» и в итоге спасает положение.
  - **Кульминация:** момент, когда наступает развязка и дается ответ на главный вопрос сюжета. Люк остается один на один со «Звездой Смерти», потеряв обоих ведомых и R2-D2. На него нападают истребители из эскадрильи Дарта Вейдера, но Хан и Чубакка, в свою очередь, атакуют вражеские истребители и дают Люку возможность сделать выстрел. Люк решается довериться Силе, закрывает глаза и делает чрезвычайно трудный и удачный выстрел, уничтожая «Звезду Смерти».

В большинстве современных фильмов и почти во всех видеоиграх кульминационный момент находится очень близко к концу повествования, почти не оставляя времени для нисходящего действия или развязки. Одним из ярких примеров может служить игра *Red Dead Redemption* студии Rockstar Games. После мощного кульминационного момента, когда главный герой, Джон Марстон, наконец побеждает человека, для убийства которого он был нанят правительством, и получает разрешение вернуться домой к семье, в игре звучит единственный музыкальный трек, пока Джон медленно бредет к дому по снегу. Затем игроку предлагается выполнить несколько скучных заданий, например избавить семейное зернохранилище от ворон, научить вздорного сына пасти скот и сделать массу других дел по хозяйству. Игрок чувствует, какую скуку испытывает Джон, занимаясь всем этим. Затем те же правительственные агенты, что нанимали Джона, приходят к нему на ферму и убивают его, выполнив наконец свою задачу. После смерти Джона изображение на экране медленно гаснет, затем вновь появляется, и игрок оказывается уже в роли Джека (сына Джона), через три года после гибели отца. Игра возвращается к более активным заданиям, и теперь уже Джек пытается выследить агентов, убивших его отца. Такой вид нисходящего действия редко встречается в играх, и он сделал повествование в игре *Red Dead Redemption* одним из самых запоминающихся для меня.

Отличия интерактивного и линейного сюжетов

Интерактивное и линейное повествования по своей сути совершенно разные, потому что аудитория и игрок играют в них разные роли. Несмотря на то что зритель или читатель безусловно использует свой опыт для интерпретации любой потребляемой информации, он может лишь изменить свое отношение, но не ход повествуемых событий. Игрок, напротив, постоянно влияет на события, в которых участвует, и поэтому обладает свободой выбора в интерактивном повествовании. Это означает, что авторы интерактивных сюжетов должны знать некоторые основные отличия, важные для их интерактивных историй.

Сюжет против воли

Разрабатывая интерактивный сюжет, сложнее всего отказаться от попыток контролировать его. И авторы, и читатели/зрители привыкли к сюжетам с такими элементами, как предзнаменование, судьба, ирония и другими, когда ожидаемый исход влияет на ранние части истории. В по-настоящему интерактивном окружении это невозможно из-за свободы воли игрока. Не зная, какой выбор сделает игрок, трудно предвидеть результаты этого выбора. Тем не менее есть определенные способы решения этой двойственности, часть которых уже широко используется в цифровых играх, а другие уже использовались в настольных ролевых играх, но пока не нашли применения в цифровых:

- **Ограничение возможностей** — важная часть почти всех интерактивных повествований. Фактически большинство игр на их фиксированном уровне не

являются интерактивными повествованиями. Все самые популярные серии игр последнего десятилетия (*Prince of Persia*, *Call of Duty*, *Halo*, *Uncharted* и др.) основаны исключительно на линейной истории. Что бы вы ни делали в игре, фактически на выбор у вас всего два варианта: продолжать следовать за сюжетом или покинуть игру. Разработчики *Spec Ops: The Line* из студии Yager Development нашли красивое решение этой проблемы, поставив игрока и главного героя в позицию, когда у него на выбор только два реальных варианта: продолжить творить зло или прекратить игру. В *Prince of Persia: The Sands of Time* проблема решается с помощью рассказчика (принца и главного героя), который говорит: «Нет, нет, нет; все должно быть совсем не так. Начнем сначала?» — когда персонаж погибает и игра возвращается к последней контрольной точке. В серии *Assassin's Creed* в таких ситуациях игроку сообщается, что он «отклонился» от истории предшественника, если (из-за отсутствия навыков) предшественник погиб.

Есть также несколько примеров игр, ограничивающих выбор несколькими вариантами, которые основываются на действиях игрока на протяжении всей игры. Например, в Lionhead Studios (игра *Fable*) и в BioWare (игра *Star Wars: Knights of the Old Republic*) заявили, что наблюдают за игроком в течение всей игры, чтобы определить финальный исход. Но даже при том, что в обеих играх действия игроков оцениваются по шкале добра и зла за всю игру, в обоих случаях (как и во многих других играх) единственный выбор в конце игры может переопределить результат всей игры.

Другие игры, такие как японские ролевые игры *Final Fantasy VII* и *Chrono Trigger*, имеют более тонкие и разнообразные возможности. В *Final Fantasy VII* есть момент, когда главный герой, Клауд, отправляется на встречу с другим главным персонажем в парк аттракционов Golden Saucer. По умолчанию на встречу с Клаудом должна прийти Айрис; но если игрок игнорировал Айрис на протяжении игры и не принял ее в свой отряд, на встречу с Клаудом придет Тифа. Вообще на встречу может прийти один из четырех персонажей: Айрис, Тифа, Юффи или Баррет, хотя для встречи с Барретом требуется приложить целенаправленные усилия. В описании игры нигде не объясняется такое поведение, но оно всегда имеет место, а кроме того, команда *Final Fantasy* использовала аналогичную стратегию в *Final Fantasy X*, чтобы определить, с кем главный герой Тидус совершит романтическую прогулку на снегоходах. В *Chrono Trigger* используется несколько показателей, чтобы определить, какое из тринадцати окончаний игры выбрать (и некоторые из них имеют по несколько вариантов). И снова вычисления, связанные с этим, выполняются незаметно для игрока.

- **Предоставление на выбор нескольких побочных линейных квестов:** эта стратегия используется во многих играх с открытым миром студии Bethesda Softworks, включая *Fallout 3* и *Skyrim*. Даже при том, что главный квест в этих играх почти линейный, он занимает лишь малую долю всей игры. В *Skyrim*, например, прохождение главного квеста занимает примерно 12–16 часов, а для прохождения побочных квестов требуется более 400 часов. Репутация игрока и история его

действий в игре могут повлечь разблокировку одних побочных квестов и отключение других. То есть каждый игрок может иметь свою комбинацию линейных квестов, непохожую на комбинации у других игроков.

- **Предвестие нескольких событий:** если вы предчувствуете, что могут произойти какие-то события, некоторые из них, вероятно, произойдут. Игроки обычно игнорируют предзнаменования, которые не сбываются, и обращают внимание на те, которые сбываются. Это часто происходит в телевизионных сериалах и шоу, закладывающих несколько возможностей для будущих сюжетов, из которых реализуется лишь часть (например, в *Farscape*, в серии «A Clockwork Nebari», рассказывается о расе Nebari, желающей захватить власть над Вселенной, но потом она больше нигде не упоминается; или титульный персонаж из серии «The Doctor's Daughter» в сериале *Doctor Who* тоже больше не появляется в других сериях).
- **Превращение второстепенных персонажей, которыми управляет компьютер, в основных:** мастера игры в настольных ролевых играх часто пользуются этим приемом. Например, представьте, что на игроков напала банда из десяти человек. Игроки уничтожили банду, но одному бандиту удалось уйти. Позднее мастер игры мог бы вернуть бандита в игру с целью отомстить игрокам за убитых друзей. Это важное отличие от таких игр, как *Final Fantasy VI* (первоначально названной *Final Fantasy III in the U.S.*), где с самого начала очевидно, что Кефка будет возвращаться снова и снова и, наконец, превратится в главного отрицательного героя. Персонажи, действующие на стороне игрока, не понимают этого, но сам факт, что разработчики придали смеху Кефки особое звучание, делает это очевидным для игрока.

[+] Настольные ролевые игры предлагают игрокам уникальный опыт интерактивной игры, и я настоятельно рекомендую их. В действительности, когда я преподавал в Южно-Калифорнийском университете, я требовал, чтобы все мои студенты попробовали сыграть в ролевую игру со своими сокурсниками. Примерно 40 % студентов в каждом семестре отметили, что это стало их любимым занятием.

Поскольку настольные ролевые игры протекают под управлением человека, мастер игры может оперативно создать для игроков свой сюжет, не соответствующий сюжету в компьютерной игре. Мастера игры используют все стратегии, перечисленные выше, для руководства своими игроками и создания у них впечатления предчувствия, предопределенности или иронии, которые обычно используются в линейном повествовании.

Вечная игра *Dungeons & Dragons*, созданная в Wizards of the Coast, может послужить отличной отправной точкой, и для нее написано бесчисленное множество справочников и руководств. Однако, как мне кажется, кампании в *D&D* больше ориентированы на борьбу, а борьба отнимает очень много времени. Поэтому за опытом создания и переживания интерактивных историй я рекомендую обратиться к системе *FATE Accelerated*, созданной в Evil Hat Productions.[1]

[1] http://www.evilhat.com/home/fae/, была доступна в марте 2018 года.

Эмпатия против олицетворения

В линейных повествованиях главным героем обычно является персонаж, которому аудитория должна сопереживать. Когда зрители наблюдают, как Ромео и Джульетта принимают неразумные решения, они вспоминают свою молодость и сопереживают чувствам, которые ведут двух юных влюбленных по фатальному пути. В интерактивном повествовании главный герой, напротив, является не персонажем, отделенным от игрока, а его *аватаром* в игровом мире. (Слово «аватар» на санскрите означает физическое воплощение бога на Земле; в играх это виртуальное воплощение игрока в игровом мире.) Это может вызывать диссонанс между действиями и индивидуальностью игрока в реальном мире и индивидуальностью игрового персонажа, управляемого игроком. В моем случае такой диссонанс наблюдался в отношении Клауда Страйфа, главного героя игры *Final Fantasy VII*. В игре Клауд был более вздорным, чем мне хотелось бы, но в целом его молчаливость позволяла мне проецировать на него свой характер. Однако после ключевой сцены, в которой Клауд теряет кого-то из близких, он предпочел сесть в инвалидное кресло и сдаться, вместо того чтобы сражаться и спасти мир от Сефирот, как мне хотелось бы. Это противоречие между выбором моего персонажа и тем, что хотел бы выбрать я, было для меня очень огорчительно.

Отличный пример такого противоречия, используемого для большего эффекта, есть в фантастической игре *Okami* (2006), созданной студией Clover Studio. В *Okami* персонаж игрока — Аматэрасу — реинкарнация богини Солнца в обличье белого волка. За последние 100 лет Аматэрасу утратила часть своей силы, и игрок должен потрудиться, чтобы вернуть ее. Примерно через четверть повествования главный отрицательный герой, демон Ороти, выбрал деву для принесения в жертву. И игрок, и соратник Аматэрасу — Иссун знают, что на данном этапе Аматэрасу восстановила лишь малую часть своей силы, и игрок чувствует настороженность перед лицом Ороти, находясь в таком ослабленном состоянии. Однако, несмотря на протесты Иссун, Аматэрасу бросается в бой. По мере нарастания музыки в поддержку ее решения мои чувства менялись от трепета до безрассудства, и фактически я почувствовал себя героем, потому что знал, что обстоятельства складываются против меня, но я делал то, что должен был делать.

Дизайнеры игр и интерактивных сюжетов решили проблему диссонанса между игроком и аватаром, применив следующие приемы:

- **Вживание в роль:** безусловно, самый распространенный прием в играх — заставить игрока играть роль игрового персонажа. Играя в игры, развитие событий в которых зависит от персонажа, такие как *Tomb Raider* или *Uncharted*, игрок играет не себя, а роль Лары Крофт или Натана Дрейка. Игрок отделяет свою личность, чтобы вжиться в фиксированную личность главного героя игры.
- **Молчаливость главного героя:** по традиции, заведенной еще во времена первой игры *Legend of Zelda*, многие главные герои безмолвствуют. Другие персонажи говорят с ними и реагируют, как если бы слышали их слова, но игрок никогда не слышит слов, произнесенных персонажем игрока. Так делается потому, что

иначе игрок мог бы наложить свою личность на главного героя, а не вжиться в личность, созданную разработчиками игры. Однако независимо от того, что говорит или не говорит Линк, его личность достаточно четко демонстрируется его действиями, а в случае с Клаудом, который не говорит ни слова, игроки все равно испытывают диссонанс между их желаниями и действиями, как я описал выше.

- **Выбор из нескольких вариантов развития диалога:** многие игры предлагают игроку несколько вариантов развития диалога, что, безусловно, помогает игроку полнее почувствовать, что он управляет личностью персонажа. Однако при этом важно соблюсти следующие важные требования:
  - **Игрок должен понимать последствия своих слов:** иногда линия поведения, совершенно понятная для создателей игры, не выглядит такой же очевидной для игроков. Если игрок выбирает ответ, который кажется ему приветственным или одобрительным, а писатель расценивал его как оскорбление, поведение персонажа, управляемого компьютером, может показаться игроку более чем странным.
  - **Выбор ответа должен иметь значение:** некоторые игры предлагают игроку фиктивный выбор, ожидая, что тот сделает выбор, нужный игре. Если, например, его просят спасти мир, а он отвечает просто «Нет», игра отвечает примерно такой фразой: «Нет, вы не можете отказать», — фактически отказывая игроку в выборе.

  Одним из фантастических примеров воплощения этих требований является колесо диалога в серии игр *Mass Effect* студии BioWare. Эти игры предлагают игроку колесо с вариантами выбора продолжения диалога, а разделы колеса кодируют определенный смысл. Вариант в левой части колеса обычно растягивает общение, а вариант в правой части, наоборот, сокращает. Вариант в верхней части является дружественным, а в нижней — неприветливым или даже враждебным. Благодаря такому позиционированию вариантов ответов в диалоге, игрок получает важную информацию об интерпретации разных вариантов и уже не удивляется исходу.

  Еще одним, совершенно другим, но столь же убедительным примером может служить игра *Blade Runner* студии Westwood Studios (1997). Дизайнеры понимали, что выбор вариантов в диалоге помешает плавности восприятия событий в игре, поэтому было решено предлагать игроку не конкретные варианты ответов, а только настроение для его персонажа (дружественное, нейтральное, неприветливое или случайное). Главный герой действовал и говорил в соответствии с настроением, выбранным игроком, не нарушая плавного течения повествования, а игрок мог изменить настроение в любой момент, чтобы изменить ответ персонажа в той или иной ситуации.

- **Отслеживание действий игрока и соответствующая реакция на них:** некоторые игры следят за отношением игрока к разным группировкам, и члены этих группировок соответственно реагируют на игрока. Сделайте одолжение оркам,

и они могут позволить вам продавать свои товары в их поселениях. Арестуйте члена Гильдии воров, и в будущем вы можете подвергнуться нападению с их стороны. Это общая черта ролевых игр с открытым миром в стиле вестернов, создаваемых студией Bethesda Softworks, и в какой-то степени она основана на системе морали из восьми достоинств и трех принципов, введенной в игре *Ultima IV* студией Origin Systems, — одного из первых примеров сложной системы морали в цифровой игре.

Цели фиксированного сюжета

Фиксированный сюжет может служить нескольким целям в играх:

- **Вызывание эмоций:** за последние несколько столетий писатели научились манипулировать эмоциями своей аудитории, пользуясь приемами драматического искусства. Это верно и в отношении игр и интерактивного повествования, и даже исключительно линейное повествование, встроенное в игру, может фокусировать и формировать чувства игрока.
- **Стимулирование и обоснование:** драматургию можно использовать не только для формирования эмоций, но и для того, чтобы подтолкнуть игрока к определенным действиям или обосновать действия, если они кажутся неприятными. Это особенно верно в отношении фантастического пересказа приключенческой повести «Heart of Darkness» Джозефа Конрада в игре *Spec Ops: The Line*. Более позитивным примером может служить игра *The Legend of Zelda: The Wind Waker*. В начале игры сестра Линка, Арилл, одалживает ему свой телескоп на один день, потому что это его день рождения. В тот же день ее похищает гигантская птица, и первая часть игры управляется желанием Линка спасти ее. Описание, что она дала что-то игроку перед ее похищением, усиливает желание игрока спасти ее.
- **Обеспечение движения вперед и вознаграждение:** во многих играх используются анимационные заставки и другие встроенные фрагменты повествования, чтобы напомнить игроку, где он находится в истории, и вознаградить его за движение вперед. Если сюжет игры носит в основном линейный характер, понимание игроком традиционной структуры повествования может помочь ему понять, где в трехактной структуре повествования он находится в данный момент, благодаря чему он сможет сообщить, насколько далеко продвинулся в общем сюжете игры. Встроенные анимационные заставки также часто используются как награда, отмечая конец уровня или другого раздела игры. Это верно почти для всех продаваемых однопользовательских игр с линейными сюжетами (например, *Modern Warefare*, *Halo* и *Uncharted*).
- **Подкрепление механики:** одной из наиболее важных целей встроенных драматических элементов является подкрепление механики игры. Фантастическим примером этого может служить немецкая настольная игра *Up the River*, выпущенная компанией Ravensburger. В ней игроки пытаются переместить свои лодки вдоль доски, которая постоянно движется назад. Название «река»

для этой доски подкрепляет механику ее обратного движения в игре. Область в пределах доски, где движение останавливается, называется «отмель» (потому что корабли останавливаются, зарываясь в отмелях). Аналогично область, ускоряющая движение вперед, называется «приливом». Поскольку каждый из этих элементов имеет связанное с ним нарративное значение, их намного проще запомнить, чем, например, если бы игроку предлагалось запомнить, что область № 3 вызывает остановку лодки, а область № 7 толкает ее вперед.

Фиксированная технология

Фиксированная технология, так же как фиксированная механика, в значительной степени понимается только через ее динамическое поведение. Это верно для технологий и настольных, и цифровых игр. Выбор количества кубиков, бросаемых игроком, обретает значение лишь в процессе игры, так же как код, написанный программистом, осознается игроком, только когда он видит игру в действии. Это одна из причин, согласно которой, как отмечает Джесси Шелл в описании своей простой тетрады[1], технология — наименее заметный из фиксированных элементов.

Кроме того, у фиксированной механики и фиксированной технологии обширная область пересечения. Технология поддерживает механику, а решения в сфере механики могут управлять выбором технологии.

Фиксированная технология настольной игры

Фиксированные технологии настольных игр часто используются для внесения элемента случайности, отслеживания состояния и движения вперед.

- **Внесение элемента случайности:** внесение элемента случайности — это наиболее типичная форма технологии в настольных играх. К ней относятся кубики, карты, домино, вращающиеся барабаны и многое другое. Как дизайнер, вы вольны выбрать любой инструмент внесения случайной составляющей. Вы можете также объединить случайность с таблицами, чтобы, например, генерировать случайные встречи и персонажей для игры. Более подробно о разных способах внесения элемента случайности и о том, когда они используются, вы узнаете в главе 11 «Математика и баланс игры».
- **Отслеживание состояния:** под отслеживанием состояния может подразумеваться все что угодно: от запоминания очков, набранных разными игроками (как доска для счета при игре в криббидж), до таблиц с описанием характеристик персонажей в некоторых ролевых играх.
- **Движение вперед:** информация о движении вперед часто имеет форму графиков и таблиц. Сюда относится информация об изменении возможностей при переходе игрока на следующий уровень, изменение различных технологий и единиц

[1] О простой тетраде Джесси Шелла рассказывается в главе 2 «Методы анализа игр».

в технологическом дереве игры, такой как *Civilization*, воспроизводство ресурсов в настольной игре *Power Grid* и т. д.

Фиксированная технология цифровой игры

В последних разделах этой книги широко рассматривается технология цифровых игр в форме их программирования на языке C# с использованием Unity. Так же как в случае с технологией создания настольных игр, искусство программирования игр заключается в умении кодировать переживания, которые игрок должен испытывать, и определять правила (в форме программного кода), которые игрок затем должен декодировать, играя в игру.

Итоги

Четыре элемента фиксированного уровня составляют все, что получают игроки, когда покупают или загружают вашу игру, то есть фиксированный слой — единственный, который находится под полным контролем разработчиков игры. В следующей главе мы позволим игрокам перемещать наши игры, находящиеся в статической форме, вверх, до появления динамического уровня. Завершая обсуждение фиксированного уровня, добавлю, что, как отметил Джесси Шелл в своем обсуждении простой тетрады, структура фиксированного уровня прекрасно сочетается с организацией небольшой студии: Дизайнеры создают механику, Художники — эстетику, Писатели — сюжет, а Программисты разрабатывают технологию.

5 Динамический уровень

Когда игроки начинают играть в игру, она переходит с фиксированного на динамический уровень многоуровневой тетрады. В этом слое появляются игровой процесс, стратегия и осмысленные варианты, выбираемые игроками.

В этой главе рассматривается динамический уровень и появляющиеся на нем новые качества игры и рассказывается, как дизайнеры могут предвидеть динамику игрового процесса, вытекающую из их решений, принятых при создании фиксированного уровня.

Роль игрока

Один дизайнер однажды сказал мне, что игра — не игра вовсе, если в нее никто не играет. Хотя это может звучать как перефразированный философский вопрос: «Издает ли звук дерево при падении, если его никто не слышит?» — но на самом деле такое определение намного важнее для интерактивных пространств, чем для любых других. Фильм все еще может существовать и демонстрироваться в кинотеатре, даже если его никто не смотрит.[1] Телепередачу можно отправить в эфир, и она останется телепередачей, даже если никто ее не смотрит. Но игры не существуют без игроков, потому что только благодаря их действиям игры превращаются из коллекций фиксированных элементов в динамическое действо (рис. 5.1).

Конечно, здесь, как и везде, есть особые случаи. *Core War* — это хакерская игра, в которой игроки пытаются написать компьютерный вирус, распространяющийся в фиктивном компьютере и уничтожающий вирусы соперников. Игроки передают свои вирусы и ждут результатов их схватки друг с другом за память и выживание. В ежегодном турнире *RoboCup* команды роботов соревнуются между собой в футбол без участия программистов во время игры. В классической карточной игре *War*

[1] Некоторые фильмы, такие как *The Rocky Horror Picture Show* («Шоу ужасов Рокки Хоррора»), своей популярностью во многом обязаны выступлениям страстных поклонников перед зрителями, и реакция поклонников на эти фильмы меняет впечатление от просмотра других зрителей. Однако сам фильм никак не меняется от восприятия зрителями. Динамизм в играх, напротив, зависит от способности интерактивного пространства реагировать на действия игрока.

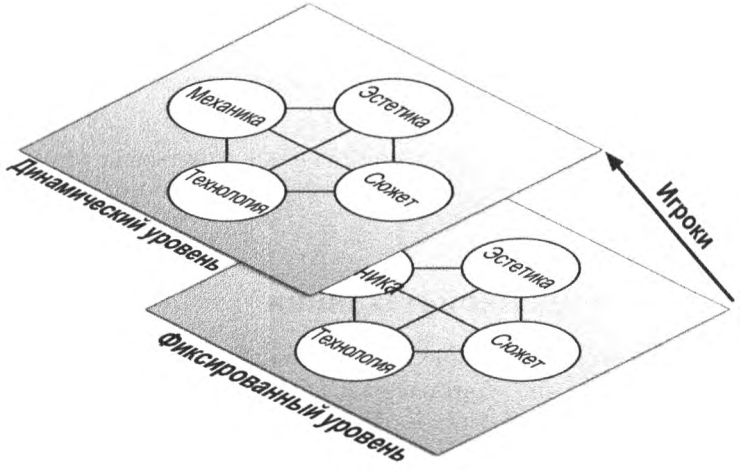


Рис. 5.1. Только игроки переносят игру с фиксированного уровня на динамический

игроки не принимают решений, кроме выбора из двух колод в начале игры, и ход игры полностью зависит от удачного перемешивания в начале. Хотя в каждом из этих случаев игроки ничего не вводят и не принимают никаких решений в ходе игры, игра по-прежнему зависит от их решений, принимаемых перед ее началом, и участники определенно заинтересованы в исходе игры и ожидают его. Кроме того, все эти случаи также требуют, чтобы игроки вступили в игру и сделали выбор, определяющий ее исход.

Игроки оказывают огромное влияние на игру и на процесс игры (включая влияние на элементы тетрады), но находятся за пределами тетрады и являются движущим механизмом, заставляющим ее работать. Игроки — причина появления игры — позволяют ей превратиться в переживания, которые разработчики закодировали в фиксированном уровне. Как дизайнеры, мы надеемся, что игроки помогут игре стать тем, чем она должна стать согласно нашим намерениям. Некоторые аспекты игрового процесса на динамическом уровне вообще неподконтрольны нам, как дизайнерам, в том числе: следование правилам со стороны игрока, заинтересованность игрока в победе, физическое окружение играющего, его эмоциональное состояние и т. д. В связи с особой важностью игроков мы, как разработчики, должны относиться к ним с уважением и заботиться, чтобы фиксированные элементы игры, особенно правила, были понятны игрокам, чтобы они могли декодировать их в запланированные нами переживания.

Непредсказуемость

Наиболее важной идеей в этой главе является *непредсказуемость*, когда даже очень простые правила могут рождать сложное динамическое поведение. Рассмотрим игру *Bartok*, в которую вы играли и с которой экспериментировали в главе 1 «Ду-

мать как дизайнер». Несмотря на небольшое количество правил, из них возникла сложная игра. Начав изменять правила и добавлять свои, вы смогли увидеть, что даже самые простые и, казалось бы, безобидные изменения в правилах могут приводить к большим изменениям в восприятии и в игре.

Динамический уровень в многоуровневой тетраде охватывает результаты пересечения игроков и игры во всех элементах тетрады: механике, эстетике, сюжете и технологии.

Появление неожиданной механики

Мой друг Скотт Роджерс, автор двух замечательных книг о проектировании игр[1], однажды сказал мне, что «не верит в неожиданное появление». Обсудив эту тему, мы вместе пришли к выводу, что в действительности он верит в это, но не верит, что дизайнеры имеют право оправдывать неожиданным появлением безответственный дизайн. Мы согласились, что, как дизайнеры систем внутри игры, мы ответственны за игру, рождающуюся из этих систем. Конечно, очень непросто понять, какие возможности возникнут из определяемых нами правил, именно поэтому так важно выполнять пробные игры. Разрабатывая игры, начинайте тестировать их как можно раньше, играйте как можно чаще и уделяйте особое внимание всему необычному, происходящему даже в одной игре. Когда ваша игра выйдет в мир, это необычное поведение будет проявляться у огромного количества людей намного чаще, чем вы могли бы ожидать. Конечно, такое случается со всеми дизайнерами: взгляните на некоторые из карт, объявленные запрещенными в игре *Magic: The Gathering*, — но, как говорит Скотт, важно, чтобы дизайнер принимал ответственность за эти проблемы и решал их.

Динамическая механика

Динамическая *механика* — это динамический уровень элементов, отделяющих игры и интерактивные пространства от других медиа, элементов, которые делают игры играми. Динамическая механика включает процедуры, осмысленную игру, стратегию, внутренние правила, намерения игрока и исход игры. Так же как в случае с фиксированной механикой, многое из перечисленного является расширенными версиями элементов, описанных в книге Трейси Фуллертон «Game Design Workshop»[2]. Далее мы рассмотрим следующие элементы динамической механики:

---

[1] Scott Rogers, *Level up!: The Guide to Great Video Game Design* (Chichester, UK: Wiley, 2010) и Scott Rogers, *Swipe this! The Guide to Great Tablet Game Design* (Hoboken, NJ: John Wiley & Sons, 2012).

[2] Tracy Fullerton, Christopher Swain и Steven Hoffman, *Game Design Workshop: A Playcentric Approach to Creating Innovative Games* (Burlington, MA: Morgan Kaufmann Publishers, 2008), главы 3 и 5.

- **Процедуры:** действия, выполняемые игроком.
- **Осмысленная игра:** придает вес решениям игрока.
- **Стратегия:** планы, разрабатываемые игроками.
- **Внутренние правила:** простые изменения в игре, производимые игроками.
- **Намерения игрока:** мотивы и цели игроков.
- **Исход:** результат(ы) игры.

Процедуры

Механика фиксированного уровня включает *правила* — инструкции, написанные дизайнером и регламентирующие порядок игры. *Процедуры* — это динамические действия, выполняемые игроками в ответ на эти правила. Можно сказать, что процедуры вытекают из правил. Рассмотрим следующее, дополнительное правило для игры *Bartok* из главы 1:

- **Правило 3:** игрок должен объявить: «Последняя карта!», если у него на руках осталась только одна карта. Если кто-то другой сказал эти слова первым, опоздавший игрок должен взять из ничейной колоды две карты (чтобы общее число карт у него на руках достигло трех).

Это правило напрямую инструктирует активного игрока следовать процедуре объявления, что у него на руках осталась одна карта. Однако это правило косвенно подразумевает другую процедуру: другие игроки могут наблюдать за картами в руках активного игрока, чтобы поймать его, если он забудет объявить о последней карте. До ввода этого правила у игроков не было явной причины следить за игрой, когда ход делает кто-то другой, но это простое правило изменило процедуру игры для активных и неактивных игроков.

Осмысленная игра

В своей книге «Rules of Play» Кэти Сален и Эрик Циммерман определили *осмысленную игру* как игру, *различимую* игроком и *интегрированную* в большую игру[1].

- **Различимость:** действие считается различимым, если игрок может сказать, что оно было. Например, нажимая кнопку вызова лифта, вы можете распознать реакцию на ваше действие по появлению подсветки кнопки. Если вам приходилось вызывать лифт, когда подсветка кнопки не работает, значит, вы понимаете, сколько путаницы может вызывать попытка выполнить действие при невозможности распознать, было ли оно интерпретировано игрой.

[1] Katie Salen и Eric Zimmerman, *Rules of Play: Game Design Fundamentals* (Cambridge, MA: MIT Press, 2003), 34.

- **Интегрированность:** действие считается интегрированным, если игрок может сказать, что от него зависит исход игры. Например, нажатие кнопки вызова лифта является интегрированным действием, потому что известно, что в результате этого действия лифт остановится на вашем этаже. В игре *Super Mario Bros.* решение о схватке с отдельным врагом или уклонении от встречи с ним в общем случае не очень значимо, потому что это конкретное действие не интегрировано в исход игры. В *Super Mario Bros.* не ведется подсчет побежденных врагов; в ней требуется лишь дойти до конца уровня за ограниченное время и не погибнуть. Однако в серии игр *Kirby* студии HAL Laboratories главный персонаж Кирби приобретает дополнительные возможности при уничтожении врагов, поэтому решение об уничтожении каждого конкретного врага интегрировано, что делает его более значимым.

Если действия игрока в игре не имеют смысла, он быстро теряет интерес. Идея осмысленной игры, выдвинутая Сален и Циммерманом, напоминает дизайнерам о необходимости постоянно думать о мышлении игрока и прозрачности или непрозрачности взаимодействий с играми с его точки зрения.

Стратегия

Когда игра гарантирует осмысленность действий, игроки начинают разрабатывать выигрышные стратегии. *Стратегия* — это рассчитанный набор действий, помогающих игроку добиться своей цели. Однако целью игрока может быть все что угодно, и необязательно это будет победа в игре. Например, играя в игру с ребенком или любым другим, более слабым соперником, целью может быть доставить другому человеку удовольствие от игры и научить его чему-то, иногда ценой своей победы.

Оптимальная стратегия

Когда игра очень проста и имеет мало допустимых действий, игрок может разработать *оптимальную стратегию*. Если два игрока играют рационально с целью победить, оптимальной является возможная стратегия с наибольшей вероятностью выигрыша.

Большинство игр слишком сложны, чтобы выработать по-настоящему оптимальную стратегию, но некоторые, такие как крестики-нолики, достаточно просты для этого. Фактически игра крестики-нолики настолько проста, что обучить играть в нее можно даже курицу, и почти каждый раз она будет выигрывать или сводить ее вничью[1].

Часто под оптимальной стратегией ошибочно понимают все, что увеличивает вероятность победы игрока. Например, в настольной игре *Up the River* Манфреда Людвига игроки пытаются переправить три лодки вверх по реке в док, находящийся

[1] Kia Gregory, «Chinatown Fair Is Back, Without Chickens Playing Tick-Tack-Toe», *New York Times*, June 10, 2012.

в верхней части игровой доски. Прибытие в док дает игроку 12 очков за первую лодку, 11 очков — за вторую, и так далее до 1 очка за двенадцатую. В каждом раунде (то есть когда все игроки сделают ход) река сдвигается на одну клетку вниз, и все лодки, оказавшиеся за краем игровой доски (упавшие в водопад), теряются. Выполняя ход, игрок бросает шестигранный кубик и выбирает, какую лодку двинуть. Так как в среднем бросок кубика дает 3,5 очка, и игрок должен выбирать, какую из трех лодок двинуть, каждая лодка будет перемещаться вверх в среднем на 3,5 клетки каждые три хода. Но за эти же три хода доска сместится назад на 3 клетки, то есть каждая лодка будет двигаться вверх в среднем на 0,5 клетки за три хода (или 0,1666 (или 1/6) клетки за каждый ход)[1].

Оптимальная стратегия в этой игре состоит в том, чтобы бросить одну лодку и просто позволить ей упасть в водопад. В этом случае каждую из оставшихся лодок можно перемещать вверх со средней скоростью 3,5 клетки каждые два хода вместо трех. Учитывая обратное движение доски на 2 клетки за эти же два хода, каждая из оставшихся лодок будет продвигаться вверх на 1,5 клетки за два хода (или 0,75 клетки за один ход), что намного лучше, чем 0,1666, если игрок будет стараться сохранить все лодки. Игрок, использующий эту стратегию, имеет больше шансов добраться до дока, заняв первое и второе места, и получить 23 очка (12 + 11). Когда в игре участвуют два игрока, эта стратегия не работает, потому что второй игрок может переправить все три свои лодки и получить 10, 9 и 8, всего 27 очков, но в игре с тремя или четырьмя участниками описанная стратегия является самой близкой к оптимальной для *Up the River*. Однако из-за свободы выбора других игроков, случайного характера выпадения очков на кубике и наличия дополнительных факторов данная стратегия не гарантирует победу; она лишь делает выигрыш более вероятным.

Разработка стратегии

Как дизайнер, вы можете увеличить важность стратегии в своей игре. Пока просто запомните, что, давая игроку несколько возможных способов победить, вы заставите его принимать более сложные стратегические решения. Кроме того, если некоторые из целей сделать противоречащими друг другу, а другие взаимодополняющими (то есть когда требования к двум целям одинаковы), можно заставить игроков выбирать определенные роли по ходу игры. Когда игрок видит, что начинает достигать одной из целей, он будет выбирать дополняющие ее цели, а это заставит его принимать тактические решения, соответствующие той роли, для которой эти цели были предусмотрены. Если цели заставляют игрока выполнять в игре определенные действия, это может изменить его внутриигровые отношения с другими игроками.

Примером такого подхода может служить игра *Settlers of Catan*, созданная Клаусом Тойбером. В ней игроки приобретают ресурсы, торгуя или бросая кубик, при этом некоторые из пяти ресурсов выгоднее приобретать в начале игры, а некоторые — в конце. К трем ресурсам, наименее выгодным для приобретения в начале, относятся

[1] Ради простоты в этом примере я опустил другие правила.

шерсть, зерно и руда; однако все вместе их можно продать за карточку развития. Наиболее распространенной карточкой развития является карточка солдата, которая позволяет поставить фишку грабителя в любое место и перемещать его для захвата ресурсов у других игроков. Как результат, игрок, обладающий избытком руды, зерна и шерсти в начале игры, обычно стремится покупать карточки развития. Обладая большим количеством карточек солдат, игрок может заработать больше очков за победы. Эта комбинация ресурсов и стремление заработать больше очков на победах может подталкивать игрока чаще грабить других игроков и исполнять роль агрессора.

Внутренние правила

Внутренние правила устанавливаются самими игроками путем изменения фиксированных правил. Как было показано на примере игры *Bartok*, даже простое изменение в правилах может оказывать существенное влияние на игру. Например, большинство играющих в *Monopoly* устанавливают внутренние правила, которые отменяют аукцион на приобретение недвижимости (который обычно проводится, когда фишка игрока оказывается на поле с ничейной недвижимостью, и игрок принимает решение не приобретать ее) и добавляют штраф за бесплатное место на парковке, если фишка игрока попадет на него. Правило, отменяющее аукцион, устраняет почти все потенциальные стратегии, которые могли бы проводиться в начале игры (и превращает ее в чрезвычайно медленную систему случайного приобретения недвижимости), а второе исключает определенный детерминизм из игры (и тем самым может приносить пользу всем игрокам — и лидерам, и аутсайдерам). Несмотря на то что первое правило в этом примере немного ухудшает игру, большинство внутренних правил все же направлено на то, чтобы сделать игру интереснее.[1] Во всех случаях внутренние правила являются примером, как игроки начинают приобретать власть над игрой, делая ее чуть больше своей собственностью и чуть меньше — дизайнера. Самое замечательное во внутренних правилах, что для многих они являются первой попыткой поэкспериментировать с проектированием игр.

Намерения игрока: типы по Бартлу, читеры и зануды

Единственное, что мы почти или вообще не контролируем, — это намерения наших игроков. Большинство игроков будет стремиться играть рационально, чтобы победить, но вам также придется бороться с читерами и занудами. Даже среди законопослушных игроков можно выделить четыре типа, как их определил Ричард Бартл, один из дизайнеров первого многопользовательского мира (Multi-User Dungeon, MUD — текстовой онлайн-игры, предка современных массовых многопользовательских ролевых игр).

---

[1] Если вы иногда играете в игру *Lunch Money* компании Atlas Games, попробуйте дать игрокам возможность атаковать других игроков, исцеляться и сбрасывать любые ненужные им карты (вместо обязательного выбора одной из трех). Это сделает игру более увлекательной!

Четыре типа игроков, которые он определил, существуют со времен его первого многопользовательского мира и встречаются во всех современных многопользовательских онлайн-играх. В его статье «Hearts, Clubs, Diamonds, Spades: Players Who Suit MUDs»[1], опубликованной в 1996-м, содержится потрясающая информация о том, как игроки этих типов взаимодействуют друг с другом и с игрой и как правильно развивать сообщество игроков.

Ниже перечислены четыре психотипа по Бартлу (которым дополнительно присвоены карточные масти):

- **Карьерист (♦ Бубны):** старается набрать большее количество очков. Стремится доминировать над игрой.
- **Исследователь (♠ Пики):** стремится найти все скрытые места в игре. Пытается понять игру.
- **Социофил (♥ Червы):** любит играть с друзьями. Пытается понять других игроков.
- **Киллер (♣ Трефы):** любит провоцировать других игроков. Стремится доминировать над другими игроками.

Эти четыре типа можно представить в виде диаграммы, как показано на рис. 5.2 (взятой из статьи Бартла).

Рис. 5.2. Четыре типа игроков по классификации Ричарда Бартла, играющих в многопользовательские онлайн-игры

[1] Richard Bartle, «Hearts, Clubs, Diamonds, Spades: Players Who Suit Muds», http: // www.mud.co.uk/richard/hcds.htm. Была доступна в марте 2018. (Перевод на русский язык можно найти по адресу http: // dailytelefrag.ru/articles/read.php?id=44593. — *Примеч. пер.*)

Разумеется, существуют также другие теории мотивации игроков и их типов[1], но имя Бартла наиболее широко известно и признано в игровой индустрии.

Два других типа игроков, с которыми вы можете столкнуться, — это читеры и зануды:

- **Читеры:** их интересует победа и не интересует целостность игры. Готовы игнорировать или нарушать правила ради победы.
- **Зануды:** их не интересует ни победа, ни игра. Зануды часто ломают игру, просто чтобы испортить настроение другим игрокам.

Никто не хотел бы видеть игроков этих двух типов в своей игре, но вы должны понимать их мотивацию. Например, если читер чувствует, что у него есть шанс выиграть законным путем, он может не испытывать потребности пускаться в обман. С занудами ситуация намного сложнее, потому что их не интересует ни игра, ни победа, но вам редко придется сталкиваться с занудами в однопользовательских цифровых играх, потому что для них бессмысленно играть в игру, которая их не интересует. Однако даже самые лучшие игроки могут превращаться в зануд, встречая ужасную игровую механику... часто непосредственно перед тем, как они решили покинуть игру.

Исход

Исход — это результат игры. У всех игр есть исход. Многие традиционные игры являются играми с *нулевой суммой*, в том смысле, что один игрок выигрывает, а остальные проигрывают. Однако это не единственный вид исхода, который может иметь игра. Фактически каждый отдельный момент в игре имеет свой исход. Большинство игр имеет несколько разных уровней исходов:

- **Немедленный исход:** каждое отдельное действие имеет исход. Когда игрок стреляет в врага, исходом этого действия является или промах, или результативное попадание. Когда игрок покупает недвижимость в *Monopoly*, исходом является уменьшение количества денег у него, но теперь он получает потенциальную возможность заработать больше денег.
- **Исход квеста:** многие игры предлагают игроку выполнить задание или пройти квест и предлагают некоторое вознаграждение за успешное завершение. Задания и квесты часто имеют сюжеты, построенные вокруг них (например, в *Spider-Man 2* [Treyarch, 2004] маленькая девочка выпускает из рук шарик, и Человек-паук должен достать его), и исход квеста отмечает окончание короткого сюжета, окружающего его.

---

[1] См. статью Ника Йи (Nick Yee) «Motivations of Play in MMORPGs: Results from a Factor Analytic Approach», http: // www.nickyee.com/daedalus/motivations.pdf. Была доступна в марте 2018 года. Еще один замечательный ресурс: статья Скотта Ригби (Scott Rigby) и Ричарда Райана (Richard Ryan) «The Player Experience of Need Satisfaction (PENS)», http: // immersyve.com/white-paper-the-player-experience-of-need-satisfaction-pens-2007/. Была доступна в марте 2018 года.

- **Кумулятивный исход:** имеет место, когда игрок постепенно движется к цели и, наконец, достигает ее. Один из типичных примеров — повышение статуса с увеличением количества очков опыта. Каждое достижение игрока в игре оценивается некоторым количеством очков опыта, и когда общее их число превышает некоторый пороговый предел, игровой статус персонажа увеличивается и он получает прирост в статистике или новые возможности. Главное отличие кумулятивного исхода от исхода квеста состоит в том, что кумулятивный исход обычно не окружается отдельным сюжетом, и игрок часто достигает кумулятивного исхода пассивно, активно решая какие-то другие задачи (например, *игрок 4-го издания Dungeons & Dragons* активно участвует в сеансе игры и затем, подсчитывая заработанные очки опыта в конце вечера, может заметить, что заработал больше 10 000 очков и достиг уровня 7)[1].
- **Финальный исход:** большинство игр имеет исход, завершающий игру: игрок выигрывает шахматную партию (а его соперник проигрывает), игрок завершает игру *Final Fantasy VII* и спасает мир от Сефирота, и т. д. В некоторых играх финальный исход не завершает игру (например, когда игрок завершает главный квест в игре *Skyrim*, он может продолжить играть и попробовать пройти другие квесты)[2]. Самое интересное, что смерть персонажа очень редко является финальным исходом игр.

В немногих играх, где смерть — это финальный исход (например, в игре *Rogue* [A. I. Design, 1980], где единственный проигрыш приводит к потере всех достижений в игре), отельные игровые сеансы обычно делаются очень короткими, чтобы игрок не испытывал чувства большой потери в случае смерти персонажа. Однако в большинстве игр смерть — это лишь временная неудача и контрольная точка, гарантирующая, что игрок потеряет не больше пяти минут на повторную попытку.

Динамическая эстетика

Так же как динамическая механика, динамическая *эстетика* возникает в процессе игры. Она делится на две основные категории:

- **Процедурная эстетика:** эстетика, программно генерируемая цифровым кодом игры (или с применением механики в настольной игре). Включает процедурное звуковое сопровождение и графику, возникающие непосредственно из фиксированной эстетики и технологии.
- **Эстетика окружения:** эстетика окружения, в котором протекает игровой процесс, в значительной степени неподконтрольна разработчикам игры.

---

[1] Rob Heinsoo, Andy Collins и James Wyatt, *Dungeons & Dragons Player's Handbook: Arcane, Divine, and Martial Heroes: Roleplaying Game Core Rules* (Renton, WA: Wizards of the Coast, 2008).

[2] Первоначально игра *Fallout 3* завершалась, когда игрок достигал финального исхода главного повествования, но затем был выпущен загружаемый контент (downloadable content, DLC), позволивший игрокам продолжать игру после этой точки.

Процедурная эстетика

Процедурная эстетика, которая, по нашим представлениям, свойственна в основном цифровым играм, создается программно, как результат объединения технологии и фиксированной эстетики[1]. Она называется *процедурной*, потому что порождается процедурами (или функциями) в программном коде. Каскадный водопад объектов, который будет создан в главе 18 «Привет, мир: ваша первая программа», можно рассматривать как процедурную эстетику, потому что этот интересный визуальный эффект создается программным кодом, написанным на C#. В профессиональных играх двумя наиболее распространенными формами процедурной эстетики являются звуковое сопровождение и визуальные эффекты.

Процедурное звуковое сопровождение

Процедурное звуковое сопровождение очень распространено в современных видеоиграх и в настоящее время создается тремя разными способами:

- **Горизонтальное переупорядочивание (Horizontal Re-Sequencing, HRS):** перестраивает порядок следования предварительно записанных звуков или музыкальных фрагментов в зависимости от того, какое эмоциональное воздействие на игрока хотели оказать дизайнеры в данный момент в игре. Примером может служить механизм iMUSE (Interactive MUsic Streaming Engine — интерактивный механизм потоковой музыки) студии LucasArts, использованный в серии игр *X-Wing*, а также во многих приключенческих играх студии LucasArts. В *X-Wing* используются предварительно записанные фрагменты музыки, написанной Джоном Уильямом, к фильмам *Star Wars*. С помощью iMUSE дизайнеры могут реализовать проигрывание безмятежной музыки, когда игрок просто летит в космическом пространстве, зловещей — когда враг собирается его атаковать, победной — когда игрок уничтожает вражеский корабль или достигает цели, и т. д. Музыкальные фрагменты могут быть длинными и воспроизводиться в цикле, чтобы поддержать неизменность настроения, или очень короткими (продолжительностью один-два такта), чтобы смягчить переход от одного настроения к другому. В настоящее время это наиболее распространенная технология процедурного звукового сопровождения и известна по крайней мере еще со времен *Super Mario Bros.* (Nintendo, 1985), где проигрывался короткий музыкальный пассаж и затем начиналось воспроизведение ускоренной версии фоновой музыки, когда у игрока оставалось менее 99 секунд, чтобы дойти до конца уровня.
- **Вертикальное комбинирование (Vertical Re-Orchestration, VRO):** заключается в комбинировании разных треков одной песни, которые можно включать

[1] В качестве примера процедурной эстетики в настольных играх можно привести карты, создаваемые последовательной выкладкой плиток в *Carcassonne* (Klaus-Jürgen Wrede, 2000), но цифровая процедурная эстетика имеет намного более широкое распространение.

и выключать по отдельности. Широко используется в ритмических играх, таких как *PaRappa the Rapper* и *Frequency*. В *PaRappa* четыре разных музыкальных трека представляют четыре уровня успеха игрока. Успех игрока оценивается по нескольким параметрам, и если степень успеха уменьшается или увеличивается, фоновая музыка переключается на хуже или лучше звучащий трек. В *Frequency* и в ее продолжении — *Amplitude* игрок управляет космическим кораблем, движущимся в туннеле, стены которого представляют разные треки студийных записей песен. Когда игрок успешно преодолевает ритмическую игру вдоль определенной стены, включается соответствующий трек[1]. Подобное вертикальное комбинирование встречается в ритмических играх почти повсеместно — заметными исключениями являются фантастическая японская ритм-игра *Osu Tatake Ouendan!* и ее западная последовательница *Elite Beat Agents*, а также все шире стало использоваться в других играх для организации музыкальной обратной связи, сообщающей игроку о состоянии здоровья, скорости движения и т. д.

- **Процедурная композиция (Procedural Composition, PCO):** редчайшая форма процедурного звукового сопровождения, потому что для ее реализации требуется больше всего времени и навыков. В процедурной композиции вместо переупорядочивания различных готовых музыкальных фрагментов или включения/выключения отдельных треков компьютерная программа действительно составляет музыку из отдельных нот, опираясь на запрограммированные правила, заданный темп и т. д. Одним из первых коммерческих экспериментов в этой области стала игра *C.P.U. Bach* (1994) Сида Мейера и Джеффа Бригса для игровой приставки 3DO. В *C.P.U. Bach* слушатель/игрок мог выбирать разные инструменты и параметры, а игра создавала музыкальную композицию, напоминающую музыку Баха, следуя процедурным правилам.

Другим фантастическим примером процедурной композиции может служить музыка, созданная композитором и дизайнером игр Винцентом Диамантэ для игры *Flower* студии thatgamecompany (2009). Для этой игры Диамантэ написал готовые фрагменты музыки и правила процедурной композиции. Во время игры обычно проигрывается фоновая музыка (отдельные фрагменты которой переупорядочиваются с использованием технологии горизонтального переупорядочивания, в зависимости от ситуации), когда игрок летает над цветами в поле и открывает их. Каждый открываемый цветок воспроизводит одну ноту, а механизм процедурной композиции Диамантэ выбирает ноту для этого цветка, гармонично вписывающуюся в готовый музыкальный фрагмент, и создает мелодию в комплексе с другими цветочными нотами. Когда бы игрок ни пролетал над цветком, система выбирает ноту, вписывающуюся в текущее музыкальное оформление, а пролет над несколькими цветками генерирует приятные мелодии.

[1] Игра *Amplitude* также поддерживает режим, в котором игроки могут выбирать, какие треки и с какого места в песне включать, — и таким способом создавать собственные ремиксы из треков, имеющихся в игре.

Процедурные визуальные эффекты

Процедурные визуальные эффекты создаются в процессе работы игрового программного кода. Вероятно, вы уже знакомы с некоторыми формами процедурных визуальных эффектов.

- **Системы частиц:** системы частиц, как наиболее распространенная форма процедурных визуальных эффектов, присутствуют во многих играх, созданных за последнее десятилетие. Облако пыли, поднимающееся, когда Марио приземляется в *Super Mario Galaxy*, эффекты пламени в *Uncharted 3* и искры, появляющиеся, когда автомобили сталкиваются друг с другом в *Burnout*, все это разные версии эффектов частиц. В Unity имеется очень быстрый и надежный механизм эффектов частиц (рис. 5.3).

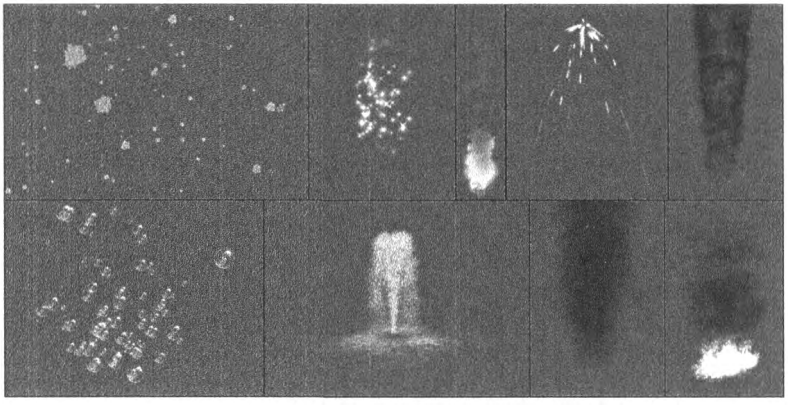


Рис. 5.3. Разнообразные эффекты частиц, созданные в Unity

- **Процедурная анимация:** охватывает широкий спектр визуальных эффектов, от стадного поведения групп созданий до блестящего движка процедурной анимации персонажей в игре *Spore* Уилла Райта, создающего эффект ходьбы, бега, нападения и других движений для любых созданий, придуманных игроком. При обычной анимации персонажи всегда точно повторяют движения, как их нарисовал аниматор. В процедурной анимации персонажи следуют процедурным правилам, которые порождают сложные движения и поведение. В главе 27 «Объектно-ориентированное мышление» вы получите некоторый опыт реализации стадного поведения, известного как «рой» (рис. 5.4).

- **Процедурное окружение:** наиболее очевидным примером процедурного окружения в играх является мир *Minecraft* студии Mojang (2011). Каждый раз, когда игрок начинает новую игру *Minecraft*, создается целый мир (площадью в миллионы квадратных километров) из единственного числа (известного как *начальное случайное число*). Так как цифровые генераторы случайных чисел в действительности не генерируют по-настоящему случайных чисел, это означает, что любой использовавший то же начальное случайное число получит тот же самый мир.

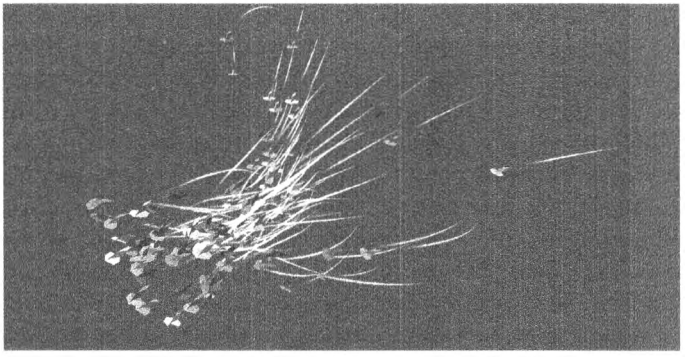


Рис. 5.4. Рой как пример процедурной анимации из главы 27 «Объектно-ориентированное мышление»

Эстетика окружения

Другой важной разновидностью динамической эстетики является та, что определяется действительным физическим окружением, в котором протекает игровой процесс. Несмотря на то что это окружение никак не управляется дизайнерами, понимание законов ее возникновения и максимальное их использование все еще является областью ответственности дизайнера.

Визуальное окружение игры

Игроки запускают игры на разных устройствах и с разными настройками, поэтому, как дизайнер, вы должны знать о проблемах, которые могут возникнуть. В частности, вы должны учесть два обстоятельства:

- **Окружающее освещение:** большинство разработчиков игр стремятся работать в обстановке с контролируемым уровнем освещения, чтобы изображения на экране выглядели максимально отчетливо. Игроки не всегда взаимодействуют с игрой в окружениях с идеальным освещением.

  Если игрок с компьютером находится на открытом воздухе, использует для игры проектор или находится в любом другом месте, где не имеет возможности управлять освещенностью, ему будет трудно видеть сцены в игре с низким уровнем яркости (например, сцены в темных пещерах). Поэтому вы должны гарантировать высокую контрастность между светлыми и темными визуальными элементами или дать игроку возможность регулировать уровень насыщенности или яркости изображения. Это особенно важно для мобильных игр, потому что нередко процесс игры на мобильном устройстве может протекать при прямом солнечном освещении.

- **Разрешение экрана:** если вы разрабатываете игру для устройства с конкретным размером экрана, например для определенной модели iPad или портативной игровой приставки (такой, как Nintendo DS), это не вызовет никаких проблем.

Но если игра разрабатывается для компьютера или обычной игровой приставки, у вас практически не будет никакой возможности влиять на разрешение или качество монитора, используемого игроком, особенно если это игра для приставки. Нельзя с полной уверенностью предполагать, что у игрока имеется монитор с разрешением 1080p или хотя бы 720p. Все современные консоли, предшествовавшие PS4 и Xbox One, все еще могут выводить стандартный ТВ-сигнал, существующий с момента появления телевидения в 1950-х. Если вы предполагаете наличие таких игроков в своей аудитории, вы должны использовать более крупные шрифты, чтобы текст в игре был разборчивым. Даже ААА-игры[1], такие как *Mass Effect* и *Assassin's Creed*, не соответствовали в прошлом этим требованиям, из-за чего в них невозможно было прочитать важный текст на телевизорах, произведенных более чем за 10 лет до выпуска этих игр. Невозможно заранее знать, захочет ли кто-то сыграть в вашу игру на старом оборудовании, но вы можете позволить это в своей игре и изменить размер шрифта, чтобы помочь своим игрокам.

Акустическое окружение игры

По аналогии с визуальным окружением, вы едва ли сможете контролировать акустическое окружение игры. Даже при том, что этот аспект особенно важен для мобильных игр, его необходимо также учитывать при создании любых игр. Кроме всего прочего, вы должны принять во внимание следующее:

- **Окружающий шум:** вокруг игрока, играющего в вашу игру, может происходить множество разных событий, поэтому вы должны обеспечить возможность продолжения игры, даже если игрок не услышит или пропустит какие-то звуки. Вы также должны гарантировать, что игра не создает избыточного шумового фона, из-за которого игрок может пропустить важную информацию. Вообще, самыми громкими звуками в вашей игре должны быть важные диалоги и устные инструкции, а все остальное может воспроизводиться с пониженной громкостью. Также избегайте едва уловимых, тихих аудиосигналов, обозначающих что-либо важное.
- **Регулировка громкости игроком:** игрок может приглушить громкость игры. Это особенно характерно для мобильных игр — никогда нельзя быть уверенным, что игрок будет слышать звуки, воспроизводимые игрой. Независимо от вида игры обеспечьте альтернативу звуку. Если в игре есть важные диалоги, дайте игроку возможность включить титры. Если звуки обозначают какие-то важные события, добавьте также визуальное оповещение.

Игроки

Еще одним важным аспектом окружения, который необходимо учитывать, является сам игрок. Не все игроки имеют равный уровень развития всех пяти органов чувств. Игрок с ограничениями слуха должен иметь возможность играть в вашу игру без

[1] Неформальный термин, обозначающий класс высокобюджетных компьютерных игр (https://ru.wikipedia.org/wiki/ААА_(компьютерные_игры)). — *Примеч. пер.*

лишних неудобств, особенно если следовать советам в нескольких предыдущих абзацах. Однако есть еще два замечания, которые пропускают многие дизайнеры:

- **Дальтонизм:** в США до 8 % мужчин и 0,5 % женщин, выходцев из Северной Европы, страдают некоторой формой дальтонизма[1]. Известно несколько форм недостатка восприятия цвета, из которых чаще встречается форма неразличения оттенков красного и зеленого цвета. Из-за такой распространенности дальтонизма вы должны найти друга с таким недостатком, которого сможете попросить попробовать сыграть в игру и убедиться, что с цветами, вызывающими путаницу, не передается никакой важной информации. Другой интересный способ проверить игру — загрузить приложение для смартфона, имитирующее разные формы дальтонизма за счет изменения изображения, получаемого с помощью камеры[2].
- **Эпилепсия и мигрень:** быстро мигающие огни могут вызывать мигренозные боли и эпилептические припадки, и особенно уязвимы в этом отношении дети, страдающие эпилепсией. В 1997 году в Японии эпизод с мерцающими изображениями в одной из сцен в телевизионном сериале *Pokémon* стал причиной припадков у сотен телезрителей одновременно[3]. Почти все игры сопровождаются предупреждением, что могут стать причиной эпилептических припадков, но в настоящее время это случается крайне редко, потому что разработчики приняли на себя ответственность за влияние их игр на игроков и почти повсеместно устранили мерцающие сцены.

Динамический сюжет

Есть несколько способов взглянуть на сюжет с динамической точки зрения. Ярким образцом может служить опыт игроков и их мастеров игры в настольной ролевой игре. Несмотря на многочисленные эксперименты по созданию по-настоящему интерактивных цифровых повествований, по прошествии более чем 30 лет так и не был достигнут уровень интерактивности хорошо известной игры *Dungeons & Dragons* (*D&D*). Причина, почему в *D&D* удалось добиться такого фантастически динамического повествования, состоит в том, что Мастер Подземелий (Dungeon Master, DM: мастер игры в *D&D*) постоянно оценивает желания, страхи и развитие навыков персонажей игроков и корректирует повествование с учетом этих оценок. Как отмечалось выше в этой книге, если игроки сталкиваются со слабым врагом, в сражении с которым (в силу случайного выпадения очков на кубике в пользу врага) испытывают сложности, мастер игры может убрать врага в последний момент

[1] Мужчины страдают дальтонизмом чаще, чем женщины. https://nei.nih.gov/health/color_blindness/facts_about. Статья была доступна в марте 2018 года.

[2] В качестве примеров таких приложений можно назвать приложение для iOS и Android *Chromatic Vision Simulator*, созданное Казунори Асадом, и приложение для iOS *Color DeBlind*, созданное electron software.

[3] Sheryl WuDunn, «TV Cartoon's Flashes Send 700 Japanese Into Seizures», *New York Times*, December 18, 1997.

и вернуть его позже, чтобы игрок довел схватку до конца. Мастер игры, являясь человеком, может скорректировать игру и повествование для каждого игрока индивидуально, что очень сложно воспроизвести на компьютере.

Зарождение интерактивного сюжета

В 1997 году Джанет Мюррей (Janet Murray), профессор Технологического института Джорджии, опубликовала книгу «Hamlet on the Holodeck»[1], где рассматривала раннюю историю интерактивного повествования в сопоставлении с ранней историей других форм средств информации. В своей книге Мюррей исследовала ранние периоды развития других средств информации между их созданием и обретением зрелой формы. Например, в ранний период развития кино режиссеры старались снимать 10-минутные версии «Гамлета» и «Короля Лира» (потому что на одну катушку 16-миллиметровой пленки умещался только 10-минутный фильм), а на начальном этапе развития телевидения телепередачи в основном были лишь телевизионными версиями популярных радиопередач. Путем сопоставления множества примеров из разных средств информации Мюррей пришла к выводу, что цифровой интерактивный жанр успешно развивается, а сейчас находится на начальном этапе. В своем исследовании она охватила ранние текстовые приключенческие игры компании Infocom, такие как *Zork* и *Planetfall*, и указала на два привлекательных аспекта, которые делают интерактивный жанр уникальным.

Интерактивное действо творится в голове игрока

В отличие от почти любых других форм повествования, интерактивное действо творится непосредственно в голове игрока. В начале игры *Zork* компании Infocom происходит следующее[2]. (Строки, начинающиеся с правой угловой скобки (например, `> открыть люк`) — это команды, вводимые игроком.)

```
...Коврик сдвигается, и появляется запертая крышка люка, покрытая слоем пыли.
> открыть люк
Крышка со скрипом открывается, и за ней появляется шаткая лестница, уходящая вниз,
в темноту.
> вниз
Здесь чертовски темно. Жуткое место, того и гляди появится чудовище.
> зажечь лампу
Лампа зажглась.
Вы в темном сыром подвале с узким проходом, ведущим на восток и постепенно
поворачивающим на юг. На западе - нижний край наклонной металлической плиты
непонятного назначения.
Люк с грохотом падает, и слышится звук задвигаемого засова.
```

[1] Janet Horowitz Murray, *Hamlet on the Holodeck* (New York: Free Press, 1997).

[2] Игра *Zork* была создана в Массачусетском технологическом институте в 1977–1979 годах Тимом Андерсоном, Марком Бланком, Брюсом Дэниелсом и Дэйвом Деблингом. В 1979 году они создали компанию Infocom и выпустили *Zork* как коммерческий продукт.

Главное здесь то, что *вы* слышите звук задвигаемого засова. Теперь *вы* в ловушке. Интерактивный жанр — единственная повествовательная среда, в которой игрок/читатель является персонажем, выполняющим какие-то действия и испытывающим на себе последствия этих действий в повествовании.

Отношения развиваются через общие переживания

Еще одной восхитительной чертой интерактивного жанра является возможность для игрока развивать отношения с другими персонажами через общие переживания. Мюррей приводит *Planetfall*[1], еще одну текстовую приключенческую игру компании Infocom, как фантастический пример этой черты. После уничтожения космического корабля, на котором он работал уборщиком в первой части *Planetfall,* игрок действует почти в одиночку. Наконец, он находит машину по производству боевых роботов, но когда запускает ее, в машине происходит сбой и она создает робота по имени Флойд, похожего на ребенка и почти бесполезного. Флойд следует за игроком оставшуюся часть игры и в какой-то мере скрашивает его одиночество. Много позже игрок должен забрать устройство из биолаборатории, но в ней свирепствуют радиационное излучение и злобные пришельцы. Флойд просто говорит себе: «Иди!» — и входит в лабораторию за прибором. Вскоре он возвращается, но из него вытекает масло и он едва двигается. Он умирает на руках игрока, когда тот поет ему «Балладу о несчастном шахтере». Многие игроки писали дизайнеру Стивену Мерецки, что плакали, когда Флойд умирал, и Мюррей приводит это как один из главных примеров ощутимой эмоциональной связи между игроком и игровым персонажем.

Сюжетная непредсказуемость

По-настоящему динамическим повествование становится, когда игроки и игровая механика вместе вносят вклад в развитие сюжета. Несколько лет назад я с друзьями играл в *Dungeons & Dragons* версии 3.5. Мастер игры завел нас в трудное положение. Мы только что отобрали артефакт у сил зла в другом измерении и удирали от большого балрога[2] на нашем ковре-самолете вниз по узкой пещере в направлении портала, ведущего в наше измерение. Он быстро нагонял нас, и наше оружие почти не наносило ему ущерба. Однако я вспомнил о бывшем у меня и редко используемом магическом жезле. Раз в неделю я мог с его помощью создать «большой шатер из шелка, имеющий 60 футов в поперечнике, с мебелью и едой для празднования, в котором могли принять участие 100 человек»[3]. Часто мы использовали это свой-

[1] Игра *Planetfall* была создана Стивом Мерецки и выпущена компанией Infocom в 1983 году.

[2] Балрог — гигантский крылатый демон, способный окутывать себя огнем и дымом, с которым сразился Гэндальф в сцене «Тебе не пройти» в книге «Властелин колец: Братство кольца» Дж. Р. Р. Толкина.

[3] Описание жезла можно найти в руководстве *Dungeons & Dragons 3.5e System Reference Document*: http://www.d20srd.org/srd/magicItems/rods.htm#splendor. Было доступно в марте 2018 года.

ство жезла, чтобы отпраздновать успешное выполнение очередного задания, но на этот раз я вызвал шатер прямо позади нас в туннеле. Поскольку туннель имел всего 30 футов в ширину, балрог врезался в шатер и запутался в нем, что позволило нам благополучно смыться.

Подобные неожиданные истории возникают из сочетания ситуаций, созданных мастерами игры, правил и творческого мышления отдельных игроков. Я сталкивался со многими похожими историями в ролевых играх, в которых мне доводилось участвовать (и в роли игрока, и в роли мастера игры), и вы можете разными способами способствовать такому совместному развитию повествования. За дополнительной информацией о ролевых играх и о том, как провести хорошую кампанию, обращайтесь к разделу «Ролевые игры» в приложении Б «Полезные идеи».

Динамическая технология

Технологии цифровых и физических игр почти не затрагиваются в этой главе, как и в предыдущей, потому что им посвящены другие большие разделы книги. Основная идея, которую достаточно знать на данном этапе: ваш игровой код (фиксированная технология) станет действующей системой с началом игрового процесса. Как это характерно для всех динамических систем, появится непредсказуемость и начнут происходить неожиданные события — и приятные, и ужасные. Динамическая технология охватывает все виды поведения вашего кода во время выполнения и способы воздействия на игрока. Это может быть все что угодно, от имитации законов физики до искусственного интеллекта и всего остального, реализованного в вашем коде.

Информацию о динамическом поведении технологий физических игр, таких как кубики, рулетки, карты и другие источники случайности, вы найдете в главе 11 «Математика и баланс игры». За информацией о технологиях цифровых игр обращайтесь к последним двум частям книги, а также к приложению Б «Полезные идеи».

Итоги

Динамическая механика, эстетика, сюжет и технология — все они вытекают из акта игры. Даже при том, что очень трудно предсказать, что появится в результате, дизайнеры обязаны провести пробные игры и понять границы непредсказуемости.

Следующая глава исследует культурный уровень многоуровневой тетрады, лежащий за рамками игрового процесса. На культурном уровне игроки получают больше власти над игрой, чем ее разработчики, и культурный уровень — единственный уровень тетрады, влияющий на членов общества, даже не игравших в игру.

6 Культурный уровень

Культурный уровень является последним в многоуровневой тетраде, и до него практически не дотягиваются руки дизайнеров. Тем не менее он является важным для целостного понимания дизайна и последствий разработки игры.

Эта глава исследует культурный уровень — область, где игрок и общество устанавливают над игрой свой контроль.

За рамками игры

Фиксированный и динамический уровни очевидны для всех дизайнеров игр, потому что оба являются неотъемлемой частью идеи интерактивного опыта. Однако культурный уровень менее очевиден. Культурный уровень находится на стыке игры и общества. Поклонники игры, объединенные общим интересом, образуют сообщество, и это сообщество переносит идеи и интеллектуальную собственность игры во внешний мир. Культурный уровень игры виден, с одной стороны, сообществу игроков, обладающих сокровенными знаниями об игре, а с другой — членам общества, которые вообще не знакомы с игрой и впервые сталкиваются с ней не через игру, а через атрибутику, созданную сообществом игроков (рис. 6.1).

Как отмечает Констанс Стейнкуэлер в своей статье «The Mangle of Play»[1], динамическая игра, в частности массовая многопользовательская игра, это «интерактивно устойчивые вальцы практики». Этими словами она отмечает, что, как обсуждалось в предыдущей главе, динамический уровень игры состоит не только из намерений разработчиков игры, но также из намерений игроков и общей ответственности игроков и разработчиков за переживания и управление ими. Продолжая эту идею, в культурном уровне игроки (и общество в целом) получают больше власти над игрой, чем ее разработчики. Именно на культурном уровне сообщество игроков фактически изменяет фиксированную игру с помощью модов (программного обеспечения, изменяющего фиксированные элементы игры), именно здесь сообщества игроков обретают власть над сюжетом игры, придумывая свои фанфики, и именно здесь игроки создают свою эстетику, связанную с игрой, в виде художественных и музыкальных произведений.

---

[1] Constance Steinkuehler. «The Mangle of Play». *Games and Culture 1*, no. 3 (2006): 199–213.

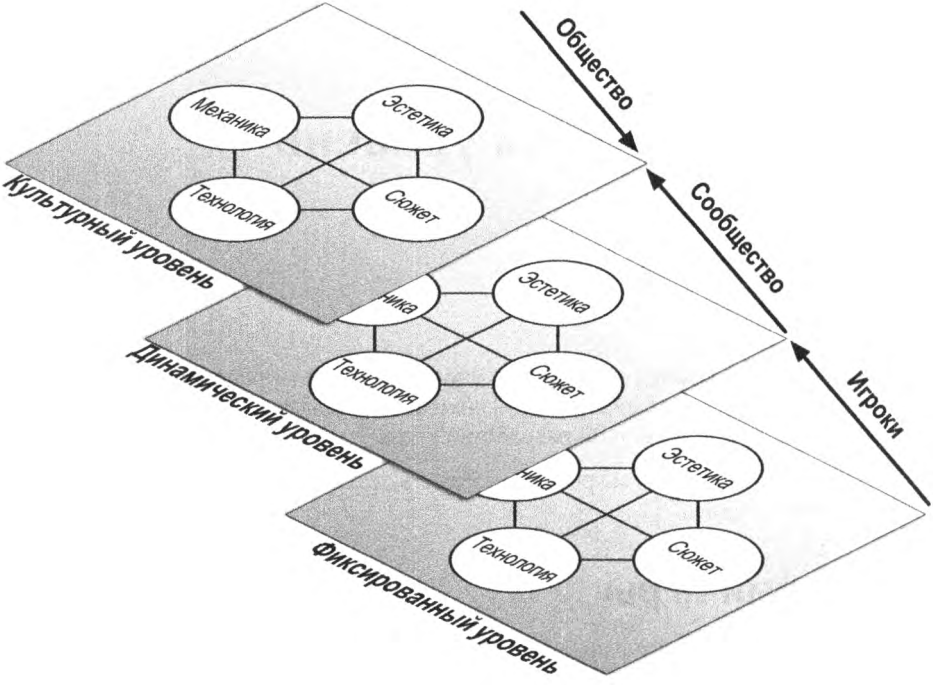


Рис. 6.1. Культурный уровень, созданный сообществом игроков и наблюдаемый обществом

В отличие от фиксированного уровня, где четыре элемента (механика, эстетика, сюжет и технология) четко делятся между разработчиками разных специальностей, элементы в культурном уровне имеют множество перекрытий и более размытые границы. Моды, создаваемые фанатами и занимающие самое видное место в культурном уровне, часто являются комбинациями всех четырех элементов, и ответственность за каждый элемент внутри мода часто делится между игроками и сообществами, создающими их[1].

В оставшейся части главы мы рассмотрим четыре элемента по отдельности, чтобы сохранить единообразие с предыдущими главами и помочь вам взглянуть на примеры сквозь призму конкретного элемента. Но имейте в виду, что примеры для одного элемента культурного уровня также можно было бы отнести к элементам другого.

[1] Я отнюдь не стремлюсь принизить разработчиков игровых модов, называя их в этой главе *игроками*. Я лишь хочу избежать путаницы между *разработчиками* (то есть создателями фиксированного контента) и *игроками* (теми, кто играет в игру и может заниматься разработкой модов). Многие потрясающие дизайнеры и разработчики начинали с создания модов, и в действительности это фантастически эффективный способ получить практические навыки.

Культурная механика

Культурная механика возникает, когда игроки берут механику игры в свои руки, иногда даже извлекая новый опыт из игры. Наиболее типичными примерами могут служить:

- **Игровые моды:** игроки переиначивают игру, внедряя в нее свою механику. Это наиболее распространено в играх для персональных компьютеров с Windows. Игроки модифицировали игру *Quake 2* компании Id и создали десятки, если не сотни, новых игр, использующих технологию *Quake 2*, но дополнивших механику новыми игровыми уровнями (часто меняющими эстетику и сюжет).

  Некоторые особенно удачные моды превратились в самостоятельные коммерческие продукты. Игра *Counter Strike* начиналась как мод для *Half-Life*, впоследствии приобретенный компанией Valve, разработавшей *Half-Life*[1]. Точно так же игра *Defense of the Ancients* (*DotA*) начиналась как мод для игры *Warcraft III* компании Blizzard и в конечном итоге способствовала популяризации жанра многопользовательских онлайн-битв в целом[2].

  Кроме того, многие компании выпускают редакторы для своих игр, поощряя игроков создавать свой игровой контент. Например, компания Bethesda Softworks выпустила Creation Kit™ для своей игры *The Elder Scrolls V: Skyrim*. Creation Kit позволяет игрокам создавать свои уровни, квесты, персонажей, управляемых компьютером, и т. д. Bethesda пошла этим путем раньше других, в том числе в игре *Fallout 3* и более ранних играх в серии *Elder Scrolls*.

- **Пользовательские уровни игры:** даже без изменения другой механики в некоторых играх предусмотрена возможность добавления пользовательских уровней. Фактически некоторые игры предполагают, что пользователи будут создавать свои уровни. Например, *Little Big Planet* компании Media Molecule и *Sound Shapes* компании Queasy Games включают простые инструменты редактирования уровней, с помощью которых, как предполагалось, игроки смогут создавать новые уровни. Обе игры также имели специальные системы, помогающие игрокам распространять свои уровни и оценивать уровни, созданные другими. Редакторы игр и комплекты инструментов для создания модов, подобные тем, что выпускались для *Skyrim* и *Fallout 3*, тоже включали редакторы уровней, а сообщество, занимающееся созданием уровней для шутера от первого лица *Unreal* компании Epic, считается одним из самых многочисленных и зрелых в современном мире компьютерных игр.

Основной чертой, отличающей культурную механику, например моды, от внутренних правил (относятся к динамической механике), является степень фактического

[1] http://www.ign.com/articles/2000/11/23/counter-strike-2 и https://ru.wikipedia.org/wiki/Counter-Strike. — *Примеч. пер.*

[2] Согласно http://en.wikipedia.org/wiki/Multiplayer_online_battle_arena и http://themittani.com/features/dota-2-history-lesson, в действительности игра *DotA* была ремейком мода *Warcraft III* для *Starcraft*, известного как *Aeon of Strife*.

изменения фиксированной механики игры. Если фиксированная механика почти не изменяется, но игроки преследуют свои собственные цели (например, выбирая «быстрые ходы» и завершая игру как можно быстрее или пытаясь играть в такие жестокие игры, как *Skyrim*, не убивая врагов), такое поведение все еще укладывается в определение динамической механики. И только когда игроки отбирают управление у дизайнеров, модифицируя фиксированные элементы игры, их поведение перемещается в культурный уровень.

Культурная эстетика

Культурная эстетика возникает, когда сообщество игроков создает свою эстетику, имеющую отношение к игре. Часто она имеет форму своих версий изображения персонажей, звукового сопровождения или других эстетических элементов игры, но иногда может принимать форму использования игрового движка сообществом для достижения своих эстетических целей:

- **Фан-арт:** многие художники черпают вдохновение в играх и игровых персонажах и создают свои произведения, изображающие этих персонажей.
- **Косплей:** подобно фан-арту, косплей (от англ. слова *cosplay*, составленного из двух слов *costume* и *play* — костюмированная игра) — это обычай фанатов игры (или комиксов, аниме, манги или фильмов) одеваться в стиле персонажей игры. Человек, участвующий в такой костюмированной игре, принимает на себя роль и индивидуальность игрового персонажа в реальном мире, как бы продолжая виртуальный мир игры. Косплей чаще встречается среди фанатов игр, аниме и комиксов.
- **Игровой процесс как искусство:** в своей книге «Game Design Theory» Кит Бургун предлагает рассматривать некоторых разработчиков игр наравне с создателями музыкальных инструментов: мастерами, изготавливающими инструменты, с помощью которых исполнители творят искусство. По его словам, искусством является не только создание игр, но и сам процесс игры, а элегантность, с какой способны играть некоторые высококлассные игроки, можно рассматривать как эстетику саму по себе. Игры с богатым набором движений или действий игрока могут вызвать такую художественную игру. В качестве примеров можно привести игры-единоборства, такие как *Street Fighter*, и творческие игры-симуляторы, такие как *Tony Hawk's Pro Skater*.

Культурный сюжет

Иногда сообщество игроков использует игру или игровой мир для изложения своих рассказов и создания своих сюжетов. В настольных ролевых играх, таких как *Dungeons & Dragons*, это неотъемлемая часть динамики игры. Однако есть также примеры игроков, которые делают это далеко за рамками стандартной или ожидаемой динамики игрового процесса:

- **Фанфики:** так же как в случае с кино, телевидением или любыми другими формами повествовательных средств информации, некоторые фанаты игр пишут собственные рассказы об игровых персонажах или игровом мире.
- **Сюжетные игровые моды:** некоторые игры, такие как *Skyrim* и *Neverwinter Nights*, позволяют игрокам использовать штатные инструменты для создания своих интерактивных сюжетов внутри игрового мира. Благодаря этому игроки могут рассказывать свои истории с участием игровых персонажей, а поскольку они создаются с применением инструментов, подобных тем, которыми пользуются разработчики игры, эти истории по своей глубине и сложности могут быть сопоставимы с сюжетами, первоначально встроенными в игру.

  Одним из особенно вдохновляющих сюжетных игровых модов было простое изменение, реализованное Майком Хоем, отцом и поклонником *The Legend of Zelda: The Windwaker*. Хой играл в эту игру со своей дочерью Майей, и игра очень нравилась ей, но его очень беспокоило, что персонаж Линк в игре (за которого играла Майя) был мальчиком. Майк взломал игру, чтобы создать версию, в которой Линк был бы девочкой. По словам Хоя: «Понимаете, я не хочу, чтобы моя дочь росла, думая, что девочки не могут быть героями и спасать своих младших братьев». Это небольшое изменение в игре позволило его дочери почувствовать себя героем истории, чего не получалось в оригинальной версии, ориентированной на мальчиков[1].
- **Машинима:** еще один интересный пример драматического искусства в культурном уровне — машинима (machinima)[2]. Так называют линейные видеоролики, созданные путем захвата видеоклипов игрового процесса. Одним из самых известных произведений в этом жанре является *Red vs. Blue* (*RvB*), комедийный сериал, созданный компанией Rooster Teeth Productions из фрагментов игры внутри шутера от первого лица *Halo*, созданного компанией Bungie. В оригинальной версии в видеоролики добавлялись черные полосы, узкая сверху и широкая снизу. Нижняя полоса добавлялась, чтобы прикрыть оружие игрока, которое могло присутствовать в сцене, потому что первое время создатели *Red vs. Blue* использовали кадры, вырезаемые непосредственно из игры. В этих ранних роликах все еще можно видеть прицельную сетку.

  Первая серия *Red vs. Blue* вышла в апреле 2003 года. С течением времени сериал все больше оттачивался и пользовался все большим успехом и в итоге даже получил поддержку от компании Bungie. Оригинальная версия *Halo* содержала ошибку — персонаж поднимал голову и смотрел прямо перед собой, когда оружие было нацелено вниз. Она использовалась в Rooster Teeth, чтобы создать ощущение, будто персонажи кивают при разговоре (не направляя оружие друг на друга). В *Halo 2* эта ошибка была исправлена, но были добавлены позы, кото-

[1] Об этом Майк Хой написал в своем блоге, а его патч к игре можно загрузить по адресу http://exple.tive.org/blarg/2012/11/07/flip-all-the-pronouns/.

[2] https://ru.wikipedia.org/wiki/Машинима. — *Примеч. пер.*

рые принимали персонажи, когда не прицеливались, специально для удобства создания роликов, таких как *RvB*.

Другие игровые движки также стали реализовывать поддержку машинимы. *Quake* стал одним из первых движков, широко использующих машиниму. *Uncharted 2: Drake's Deception* студии Naughty Dog имела многопользовательский режим машинима, поощрявший пользователей создавать свои машинимы с помощью игрового движка и поддерживавший возможность изменения направления камеры, анимацию и многое другое.

Культурная технология

Как отмечалось выше в этой главе, элементы культурного уровня имеют множество перекрытий, поэтому большинство примеров культурной технологии уже оказались перечисленными в трех предыдущих разделах (например, игровые моды упоминались в разделе о культурной механике, но для их реализации, как вы понимаете, необходимо применение технологии). Так же как в случае с другими элементами, в своей основе культурная технология носит двойной характер: она охватывает влияние технологии игры на жизнь игроков за рамками игры и технологии, которые сообщества игроков разрабатывают для изменения фиксированных элементов игры или динамических переживаний.

- **Технология игры за ее пределами:** в последние несколько десятилетий игровые технологии развивались семимильными шагами. Увеличивающееся разрешение экранов (например, переход телевидения от разрешения 480p к 1080p и 4K) и желание игроков иметь все более реалистично выглядящие игры заставляли разработчиков постоянно совершенствовать приемы быстрого отображения высококачественной графики. Эти приемы, разработанные для игр, нашли применение почти повсюду, от диагностической визуализации в медицине до предварительной визуализации фильмов (практика использования игровой анимации и графики реального времени для компоновки сложных снимков).
- **Внешние инструменты, создаваемые игроками:** внешние инструменты, создаваемые игроком, способные изменить восприятие игры, но не являющиеся модами, потому что не изменяют фиксированную механику. В числе примеров таких инструментов можно назвать:
  - Приложения для *Minecraft*, позволяющие игроку видеть карту большего размера, что упрощает поиск конкретных географических объектов или минералов.
  - Калькуляторы, вычисляющие урон в секунду (Damage per second, DPS), для массовых многопользовательских онлайн-игр, таких как *World of Warcraft*, которые помогают игрокам определять лучшие способы прокачки своих персонажей и выбора лучшей экипировки, чтобы добиться максимального среднего урона в секунду боя.

- Любой из нескольких инструментов для онлайн-игры *Eve Online*, доступных для мобильных устройств, включая инструменты управления обучением навыкам, ресурсы, почту внутри игры и т. д.[1]
- Руководства от фанатов, как, например, доступные на сайте http://gamefaqs.com, которые помогают игрокам лучше понять игру и повысить свои возможности в игре, но фактически не изменяют фиксированную игру.

Авторизованный перенос не является частью культурного уровня

Слово *перенос* означает существование сюжета или авторских прав в нескольких средствах информации. Хорошим примером может служить игра *Pokémon*, сюжет которой, с момента создания в 1996-м, успешно был распространен на телевизионный сериал, карточную игру, серию игр для портативной приставки Nintendo и мобильных устройств, а также на многочисленные серии манги. Существует много других примеров переноса, включая видеоигры, сопровождающие почти каждый новый фильм Диснея, а также фильмы, созданные по мотивам известных игр, таких как *Resident Evil* и *Tomb Raider*.

Перенос может быть важной частью фирменного стиля игры и использоваться как стратегия расширения рынка и увеличения продолжительности присутствия на рынке. Однако важно различать авторизованный перенос (как в примере с игрой *Pokémon*) и неавторизованный, выполняемый фанатами. Последний принадлежит культурному уровню, а первый — нет (рис. 6.2).

Деление на фиксированный, динамический и культурный уровни многоуровневой тетрады определяется развитием в направлении от фиксированных элементов, созданных разработчиками игры, через динамическую игру к культурному влиянию игры на игроков и общество. Авторизованный перенос, напротив, — это перенос фирменного стиля игры на что-то иное владельцами этого стиля и авторских прав. Это обстоятельство прочно закрепляет авторизованный перенос на фиксированном уровне подобно новой игре из той же серии. Каждый конкретный перенос — это еще один продукт на фиксированном уровне, который может иметь свои динамический и культурный уровни. Здесь важно, кому принадлежит контроль. Контроль над фиксированным уровнем и над сопровождающим его авторизованным переносом принадлежит компании, разработавшей игру. Когда игра перемещается на динамический уровень, контроль делится между технологией и механикой, созданными разработчиками, и фактическими действиями, процедурами и стратегиями,

[1] В *Eve Online* навыки тренируются в режиме реального времени, независимо от того, выполнил ли игрок вход; поэтому возможность сообщить игроку, что навык получен и он может приступать к тренировке следующего навыка, может оказаться очень полезной (дополнительную информацию вы найдете по адресу http://pozniak.pl/wp/?p=4882 и https://itunes.apple.com/us/app/neocom/id418895101).

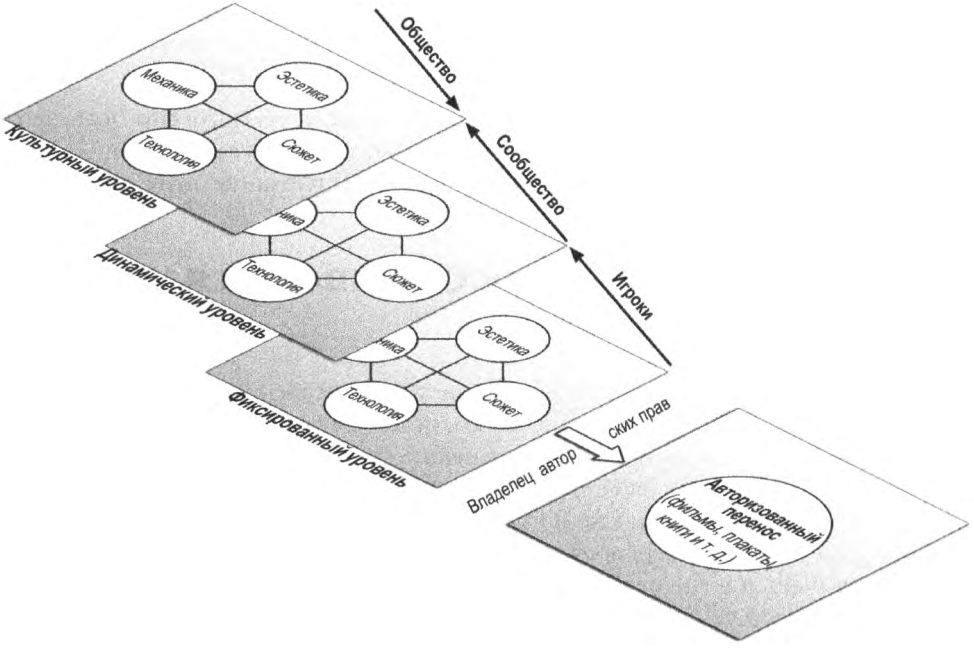


Рис. 6.2. Местоположение авторизованного переноса относительно многоуровневой тетрады

осуществляемыми игроками. На культурном уровне контроль над игрой полностью переходит из рук разработчиков игры в руки сообщества игроков. По этой причине фанфики, косплей, игровые моды и переносы, выполненные поклонниками игры, принадлежат культурному уровню, а продукты, полученные в результате авторизованного переноса, — нет.

Узнать больше об авторизованном переносе вы сможете в книгах и статьях профессора Генри Дженкинса, посвященных этой теме.

Культурное влияние игры

До сих пор мы рассматривали культурный уровень как способ, каким игроки обретают власть над игрой и переносят ее в культуру в целом. Однако на него можно взглянуть совершенно иначе и рассмотреть влияние процесса игры на игроков. К сожалению, в течение последних десятилетий игровая индустрия легко признавала и продвигала психологические исследования, доказывающие наличие положительных эффектов от игр (например, улучшенные навыки решения сразу нескольких задач, улучшенная ситуационная ориентация), и одновременно отвергала результаты исследований, доказывающие отрицательное влияние на играющих (например, зависимость от игр

и негативные последствия насилия в видеоиграх)[1]. В отношении жестоких видеоигр это, по всей видимости, объясняется защитной реакцией. Почти каждая компания, входящая в ассоциацию Entertainment Software Association (группа, лоббирующая интересы компаний, производящих видеоигры), создавала игры, в основе которых лежит тот или иной вид насилия, и, похоже, что журналисты склонны обвинять «жестокие видеоигры», когда люди совершают ужасные акты[2]. Однако в 2011 году тональность этой дискуссии кардинально изменилась, когда Верховный суд США, рассмотрев дело Эдмунда Джерри Брауна младшего (Edmund G. Brown, Jr.), губернатора штата Калифорния, и других просителей против ассоциации продавцов развлечений Entertainment Merchants Association и др., № 564 U.S. (2011), постановил, что игры являются произведениями искусства и поэтому подпадают под защиту Первой поправки к Конституции США. До этого момента члены ассоциации ESA и другие разработчики игр имели веские основания опасаться борьбы правительства с жестокими играми. Теперь, как и большинство средств информации, игры защищены как искусство и разработчики могут создавать игры любой направленности, не опасаясь запретов со стороны правительства.

Примеры отрицательного культурного влияния

Конечно, об руку с этой свободой идет ответственность признания, что игры, которые мы создаем, оказывают влияние на общество, и это относится не только к жестоким играм. В 2011 году по групповому иску Гарен Мэгуэриан против Apple Inc[3]. Apple выплатила более 100 миллионов долларов США ради урегулирования претензий, потому что игры, разработанные сторонними компаниями и одобренные в Apple, включали механику, побуждающую детей платить сотни долларов за покупки во время игры. Apple урегулировала иск и избежала судебного преследования, а суть дела состояла в том, что некоторые игры использовали слабое понимание детьми реальных денег, и некоторые дети перечислили более 1000 долларов менее чем за месяц без ведома или одобрения их родителей. Также было отмечено, что пик, когда люди играют в случайные игры в социальных сетях (например, в Facebook), приходится на рабочие часы, и многие из этих игр разработаны с применением механики «энергии» и «порчи», вынуждающей игроков возвращаться в игру каждые пятнадцать минут, что, как можно догадаться, отрицательно сказывается на производительности труда.

---

[1] Даже при поверхностном знакомстве с архивом новостей ассоциации производителей ПО и компьютерных игр (Entertainment Software Association, ESA; http://www.theesa.com/category/around-the-industry/, статья была доступна в марте 2018 года) можно заметить большое количество статей о положительных сторонах игр и почти ничего — об отрицательном их влиянии.

[2] Dave Moore и Bill Manville. «What role might video game addiction have played in the Columbine shootings?» *New York Daily News*, April 23, 2009. Kevin Simpson и Jason Blevins. «Did Harris preview massacre on Doom?» *Denver Post*, May 4, 1999.

[3] Meguerian v. Apple Inc., дело № 5:11-cv-01758, в Окружном суде Северного округа Калифорнии.

Сообщения, которые наши игры — и поклонники — посылают

Первый материал, связанный с видеоиграми, появился на первой полосе New York Times 15 октября 2014 года. Это была статья под заголовком «Feminist Critics of Video Games Facing Threats in 'GamerGate' Campaign»[1] (Феминистки-критики столкнулись с угрозами во время Геймергейтской кампании). GamerGate (Геймергейт[2]) — небольшое, но очень громкое женоненавистническое движение, прикрывавшееся заботой о соблюдении этических норм в игровой журналистике, но на практике бывшее местом сбора мужчин, боявшихся, что игры перейдут под контроль женщин, либералов и других «борцов за социальную справедливость».

Статья была написана в ответ на отмену выступления феминистки и критика СМИ Аниты Саркисян в университете штата Юта. До отмены Саркисян в течение нескольких месяцев получала угрозы убийства в ответ на серию ее исследований *Feminist Frequency* на YouTube, анализировавших скрытое женоненавистничество, пронизывавшее многие игры, выпущенные в последние десятилетия. Однако теперь некто пригрозил устроить массовую бойню на ее выступлении, а университет отказался запретить проносить оружие на ее выступление, поэтому оно было отменено.

Я хочу, чтобы вы поняли: GamerGate — это волк женоненавистничества, одетый в шкуру овцы недовольства низким уровнем этики в игровой журналистике. Я разговаривал о GamerGate со многими разработчиками игр, и подавляющее большинство из них было глубоко обеспокоено ненавистью, пропагандируемой участниками этого движения. Однако я считаю, что мы, как члены сообщества разработчиков игр, должны признать, что сами своими действиями создали GamerGate. Представляя женщин как предметы в играх и в рекламе игр, представляя сильных белых мужчин в роли героев, а женщин как объекты для спасения и трофеи победителей, мы воспитали аудиторию, которая считала, что это истина, и чувствовала угрозу, когда такие люди, как Саркисян, указывали на обратное. В последней игре из серии *Mario* — *Super Mario Odyssey* — главный герой Марио бродит по реалистичному Нью-Йорку и общается с говорящими вилками. Но даже спустя 30 лет сюжет все еще основан на его борьбе с Боузером за спасение в целом пассивной принцессы Пич, которой на протяжении всей серии отведена единственная роль — быть похищенной и служить призом для Марио[3]. Конечно, женоненавистничество не единственная проблема, с которой мы сталкиваемся

---

[1] Статью можно найти по адресу http://nyti.ms/1wHspJH, была доступна в марте 2018 года.

[2] https://ru.wikipedia.org/wiki/Геймергейт. — *Примеч. пер.*

[3] Принцесса Пич была активным игровым персонажем в американском выпуске *Super Mario Bros. 2* для NES; однако эта игра была всего лишь обновленной и усовершенствованной версией японской игры *Yume Kōjō: Doki Doki Panic* (*Dream Factory: Heart-Pounding Panic*, также спроектированной Сигэру Миямото (Shigeru Miyamoto)), а не настоящей игрой *Super Mario*. Пич также была активным персонажем во многих побочных играх с участием Марио (например, *Mario Kart*, *Mario Party*, *Mario Tennis*), но в основных играх с Марио

в плане представления женщин в играх; подавляющее большинство главных героев в играх — белые мужчины. Ситуация постепенно исправляется в коммерческих играх, но эта проблема все еще стоит достаточно остро.

Кроме того, я считаю, что мы должны внимательнее относиться к сообщениям, встраиваемым в механику наших игр. *Minecraft* — превосходная игра, поощряющая творчество и стремление к познанию окружающего мира, — включает также идею, что весь мир — это шахта с ресурсами для потребления игроками. *2b2t.org* — один из старейших и непрерывно действующий сервер игры *Minecraft*, и вступающие в его мир новые игроки оказываются в голой каменной пустыне, полностью лишенной каких-либо ресурсов. Все ресурсы в мире исчерпаны на многие километры во все стороны, и осталась только пустая шелуха каменных мостов, плывущих в воздухе, по которым игроки должны идти часами, чтобы достичь областей игры, где еще остались ресурсы. Один опытный игрок как-то отметил: «Миллионная отметка [блок]... вот где все самое интересное»[1]. При стандартной скорости ходьбы около 5 м/с игроку понадобится 2,3 дня реальной жизни (около 166 дней в мире *Minecraft*), чтобы пройти 1000 километров и достичь миллионного блока (и это при условии, что на пути не встретилось никаких ловушек, препятствий или других игроков, пытающихся убить вас на этом пути). Хотя это необычный опыт в *Minecraft*, он также является окончательным выражением основной механики игры: извлечение из земли необходимых ресурсов и создание желаемого, без несения ответственности за то, что останется после.

Как и все формы средств информации, игры могут влиять и влияют на действия и мировоззрение тех, кто в них играет. Игры никого не заставляют брать в руки оружие и устраивать массовые расстрелы, но игры и многие полицейские сериалы возводят насилие в ранг нормы, почти каждый день показывая, как полицейские применяют оружие. В действительности же в 2013 году только один из каждых 850 полицейских в Нью-Йорке применил оружие в отношении подозреваемого; то есть в среднем 0,00032 % полицейских стреляют в подозреваемых каждый день (или один из 310 250). Я считаю, что мы, как дизайнеры и создатели информации, несем ответственность за игры, которые создаем, и за сообщения, которые они посылают во внешний мир.

Механика игры может возводить в ранг нормы не только потребительское отношение или насилие, но и социальное и экологическое поведение. Версия *Minecraft*, например, могла бы иметь мир меньшего размера, требующий применения технологии севооборота и рачительного отношения к ограниченным водным ресурсам для поддержания мира и привлечения игроков на сервер. В социальной сети можно создавать игры, в которых ресурсы, доступные игроку, сокращаются с течением времени, и он мог бы получать дополнительные очки за передачу

она все еще рассматривается как объект. Источник: http://www.ign.com/articles/2010/09/14/ign-presents-the-history-of-super-mario-bros, статья была доступна в марте 2018 года.

[1] http://www.newsweek.com/2016/09/23/minecraft-anarchy-server-2b2t-will-kill-you-498946.html, статья была доступна в марте 2018 года.

части ресурсов нуждающимся, возможно, в обмен на другие ресурсы, которые он не смог приобрести самостоятельно. При распространении в играх и других средствах информации подобная механика может нормализовать рациональное отношение и альтруизм.

Итоги

Фиксированный и динамический уровни в многоуровневой тетраде обсуждались в некоторых книгах, предшествовавших этой, но культурному уровню уделялось намного меньше внимания. Фактически даже в своей практике дизайнера игр и профессора, преподающего их проектирование, я больше думаю о фиксированном и динамическом уровне и существенно меньше времени уделяю оценке культурных последствий моей работы и изменений, которые игроки могут вносить в мои игры.

Подробное исследование этики дизайна игр в значительной степени лежит за рамками этой книги, но дизайнеры должны задумываться о последствиях создаваемых ими игр, особенно потому, что когда игрок закончит и отложит игру в сторону, останется только культурный уровень.

7 Действовать как дизайнер

Теперь, когда вы немного больше узнали о том, как думают дизайнеры и как они анализируют игры, пришло время посмотреть, как дизайнеры создают интерактивные переживания.

Как упоминалось в предыдущих главах, проектирование игр — это практика, поэтому чем больше вы будете практиковаться, тем лучше у вас будет получаться. При этом важно также начинать с эффективных практик, которые дадут вам наибольший выигрыш в будущем. Знакомство с ними — цель этой главы.

Итеративное проектирование

> Проектирование игр — это 1 % вдохновения и 99 % итераций.
>
> — *Крис Свейн*

Помните эти слова, приведенные в главе 1? В этом разделе мы исследуем их подробнее.

Ключ номер один к хорошему дизайну — по сути, самое важное, о чем рассказывается в этой книге, — итеративный процесс проектирования, изображенный на рис. 7.1. Мне приходилось видеть, как итеративное проектирование превращало поначалу ужасные игры в замечательные и как этот подход с успехом применялся к любым формам дизайна, от мебели до иллюстраций.

Итеративный процесс проектирования делится на четыре этапа:

- **Анализ:** этап анализа помогает понять, где вы находитесь и куда стремитесь. Вы должны четко понимать проблему, которую пытаетесь решить (или возможность, которую пытаетесь использовать) в ходе проектирования. Вы должны также понимать, какие ресурсы сможете привнести в проект и сколько времени вы сможете выделить на проектирование.
- **Проектирование:** теперь, имея более четкое представление о том, где вы находитесь и к чему стремитесь, создайте проект, который решит вашу проблему, с использованием доступных вам ресурсов. Этот этап начинается с мозгового штурма и завершается конкретным планом реализации.

Рис. 7.1. Итеративный процесс проектирования[1]

- **Реализация:** проект в ваших руках; теперь воплотите его. Есть старая поговорка: «Игра становится игрой, только когда в нее кто-то играет». Цель этапа реализации — как можно быстрее перейти от идеи игры к действующему прототипу. Как вы увидите в учебных примерах цифровых игр далее в этой книге, первые реализации иногда просто перемещают персонажа по экрану — без врагов и без конкретной цели, просто чтобы дать возможность оценить естественность и отзывчивость движений. Реализация лишь небольшой части игры перед тестированием часто помогает сосредоточить внимание на более узком круге проблем, чем реализация большими фрагментами. Закончив реализацию, можно запустить пробную игру.
- **Тестирование:** дайте человеку поиграть в вашу игру и понаблюдайте за его реакцией. С опытом вы лучше будете понимать, какой эффект вызывают проектируемые вами игровые механики, но даже имея многолетний опыт, вы не будете знать этого наверняка. Тестирование поможет выяснить это. Всегда начинайте тестировать на самых ранних этапах, когда еще возможно что-то изменить и вернуться на правильный путь. Тестируйте как можно чаще, чтобы лучше понять причины изменения отзывов игроков. Если слишком многое изменить между двумя циклами тестирования, может быть трудно разобраться, какое именно изменение стало причиной изменения отзыва игрока.

Давайте рассмотрим каждый из этапов более детально.

---

[1] Основано на иллюстрации из книги Трейси Фуллертон (Tracy Fullerton), Кристофера Свейна (Christopher Swain) и Стивена Хоффмана (Steven Hoffman) *Game Design Workshop: A Playcentric Approach to Creating Innovative Games* (Burlington, MA: Morgan Kaufmann Publishers, 2008), 36.

Анализ

Цель любого дизайна — решить какую-либо проблему или воспользоваться какими-либо возможностями, и прежде чем приступать к проектированию, нужно ясно понять, что это за проблема или что это за возможности. Вы можете сказать себе: «Я просто хочу сделать хорошую игру», — как сказали бы многие из нас, но даже отталкиваясь от этого начального заявления, можно углубиться в свою проблему и проанализировать ее.

Для начала спросите себя о следующем:

1. **Для кого я проектирую игру?** Знание целевой аудитории поможет выбрать разные элементы дизайна. Если игра предназначена для детей, более вероятно, что родители разрешат им использовать мобильные устройства, чем компьютеры, подключенные к интернету. Если игра предназначена для любителей стратегий, они, скорее всего, будут играть на компьютерах. Если игра для мужчин, не забывайте, что около 10 % белых мужчин страдает той или иной формой дальтонизма.

   Есть одна опасность, о которой нужно помнить всегда, — опасность разработки игры для самого себя. Если вы создадите игру для себя, высока вероятность, что *только* вы и захотите сыграть в нее. Исследование целевой аудитории и ее мотивов поможет вам понять, куда должен двигаться проект игры и как сделать игру лучше.

   Также важно понимать, чего хотят игроки и что им действительно нравится, потому что иногда это две совершенно разные вещи. В процессе исследований важно различать желания, высказанные аудиторией, и то, что на самом деле мотивирует их.

2. **Какими ресурсами я обладаю?** У большинства из нас нет десятков миллионов долларов, чтобы нанять студию с 200 сотрудниками, которые за пару лет создадут игру. Но у вас, вероятно, есть немного времени и талант и, быть может, даже несколько талантливых друзей. Будьте честными сами перед собой. Точное знание того, чем вы располагаете, своих сильных и слабых сторон поможет вам в разработке вашего проекта. Ваши главные ресурсы независимого разработчика — талант и время. Любой из них можно купить за деньги, наняв специалиста или приобретя активы, но, особенно если работаете в составе небольшой независимой команды, вы должны убедиться, что задуманная вами игра максимально точно соответствует ресурсам вашей команды. Работая над игрой, расценивайте свое время и время членов вашей команды как самый драгоценный ресурс, не тратьте его попусту.

3. **Какие предшествующие произведения уже существуют?** Это единственный вопрос, о котором забывает большинство моих студентов (часто в ущерб себе). Термин *предшествующие произведения* описывает существующие игры и другие произведения, так или иначе связанные с вашей игрой. Игры не возникают из ничего, и вы, как дизайнер, должны знать не только о существовании игр, вдохновивших вас (которые вы, конечно же, знаете), но и о существовании других игр в том же пространстве, которые появились до или после произведений, вдохновивших вас.

Например, если вы хотите создать шутер от первого лица для игровой приставки, вы, конечно, в первую очередь посмотрите на серии *Destiny*, *Titanfall* и *Call of Duty*, но вам также нужно познакомиться с *Halo* (самый ранний шутер от первого лица, прекрасно работавший на приставках, когда, по общему мнению, это было невозможно), *Marathon* (игра компании Bungie, появившаяся перед *Halo* и образовавшая фундамент для большого количества дизайнерских решений и мифологии в *Halo*) и другие шутеры, предшествовавшие *Marathon*.

Исследование предшествовавших произведений необходимо, потому что вы должны понять все, что можно, о том, как другие пытались подойти к проблеме, которую вы решаете. Даже если других посетила та же идея, что и вас, они почти наверняка подошли к ней с другой стороны, и понимание их успехов и неудач поможет вам сделать свою игру лучше.

4. **Как быстрее получить демонстрационную версию игры, чтобы проверить желаемое?** Этот вопрос часто игнорируется, хотя имеет решающее значение по очевидным причинам. В сутках всего 24 часа, из которых я, например, могу потратить на разработку игр лишь малую долю. Поэтому важно эффективно расходовать свое время, если вы хотите закончить игру. Подумайте об основной механике: что будет делать игрок большую часть игры (например, основная механика в *Super Mario Bros.* — прыжки) — и спроектируйте и протестируйте ее в первую очередь. Разрабатывая и опробуя ее, вы поймете, стоит ли добавлять в игру что-то большее. Графика, музыка и все другие эстетические элементы будут важны для окончательной версии игры, но на данном этапе сосредоточьтесь на механике — процессе игры — и в первую очередь обеспечьте ее работоспособность. Это ваша цель как дизайнера.

У вас также будет множество своих вопросов, которые вы добавите позже, но эти четыре — решающие на этапе анализа для любой игры.

Проектирование

Большая часть книги посвящена проектированию, но в этом разделе я сосредоточусь на отношении к профессиональному дизайнеру. (Глава 15 «Индустрия цифровых игр» охватывает саму индустрию более подробно.)

Работа дизайнера не в том, чтобы выдвигать свои требования, каким бы гениальным или авторитетным вы ни были, и даже не в том, чтобы высказывать свое видение остальной команде. Работа дизайнера — это соблюдение интересов проекта. Работа дизайнера — это сотрудничество с остальной командой, поиск компромиссов и, прежде всего, умение слушать.

На первых нескольких страницах своей книги «The Art of Game Design» Джесси Шелл заявляет, что умение слушать является самым важным для дизайнера, и я полностью согласен с ним. Шелл перечисляет пять видов умения слушать, которые вы должны развивать в себе[1].

[1] Jesse Schell, *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), 4–6.

- **Умение слушать аудиторию:** чье внимание вы хотели бы привлечь к вашей игре? Кому бы вы хотели продавать игру? Как упоминалось ранее, это решающие вопросы, и вы должны ответить на них, а когда вы ответите, вы должны прислушаться к мнению тех, кого хотели бы видеть в составе своей аудитории. Главная цель итеративного процесса проектирования состоит в том, чтобы создать что-то, представить это тестировщикам и получить от них отзывы. Внимательно выслушайте все, что они вам скажут, даже (особенно!) если их слова не совпадут с вашими ожиданиями.
- **Умение слушать команду:** в большинстве игровых проектов вам придется работать в команде других талантливых людей. Ваша задача как дизайнера — выслушать все их мысли и идеи и переработать, чтобы выявить те, которые помогут создать лучшую игру для вашей аудитории. Если вам удастся окружить себя людьми, не боящимися высказать свое мнение, отличающееся от вашего, вы получите более удачную игру. При этом важно, чтобы команда не погрязла в спорах; это должна быть команда творческих индивидуальностей, трепетно относящихся к игре и друг к другу.
- **Умение слушать клиента:** как профессиональный дизайнер, большую часть времени вы будете работать на клиента (начальника, комитет и т. д.) и должны прислушиваться к его мнению. Обычно среди них нет опытных дизайнеров — именно поэтому они наняли вас, — но у них есть конкретные потребности, которые вы должны удовлетворить. В конце концов, это ваша работа — слушать их на нескольких уровнях: что они говорят о своих желаниях; чего они хотят, но не высказывают вслух; и даже чего они хотят в глубине души, но боятся признаться в этом даже самим себе. Умение внимательно слушать клиентов поможет вам создать для них не только великолепную игру, но и оставить отличное впечатление от работы с вами.
- **Умение слушать игру:** иногда элементы дизайна игры сочетаются друг с другом как пальцы в перчатке, а иногда — как лапа росомахи в бумажном пакете (то есть *плохо*). Являясь дизайнером, вы будете тем членом команды, находящимся ближе всего к игровому процессу, и понимание игры в *комплексе* — ваша задача. Даже если у какого-то аспекта игры блестящий дизайн, он может плохо сочетаться со всем остальным. Не волнуйтесь: если дизайн действительно хорош, вполне возможно, что вы найдете применение ему в другой игре. У вас будет множество возможностей в других играх, которые вам предстоит создать на протяжении своей карьеры.
- **Умение слушать себя** подразумевает следующее:
  - **Прислушивайтесь к своей интуиции:** иногда у вас будет появляться чувство, что что-то идет не так. Иногда это чувство будет ошибочным, а иногда нет. Когда ваше чутье подсказывает вам что-то, проверьте это. Возможно, какая-то часть вашего разума заметила что-то раньше, чем это смог осознать ваш ум.
  - **Прислушивайтесь к своему здоровью:** заботьтесь о себе и берегите свое здоровье. Я не шучу. Огромное число исследований доказывает, что работа

по ночам, стрессы и отказ от физической активности оказывают большое отрицательное влияние на способность выполнять творческую работу. Чтобы стать лучшим дизайнером, вы должны быть здоровым и хорошо отдохнувшим. Не позволяйте себе зацикливаться на выходе из череды кризисов, работая по ночам.

- **Слушайте, как звучит ваш голос, когда вы разговариваете с другими:** когда вы что-то говорите своим коллегам, сверстникам, друзьям, родным и знакомым, прислушивайтесь порой к тому, как звучит ваш голос. Я не призываю делать это постоянно, но иногда прислушивайтесь к себе и ответьте на эти вопросы:

  В моем голосе чувствуется уважение?

  В моем голосе слышится забота о других?

  В моем голосе слышна забота о проекте?

  При прочих равных условиях больших успехов добиваются те, кто последовательно демонстрирует уважение и заботу о других. Я знал по-настоящему талантливых людей, которые не понимали этого; они с самого начала делали все правильно, но их карьера не выдерживала и терпела неудачу, потому что все меньше и меньше людей хотели работать с ними. Разработка игр — это сотрудничество на основе взаимного уважения.

В работе профессионального дизайнера есть, конечно, и множество других аспектов, кроме умения слушать, но я согласен с Шеллом, что этот — один из самых важных. В оставшейся части книги больше внимания будет уделяться техническим аспектам дизайна, но вы должны подходить к их применению скромными, здоровыми, готовыми к совместному и творческому труду.

Реализация

Последние две трети этой книги описывают цифровую реализацию, но важно понимать, что ключом к эффективной реализации в процессе итеративного проектирования является *максимально быстрый* переход от проекта к тестированию. Если вы тестируете прыжок персонажа на такой платформе, как *Super Mario Bros.* или *Mega Man*, вы должны создать цифровой прототип. Но для тестирования графического интерфейса системы меню нет необходимости создавать полную работающую цифровую версию; достаточно напечатать изображения с разными разделами меню и попросить тестировщиков походить по ним с вами (меняющим изображения вручную) в роли компьютера (см. раздел «Прототипирование интерфейсов на бумаге» в главе 9 «Прототипирование на бумаге»).

Бумажный прототип позволит быстро проверить идею и получить обратную связь. Обычно прототипирование на бумаге занимает намного меньше времени, чем реализация цифрового прототипа, и дает уникальную возможность изменить правила игры в середине сеанса, если начальная их версия оказалась неудачной. Подробная информация о методах прототипирования на бумаге, а также удачные и не очень примеры их использования вы найдете в главе 9 «Прототипирование на бумаге».

Еще один способ сократить время на реализацию — понять, что не нужно все делать самостоятельно. Многие мои студенты приступают к разработке игр с желанием изучить всё и вся: они хотят спроектировать игру; написать весь код; создать модель, текстуру, внешний вид и движения игровых персонажей; создать окружающую среду; написать сценарий; создать код игры и иногда даже свой игровой движок. Если бы вы работали в студии с многомиллионным бюджетом и запасом времени в несколько лет, тогда в таком стремлении не было бы ничего предосудительного, но для независимого дизайнера это бредовая идея. Даже такие независимые разработчики, как Notch (создатель *Minecraft*)[1], которых часто считают гениями-одиночками, стояли на плечах гигантов. Первоначально игра *Minecraft* создавалась как проект с открытым исходным кодом, в котором участвовало много людей. Если вы хотите создать компьютерную игру, вы могли бы начать с создания компьютера из отдельных транзисторов, но это было бы совсем глупо. Почти такой же нелепой выглядит мысль написать свой игровой движок. Для этой книги на роль игрового движка я выбрал *Unity*, потому что сотни людей ежедневно работают в Unity Technologies, чтобы упростить жизнь нам, создателям игр. Доверяя им выполнить свою работу, я позволяю себе сосредоточиться на более интересной работе — проектировании и разработке игр, с которой я справляюсь намного лучше, чем с созданием игровых движков[2].

Аналогично, *Unity Asset Store* — потрясающее место, где можно обменять деньги на время. В онлайн-магазине Asset Store можно приобрести тысячи разных ресурсов (и сэкономить кучу времени), в том числе модели, анимации и программные библиотеки для всего чего угодно: от поддержки пультов управления до улучшенного отображения текста и великолепных библиотек отображения, основанных на законах физики[3]. Здесь также есть множество бесплатных ресурсов, которые можно успешно использовать в своих прототипах. Каждый раз, когда вы думаете потратить время на написание для своего прототипа надежного кода, пригодного для повторного использования, я рекомендую заглянуть на Asset Store — возможно, кто-то другой уже сделал это. Заплатив этому человеку несколько баксов, вы сможете сэкономить десятки часов времени.

Тестирование

После создания минимального действующего прототипа наступает время протестировать его. Главное, помните: что бы вы ни думали о своей игре, вы ничего не будете знать наверняка, пока игрок не опробует ее и не поделится с вами впечатлениями. Чем дольше он будет играть в вашу игру, тем точнее будет обратная связь.

---

[1] Https://minecraft-ru.gamepedia.com/Маркус_Перссон. — *Примеч. пер.*

[2] Для тех, кто действительно хочет написать свой игровой движок, я рекомендую фантастическую книгу по этой теме моего друга Джейсона Грегори: Jason Gregory, *Game Engine Architecture, 2nd Edition* (Boca Raton, FL: CRC Press, 2014).

[3] Для поддержки пультов управления я рекомендую приобрести библиотеку *InControl* студии Gallant Games, для улучшенного отображения текста — библиотеку *TextMeshPro* студии Digital Native Studios, для отображения с учетом законов физики — библиотеку *Alloy* студии RUST, LTD.

В моем курсе «Game Design Workshop», который я читаю в Южно-Калифорнийском университете, каждый проект настольной игры разрабатывался за четыре недели лабораторных занятий. На первом занятии студенты объединяются в команды и им дается время на мозговой штурм игровых идей. Каждое последующее занятие целиком посвящено тестированию последних прототипов их игр. К концу четвертой недели каждая команда набирала почти шесть часов тестирования в классе и значительно улучшала свой проект. Лучшее, что вы можете сделать для своих проектов, — предложить людям поиграть в них и делиться своими впечатлениями как можно чаще. И ради всего святого, записывайте все, что говорят тестировщики. Если вы забудете, что они сказали, такое тестирование не принесет пользы.

Также важно следить, чтобы тестировщики давали честные отзывы. Иногда тестировщики могут давать чересчур положительные отзывы, боясь вас обидеть. В своей книге «The Art of Game Design» Джесси Шелл рекомендует говорить тестировщикам примерно такие слова, чтобы побудить их быть честными в отношении недостатков, которые они заметят в игре:

> «Мне нужна ваша помощь. В этой игре есть проблемы, но мы пока не знаем, какие именно. Пожалуйста, если вам что-то не понравится в этой игре, вы мне сильно поможете, если расскажете, что именно»[1].

Более подробно о разных аспектах тестирования рассказывается в главе 10 «Тестирование игры».

Повторяйте, повторяйте, повторяйте, повторяйте, повторяйте!

После тестирования игры у вас должно появиться множество замечаний, записанных со слов ваших тестировщиков. Теперь самое время проанализировать их. Что понравилось игрокам? Что не понравилось? Были ли в игре места слишком простые или слишком сложные? Была ли игра интересной и захватывающей?

Ответив на все эти вопросы, вы сможете очертить новую проблему для последующего решения. Уделите время на интерпретацию и обобщение отзывов игроков (подробнее об этом рассказывается во врезке в главе 10). После этого попробуйте выбрать конкретную и достижимую цель для следующей итерации. Например, вы можете решить сделать более захватывающей вторую половину первого уровня или уменьшить влияние случайности на игру.

Каждая последующая итерация должна включать некоторые изменения, но не старайтесь изменить слишком много или решить сразу множество проблем. Самое главное — как можно скорее дойти до следующего этапа тестирования игры и определить, удалось ли вам своими изменениями решить проблемы, которые вы собирались решить.

---

[1] Jesse Schell, *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), 401.

Новаторство

В своей книге «The Medici Effect»[1] ее автор Франс Йоханссон пишет о двух видах инноваций: постепенных и лежащих на стыке (идей, технологий, концепций).

- **Постепенные инновации** способствуют развитию чего-либо предсказуемым способом. Плавное совершенствование процессоров Pentium в компании Intel на протяжении 1990-х — это постепенные инновации: каждый год выпускался новый процессор Pentium, имевший больше транзисторов, чем предыдущее поколение. Постепенные инновации надежны и предсказуемы, и если вы ищете инвестиционный капитал, с помощью таких инноваций вам проще будет убедить инвесторов, что их деньги будут работать. Однако, как следует из названия, постепенные инновации никогда не совершают резких скачков вперед, потому что от них ожидается именно постепенное поведение.
- **Инновации на стыке** возникают при столкновении двух разрозненных идей, и именно здесь может появиться множество других величайших идей. Однако из-за того, что результаты инноваций, лежащих на стыке, часто непредсказуемы, обычно сложно бывает убедить других в достоинствах этих идей.

В 1991 году Ричард Гарфилд попытался найти издателя для своей игры *RoboRally*. Одним из тех, к кому он обратился, оказался Питер Адкинсон, основатель и директор компании Wizards of the Coast. Несмотря на то что Адкинсону очень понравилась игра, он посчитал, что его компания Wizards не обладает достаточными ресурсами для выпуска такой игры, как *RoboRally*, содержащей так много разных деталей. Но он вспомнил о Ричарде, когда они искали новую игру, которая содержала бы немного деталей и которую можно было бы пройти за 15 минут.

Ричард попробовал совместить идею короткой карточной игры, содержащей небольшое количество деталей, с другой идеей, которую он вынашивал в своей голове, — идеей игры с карточками, собираемыми подобно бейсбольным, — и в 1993 году Wizards of the Coast выпустила игру *Magic: The Gathering*, давшую начало целому жанру коллекционных карточных игр (Collectible Card Games, CCG).

До встречи с Адкинсоном Гарфилд уже подумывал о создании коллекционной карточной игры, в результате возникло столкновение его идеи с конкретным пожеланием Адкинсона получить короткую игру, из которой родился жанр коллекционных карточных игр, и почти все такие игры, выпущенные с тех пор, следуют одной и той же базовой формуле: базовый набор правил, карточки с напечатанными правилами, которые отменяют базовые правила, конструкция колоды и быстрая игра.

---

[1] Frans Johansson, *The Medici Effect: What Elephants and Epidemics Can Teach Us about Innovation*; Boston, MA: Harvard Business School Press, 2006 (*Йоханссон Франс.* Эффект Медичи: Возникновение инноваций на стыке идей, концепций и культур. М.: Вильямс, 2008. — *Примеч. пер.*).

Далее описывается процедура мозгового штурма с применением инноваций обоих видов, которая поможет вам генерировать удачные идеи.

Мозговой штурм и формирование идей

> Лучший способ найти хорошие идеи — найти много идей и выкинуть плохие.
>
> — *(Linus Pauling), лауреат Нобелевских премий по химии и мира.*

Очевидно, что не все ваши идеи будут удачными, поэтому лучшее, что можно сделать, — сгенерировать множество идей и затем просеять их, чтобы отобрать лучшие. В этом заключается идея мозгового штурма. Далее я расскажу о конкретном процессе мозгового штурма, который прекрасно подходит для большинства команд, особенно для групп творческих людей.

Для мозгового штурма вам понадобится белая доска, стопка карточек для заметок размером 3 × 5 дюймов (или просто стопка листов бумаги), блокнот для записи идей и разноцветные маркеры для рисования на доске, ручки, карандаши и т. д. Лучше всего, когда в процессе участвует от пяти до десяти человек, но вы можете изменить число, адаптировав процесс для группы меньшего размера путем повторения задач. В прошлом я так и поступал, адаптируя процедуру мозгового штурма для аудитории с 65 студентами. (Например, если вы один, а в описании говорится, что каждый в группе должен сделать что-то один раз, просто сделайте это сами несколько раз, пока не удовлетворитесь.)

Шаг 1: этап расширения

Допустим, вы только начинаете 48-часовой игровой джем с несколькими друзьями. Тема джема — уроборос (змея, свернувшаяся в кольцо и кусающая себя за хвост; этот символ был темой глобального игрового джема Global Game Jam в 2012 году). Не так много, правда? Итак, вы начинаете мозговой штурм, с которым познакомились еще в школе. Нарисуйте уробороса в центре доски, нарисуйте окружность вокруг него и начинайте высказывать возникающие ассоциации. Не волнуйтесь о том, что вы записываете на этом этапе, не отклоняйте ничего — просто записывайте все, что приходит на ум. На рис. 7.2 показан пример таких записей.

Помните о тирании маркера. Если в мозговом штурме участвует больше людей, чем у вас есть цветных маркеров, постарайтесь гарантировать, что любой из участников будет услышан. Творческие люди бывают разных типов, и у самого что ни на есть интроверта в вашей группе могут родиться самые лучшие идеи. Если вы руководите группой творческих людей, постарайтесь сделать так, чтобы самые интровертивные получили по маркеру. Возможно, они захотят что-то написать на доске, не высказывая мысли вслух.

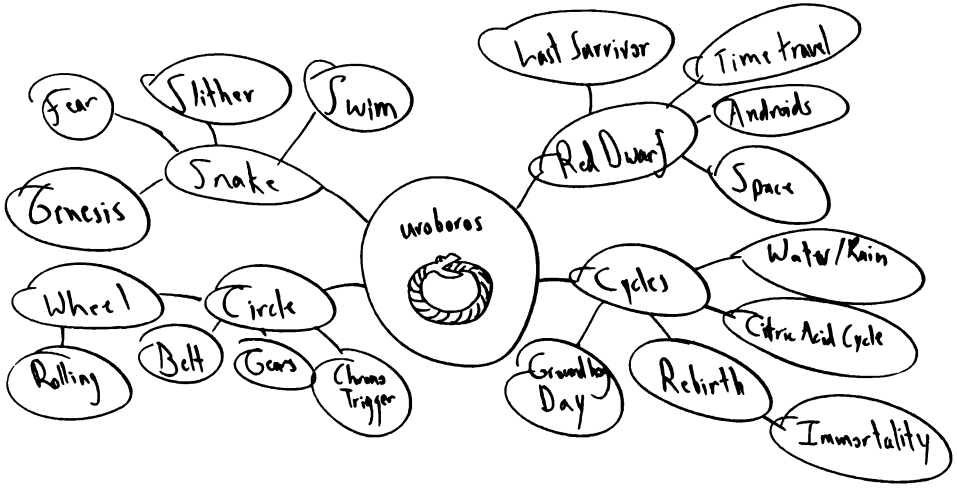


Рис. 7.2. Этап расширения мозгового штурма игры в уробороса

Закончив, сфотографируйте доску с набросками. У меня в телефоне скопились сотни таких фотографий, и я никогда не жалел о том, что сделал их. Сделав фотографию, разошлите ее по электронной почте всем участникам.

Шаг 2: этап сбора данных

Соберите все узлы, нарисованные во время этапа расширенного мозгового штурма, и запишите их по одному на карточки для заметок 3 × 5. Такие карточки называют *карточками идей* (рис. 7.3), и они будут использоваться на следующем этапе.

Страх	Скользкий	Плыть	Происхождение	Последний выживший	Путешествие во времени
Андроиды	Космос	Вода/Дождь	Цикл лимонной кислоты	Бессмертие	День сурка
Секундомер	Механизмы	Ремень	Вращение	Змея	Красный карлик
Круги	Возрождение	Окружность	Колесо		

Рис. 7.3. Карточки идей на тему уробороса

КОРОТКОЕ ОТСТУПЛЕНИЕ И ОДНА-ДВЕ НЕУДАЧНЫЕ ШУТКИ

Начнем с неудачной шутки:

> Идут по дороге два атома лития, и один говорит другому: «Мне кажется, я потерял электрон». В ответ другой его спрашивает: «Ты уверен?» На что первый отвечает: «Положительно, да!»

А вот еще одна:

– Вы слышали о пожаре в цирке?

– О да, зрелище огонь!

Извините, я знаю, что это глупые шутки.

Возможно, вам интересно, зачем я рассказал эти неудачные шутки. А сделал я это потому, что шутки, подобные этим, работают так же, как инновации на стыке. Людям нравится придумывать и смешивать самые дикие идеи. Шутки смешны потому, что сначала ведут наши мысли по одному пути, а потом резко вбрасывают чужеродную идею. Ваш разум создает связь между двумя разрозненными понятиями, кажущимися несвязанными, и возникает радость, которая называется юмором.

То же происходит при смешивании двух идей, и именно поэтому так приятен нам момент озарения, возникший на стыке двух обычных идей и породивший новую, необычную.

Шаг 3: этап столкновения

Здесь начинается самое интересное. Перемешайте карточки идей и раздайте по две каждому в группе. Каждый должен открыть свои карточки и повесить их на доску, чтобы было видно всем. Затем группа вместе придумывает три разные идеи игры, возникшие из столкновения карточек. (Если пара карточек содержит тесно связанные идеи или идеи на них никак не связываются, такую пару можно пропустить.) На рис. 7.4 показана пара примеров.

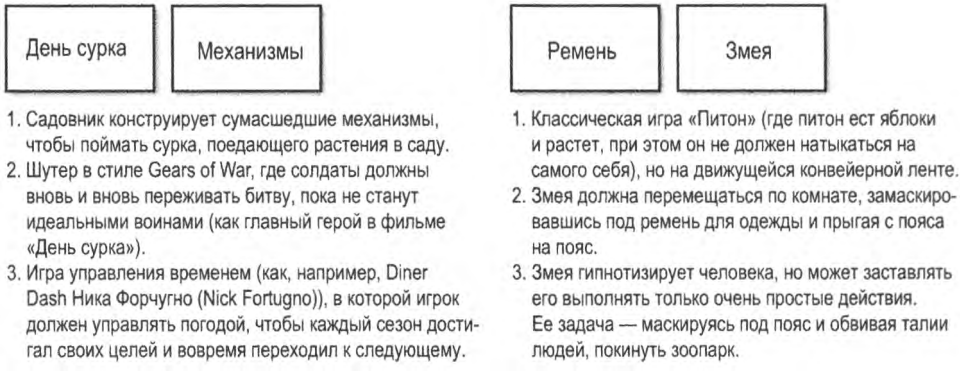


Рис. 7.4. Столкновение идей

Примеры на рис. 7.4 — это первое, что пришло мне в голову, и нечто похожее должно бы посетить вас. На этом этапе мы тоже ничего не отбрасываем. Запишите все идеи, пришедшие вам в голову на этом этапе.

Шаг 4: этап оценки

Теперь, когда у вас есть множество идей, можно заняться их отбором. Каждый участник должен написать на доске две идеи, родившиеся на шаге 3, заслуживающие наибольшего внимания с его точки зрения.

Когда все закончат, каждый должен поставить галочку рядом с тремя идеями, которые нравятся ему больше других. В результате одни идеи получат больше галочек, другие — меньше.

Шаг 5: обсуждение

Продолжите процесс отбора, изменяя и объединяя идеи с самым высоким рейтингом. Из десятков разных сумасшедших идей вы сможете выбрать две-три особенно хороших и, объединив их, получить отличную отправную точку для проекта.

Изменение мнения

Изменение мнения — решающий аспект итеративного процесса проектирования. На разных итерациях вы неизбежно будете вносить изменения в свой дизайн.

Как показано на рис. 7.5, никто и никогда не воплощает исходную идею непосредственно в игру без всяких изменений (как показано в верхней части рисунка), и даже если кто-то попытается пойти таким путем, в результате получится плохая игра. В действительности проектирование идет извилистым путем, как показано в нижней части рис. 7.5. Изначально у вас есть идея, и вы создаете начальный прототип. В процессе работы над прототипом появляются новые идеи, и на их основе вы создаете другой прототип. Может быть, у вас что-то не получится, вы отступите на шаг и переделаете это. Вы продолжаете этот процесс, пока не воплотите идею в замечательную игру, и если вы будете придерживаться этого процесса, прислушиваться, участвовать в творческом сотрудничестве, ваша игра получится намного лучше, чем задумывалось.

По мере движения вперед блокируется все большая часть проекта

Только что описанный процесс прекрасно подходит для разработки небольших проектов или для этапа опытного проектирования любого проекта, но после того, как много людей вложит много времени и сил во что-то, изменять свое мнение

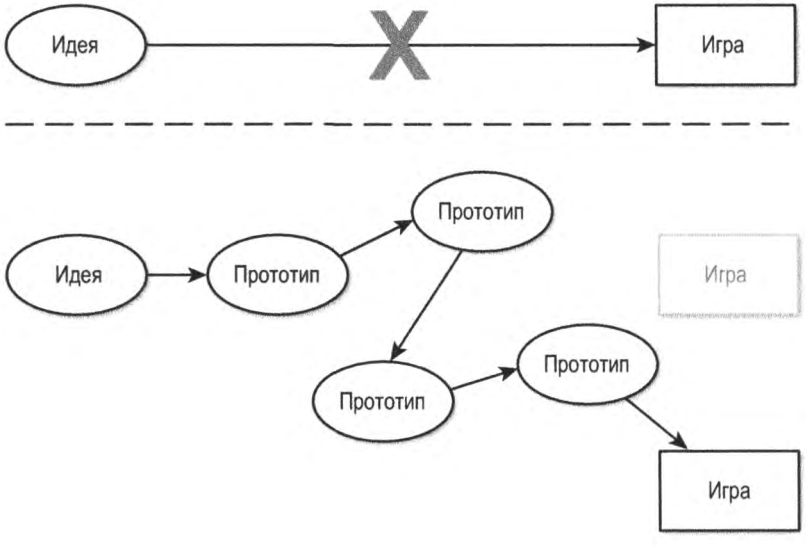


Рис. 7.5. Действительность проектирования игры

становится все труднее и дороже. Обычная профессиональная игра разрабатывается в несколько этапов:

- **Опытное проектирование:** этому этапу посвящена большая часть книги. На этапе опытного проектирования вы экспериментируете с разными прототипами и пытаетесь найти что-то, что было бы наглядно, увлекательно и доставляло удовольствие. Этап опытного проектирования прекрасно подходит для изменения мнения. В работе над крупным промышленным проектом на этапе опытного проектирования принимают участие от 4 до 16 человек, и в конце этого этапа обычно получается *вертикальный срез* — короткий пятиминутный фрагмент игры с тем же уровнем качества, что и окончательная версия игры. Это напоминает демонстрационный уровень для руководителей и других лиц, принимающих решение, чтобы дать им возможность поиграть и решить, стоит ли производить эту игру. На этом этапе уже должны быть спроектированы другие разделы игры, но они могут быть пока не реализованы.
- **Производство:** в игровой индустрии при переходе к этапу производства игры команда разработчиков быстро увеличивается. Над большими играми может работать до 100 человек, многие из которых могут жить в разных городах и даже странах. На этапе производства с самого начала должен быть заблокирован *системный дизайн* (то есть механика игры), а затем последовательно, по мере готовности, должны блокироваться другие аспекты дизайна (например, дизайн уровней, настраиваемые возможности персонажей и т. д.). С эстетической стороны этап производства — это период, когда происходит изменение моделей, текстур, анимации и реализаций других эстетических элементов.

Этап производства доводит до высокого качества вертикального среза весь остальной проект.

- **Альфа:** к моменту перехода на этап альфа-версии все функциональные возможности и механика игры должны быть заблокированы на 100 %. На этом этапе не может вноситься никаких изменений в системный дизайн игры, изменять допускается только такие аспекты, как дизайн уровней, — в ответ на проблемы, открывшиеся во время тестирования. На этом этапе игровое тестирование переходит в проверку качества, чтобы найти проблемы и ошибки (дополнительную информацию ищите в главе 10). На этапе альфа-версии игра все еще может содержать какие-то ошибки (например, ошибки программирования), и вы должны выявить их все и уметь воспроизвести.
- **Бета:** к моменту перехода на этап бета-версии работа над игрой должна быть практически завершена. В бета-версии должны быть исправлены все ошибки, способные вызвать крах игры, и могут оставаться только мелкие и несущественные ошибки. Цель этапа бета-версии — поиск и исправление последних ошибок в игре. С художественной стороны это означает, что все текстуры должны отображаться правильно, каждый фрагмент текста должен читаться без затруднений, и т. д. На этапе бета-версии не вносится никаких новых изменений — только исправления любых оставшихся проблем.
- **Золото:** достигнув золотого этапа, ваш проект готов к выпуску. Это название этапа сохранилось со времен, когда мастер-диском для производства компакт-дисков с игрой фактически был диск из золота, к поверхности которого физически прижимался фольгированный слой обычных компакт-дисков для получения оттиска. Теперь, когда даже для игровых приставок обновления поставляются через интернет, золотой этап утратил свой окончательный характер, но название «золото» все еще используется для обозначения этапа готовности игры к дистрибуции.
- **После выпуска:** благодаря повсеместности интернета все игры, которые распространяются не на картриджах (как игры для Nintendo DS и некоторые игры для 3DS), могут подвергаться *тюнингу*[1] после выпуска. Период после выпуска можно также использовать для создания загружаемого контента (Downloadable Content, DLC). Так как DLC часто включает новые задания и уровни, каждый выпуск DLC преодолевает те же этапы разработки, что и большая игра (хотя и в меньшем масштабе): опытное проектирование, производство, альфа, бета и золото.

Несмотря на то что ваши первые проекты будут намного меньше профессиональных, по-прежнему важно, чтобы вы блокировали дизайнерские решения как можно раньше. В профессиональной команде большое изменение дизайна на этапе про-

[1] Термин *тюнинг* подразумевает небольшие корректировки в механике игры на заключительных стадиях. Даже при том, что многие игры для Nintendo Switch распространяются на картриджах, они все еще могут подвергаться тюнингу через загружаемые обновления.

изводства может стоить миллионы долларов, а в независимых командах это может отсрочить выпуск игры на несколько месяцев, лет или навсегда. Пока вы двигаетесь по своей игровой карьере, никого не будут интересовать ваши незавершенные игры или нереализованные идеи, но всем будет интересно познакомиться с завершенной и выпущенной игрой. Выпуск игр создает репутацию эффективности, и это то, что люди ищут в разработчиках игр.

Ограничение объемов работ

Играя роль дизайнера игр, вы должны понимать одну важную идею — как определять объем работ. *Ограничение объемов работ* — это процесс ограничения дизайна тем, что можно выполнить за разумное время при имеющихся ресурсах. *Переоценка своих сил* — убийца номер один любительских игровых проектов.

Скажу еще раз: *переоценка своих сил — убийца номер один любительских игровых проектов.*

Большинство игр, которые вы видите и в которые играете, создавались десятками людей в течение месяцев. Разработка некоторых больших игр для приставок стоит до 500 миллионов долларов. Команды в этих проектах состоят из уникальных специалистов с многолетней практикой.

Я не пытаюсь отговорить вас, я лишь хочу, чтобы вы приучали себя мыслить менее масштабно. Ради себя самого не пытайтесь создать вторую игру *Titanfall* или *World of Warcraft* или другую, столь же огромную. Вместо этого найдите небольшую, по-настоящему интересную механику и исследуйте ее в небольшой игре.

Для вдохновения можете посмотреть игры, которые каждый год участвуют в фестивале независимых игр IndieCade Game Festival. IndieCade — это главный фестиваль независимых игр разного размера, и я думаю, что он является авангардом независимых игр[1]. На веб-сайте фестиваля (http://indiecade.com) можно найти огромное количество потрясающих игр, каждая из которых по-своему раздвигает границы игрового опыта. Каждая — чей-то страстный проект, и на многие из них были потрачены сотни и тысячи часов работы небольшой команды или отдельного человека.

Удивляешься, насколько некоторые из них малы. И это нормально. Несмотря на малый размер, они достаточно чудесны, чтобы удостоиться награды фестиваля IndieCade.

По мере профессионального развития вы можете перейти к созданию крупных игр, подобных *Starcraft* или *Grand Theft Auto*, но не забывайте, что все мы с чего-то начинаем. До создания «Звездных войн» Джордж Лукас был просто талантливым молодым человеком, обучавшимся на факультете кинематографических искусств

[1] В духе полной открытости заявляю, что начиная с 2013 года я занимаю пост председателя IndieCade по образованию и развитию, и я горд, что имею отношение к этой замечательной организации.

в Южно-Калифорнийском университете. В сущности, даже снимая «Звездные войны», он настолько ограничил объем работ, что сумел создать один из самых потрясающих фильмов всего за 11 миллионов долларов. (Он заработал на прокате фильма более 775 миллионов и много, много больше на продаже игрушек и видеофильмов для домашнего просмотра.)

Поэтому старайтесь мыслить менее масштабно. Придумайте что-то, что вы наверняка осилите за обозримое время, трудитесь эффективно и, самое главное, **закончите это**. Если вы создадите что-то замечательное, вы всегда сможете вернуться к этому позже.

Итоги

Инструменты и теории, о которых вы прочитали здесь, — это все то, что я преподаю своим студентам и сам использую в своей работе дизайнера. Я видел, как перечисленные выше методики мозгового штурма работают в больших и маленьких группах, помогают генерировать интересные, непривычные и все же реализуемые идеи, и каждый опыт, который я получал в индустрии и в академических кругах, доказывал, что итеративное проектирование, быстрое прототипирование и правильное ограничение объемов работ — ключевые принципы, которым вы можете следовать для улучшения своего дизайна. Нет ничего другого, что я мог бы рекомендовать столь же настоятельно.

8 Цели проектирования

В этой главе рассматриваются несколько важных целей проектирования, которые могут быть у вас для ваших игр. Здесь мы охватим все — от обманчиво сложной цели *забавы* до цели *эмпирического понимания*, которая может быть уникальной для интерактивного опыта.

Читая эту главу, подумайте, какие из этих целей важны для вас. Важность целей друг относительно друга будет меняться от проекта к проекту, и часто будет меняться на разных этапах разработки. Но всегда помните, что все они, и даже неважные для вас, должны выбираться сознательно, а не в результате неумышленного упущения.

Цели проектирования: неполный список

Проектируя игру или другую интерактивную среду, вы можете ставить перед собой сколько угодно целей, и я уверен, что у каждого из вас найдется своя цель, которой не нашлось места в этой главе. Тем не менее я постараюсь охватить как можно больше целей, которые я наблюдал в своей работе дизайнера и в работе моих студентов и друзей.

Цели для дизайнера

Эти цели часто занимают вас как дизайнера. Что лично вы хотели бы получить от разработки игры?

- **Удача:** вы хотите заработать денег.
- **Слава:** вы хотите, чтобы люди знали о вас.
- **Сообщество:** вы хотите быть частью чего-то большего.
- **Самовыражение:** вы хотите общаться с другими через игры.
- **Высшее благо:** вы хотите сделать мир лучше.
- **Стать лучшим дизайнером:** вы просто хотите создавать игры и совершенствовать свои умения.

Цели для игрока

Эти цели определяют, что вы хотели бы дать игрокам своей игрой:

- **Забава:** вы хотите, чтобы игроки получали удовольствие от вашей игры.
- **Игривый настрой:** вы хотите сделать игроков частью фантастического сюжета вашей игры.
- **Потоковое состояние:** вы хотите добиться оптимальной сложности для игроков.
- **Организованный конфликт:** вы хотите дать игрокам возможность сразиться с другими или бросить вызов вашей игровой системе.
- **Раскрепощение:** вы хотите, чтобы игрок почувствовал себя могущественным и в игре, и в метаигре.
- **Интерес/внимание/вовлеченность:** вы хотите вовлечь игрока в свою игру.
- **Осмысленные решения:** вы хотите, чтобы выбор игрока в той или иной ситуации имел смысл для него и для игры.
- **Эмпирическое понимание:** вы хотите, чтобы игроки обретали понимание чего-либо через игру.

А теперь рассмотрим каждую из целей подробнее.

Цели для дизайнера

У вас, как дизайнера и разработчика, есть некоторые цели в жизни, которых вы надеетесь достичь, создавая игры.

Удача

Мой друг Джон Хованек (John «Chow» Chowanec) уже много лет работает в игровой индустрии. Когда я впервые встретил его, он дал мне несколько советов о том, как зарабатывать деньги в игровой индустрии. Он сказал: «На игровой индустрии вы можете сделать буквально... сотни долларов»

Как намекает эта шутка, есть много более простых и быстрых способов зарабатывания денег. Я часто говорю своим студентам, обучающимся программированию, что если они хотят зарабатывать деньги, им лучше пойти работать в банк — у банков очень много денег и они с радостью платят тем, кто помогает им сохранить их. Работа в игровой индустрии похожа на любую другую в сфере развлечений: рабочих мест меньше, чем желающих занять их, и обычно люди получают удовольствие от своей работы, поэтому компании, занимающиеся выпуском игр, платят меньше, чем другие компании, сотрудникам той же квалификации. Конечно, в игровой индустрии есть люди, делающие много денег, но их очень немного.

Работая в игровой индустрии, вполне можно жить на достойном уровне, особенно если вы одиноки и у вас нет детей. Это особенно верно, если вы работаете в большой игровой компании, где платят хорошие зарплаты и предоставляют разные льготы. Работа в маленькой компании (или создание своей небольшой компании) обычно сопряжена с большими рисками и часто оплачивается хуже, зато есть шанс заработать долю в компании, что дает небольшой шанс начать зарабатывать больше.

Слава

Буду честен: очень, очень немногие прославляются на поприще проектирования игр. Желание стать игровым дизайнером ради известности немного похоже на желание стать художником по спецэффектам в фильмах, чтобы прославиться. Даже если вашу работу увидят миллионы, очень немногие запомнят вас.

Конечно, есть известные имена, такие как Сид Мейер, Уилл Райт и Джон Ромеро, но все эти люди уже много лет занимаются играми и за это время прославились. Есть также новые имена, которые могут быть вам известны, такие как Женова Чен, Джонатан Блоу и Маркус Перссон; но даже в этом случае людям более известны их игры (*Flow/Flower/Journey*, *Braid/The Witness* и *Minecraft* соответственно), чем их имена.

Однако, на мой взгляд, есть кое-что получше, чем слава, — сообщество, и в игровой индустрии этого сколько угодно. Игровая индустрия намного меньше, чем предполагают внешние наблюдатели, и это отличное сообщество. В частности, меня всегда восхищали дружелюбие и открытость сообщества независимых игр и игровой конференции IndieCade.

Сообщество

Конечно, в игровой индустрии много разных сообществ, но в целом это чудесное место, наполненное замечательными людьми. Многие из моих близких друзей — это люди, с которыми я встретился, работая в игровой индустрии или обучая созданию игр. Несмотря на то что большое количество высокобюджетных AAA-игр выглядят сексистскими и жестокими, по моему опыту, большинство тех, кто работает над созданием игр, действительно хорошие люди. Существует также большое и активное сообщество разработчиков, дизайнеров и художников, старающихся сделать игры более прогрессивными и с более широкими перспективами. За последние несколько лет на фестивале независимых игр IndieCade имели место выступления, пользовавшиеся самым пристальным вниманием, посвященные разнообразию игр, которые мы создаем, и команд разработчиков, которые делают эти игры. Независимое игровое сообщество — это меритократия; если вы успешно

справляетесь со своей работой, независимое сообщество будет с уважением относиться к вам независимо от пола, расы, сексуальной ориентации, религии и прочих дурацких признаков, которые люди могут использовать для дискриминации. Разумеется, еще есть куда расти степени открытости сообщества разработчиков игр — как известно, в семье не без урода, — но оно полно людей, желающих сделать его уютнее для всех.

Самовыражение и общение

Эта цель — оборотная сторона направленной на игрока цели эмпирического понимания, которая описывается ниже в этой главе. Однако самовыражение и общение могут принимать более разнообразные формы, чем эмпирическое понимание (являющееся исключительной особенностью интерактивных средств информации). Дизайнеры и художники уже тысячи лет выражают себя всеми возможными способами. Если вы хотите что-то выразить, задайте себе два важных вопроса:

Какое средство информации поможет лучше выразить мою идею?

Какими средствами информации я умею пользоваться?

Где-то между этими двумя вопросами вы найдете ответ, который подскажет, является ли интерактивное пространство лучшим способом самовыражения для вас. К счастью, в интерактивной сфере существует очень активная аудитория, ищущая новые плоды самовыражения. Примерами очень личного самовыражения могут служить *Papo & Yo*, *Mainichi* и *That Dragon, Cancer*, привлекшие много внимания и высоко оцененные критиками, что свидетельствует о готовности интерактивной среды как средства самовыражения[1].

Высшее благо

Многие занимаются созданием игр, потому что хотят сделать мир лучше. Эти игры часто называют *серьезными играми* или *играми, меняющими мир*, и они часто — предмет обсуждения нескольких конференций разработчиков, включая конференцию Meaningful Play в Университете штата Мичиган. Этот жанр также может стать для

[1] Игра *That Dragon, Cancer* (2014), Райана Грина (Ryan Green) и Джоша Ларсона (Josh Larson) рассказывает об опыте пары, узнавшей, что у их сына терминальная стадия рака. Это помогло Райану справиться с заболеванием раком у его собственного сына. Игра *Mainichi* (2013), Мэтти Брис (Mattie Brice) была создана Мэтти с целью показать своему другу переживания трансгендерной женщины, живущей в Сан-Франциско. Игра *Papo & Yo* (2014, студия Minority Media) помещает игрока в вымышленный мир мальчика, который пытается защитить себя и свою сестру от иногда добродушного, а иногда жестокого монстра, являющегося воплощением его отца-алкоголика (отца автора игры Вандера Кабальеро (Vander Caballero). — *Примеч. пер.*).

маленькой студии отличным средством взлететь и сделать что-то хорошее для мира; многие правительственные учреждения, компании и некоммерческие организации предлагают гранты и контракты для разработчиков, интересующихся созданием серьезных игр.

Для описания игр, созданных во имя высшего блага, используется много разных терминов. Вот три наиболее распространенных:

- **Серьезные игры:** одно из самых старых и самых общих названий игр этого вида. Конечно, эти игры могут быть забавными; слово «серьезные» лишь отмечает, что за игрой стоит более высокая цель, чем просто увеселение. Ярким примером этой категории могут служить обучающие игры.
- **Игры, меняющие общество:** к этой категории обычно относят игры, призванные влиять на людей или изменять их отношение к некоторой теме. В эту категорию попадают игры, поднимающие проблему глобального потепления, дефицита государственного бюджета или описывающие достоинства и недостатки разных политиков.
- **Игры, меняющие поведение:** цель этих игр не в том, чтобы изменить мнение или образ мышления игрока (как у игр, меняющих общество), а в том, чтобы изменить поведение игрока за рамками игры. Например, в медицинской сфере было создано множество игр, направленных на предотвращение ожирения у детей, на развитие внимательного отношения к другим, на борьбу с депрессией и выявление таких недостатков, как ослабление зрения у детей. Увеличивающееся число исследований показывает, что игры могут оказывать существенное влияние (и положительное, и отрицательное) на умственное и физическое здоровье.

Стать лучшим дизайнером

Самое главное, что нужно делать, чтобы стать отличным дизайнером, — создавать игры... точнее, создавать много игр. Цель моей книги — помочь вам в этом, что стало одной из причин, почему в конце книги описывается несколько учебных примеров разных игр вместо одного большого примера, как это делают многие другие книги, посвященные разработке игр. В каждом учебном примере основное внимание уделяется созданию прототипа игры определенного вида, и каждый охватывает несколько специальных тем. Прототипы, которые вы создадите в этих главах, призваны служить не только инструментами обучения, но и основой для ваших собственных игр в будущем.

Цели для игрока

Как дизайнер и разработчик, вы будете ставить перед своими играми цели, направленные на оказание некоторого влияния на игрока.

Забава

Многие считают забаву единственной целью игр, но вы, как читатель этой книги, уже наверняка понимаете, что это не так. Как рассказывается далее в этой главе, игроки готовы играть даже в не особо веселые игры, если в них есть то, что приковывает внимание. Это верно для всех форм искусства; я с большим вниманием смотрел фильмы *Schindler's List* («Список Шиндлера»), *Life is Beautiful* («Жизнь прекрасна») и *What Dreams May Come* («Куда приводят мечты»), но ни один из них я бы не назвал забавным. Несмотря на то что забава — не единственная цель игр, расплывчатая ее идея по-прежнему весьма значима для дизайнеров игр.

В своей книге «Game Design Theory» Кит Бургун (Keith Burgun) определяет три аспекта, которые, по его мнению, делают игру забавной. По его словам, игра должна быть приятной, увлекательной и доставляющей удовлетворение:

- **Приятная:** игра может быть приятной во многих отношениях и доставлять удовольствие в той или иной форме — это то, что большинство игроков ищет, начиная играть в игру. В своей книге «Les Jeux et Les Hommes»[1] (1958) Роже Кайуа (Roger Caillois) определяет четыре вида игры:
  - **Agon:** состязательные игры (например, шахматы, бейсбол, серия игр *Uncharted*).
  - **Alea:** игры, базирующиеся на удаче (например, рулетка или «камень, ножницы, бумага»).
  - **Ilinx:** головокружительные (в прямом смысле слова) игры (например, американские горки и другие игры, вызывающие головокружение).
  - **Mimicry:** игры, основанные на вере в фантазию и симуляции реальности (например, игра в дочки-матери, игра с оловянными солдатиками, ролевые игры).

  Каждая из этих игр по-своему приятна, хотя забавность каждой зависит от игривого отношения (то есть отношения к процессу как к игре, о чем рассказывается в следующем разделе). Как утверждает Крис Бейтман в своей книге «Imaginary Games», в играх ilinx (головокружительных) есть тонкая грань между страхом и восторгом, определяемая игривым отношением игрока[2]. Аттракцион *Tower of Terror* («Башня ужаса») в парках Диснея — это забавная имитация лифта, потерявшего управление, но тому, кто входит в этот лифт, становится не до забавы.
- **Увлекательная:** игра должна захватывать и удерживать внимание игрока. В своем докладе «Attention, Not Immersion», сделанном в 2012 году на конференции разработчиков игр Game Developers Conference в Сан-Франциско,

[1] Roger Caillois, *Le Jeux et Les Hommes* (*Man, Play, & Games*); Paris: Gallimard, 1958. (*Кайуа Р.* Игры и люди: Статьи и эссе по социологии культуры / Сост., пер. с фр. и вступ. ст. С. Н. Зенкина. М.: ОГИ, 2007. — *Примеч. пер.*)

[2] Chris Bateman, *Imaginary Games* (Washington, USA: Zero Books, 2011), 26–28.

Ричард Лемарчанд, ведущий разработчик первых трех игр в серии *Uncharted*, особо подчеркнул слово «внимание», и это очень важный аспект проектирования игр. Я подробнее расскажу о его докладе далее в этой главе.

- **Приносящая удовлетворение:** игра должна удовлетворять какие-то потребности или желания игрока. У нас, людей, много потребностей, которые можно удовлетворить через игру как реальными, так и виртуальными способами. Потребность в социализации и в том, чтобы быть частью сообщества, например, можно удовлетворить, играя в настольные игры с друзьями или получая жизненный опыт в *Animal Crossing* с виртуальными друзьями, живущими в вашем городе. Чувства триумфа[1] можно достичь, помогая своей футбольной команде выиграть матч, победив друга в поединке, как в игре *Tekken*[2], или, наконец, пройдя заключительный уровень сложной ритмической игры, такой как *Osu! Tatake! Ouendan*. У разных игроков разные потребности, и один и тот же игрок может иметь разные потребности в разные дни.

Игривый настрой

В книге «The Grasshopper» Бернард Сьютс подробно рассказывает об *игривом настрое*: настрое, которого нужно достичь, чтобы принять участие в игре. Находясь в игривом настрое, игроки с удовольствием следуют правилам игры ради радости победы по правилам (а не наперекор им). Как указывает Сьютс, ни читеры, ни зануды не имеют игривого настроя; читеры просто хотят победить, даже нарушая правила, а зануд не интересует победа в игре, и они могут следовать или не следовать правилам (часто они лишь хотят испортить веселье другим игрокам).

Как дизайнер, вы должны работать над играми, которые побуждают игроков поддерживать этот игривый настрой. Это означает, как мне кажется, что вы должны проявлять уважение к своим игрокам, а не пытаться заработать на них. В 2008 году мой коллега Брайан Кэш и я сделали два доклада на конференции Game Developers Conference на тему так называемых спорадических игр[3] — игр, в которые игрок играет урывками, в течение дня. Оба доклада были основаны на нашем опыте проектирования игры *Skyrates*[4] (название «скайраты» созвучно слову «пираты»),

---

[1] Николь Лазарро (Nicole Lazzaro) часто использует понятие триумфа в своих докладах на конференции GDC, говоря об эмоциях, управляющих игроками.

[2] Спасибо моим друзьям, Дональду Маккаскилу (Donald McCaskill) и Майку Вабшаллу (Mike Wabschall), которые познакомили меня с некоторыми тонкостями *Tekken 3* и за тысячи матчей, которые мы сыграли вместе.

[3] Cash, Bryan and Gibson, Jeremy. «Sporadic Games: The History and Future of Games for Busy People» (представлен в рамках саммита социальных игр Social Games Summit на конференции Game Developers Conference, Сан-Франциско, Калифорния, 2010). Cash, Bryan and Gibson, Jeremy «Sporadic Play Update: The Latest Developments in Games for Busy People» (представлен на конференции Game Developers Conference Online, Остин, Техас, 2010).

[4] *Skyrates* была написана за два семестра в 2006 году, когда мы были аспирантами центра развлекательных технологий в Университете Карнеги-Меллона. В разработке участвовали:

выпускного проекта, за который наша команда получила несколько наград за дизайн в 2008 году. Проектируя *Skyrates*, мы стремились сделать продолжительную онлайн-игру (подобную массовым многопользовательским онлайн-играм того времени, таким как *World of Warcraft* студии Blizzard), в которую могли бы играть занятые люди. В *Skyrates* у игроков есть роль воздушных каперов, взлетающих с плавучего острова, торгующих товарами и сражающихся с пиратами. Спорадический характер игры выражается в том, что игрок может периодически заглядывать в игру на несколько минут, отдавать распоряжения своему персонажу, поучаствовать в нескольких битвах с пиратами, усовершенствовать свой самолет или персонажа и затем позволить игре выполнять распоряжения, а самому заняться своими делами. В разные моменты времени игроку могут посылаться текстовые сообщения на телефон, информирующие о нападении на его персонажа, чтобы тот мог вернуться в игру и вступить в схватку или дать своему персонажу возможность провести бой самостоятельно.

Как дизайнеры игровой индустрии того времени, мы стали свидетелями появления в социальных сетях игр, таких как *FarmVille* и других, которые совсем или почти не уважали время своих игроков. Для игр в социальных сетях было обычным делом требовать (через механику) постоянного присутствия пользователя в игре в течение дня и наказывать их, если они не возвращались в игру вовремя. Это достигалось применением некоторых мерзких механик, главными из которых были *энергия и порча*.

В играх социальных сетей с энергией в качестве ресурса уровень энергии игрока медленно повышался с течением времени, независимо от того, играл он или нет. Но был определенный предел энергии, которую можно было накопить ожиданием, и этот предел часто был намного меньше, чем могло бы быть накоплено за день, и меньше, чем нужно для оптимальной игры. В результате игроки были вынуждены заходить в игру по несколько раз в день, чтобы израсходовать накопленную энергию и не потерять ничего из-за установленного предела. Конечно, игроки могли покупать дополнительную энергию, которая не заканчивалась и не ограничивалась сроком действия, и это способствовало росту продаж в этих играх.

Механику порчи проще всего объяснить на примере игры *FarmVille*, в которой игроки сажали сельскохозяйственные культуры и потом собирали урожай. В этой игре, если оставить урожай неубранным, через какое-то время он начинал портиться, и игрок терял свои инвестиции — семена и время, потраченное на уход за растениями. Культуры с высокой стоимостью портились намного быстрее, чем культуры

Ховард Брахам (Howard Braham), Брайан Кэш (Bryan Cash), Джереми Гибсон Бонд (Jeremy Gibson Bond), Чак Хувер (Chuck Hoover), Генри Клай Рейстер (Henry Clay Reister), Сет Шайн (Seth Shain) и Сэм Спиро (Sam Spiro), а художественное оформление персонажей выполнил Крис Дэниел (Chris Daniel). Нашими преподавателями были Джесси Шелл и доктор Дрю Дэвидсон (Drew Davidson). После выпуска *Skyrates* к нам примкнули разработчики Фил Лайт (Phil Light) и Джейсон Бакнер (Jason Buckner). Вы можете сыграть в игру по адресу http://skyrates.net.

с небольшой стоимостью, поэтому пристрастившиеся игроки вынуждены были возвращаться в игру в течение все более сокращающихся промежутков времени, чтобы получить максимальную отдачу от своих инвестиций.

Мы с Брайаном надеялись, что наши выступления на конференции GDC помогут остановить эти тенденции или хотя бы поспособствуют появлению альтернатив. Идея спорадической игры состоит в том, чтобы дать игроку наибольшее влияние (возможность выбора) за наименьшее время. Наш профессор, Джесси Шелл, однажды заметил, что *Skyrates* похожа на друга, который периодически предлагает оторваться от работы, но спустя несколько минут игры напоминает, что пора вернуться к работе. Такое уважительное отношение привело к тому, что уровень конверсии составил более 90 %, то есть в 2007 году более 90 % игроков, попробовавших сыграть в нее, становились постоянными пользователями.

Уважение игроков помогает удержать их в игривом настрое, а игривый настрой — это то, что поддерживает существование *магического круга*.

Магический круг

Как отмечалось в главе 2 «Методы анализа игр», в своей книге «Homo Ludens» (1938) Йохан Хейзинга предложил идею под названием *магический круг*. Магический круг — это область, где протекает игра, которая может иметь ментальные или физические границы или и те и другие. Внутри магического круга господствуют правила игры, и суммы, которыми поощряются или не поощряются определенные действия, отличаются от реального мира.

Например, когда два друга играют в покер друг против друга, они часто блефуют (то есть лгут), демонстрируя друг перед другом уверенность, что у них хорошие комбинации карт и они с легкостью возьмут банк. Однако за рамками игры эти же друзья могут считать ложь недопустимой в дружеских отношениях. Аналогично, в хоккее с шайбой на льду игроки широко применяют силовые приемы друг к другу (в пределах правил, конечно), но за пределами игры те же игроки будут пожимать друг другу руки, а некоторые даже могут быть хорошими друзьями.

Ян Богост и многие другие теоретики игры отмечали, что магический круг — это неустойчивая и недолговечная штука. Даже дети понимают это и иногда объявляют «тайм-аут» в своих играх понарошку. Тайм-аут в этом случае означает приостановку действия правил и временное прекращение существования магического круга, часто для того, чтобы игроки могли договориться о правилах на оставшуюся часть игры. Когда обсуждение заканчивается, объявляется возобновление игры, после чего игра и магический круг продолжают действовать с того места, на котором они были прерваны.

Несмотря на то что магический круг можно приостановить и возобновить, восстановить целостное восприятие магического круга после долгой паузы иногда бывает непросто. Во время длительных задержек в футбольном матче (например, когда игра приостанавливается на 30 минут из-за погодных условий в середине

второго периода) комментаторы часто обсуждают, насколько сложно игрокам поддерживать в себе настрой на игру во время задержки или вернуть его после возобновления игры.

Потоковое состояние

Американский психолог Михай Чиксентмихайи (Mihaly Csíkszentmihályi) описывал *потоковое состояние* как состояние оптимального напряжения. Оно часто обсуждается на конференции Game Developers Conference, потому что тесно связано с тем, что пытаются создать многие игровые дизайнеры в своих играх. Находясь в потоковом состоянии, игрок сосредоточен на стоящей перед ним задаче и часто теряет связь с происходящим вокруг него. Возможно, вы ощущали это на себе, когда играли или работали настолько увлеченно, что казалось, будто время меняет свой ход, начиная идти быстрее или медленнее.

Потоковое состояние было темой диссертации Дженовы Чена (Jenova Chen) на степень магистра изящных искусств в Южно-Калифорнийском университете, а также предметом его дипломной игры, названной соответственно *Flow*[1]. Дженова также рассказывал об этой идее в своих докладах на конференции GDC.

Как показано на рис. 8.1, потоковое состояние находится на границе между скукой и разочарованием. Если игра окажется слишком сложной для игрока, он расстроится; напротив, если она окажется слишком простой, — игрок заскучает.

Как отмечали Жанна Накамура и Михай Чиксентмихайи в своей совместной статье «The Concept of Flow», опубликованной в 2002 году, потоковое состояние свойственно всем людям, независимо от культуры, пола, возраста и вида деятельности, и базируется на двух условиях[2]:

- Воспринимаемые сложности или возможности для действий заставляют напрягать силы и умения (но не превосходят и не недоиспользуют их); то есть когда создается ощущение, что сложности соответствуют возможностям.
- Ясные ближайшие цели и немедленная обратная связь, демонстрирующая движение вперед.

Это в значительной степени созвучно обсуждению потокового состояния в области проектирования игр. Оба условия достаточно конкретны, чтобы дизайнеры могли понять, как их реализовать в своих играх, а путем тщательного тестирования и бесед с игроками вы легко сможете оценить, позволяет ли ваша игра войти в это состояние.

---

[1] Сыграть в оригинальную Flash-версию игры *Flow* можно по адресу http://interactive.usc.edu/projects/cloud/flowing/. Обновленная и расширенная версия для PlayStation 3 (и PS4) доступна в PlayStation Store.

[2] Jeanne Nakamura and Mihaly Csíkszentmihályi, «The Concept of Flow». *Handbook of Positive Psychology* (2002): 89–105, 90.

Рис. 8.1. Потоковое состояние как его описывал Чиксентмихайи

После 1990 года, когда Чиксентмихайи опубликовал свою книгу «Flow: The Psychology of Optimal Experience»[1], дополнительные исследования расширили наше понимание потокового состояния, и была выявлена одна важная деталь, имеющая отношение к играм: теперь дизайнеры понимают, что поддержание потокового состояния истощает игрока. Даже при том, что потоковое состояние доставляет удовольствие игроку — и игровые моменты в этом состоянии являются наиболее запоминающимися, — нахождение в потоковом состоянии дольше 15–20 минут вызывает утомление. Кроме того, если игрок всегда находится в идеальном потоковом состоянии, у него не будет возможности заметить улучшение навыков. Поэтому для большинства игроков предпочтительнее выглядит схема пребывания в потоковом состоянии, подобная изображенной на рис. 8.2.

Вдоль границы между потоковым состоянием и скукой простирается область, находясь в которой игрок чувствует себя ловким и умелым (то есть чувствует себя потрясающе!), и ему это действительно нужно. Несмотря на то что потоковое состояние доставляет удовольствие игрокам, позволяйте им выходить из него, чтобы они могли осмыслить свои достижения в потоке, — это тоже очень важно. Вспомните самую лучшую игру с поединками, в которую вам доводилось играть. Находясь в потоковом состоянии, вы, по определению, теряете связь со всем находящимся

[1] «Поток. Психология оптимального переживания». — *Примеч. пер.*

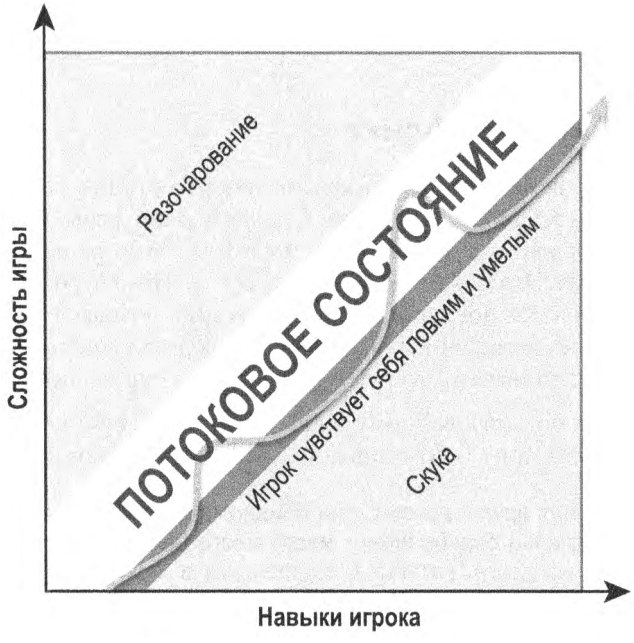


Рис. 8.2. Усовершенствованное потоковое состояние

за пределами момента, потому что потоковое состояние требует полной концентрации внимания. И только выйдя победителем из поединка, вы получали время, чтобы перевести дух и осознать, насколько потрясающим был бой. Игрокам нужны не только моменты в потоковом состоянии, но и моменты, чтобы насладиться возросшим мастерством.

Отличного баланса, например, удалось достичь игре *God of War*. Она последовательно ведет игрока к противнику нового типа, и это часто воспринимается как преодоление очередного рубежа, потому что игрок еще не имеет стратегии победы над этим типом врага. В конце концов игрок находит результативную стратегию для данного типа врага и в течение нескольких поединков с единственным противником совершенствует свое мастерство. Затем, несколькими минутами позже, игрок ставится перед необходимостью сразиться сразу с несколькими врагами этого типа, но, благодаря улучшенным навыкам, этот бой оказывается легче, чем самый первый бой с одним врагом этого типа. Способность легко выходить победителем из поединка сразу с несколькими врагами, которые прежде в одиночку доставляли немало хлопот, наглядно демонстрирует возросшее мастерство и вызывает у игрока потрясающее чувство.

Проектируя свои игры, помните, что хорошая игра дает игроку не только задачи оптимальной сложности, но также понимание, что он становится лучше, и выделяет ему время, чтобы испытать потрясающее чувство. После тяжелого боя дайте

игроку немного времени почувствовать себя сильным. Это стимулирует появление чувства раскрепощения.

Организованный конфликт

Как рассказывалось в главе 1 «Думать как дизайнер», организованный конфликт — одна из потребностей человека, которые удовлетворяют игры. Одно из основных отличий между забавой и игрой заключается в том, что игра всегда предполагает борьбу или конфликт. Это может быть конфликт с другими игроками или с игровой системой (см. раздел «Отношения между игроками» в главе 4 «Фиксированный уровень»). Этот конфликт дает игрокам шанс проверить свои умения (или умения своей команды), выступив против других игроков, системы, шансов или самого себя.

Такое стремление к организованному конфликту также проявляется в играх животных. Как отмечает Крис Бейтман в книге «Imaginary Games»:

> Когда щенки играют друг с другом, они придерживаются строго ограниченного поведения. В шутливой борьбе имеют место много нежных покусываний, щенки взбираются друг на друга, катаются, сцепившись в клубок и притворно рыча; таковы правила[1].

Даже некоторые войны велись по правилам, напоминающим правила игры. В биографии Алик-чея-ахуш (Много Подвигов), вождя племени кроу в Северной Америке, упоминаются некоторые правила подсчета «ку» (подвигов) в бою. За подвиг считалось любое опасное действие на поле битвы. Касание вооруженного и полного сил противника шестом для «ку», коротким хлыстом или луком, перед тем как нанести ему вред; захват оружия у еще живого врага; кража лошадей или оружия из вражеского лагеря; и касание первого павшего врага (до его смерти) — все это считалось «ку», или подвигами. Если при этом воин не получал ранений, его подвиг ценился еще выше. Много Подвигов также рассказывал о правилах двух символических шестов в племенных общинах.

> Один из шестов — прямой и украшен орлиным пером на тонком конце. Если в битве носитель шеста втыкал его в землю, он не должен был отступать или оставлять шест. Он должен сбросить свою накидку [умереть], если только между ним и его противником не окажется его собрат по общине. Тогда он мог с честью перенести шест, но пока тот был воткнут в землю, он представлял страну Кроу. Носители изогнутых шестов, украшенных двумя перьями, могли перемещать их по своему усмотрению для обозначения своей позиции и должны были умереть, если враг захватывал их. При подсчете «ку» в любой из таких общин число «ку», заработанных носителями, удваивалось — каждый «ку» засчитывался как два, потому что считалось, что их жизни находились в большей опасности[2].

---

[1] Christ Bateman, *Imaginary Games* (Washington, USA: Zero Books, 2011), 24.

[2] Frank Bird Linderman, *Plenty-Coups, Chief of the Crows*, New ed. (Lincoln, NE: University of Nebraska Press, 2002), 31–32.

После битвы происходил подсчет «ку», и каждый воин рассказывал о своих подвигах. Успешно совершивший «ку» и благополучно избежавший ран получал орлиное перо, которое он мог носить в волосах или прикрепить к своему шесту для «ку». Если воин получал ранение, перо окрашивалось в красный цвет.

Подсчет «ку» среди индейцев Равнин Америки придавал дополнительный смысл войнам между народами и обеспечивал структурированный способ проявления храбрости на поле боя и получение повышенного уважения после битвы.

Многие из современных популярных игр предоставляют организованный конфликт между командами игроков, включая все командные виды спорта (футбол, крикет и баскетбол, считающиеся наиболее популярными во всем мире), а также онлайн-состязания между командами, такие как *League of Legends*, *Team Fortress 2* и *Overwatch*. Но даже в отсутствие команд игры в целом дают игрокам возможность участвовать в конфликтах и преодолевать невзгоды.

Раскрепощение

В разделе о потоковом состоянии рассказывалось об одном из видов раскрепощения (ощущение ловкости и могущества в игровом мире). В этом разделе мы рассмотрим другой вид: свобода выбора того, что можно делать в игре. Я имею в виду две разновидности свободы: автотелическую (autotelic) и перформативную (performative).

Автотелическая свобода

Термин *автотелический* происходит от двух латинских слов: сам (auto) и цель (telos). Личность считается автотелической, когда она сама определяет цели для себя. Когда Чиксентмихайи только начинал развивать свою теорию потокового состояния, он знал, что важную роль в нем играет автотелизис (autotelisis). Согласно его исследованиям, автотелические личности получают больше удовольствия от потокового состояния, тогда как другие (предпочитающие не устанавливать свои цели) получают больше удовольствия от простых ситуаций, когда они убеждаются, что их уровень мастерства выше уровня сложности предложенной задачи[1]. Чиксентмихайи полагал, что автотелическая индивидуальность — это то, что помогает человеку обрести счастье в жизни независимо от ситуации[2].

Итак, какие игры развивают автотелическое поведение? Одним из фантастических примеров может служить *Minecraft*. В этой игре игрок помещается в случайно сгенерированный мир, и перед ним стоит только одна реальная цель — выжить. (Зомби и другие монстры будут атаковать игрока по ночам.) Но при этом ему дается возможность добывать природные ресурсы и использовать их для созда-

[1] Nakamura and Csíkszentmihályi, «The Concept of Flow», 98.

[2] Mihaly Csíkszentmihályi, *Flow: The Psychology of Optimal Experience*; New York: Harper & Row, 1990, 69. (*Чиксентмихайи Михай*. Поток: Психология оптимального переживания. М.: Альпина нон-фикшн, 2018. — *Примеч. пер.*)

ния инструментов и построек. Игроки в *Minecraft* строят не только замки, мосты и полноценные копии звездолета Star Trek Enterprise NCC-1701D, но и американские горки, протянувшиеся на многие километры, и даже простые компьютеры с ОЗУ[1]. В этом истинный гений *Minecraft*: она дает игрокам возможность выбирать свой путь и предоставляет гибкую игровую систему, позволяющую сделать этот выбор.

Большинство игр не такие гибкие, как *Minecraft*, но они тоже дают возможность подступиться к проблеме с нескольких сторон. Одна из причин потери популярности текстовых приключенческих игр (таких, как *Zork*, *Planetfall* и *The Hitchhiker's Guide to the Galaxy* компании Infocom) и последовавших за ними приключенческих игр, управляемых мышкой (таких, как начало серий игр *King's Quest* и *Space Quest*, выпущенных компанией Sierra OnLine) в том, что они часто предполагали только один (нередко туповатый) подход к большинству проблем. В *Space Quest II*, если вы не захватили плавки из случайного шкафчика в самом начале игры, много позднее вы не сможете использовать их как пращу и вам придется начать прохождение с самого начала. В игре *The Hitchhiker's Guide to the Galaxy* компании Infocom, когда бульдозер подъезжает к вашему дому, вы должны лечь в грязь перед ним и затем трижды ввести `"wait"`. Если не сделать *именно так*, вы погибнете и вынуждены будете перезапустить игру[2]. Сравните это с более современными играми, такими как *Dishonored*, где почти каждая проблема имеет хотя бы одно насильственное и одно ненасильственное решение. Возможность выбора способа достижения цели увеличивает заинтересованность игрока в игре и его ответственность за успех[3].

Перформативная свобода

Другой вид свободы, не менее важной для игр, — перформативная свобода. В книге «Game Design Theory» Кит Бургун отмечает, что дизайнеры игр не только сами творят искусство, но и дают игрокам возможность творить свое искусство. Создателей пассивных медиа можно сравнить с композиторами: они создают что-то, что потом будет потребляться аудиторией. Но вы, как геймдизайнер, на самом деле находитесь где-то между композитором и мастером, создающим инструменты. Вы не просто пишете ноты, которые будут играть другие, вы также создаете

[1] http://www.escapistmagazine.com/news/view/109385-Computer-Built-in-Minecraft-Has-RAM-Performs-Division.

[2] Одна из главных причин, почему был реализован такой сценарий: многократное возрастание объема контента, если игроку разрешить делать что-то еще в сюжете игры. Из того, что я видел, ближе всех к по-настоящему открытому, ветвящемуся повествованию была интерактивная драма *Façade* Майкла Матеаса (Michael Mateas) и Эндрю Штерна (Andrew Stern).

[3] При этом вы должны помнить о стоимости и времени разработки. При неосмотрительном подходе каждый вариант, который вы дадите игроку, может увеличить стоимость разработки и в смысле денег, и в смысле времени. Как дизайнер и разработчик вы должны найти приемлемый баланс.

инструменты, с помощью которых они смогут творить. Прекрасным примером таких игр может служить *Tony Hawk's Pro Skater*, где игроку дается богатый набор базовых движений, которые он может комбинировать и объединять в гармоничные трюки для получения максимальной оценки. Так же как виолончелист Йо Йо Ма (Yo-Yo Ma) является художником, игрок тоже может стать художником, если дизайнер создаст игру, в которую можно играть на художественном уровне. То же можно наблюдать в других играх, поддерживающих богатые наборы базовых движений или линий поведения, таких как игры-поединки или игры-стратегии в реальном времени.

Внимание и вовлеченность

Как упоминалось выше в этой главе, фантастический игровой дизайнер Ричард Лемарчанд говорил о внимании в своем докладе «Attention, Not Immersion: Making Your Games Better with Psychology and Playtesting, the Uncharted Way» на конференции GDC 2012. Его целью было показать распространенную ошибку в игровом дизайне — использование понятия *погружение* вместо *привлечения и удержания внимания* аудитории, из которых последнее точнее отражает то, к чему стремятся дизайнеры игр.

До выступления Лемарчанда многие дизайнеры стремились увеличить степень погружения игроков в свои игры. Это повлекло за собой, например, сокращение или удаление приборных панелей с экранов и уменьшение количества элементов, которые могли бы вывести игрока из состояния погружения в игру. Но, как отметил Лемарчанд в своем докладе, игроки никогда не достигают настоящего погружения, да и не желают этого. Если бы игрок действительно считал, что он Натан Дрейк, и на полпути *Uncharted 3* в него стреляли бы, пока он, цепляясь за грузовую сетку, пытается забраться в транспортный самолет, летящий в тысячах футов над пустыней, он испытал бы абсолютный ужас! Один из важнейших аспектов магического круга состоит в том, что вход и пребывание в нем — это выбор игрока, и игрок всегда знает, что его участие в игре является добровольным. (Как отметил Сьютс, когда участие перестает быть добровольным, действо перестает быть игрой.)

Вместо погружения Лемерчанд стремится сначала привлечь внимание игрока, а потом удержать его. В данном случае я использую слово *внимание* для описания непосредственно интереса и слово *вовлеченность* для описания долговременного интереса, который нужно удерживать (это различие теперь признает и Лемерчанд). Лемерчанд также различает *рефлекторное внимание* (непроизвольный отклик на внешнее воздействие) и *управляющее внимание* (которое возникает, когда мы решаем обратить внимание на что-то).

По его словам, элементы красоты, эстетики и контраста отлично подходят для привлечения внимания. Именно поэтому фильмы о Джеймсе Бонде всегда начинаются со сцены действия. Она начинается в разгар событий (в середине происходящего), потому что это создает заметный контраст между скукой, которую испытывает зритель, сидя в зале в ожидании начала фильма, и волнением, возникающим с первых

сцен. В этот момент происходит захват рефлекторного внимания, такой перенос внимания жестко заложен в нас эволюцией. Когда мы замечаем что-то краем глаза, мы тут же невольно переключаем свое внимание на этот объект. Затем, после привлечения первоначального внимания, фильмы о Бонде начинают демонстрировать довольно скучные сцены, подготавливающие почву для остальной части фильма. Поскольку внимание зрителя уже привлечено к фильму, он решает использовать управляющее внимание, чтобы просмотреть эти сцены.

В книге «The Art of Game Design» Джесси Шелл представил свою теорию *кривой изменения интереса*. Кривая интереса описывает захват и удержание внимания и, по мнению Шелла, оптимальная кривая интереса должна выглядеть как показано на рис. 8.3.

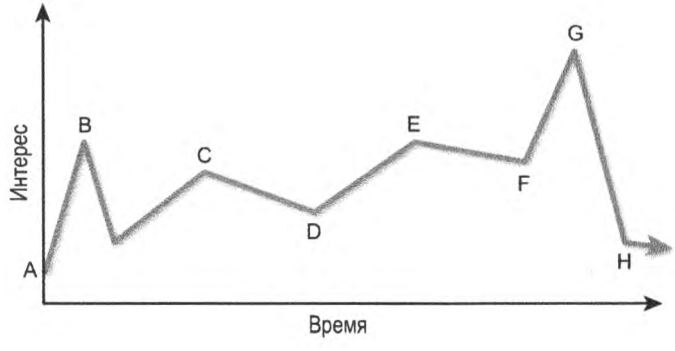


Рис. 8.3. Кривая интереса из книги Джесси Шелла

Как показывает оптимальная кривая интереса Шелла, изначально аудитория имеет низкий уровень интереса (A), и затем вы быстро захватываете его с помощью «зацепки», вследствие чего образуется пик интереса (B). Добившись начального интереса, вы можете сбросить вниз и потом постепенно повышать его, создавая невысокие пики и локальные точки минимума (C, E и D, F соответственно), до высшей точки интереса: кульминации (G). После кульминации интерес аудитории падает в развязке (H), потому что повествование подходит к концу. В действительности эта кривая интереса очень напоминает стандартную драматическую кривую трехактной структуры Сида Филда, описанную в главе 4 «Фиксированный уровень», и как показывает практика, она хорошо работает на интервалах времени от нескольких минут до пары часов. Шелл утверждает, что для охвата более длинных интервалов времени кривую интереса можно повторять на манер фракталов. Например, в большой игре можно создать структуру миссий, каждая из которых будет иметь свою оптимальную кривую интереса внутри более протяженной кривой интереса всей игры. Однако на самом деле все намного сложнее, чем описано здесь, потому что интерес, который обсуждает Шелл, — это то, что я называю *вниманием*, кроме которого мы должны также учитывать вовлеченность, если хотим надолго удержать интерес игрока.

При близком рассмотрении внимания и вовлеченности легко обнаружить, что внимание непосредственно связано с рефлекторным вниманием (невольный отклик), тогда как вовлеченность — исключительно добровольное/управляющее внимание. На рис. 8.4 изображена диаграмма, отражающая синтез идей Лемарчанда и моего личного опыта как дизайнера и игрока.

Рис. 8.4. Четыре элемента многоуровневой тетрады, плюс сообщество, в отношении внимания и вовлеченности (так как технология остается почти невидимой для игрока, на этом графике она обозначена чисто символически)

Как видно на диаграмме, эстетика (эстетический элемент в тетраде) — лучшее средство захвата внимания, которое в данном случае является в основном рефлекторным. Это объясняется тем, что эстетика воздействует непосредственно на наши чувства и привлекает внимание.

Сюжет и механика требуют управляющего внимания. Как отмечает Лемарчанд, сюжет обладает большой способностью привлекать наше внимание, но я не согласен с утверждением Лемарчанда и Джейсона Рорера, что механика обладает большей способностью поддерживать вовлеченность, чем сюжет. Одиночный фильм обычно длится пару часов, то же самое можно сказать в отношении механики одного сеанса игры. Кроме того, на своем личном опыте я убедился, что так же как хорошая механика может удерживать мою вовлеченность более 100 часов, серия сюжетов может удерживать мое внимание на протяжении 100 и более серий телевизионного сериала. Главное отличие между механикой и сюжетом в том, что сюжет должен постоянно развиваться, тогда как механика процесса игры может существовать в неизменном виде годами и все еще удерживать интерес благодаря смене обстоятельств в игре (примером может служить пожизненная приверженность игрока к шахматам или Го).

Единственное, что, по моим наблюдениям, дает еще большую вовлеченность, чем сюжет и механика, — это сообщество. Когда люди обнаруживают, что игра, фильм или род занятий имеют сообщество, и они чувствуют себя частью этого сообщества, они долго еще будут продолжать принимать участие в нем после того, как сюжет или

механика потеряют свое влияние на них. Именно сообщества еще долго поддерживали существование многих гильдий в *Ultima Online* после того, как большинство людей перекочевало в другие игры. Кроме того, когда перед членами сообщества наконец вставал вопрос, куда двигаться дальше, они часто выбирали сообщество и все вместе переходили в новую игру, чтобы продолжать играть вместе, сохраняя одно и то же сообщество на протяжении нескольких онлайн-игр.

Осмысленные решения

По словам Сида Мейера, как было написано в главе 1, игра — это серия интересных решений, и с того момента мы не раз пытались выяснить значение понятия *интересные*. До настоящего момента было представлено несколько идей, которые могут помочь разобраться в этом.

Идея *осмысленной игры* Кэти Сален и Эрика Циммермана, представленная в главе 5 «Динамический уровень», помогает кое-что понять. Решение, имеющее смысл, должно быть *различимым* и *интегрированным*[1]:

- **Различимость:** у игрока должна быть возможность уверенно сказать, что игра приняла и поняла его решение (есть немедленная обратная связь).
- **Интегрированность:** игрок должен быть уверен, что его решение имеет некоторое влияние на долгосрочный исход игры (то есть долгосрочное влияние).

В своем определении игры Кит Бургун указывает на важность *неоднозначных* решений:

- **Неоднозначность:** решение является неоднозначным для игрока, если он может догадываться, как можно повлиять на систему, но никогда не уверен точно. Решение о размещении денег на фондовом рынке — неоднозначное. Вы, как опытный инвестор, можете довольно четко представлять, какие акции будут расти, а какие падать, но рынок настолько переменчив, что вы никогда не можете точно знать этого.

Почти все интересные решения одновременно являются *обоюдоострыми*:

- **Обоюдоострый:** решение называют обоюдоострым, когда оно имеет не только достоинства, но и недостатки. В предыдущем примере с инвестициями в фондовый рынок достоинство — это долгосрочный потенциал для зарабатывания денег, а недостаток — немедленная потеря ресурса (денег), используемого для покупки акций, а также потенциальная возможность падения акций на рынке.

Еще один аспект, делающий решения интересными, — *новизна*.

- **Новизна:** решение считается новым, если достаточно отличается от других недавних решений игрока. В классической японской ролевой игре *Final Fantasy VII*

[1] Katie Salen and Eric Zimmerman, *Rules of Play* (Cambridge, MA: MIT Press, 2003), 34.

характер боя с конкретным противником мало меняется на протяжении каждой битвы, а это означает, что игроку нет нужды принимать новые решения. Если у противника недостаточно сил, чтобы порождать огонь, а у игрока достаточно маны и магии огня, он обычно в каждом раунде атакует врага огнем, пока не повергнет его. Напротив, превосходные бои в *Grandia III* делают позиционирование и местоположение важным для многих специальных выпадов, но персонажи игрока перемещаются по игровому полю автономно (независимо от ввода игрока). Всякий раз, когда игрок получает возможность принять решение, время для него останавливается, и он должен оценить позиции союзников и врагов, чтобы принять решение. Такое автономное перемещение персонажей и важность местоположения делают необходимым принятие новых решений.

Последнее требование к интересным решениям — они должны быть ясными.

- **Ясность:** несмотря на важность неоднозначности результатов выбора, сам выбор должен быть четким и ясным. Отсутствие ясности выбора можно охарактеризовать по-разному:
  - Выбор может быть неясным, если в данный момент есть слишком много его вариантов; игрок может с трудом понимать их различия. Это ведет к *параличу выбора*, невозможности сделать выбор из-за слишком большого количества вариантов.
  - Выбор может быть неясным, если игрок не понимает, каким будет вероятный результат выбора. Это часто было проблемой в древовидной структуре диалогов некоторых старых игр, которые в течение многих лет просто перечисляли возможные предложения, которые игрок мог бы произнести, без любой дополнительной информации о предполагаемом значении этих предложений. Напротив, колесо диалога в *Mass Effect* включает информацию о дружественности или неприветливости предложений, которые мог бы сказать игрок, и о том, как они могут повлиять на продолжительность диалога. Эта информация позволяет игроку выбрать вариант отношения, а не конкретные слова, и устранить двусмысленность из древовидной структуры диалога.
  - Выбор также может быть неясным, если игрок не понимает важности выбора. Одно из самых больших достижений *Grandia III* в сравнении с *Grandia II* — возможность персонажей автоматически звать на помощь другого персонажа, когда наступает его время сделать ход. Если персонаж А атакован и персонаж В может помочь ему своим ходом, персонаж А попросит помощи, когда для персонажа В наступит время хода. Игрок, управляющий персонажем В, все еще может выбрать сделать что-то другое и отказать в помощи, тем не менее игра ясно дает понять, что это был последний шанс предотвратить нападение на А.

Мы можем объединить все шесть аспектов вместе и получить довольно полное понимание, что именно делает решение интересным. Интересное решение является различимым, интегрированным, неоднозначным, обоюдоострым, новым и ясным. Сделав решения более интересными, вы сможете увеличить привлекательность

своей механики и тем самым обеспечить долгосрочную вовлеченность игрока в вашу игру.

Эмпирическое понимание

Последняя цель для игроков, которую мы обсудим в этой главе, — *эмпирическое понимание*. Это цель проектирования, намного более доступная для дизайнеров игр, чем для любых других средств информации.

В 2013 году критик и теоретик игр Мэтти Брис выпустила *Mainichi*[1], первую игру, которую она спроектировала и разработала (рис. 8.5).

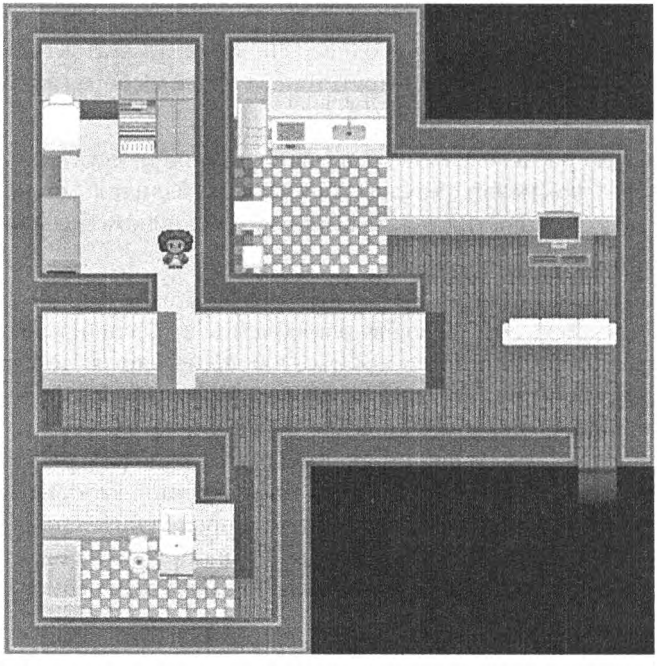


Рис. 8.5. Игра *Mainichi*, созданная Мэтти Брис (2013)

Согласно описанию Брис, игра *Mainichi* — это личное письмо от нее другу, чтобы дать ему понять, какова ее повседневная жизнь. В реальной жизни Брис — женщина-трансгендер, которая в то время жила в районе Кастро в Сан-Франциско. В *Mainichi* игрок принимает на себя роль Мэтти Брис и должен подготовиться к встрече с другом в кафетерии: что надеть, какой макияж наложить, стоит ли перекусить? Каждое из этих решений меняет то, как некоторые (не все) люди в городе реагируют на нее, когда она идет в кафетерий и делает заказ. Даже такое

[1] Игра Mainichi доступна для загрузки по адресу http://www.mattiebrice.com/mainichi/.

простое решение, как расплачиваться — наличными или кредитной картой, — имеет значение. (Попытка расплатиться кредитной картой приводит к тому, что бариста будет называть вас: «Мисс... эм-м... мистер Брис», потому что прочитает прежнее мужское имя Брис на ней.)

Игра очень короткая, и вы, как игрок, пытаетесь пробовать снова и снова, чтобы посмотреть, что получится при выборе других вариантов на всем протяжении игры. Поскольку решения игрока изменяют поведение персонажа Мэтти, вы чувствуете себя соучастником в формировании хорошего или плохого отношения окружающих к ней. Диаграмма ветвления или история, организованная подобно фильму «День сурка» (в котором главный герой Билла Мюррея должен сотни раз проживать один и тот же день, пока не найдет верный путь), могла бы передать ту же информацию о большом влиянии мелочей, из которых Брис выбирает каждый день, но она не смогла бы переложить чувство ответственности на аудиторию. На этот раз только через игру (будь то видеоигра, игра «понарошку» или ролевая игра) человек может почувствовать себя в шкуре другого и получить представление о последствиях принимаемых им решений. Такое эмпирическое понимание — одна из самых интересных целей, к которым мы можем стремиться как дизайнеры игр.

Итоги

У каждого, кто занимается созданием игр, свои представления о каждой из целей, рассмотренных в этой главе. Один хочет, чтобы игра была забавной, другой — предложить игрокам любопытные головоломки, третьему важно, чтобы игроки глубоко задумались над чем-то, а четвертый пожелает создать арену, на которой игроки будут наделены свободой выбора. Но какими бы ни были причины и мотивы, побуждающие вас создавать игры, теперь пришло время наконец-таки начать их создавать.

В следующих двух главах рассказывается о прототипировании игр на бумаге и их тестировании. Прототипирование и тестирование вместе представляют основу реальной работы по проектированию игр. Почти в любой игре, и особенно в цифровой, существуют сотни параметров, которые можно подстраивать, чтобы влиять на восприятие игры. Однако в цифровых играх даже небольшие изменения могут потребовать значительного времени на реализацию. Стратегии бумажного прототипирования, представленные в следующей главе, помогут вам быстро переходить от идеи к действующему (бумажному) прототипу и затем еще быстрее переходить от одного прототипа к другому. Во многих случаях этап прототипирования на бумаге в цифровой разработке может сэкономить массу времени, потому что прежде, чем написать хоть строку кода, вы уже обнаружите лучший вариант, пройдя несколько пробных игр на бумажных прототипах.

9 Прототипирование на бумаге

В этой главе вы узнаете о прототипировании на бумаге, одном из лучших инструментов в арсенале дизайнеров игр для быстрой проверки и развития идей. Несмотря на простоту, прототипирование на бумаге поможет вам узнать самые разные особенности вашей игры, даже если эта игра впоследствии станет цифровой.

К концу этой главы вы узнаете, как применять приемы прототипирования на бумаге и исследовать части цифровой игры, которые проще смоделировать и проверить на бумаге.

Преимущества прототипирования на бумаге

Несмотря на то что цифровые технологии позволили создать совершенно новый мир возможностей для разработки игр, многие дизайнеры по-прежнему предпочитают опробовать начальные идеи с использованием традиционных методов прототипирования на бумаге. Это может показаться странным, если учесть, что компьютеры способны выполнить расчеты и отобразить результаты на экране намного быстрее, чем человек сделает то же самое вручную. Объяснение заключено в двух причинах: скорость и простота реализации. Эти два фактора влекут за собой несколько преимуществ, в том числе:

- **Начальная скорость разработки:** по оперативности реализации игры ничто не превзойдет бумагу. Вы можете объединить несколько кубиков, карточек для заметок и другие простые предметы, чтобы организовать игру за очень короткое время. Даже при наличии огромного опыта проектирования игр, чтобы создать новый проект цифровой игры, придется потратить намного больше времени, особенно если он значительно отличается от всего, что вы делали раньше.
- **Скорость итераций:** изменения в бумажные игры вносятся очень быстро; фактически некоторые изменения можно вносить прямо в процессе игры. Благодаря простоте внесения изменений, прототипы на бумаге как нельзя лучше подходят для мозгового штурма на начальном этапе подготовки проекта (когда часто вносятся крупные изменения). Если в прототипе на бумаге что-то не получается, внесение изменений может занять всего несколько минут.

- **Техническая простота:** так как для создания прототипа на бумаге не требуется больших технических знаний или художественного таланта, любой из команды разработчиков может принять участие в этом процессе. Это поможет не упустить отличные идеи, рождающиеся у членов вашей команды, которые не смогли бы внести свой вклад в цифровой прототип.
- **Совместное прототипирование:** благодаря технической простоте и высокой скорости итераций, прототипы на бумаге можно создавать и изменять коллективно, способами, пока невозможными для цифровых прототипов. Группа людей — членов вашей команды — может совместно работать над бумажным прототипом и быстро обмениваться идеями. Как дополнительное преимущество, такое участие членов команды в процессе проектирования поможет увеличить их вовлеченность в проект и создать потрясающе сплоченную команду.
- **Сосредоточенность на прототипировании и тестировании:** даже новичку очевидно, что бумажный прототип цифровой игры будет существенно отличаться от законченного цифрового продукта. Это позволяет тестировать конкретные элементы игры, не зацикливаясь на деталях. В 1980-х внутренний документ для дизайнеров пользовательских интерфейсов в Apple Computer рекомендовал создавать грубые эскизы кнопок для их интерфейсов на бумаге, сканировать эти рисунки и затем создавать прототипы пользовательских интерфейсов с использованием сканированных изображений. Поскольку наброски и сканированные изображения элементов интерфейса, таких как кнопки и меню, очевидно не были окончательными вариантами, которые Apple могла бы выбрать, тестировщики не зацикливались на внешнем виде кнопок и сосредоточивали основное внимание на удобстве интерфейса, что Apple интересовало больше всего. Бумажный прототип может помочь направить внимание ваших тестировщиков на конкретный игровой аспект, который вы собирались проверить, чтобы он не зависел от внешнего вида прототипа.

Инструменты прототипирования на бумаге

Есть несколько инструментов для прототипирования на бумаге, которыми вы, возможно, захотите овладеть. Бумажный прототип можно сделать практически из чего угодно, но некоторые инструменты помогают ускорить этот процесс:

- **Большие листы бумаги:** в большинстве магазинов канцелярских товаров можно купить большие листы бумаги (около 60 см в ширину и 90 см в длину). Часто такие листы продаются собранными в своеобразные блокноты, а иногда у них может быть клейкая обратная сторона — для крепления на стенах и других поверхностях. Часто также можно найти большие листы бумаги с нарисованной квадратной или шестиугольной сеткой. Подробнее о том, когда могут пригодиться такие листы с сеткой и как выполнять свободное перемещение на открытом игровом поле, рассказывается во врезке «Перемещение в сетках разных видов».

- **Игровые кости:** у многих людей дома найдутся несколько игровых костей d6 (обычных шестигранных кубиков). Вам, как дизайнеру игр, пригодятся также некоторые необычные разновидности игровых костей. В местном магазине игрушек вы наверняка найдете наборы костей для ролевых игр, включающие 2d6 (два шестигранных кубика), 1d8, 1d12, 1d20 и процентильные кости (2d10, одна из которых размечена очками 0–9, а другая 00–90; они бросаются вместе и дают число в диапазоне от 00 до 99). В главе 11 «Математика и баланс игры» вы найдете много информации о разнообразных игровых костях и о том, как выглядит их пространство вероятностей. Например, при броске кубика 1d6 у вас равные шансы получить любое число от 1 до 6, но с двумя такими кубиками у вас есть шесть разных вариантов получить 7 очков (вероятность 6/36) и только один вариант получить 12 (вероятность 1/36).
- **Карточки:** карточки — фантастический инструмент для прототипирования благодаря своей податливости и универсальности. Создайте карточки с числами 1–6, и вы получите колоду 1d6. Если тщательно перетасовывать эту колоду перед каждой попыткой вытащить карточку, она будет действовать подобно игральной кости 1d6, но если вытягивать из колоды все карточки перед перемешиванием, вы гарантированно получите каждую цифру из 1, 2, 3, 4, 5 и 6, прежде чем увидите любую из них во второй раз.
- **Протекторы карт:** во многих магазинах игрушек продают протекторы карт разного вида. Первоначально протекторы были придуманы для защиты бейсбольных карточек и получили распространение в игровой индустрии с появлением коллекционных карточных игр, таких как *Magic: The Gathering* в 90-х. Каждый карточный протектор — это пластиковая обложка для отдельной карточки, внутри которой достаточно места и для обычной карточки, и для полоски бумаги. Протекторы отлично подходят для прототипирования: вы можете напечатать карточки для прототипа на обычной бумаге и затем засунуть их в обложки перед обычными игральными картами. Игральная карта придаст жесткости, благодаря которой вы сможете легко перемешивать карточки, и вам не придется тратить время и деньги на печать карточек. Протекторы для карточек также обеспечат одинаковый внешний вид со стороны рубашки, кроме того, несколько наборов протекторов карт можно использовать для раздельного хранения разных колод карт.

ПЕРЕМЕЩЕНИЕ В СЕТКАХ РАЗНЫХ ВИДОВ

Если игра предполагает перемещение игрока, необходимо решить, как будут происходить эти перемещения по игровому полю. Как видно в части A рис. 9.1, ход по диагонали на одну клетку в квадратной сетке почти на 50 % длиннее, чем ход по горизонтали или по вертикали. (Согласно теореме Пифагора, длина диагонали составляет $\sqrt{2}$, или примерно 1,414.) Однако в шестиугольной сетке ходы в любые смежные клетки имеют одинаковую длину (часть B на рис. 9.1).

В части C рис. 9.1 изображена простая альтернатива системе ходов в квадратной сетке, которую можно использовать в настольных играх, чтобы сохранить возмож-

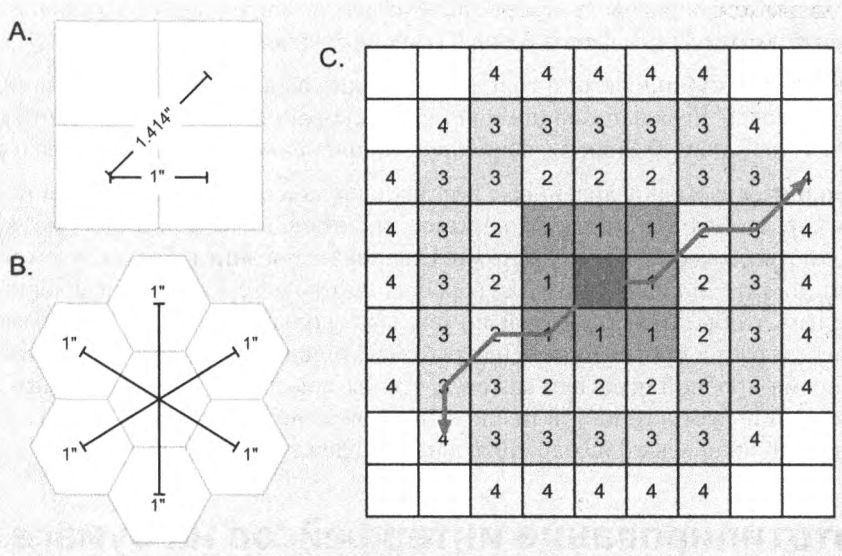


Рис. 9.1. Системы ходов

ность ходов по диагонали, но предотвратить злоупотребление ими. Игрокам разрешается выполнить диагональный ход только в сочетании с ходом по вертикали или по горизонтали. Это несколько уменьшает расстояние, которое требуется пройти, и делает возможными почти круговые перемещения за заданное количество ходов. Фиолетовая линия на диаграмме C показывает два разных возможных пути, по четыре хода в каждом. Такая система ходов используется в *Dungeons & Dragons*, созданной в Wizards of the Coast.

Шестиугольная сетка часто используется в настольных военных симуляторах, где особенно важна точность представления расстояний и движений. Однако большинство зданий в реальном мире имеют прямоугольные формы, плохо укладывающиеся в шестиугольное игровое поле. Выбор типа сетки полностью лежит на вас как на дизайнере.

- **Карточки для заметок 3 × 5:** разрезав пополам карточки 3 × 5, мы получим карточки с прекрасно подходящим для колоды размером. С исходными размерами они великолепно подходят для мозгового штурма. Сейчас в некоторых магазинах можно найти карточки 3 × 5, уже разрезанные пополам (3 × 2,5).
- **Стикеры:** эти маленькие самоклеящиеся листочки для заметок отлично подходят для быстрого упорядочивания и сортировки идей.
- **Белая доска:** нет более подходящей для мозгового штурма вещи, чем белая маркерная доска. Прикупите к ней побольше разноцветных маркеров. Нарисованное на белой доске часто стирается, поэтому все, что достойно сохранения, не забудьте сфотографировать. Если у вас есть настольная или настенная белая

доска, можете нарисовать игровое поле на ней, но я предпочитаю большие листы бумаги, потому что нарисованное на них не стирается.

- **Проволока/кубики лего:** и то и другое можно использовать для создания разных мелочей: игральных фишек, деталей декорации и вообще всего, что может прийти в голову. Кубики лего прочнее, но проволока намного дешевле и гибче.
- **Блокнот:** у вас, как у дизайнера, всегда под рукой должен быть блокнот. Я, например, люблю блокноты Moleskine с нелинованными листами, но вообще блокноты бывают самыми разными. Главное требование к блокноту: он должен быть достаточно маленьким для переноски в сумке или в кармане и достаточно пухлым, чтобы его хватало больше чем на несколько недель. Каждый раз, когда кто-то играет в прототип вашей игры, обязательно делайте заметки. Даже если во время пробной игры вам кажется, что вы запомните какие-то важные идеи, пришедшие вам в голову, или слова, произнесенные игроками, имейте в виду, что часто их не удается сохранить в памяти даже на несколько часов.

Прототипирование интерфейсов на бумаге

Одним из примеров типичного применения прототипирования на бумаге может служить поиск решений в организации интерфейса. Например, на рис. 9.2 показано несколько разных экранов из макета графического пользовательского интерфейса меню настройки игры для мобильных устройств с сенсорными экранами. Каждый тестировщик одновременно может видеть только один экран, начиная с экрана № 1,

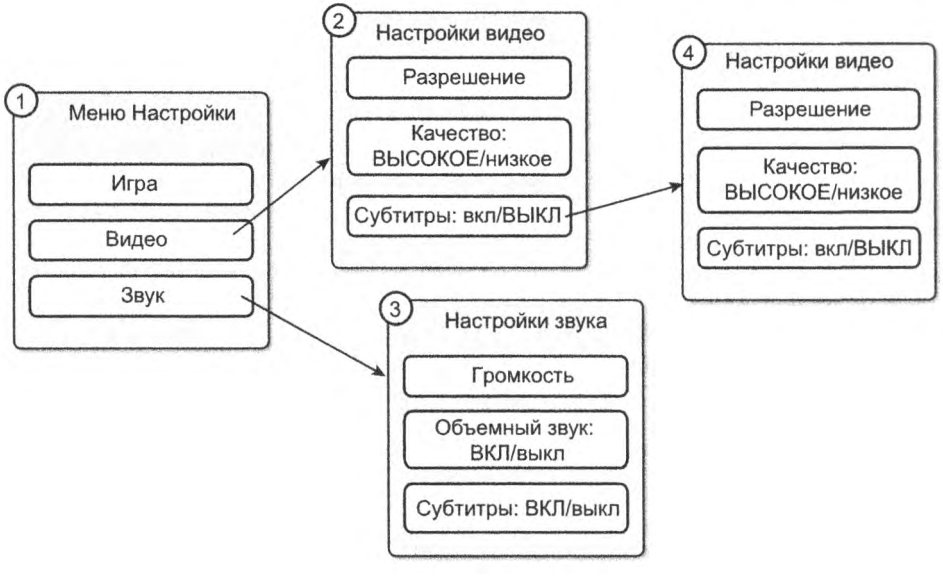


Рис. 9.2. Простой бумажный прототип графического интерфейса пользователя

меню Настройки. Когда тестировщик наблюдает экран № 1, его можно проинструктировать: «Нажмите пункты, которые, по вашему мнению, должны быть выбраны, чтобы включить субтитры». (Вы могли бы попросить тестировщиков на самом деле касаться бумаги, как если бы перед ними были сенсорные экраны.)

Кто-то из тестировщиков может нажать кнопку Видео, кто-то — кнопку Звук (кто-то даже может нажать кнопку Игра). После того как пользователь сделает выбор, вы можете заменить лист № 1 листом с выбранным разделом меню (например, № 2 Настройки видео). Затем, как предполагается, тестировщик мог бы нажать кнопку Субтитры: вкл/ВЫКЛ, чтобы включить субтитры, и тогда вы могли бы заменить лист № 2 листом № 4 Настройки видео.

Обратите внимание, что в данном примере субтитры можно включить в двух экранах с настройками, Настройки видео и Настройки звука. Для тестирования это очень удачное решение, потому что независимо от того, какой из двух пунктов выберет тестировщик (Видео или Звук), вы затем сможете проверить, насколько регистр надписей вкл/ВЫКЛ ясно сообщает, что субтитры в данный момент выключены.

Пример бумажного прототипа

В этом разделе главы я проведу вас через процесс проектирования прототипа уровня на бумаге, который мы реализуем в главе 35 «Исследователь подземелья». Вы увидите шаги превращения начальной идеи в четкий, законченный бумажный прототип, которые могут научить вас кое-чему полезному, что пригодится при создании окончательной цифровой версии игры.

Идея игры — уровень двумерной приключенческой игры

В последней главе этой книги я проведу вас через процесс создания двумерной приключенческой игры сверху вниз, используя в качестве примера оригинальную игру *Legend of Zelda* для Nintendo Entertainment System (NES). В играх, подобных этой, один из важнейших вопросов проектирования уровней связан с замками и ключами.

Как дизайнер, вы размещаете замки на дверях и ключи от них в комнатах, ожидая, что ваши игроки будут пересекать подземелье по определенному маршруту; однако часто игроки действуют непредсказуемо. Взгляните на две идентичные версии первого подземелья из *The Legend of Zelda* на рис. 9.3. Серой линией на верхней карте показан полный путь типичного игрока через подземелье. Здесь он собрал и использовал все ключи и все снаряжение, найденные в подземелье (включая лук в B1 и бумеранг в D3). Это типичный путь для начинающих игроков, исследующих уровень, но не самый короткий из ведущих к цели.

На нижней карте на рис. 9.3 показаны два коротких пути через подземелье. Следуя этими путями, игрок не подберет лук в комнате B1, но все еще сможет взять бумеранг в комнате D3. Светло-серой линией показан путь, которым игрок может пройти без использования специального снаряжения. Пунктиром показан самый

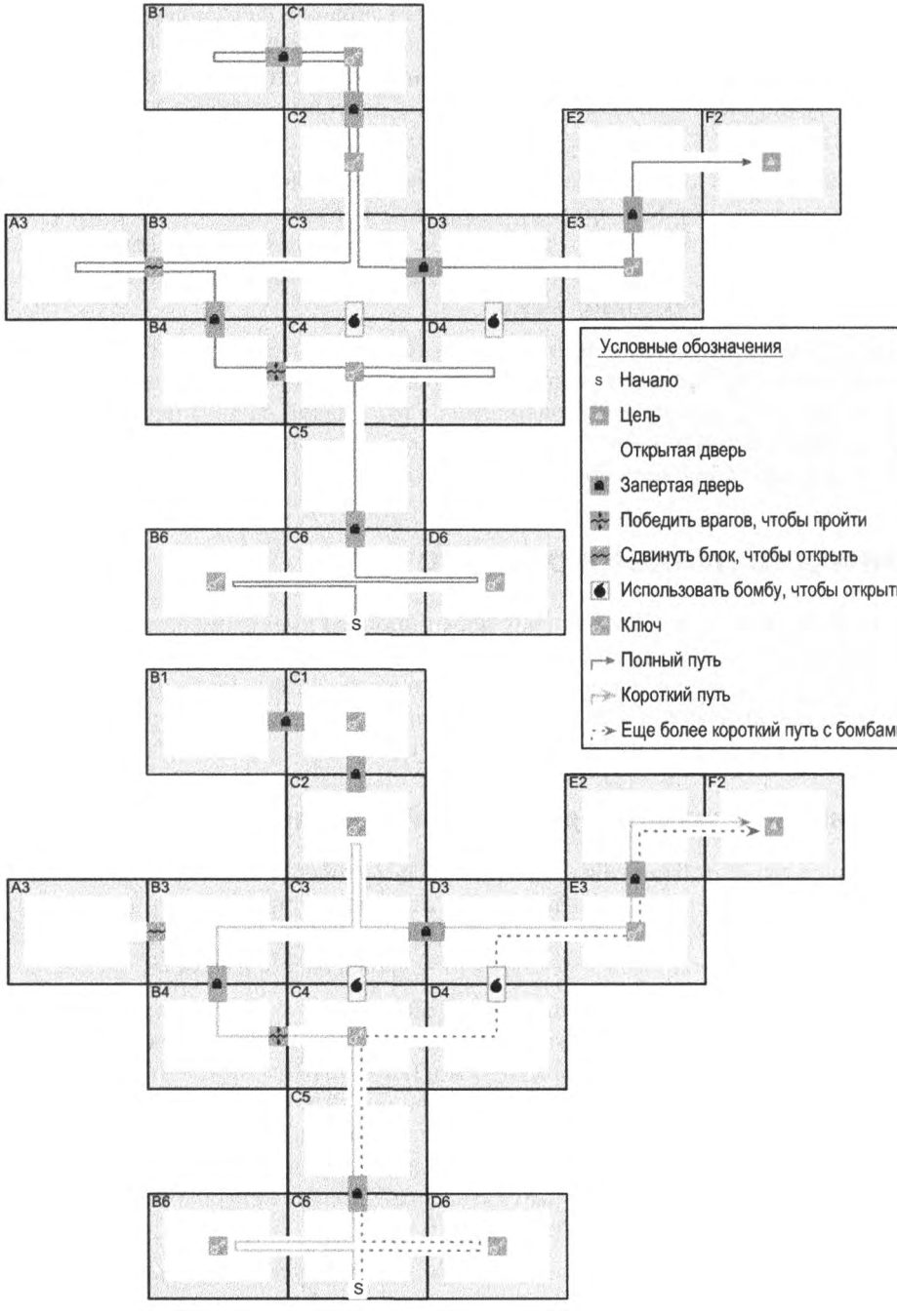


Рис. 9.3. Первое подземелье из *The Legend of Zelda* с ожидаемым маршрутом (*вверху*) и двумя возможными короткими маршрутами (*внизу*)

короткий путь, которым игрок может пройти, имея бомбы. Пунктирный путь требует найти лишь 2 из 6 ключей и посетить только 9 комнат из 17. Лук, лежащий в комнате B1, является сильным стимулом для игрока, чтобы зайти туда, но он может отказаться от лука в пользу дополнительных ключей, которые можно было бы использовать в других подземельях.

А если игрок захочет получить лук и использовать бомбы, чтобы пройти кратчайшим путем, каким маршрутом он должен пойти? Чтобы узнать это, вы должны сделать бумажный прототип! Попробуйте нарисовать план подземелья на листе бумаги. Поставьте метку в каждой комнате с ключом (например, монетку или маленький клочок бумаги с нарисованным на нем ключом). Положите скрепки (или что-нибудь другое прямоугольной формы) поверх каждой запертой двери, которые еще не были открыты. Возьмите еще какие-нибудь прямоугольные предметы и положите в местах стен, которые можно разрушить бомбами[1]. Затем поставьте в начальную комнату фишку, обозначающую вас самих. Заходите в комнаты и подбирайте встретившиеся ключи, затем убирайте в сторону по одному ключу и одной скрепке, имитируя отпирание соответствующей двери. Используйте бомбы, если захотите. Сколько комнат придется посетить, чтобы зайти в комнату B1, прежде чем попасть в F2?[2] Если у вас есть бомбы и вы хотите подобрать лук в комнате B1, можно ли пройти подземелье и не использовать все ключи?[3]

Это очень хорошо продуманное подземелье, предполагающее определенное прохождение, но, как видите, игроки могут использовать в своих интересах даже продуманный дизайн. Попробуйте спроектировать свое подземелье для *Legend of Zelda* с одними ключами и запертыми дверьми (без стен, разрушаемых бомбами) и посмотрите, смогут ли игроки проложить свой маршрут.

Прототипирование новой механики перемещения

В последних играх из серии *Legend of Zelda* появились предметы экипировки, увеличивающие возможности главного персонажа Линка перемещаться по подземельям. Классическим примером может служить *стреляющий крюк*, крюк с канатом, который Линк может зацепить за противоположную сторону и перебраться через провал. Снаряжение, подобное стреляющему крюку, легко можно исследовать с использованием бумажного прототипа.

На рис. 9.4 показано подземелье, которое я спроектировал с учетом этой идеи. Полный путь через подземелье отмечен серой линией.

---

[1] В игре *The Legend of Zelda* стены в первом подземелье, которые можно взорвать, выглядят как обычные стены, но эта тонкость не должна волновать нас при создании прототипа.

[2] На самом коротком пути, который я нашел, нужно посетить 12 комнат, можно подобрать 5 ключей и использовать 4 из них. Есть также еще один путь, лежащий через 13 комнат, с 5 ключами, из которых придется использовать всего 3.

[3] Да, можно, и вы сможете выйти из подземелья с дополнительными ключами, которые пригодятся вам в любых других подземельях в игре.

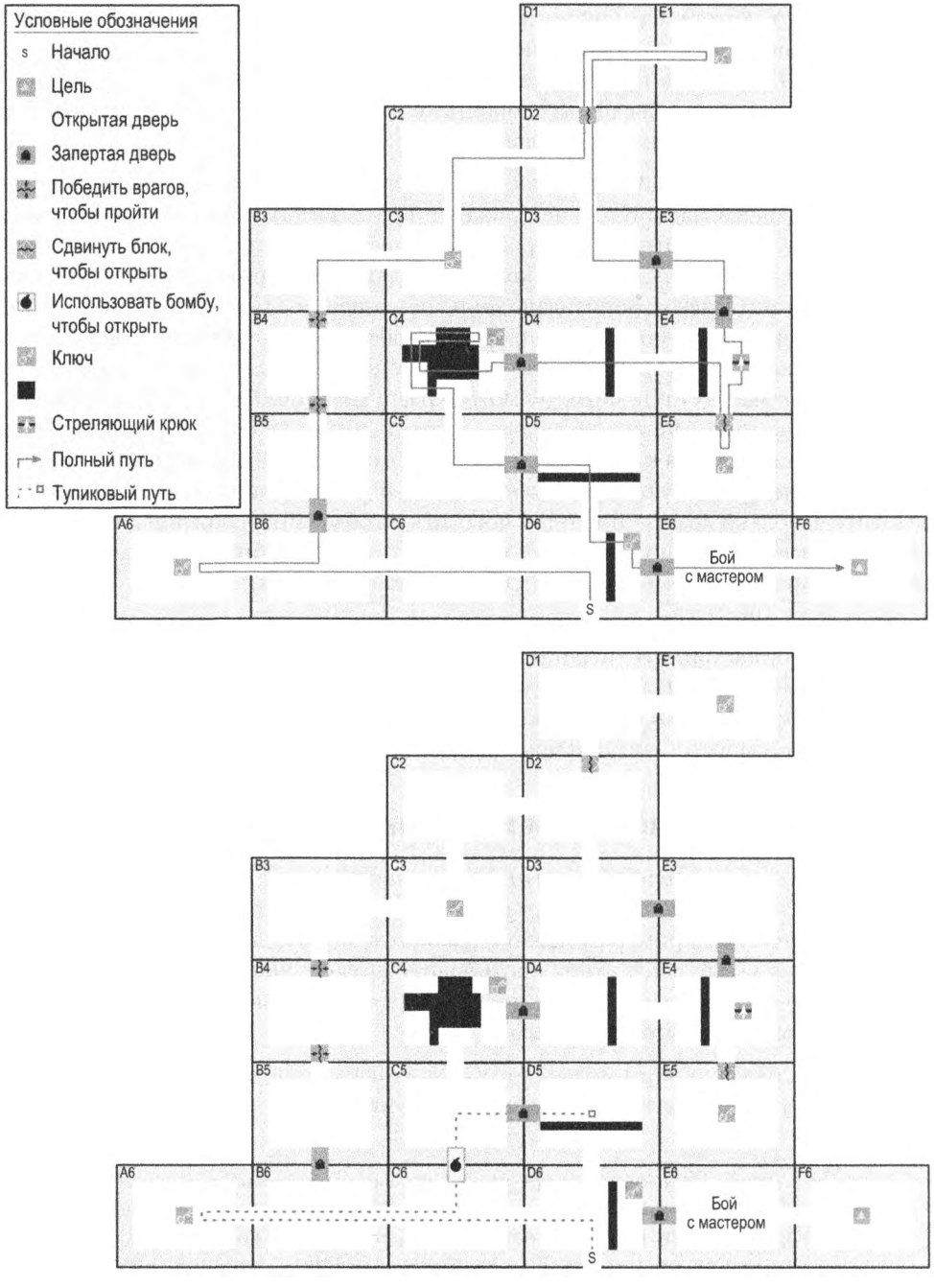


Рис. 9.4. Новое подземелье, где можно использовать стреляющий крюк (находится в комнате E4)

Опасность коротких путей. На плане, изображенном на рис. 9.4, нет стен, разрушаемых бомбами, потому что короткие пути могут заводить игроков в безвыходные ситуации. Например, внизу на рис. 9.4 добавлена единственная стена, которую можно взорвать, между комнатами C6 и C5. В результате у игрока появляется возможность пойти тупиковым маршрутом, обозначенным пунктиром, и оказаться в безвыходной ситуации, использовав единственный ключ для открытия не той двери. Это еще одна большая проблема, которую поможет решить прототип на бумаге.

Пробная игра

Попробуйте сами сыграть с прототипом несколько раундов, а потом попросите сыграть нескольких друзей. Они могут найти пути через подземелье, которые вы упустили из виду.

Единственное, чего не хватает в этом прототипе в текущем состоянии, — неожиданностей в каждой новой комнате. Я имею в виду, что игроки могут планировать свое движение вперед, видя перед собой всю схему подземелья, а не отдельные комнаты. Чтобы придать игре эффект неожиданности, можно нарисовать (напечатать) комнаты подземелья на отдельных листах бумаги. В этом случае можно было бы выкладывать комнаты на схеме, обозначая их карточками 3 × 5 (или визитками), по мере прохождения. Игроки почти всегда пробуют следовать разными путями, если у них нет изображения всей карты (именно поэтому такие элементы снаряжения, как карта и компас, так важны в *The Legend of Zelda*[1]).

Каждый раз, играя в пробную игру, задавайте себе следующие вопросы:

- Игрок последовал путем, который удивил вас?
- Можно ли оказаться в безвыходном положении?
- Увлекает ли игрока прохождение уровня?

Третий вопрос может показаться неуместным для прототипа, в котором нет ни одного врага, но уровни для игр, подобных этой, — загадки сами по себе, а загадки должны увлекать.

Обязательно делайте заметки о действиях игрока и записывайте все, что он думает о прототипе, когда играет. По мере изменения прототипа вы можете обнаружить, что ваши мнения также меняются. Делать заметки важно, потому что они позволят понять, как изменилась игра в процессе разработки, и дадут больше информации об изменении тенденций с течением времени.

Глава 10 «Тестирование игры» включает больше информации о правилах проведения пробных игр, а глава 13 «Проектирование головоломок» охватывает некоторые

[1] В оригинальной версии для NES карта отображается как сетка с синими блоками, соответствующими комнатам, а компас отмечает красной точкой комнату, где вас ждет бой с мастером.

аспекты процесса проектирования головоломок и объясняет, почему они так важны для однопользовательских игр.

Лучшие примеры использования прототипирования на бумаге

Прототипирование на бумаге при создании цифровых игр имеет свои сильные и слабые стороны. Вот несколько примеров, когда прототипирование на бумаге дает особенно положительные результаты:

- **Понимание перемещений игрока в игровом пространстве:** это главная цель прототипа, представленного в этой главе. Следите, в каких направлениях стремятся двигаться игроки в вашем подземелье. Куда они чаще поворачивают, вправо или влево, выбирая из двух вариантов? Понимание особенностей перемещения игрока в игровом пространстве поможет вам в проектировании всех аспектов уровня.
- **Балансирование простых систем:** даже при небольшом количестве параметров балансирование силы схожих элементов в игре может быть очень трудной задачей. Например, представьте балансирование двух видов оружия — дробовика и пулемета — у каждого три простых параметра: точность на расстоянии, количество выстрелов и урон, наносимый единственным выстрелом. Даже при наличии всего трех параметров сбалансировать каждое оружие относительно других намного сложнее, чем могло бы показаться. Сопоставим, к примеру, дробовик и пулемет:
  - **Дробовик:** дробовик может вызывать значительные повреждения при стрельбе с близкого расстояния, но его точность быстро падает с увеличением дистанции. Кроме того, он производит только один выстрел, поэтому в случае промаха враг вообще не получит никаких повреждений.
  - **Пулемет:** каждый выстрел из пулемета наносит небольшое повреждение, но он может выстреливать пули очередями и имеет более высокую точность на больших расстояниях. Кроме того, в игре можно учесть случайный разброс точности для каждой пули в отдельности.

  Даже если точность в игре вычисляется с учетом некоторой доли случайности, множественные шансы попадания пуль, выпускаемых пулеметом, увеличат вероятность нанесения повреждений одной очередью по сравнению с единственным выстрелом из дробовика по принципу «все или ничего». Таким образом, дробовик является рискованным, но более мощным оружием, тогда как пулемет обладает большей надежностью, но меньшей поражающей мощью. Математику, лежащую в основе этих рассуждений, мы исследуем в главе 11 «Математика и баланс игры».
- **Графический интерфейс пользователя:** как показано на рис. 9.2, легко можно напечатать несколько элементов графического интерфейса (например, кнопок,

меню, полей ввода) и затем спросить тестировщика, какую кнопку он нажал бы, чтобы выполнить поставленную задачу (например, приостановить игру, выбрать персонажа).

- **Опробование необычных идей:** благодаря высокой скорости итераций и разработки прототипов на бумаге, также можно быстро проверять разные необычные правила и видеть, как они влияют на игровой процесс.

Неудачные примеры использования прототипирования на бумаге

Несмотря на большое количество плюсов, для некоторых ситуаций прототипирование на бумаге совершенно не подходит:

- **Когда требуется отслеживать большой объем информации:** многие цифровые игры постоянно следят за сотнями параметров. В бесхитростной игре-стрелялке могут вычисляться видимость, отслеживаться уровень здоровья, оцениваться расстояния от атакующих до их целей и т. д. В прототипе на бумаге в первую очередь нужно сосредоточиться на простых системах в игре и проверять такие вещи, как макет уровня или характеристики разных видов вооружений (например, «рискованность» дробовика и «надежность» пулемета можно смоделировать с помощью одной кости d20 для дробовика против кости 4d4 для пулемета). Затем эту информацию можно точнее настроить в цифровом прототипе.
- **Ритм для быстрых и медленных игр:** бумажный прототип также может дать ложное представление о ритме игры, то есть фактический игровой процесс может протекать намного быстрее или медленнее, чем с бумажным прототипом. Например, однажды я видел, как команда придала слишком много значения бумажному прототипу игры, в которую в течение месяца должны были играть игроки, разбросанные по всему миру. В бумажном прототипе было множество интересных решений, позволявших игрокам напрямую мешать друг другу и конкурировать между собой. Механика хорошо работала, когда все игроки находились в одной комнате, а сеанс игры длился около часа. Однако для игроков, разбросанных по миру, и с фактической длительностью игры в несколько недель или даже месяцев механика оказалась менее действенной и менее удачной.
- **Физический интерфейс:** несмотря на то что бумажный прототип хорошо подходит для тестирования графического интерфейса, он почти ничего не способен сообщить о физических интерфейсах (то есть игровых пультах, сенсорных экранах, клавиатуре и мыши). Пока у вас не будет цифрового прототипа, работающего с фактическим физическим интерфейсом, который будет использоваться игроком, вы ничего не узнаете об удобстве его использования для управления вашей игрой. Это сложная проблема, о чем свидетельствуют постепенные изменения в управлении во многих сериях игр (например, все изменения, произошедшие за годы выпуска разных игр *Assassin's Creed*).

Итоги

Я надеюсь, эта глава сумела показать вам мощь и простоту прототипирования на бумаге. В некоторых лучших университетских программах обучения проектированию игр студенты проектируют свои первые игры на примерах настольных и карточных игр, оттачивая навыки в прототипировании, балансировании и настройке игры на бумаге. Бумажные прототипы помогают не только быстро исследовать идеи цифровых игр, но и получить навыки итеративного проектирования и принятия решений, которые станут для вас бесценными, когда вы начнете создавать цифровые игры.

Каждый раз, начиная проектирование новой игры (или новой системы для игры, которую вы уже разрабатываете), задайте себе вопрос, можно ли извлечь дополнительные выгоды, создав на бумаге прототип этой игры или системы. Например, мне потребовалось меньше часа, чтобы спроектировать, реализовать и несколько раз протестировать новый уровень подземелья, изображенный на рис. 9.4, но несколько дней, чтобы реализовать логику, перемещение камеры, искусственный интеллект, управление и другие детали цифровой версии в главе 35 «Исследователь подземелья». Чтобы добавить что-то вроде стреляющего крюка, нужно несколько минут в бумажном прототипе и часов или даже дней — в Unity и C#.

Кроме того, бумажные прототипы помогают не унывать, столкнувшись с неудачным дизайнерским решением. Все мы порой принимаем плохие решения, и это нормально. Преимущество плохого решения в бумажном прототипе в том, что вы можете быстро убедиться, что это решение действительно плохое, отбросить его и перейти к следующей идее.

Там вы познакомитесь с разными формами опробования игр и тестирования удобства их использования. Эти знания помогут вам получать более конкретную и достоверную информацию о ваших играх. Затем, в главе 11 «Математика и баланс игры» вы исследуете некоторые математические основы проектирования игр и увидите, как использовать программы электронных таблиц для балансирования игр.

10 Тестирование игры

В основе идей прототипирования и итеративного проектирования лежит понимание, что качественное тестирование абсолютно необходимо для хорошего дизайна игры. Но возникает вопрос: как именно должно производиться тестирование?

В этой главе вы познакомитесь с разными методами тестирования игр и с тем, как правильно их реализовать и на каком этапе разработки лучше подходит тот или иной метод.

Зачем тестировать игры?

После анализа целей, проектирования решения и реализации прототипа наступает момент, когда нужно опробовать этот прототип и узнать стороннее мнение о своем дизайне. Я понимаю, что это предложение может напугать. Игры сложны в проектировании, и вам еще нужно набираться опыта, чтобы получить хорошие результаты. Но даже став опытным дизайнером, вы все равно будете испытывать волнение, задумываясь о людях, которые будут играть в вашу игру в первый раз. Это нормально. Главное, о чем нужно помнить, — каждый играющий в вашу игру делает ее лучше; каждый комментарий, который вы получите, положительный или отрицательный, поможет вам улучшить восприятие игрока и отточить ваш дизайн.

Совершенствование дизайна — вот для чего это необходимо, поэтому у вас обязательно должна быть обратная связь от игроков. На протяжении многих лет я выступал в роли судьи на фестивалях игрового дизайна, и меня всегда поражало, насколько легко определить, проводила ли команда разработчиков тестирование в достаточном объеме. Например, если игра тестировалась недостаточно полно, ее цели часто недостаточно ясны, и сложность игры часто резко меняется. Таковы общие признаки того, что большую часть времени в игру играли люди, которые знали, как играть и преодолевать сложные участки, поэтому они не видят неоднозначности целей или резкого изменения сложности, как это видит простой тестировщик.

Эта глава расскажет и покажет вам, как проводить осмысленное тестирование и получать информацию, которая поможет сделать игру лучше.

Исследователи и тестировщики. В игровой индустрии мы часто называем *тестировщиками* людей, проводящих пробные игры и участвующих в них. Для большей ясности я хочу пояснить, что в этой книге я использую разные термины:

- **Исследователь:** человек, управляющий пробной игрой, обычно кто-то из вашей команды.
- **Тестировщик:** человек, принимающий участие в пробной игре и дающий отзывы.

Воспитайте в себе хорошего тестировщика

Прежде чем узнать, как организовывать пробные игры или на что обращать внимание при общении с тестировщиками, давайте посмотрим, как бы вы могли стать хорошим тестировщиком для других.

- **Думайте вслух:** одна из лучших черт тестировщика — способность вслух описывать свои внутренние мыслительные процессы во время игры. Это помогает исследователю, проводящему тестирование, правильно интерпретировать мотивы тех или иных ваших действий. Это особенно может пригодиться при первом столкновении с игрой.
- **Раскрывайте свои предубеждения:** у нас у всех есть свои предубеждения, сложившиеся из личного опыта и предпочтений, но исследователи часто не знают, с каким багажом опыта приходят к ним тестировщики. Играя, рассказывайте о других играх, фильмах, книгах, историях из жизни и т. д., которые вам напоминает игра. Это поможет исследователю понять предысторию и предубеждения, которые вы привносите в пробную игру с собой.
- **Самоанализ:** попробуйте помочь исследователям понять, почему вы испытываете именно такие реакции. Вместо простых фраз, таких как «я доволен», говорите что-нибудь более определенное, например «я доволен, потому что механика прыжка позволяет мне чувствовать себя сильным и ловким».
- **Отделяйте элементы:** как тестировщик, высказав общее мнение об игре, попробуйте рассмотреть каждый элемент отдельно; проанализируйте художественную сторону, механику игры, эмоциональный настрой, звук, музыку и т. д., по отдельности. Это может очень пригодиться исследователям. Это сродни тому, что вы скажете: «Виолончели не попадали в ноты» — вместо: «Мне не понравилась эта симфония». Пользуясь профессиональными знаниями дизайнера, вы сможете дать более развернутый отзыв, чем большинство игроков, поэтому не упускайте такую возможность.
- **Не беспокойтесь, если им не понравятся ваши идеи:** как дизайнер, вы должны выкладывать исследователям любые идеи, которые, по вашему мнению, могут сделать их игру лучше, но вы не должны обижаться, если они не воспользуются ими. В проектировании игр часто желательно оставлять свое эго за дверью; как оказывается, приступая к тестированию, желательно делать то же самое.

Круги тестировщиков

Тестирование игры проходит через несколько расширяющихся кругов тестировщиков, начиная с вас и постепенно охватывая ваших друзей и знакомых и, наконец, многих людей, которых вы даже не встречали. Каждый круг людей может помочь вам с определенными аспектами тестирования.

Круг первый — вы

Первым и последним тестировщиком игр, создаваемых вами как дизайнером, скорее всего, будете вы сами. Вы будете первым, кто сможет опробовать каждую из ваших идей, и первым, кто решит, насколько хорошо продуманы механика и интерфейс игры.

Главный посыл этой книги: вы всегда должны стараться как можно быстрее создать действующий прототип своей игры. Пока у вас нет прототипа, вы располагаете лишь бессвязным набором идей, но после создания прототипа у вас будет нечто конкретное, на что другие уже могут реагировать.

Далее в этой книге вы будете создавать прототипы цифровых игр в Unity. Каждый раз, щелкая по кнопке **Play** (Играть), чтобы запустить игру, вы будете выступать в роли тестировщика. Даже если вы работаете в команде и не являетесь ведущим инженером проекта, вы, будучи дизайнером, должны следить, чтобы игра двигалась в направлении создания восприятия, которое команда стремится создать. Ваши навыки как тестировщика будут наиболее полезны на самых ранних стадиях прототипирования, когда вы пытаетесь помочь другим членам команды понять идею дизайна или сами хотите нащупать базовую механику или базовый опыт игры.

Однако вы никогда не сможете получить первое впечатление от собственной игры — вы слишком многое знаете о ней. В какой-то момент вы должны показать свою игру другим. После того как вы почувствуете, что ваша игра — это скорее что-то хорошее, чем ужасное, потратьте время, найдите нескольких друзей и покажите им ее.

ОДНОРАЗОВЫЕ ТЕСТИРОВЩИКИ

Одноразовые тестировщики — это профессиональный термин, описывающий тестировщиков, которых лишь однажды пригласили, чтобы провести пробную игру и получить отзыв, после чего к ним больше не обращаются. Они приглашаются только раз. Эти тестировщики играют важную роль, потому что могут дать вам наивную реакцию на вашу игру. После того как кто-то сыграл в вашу игру, пусть даже однажды, он уже будет знать кое-что о ней, и это знание будет накладывать свой отпечаток на последующие сеансы тестирования. Первая наивная реакция имеет решающее значение, когда тестируются:

- обучающая система;
- первые несколько уровней;

- эмоциональное влияние любых сюжетных поворотов и других сюрпризов;
- эмоциональное влияние концовки игры.

Любой может быть одноразовым тестировщиком только раз

Ваша игра никогда не получит второго шанса произвести первое впечатление. Когда Дженова Чен работал над своей самой известной игрой, *Journey*, мы с ним жили по соседству. Однако он больше года, пока шла разработка, просил меня подождать, прежде чем я получил возможность сыграть в пробную игру. Позже он признался, что очень хотел получить мой отзыв, когда игра будет именно на заключительной стадии, чтобы оценить, удалось ли ему достичь желаемого эмоционального накала. Если бы я тестировал эту игру на ранних стадиях, когда эстетическая сторона была еще далека от идеала, это разрушило бы целостность моего восприятия. Помните об этом, проводя пробные игры с друзьями. Подумайте о том, на каком этапе каждый из них может дать самый ценный отзыв, и показывайте игру каждому в отдельности, в самый подходящий для этого момент.

Но никогда не используйте вышесказанное как предлог, чтобы спрятать игру от всех, «пока она не будет готова». Игру *Journey* тестировали сотни людей до того, как я увидел ее. На начальных этапах тестирования большинство тестировщиков будет вам говорить практически одно и то же, хотя и разными словами. Но вам нужны эти отзывы, и даже на самых ранних этапах разработки вам нужны одноразовые тестировщики, чтобы вы могли узнать, что в вашей игровой механике вызывает недоразумения или над чем еще нужно поработать по тем или иным причинам. Просто сохраните в неведении самых доверенных людей и обратитесь к ним потом, когда конкретно их отзывы будут для вас наиболее полезны.

Круг второй — близкие друзья

После того как вы сами сыграете в свою игру, повторите цикл разработки, внесете какие-то улучшения и действительно создадите что-то близкое к желаемому, наступает момент показать игру другим. Первыми должны быть лучшие друзья и члены семьи, предпочтительнее те из них, кто попадает в целевую аудиторию или близок к игровому сообществу. Целевая аудитория даст вам хорошие отзывы с точки зрения ваших будущих игроков, а разработчики игр могут поделиться своим опытом. Кроме того, разработчики игр часто способны игнорировать некоторые шероховатости и незавершенности в играх, что может очень пригодиться при опробовании ранних прототипов.

Круг третий — знакомые и все остальные

После нескольких итераций у вас появится нечто, что кажется довольно цельным. Теперь самое время выпустить игру на свободу. В этот момент пока рано публи-

ковать бета-версию в интернете и открывать игру для всего мира, но самое время, когда могут принести пользу отзывы от людей, с которыми у вас нет тесных связей. У круга ваших друзей или родственников опыт и взгляды часто похожи на ваши, а значит, схожи вкусы и наклонности. Если вы ограничитесь тестированием игры только на них, то получите предвзятое мнение о ней.

Следуя таким путем, житель Остина (штат Техас) мог бы удивиться, что его штат проголосовал за кандидата в президенты от республиканцев. Большинство жителей Остина придерживаются либеральных взглядов, тогда как среди остальных жителей штата больше консерваторов. Если бы вы опрашивали только жителей Остина, не пытаясь выйти за границы этого пузыря с левыми взглядами, то никогда не узнали бы мнения штата в целом. Точно так же вы должны выйти из привычного социального круга и найти побольше тестировщиков для своей игры, чтобы узнать реакцию более широкой аудитории на нее.

Итак, где можно найти больше людей, чтобы протестировать свою игру? Вот некоторые возможные места:

- **Местные университеты:** многие студенты колледжей любят играть в игры. Вы можете попробовать установить свою игру в студенческом центре и показывать ее группам людей. Конечно, вам нужно проверить безопасность кампуса, прежде чем сделать это.

  Узнайте также, есть ли в этом университете клуб разработчиков игр или группа, участники которой устраивают ночные посиделки за игрой по выходным. Вы можете обратиться к ним с просьбой протестировать вашу игру.

- **Местные магазины и торговые центры, где продаются игры:** люди приходят в эти места, чтобы покупать игры, поэтому такие магазины могут оказаться фантастическим местом для получения отзывов. В каждом из таких мест действуют свои правила в отношении подобных акций, поэтому прежде обсудите это с администрацией.

- **Ярмарки / общественные мероприятия / празднования:** на таких общественных мероприятиях могут собираться самые разные люди. Я порой получал отличные отзывы от людей, с которыми встречался на празднованиях.

Круг четвертый — интернет

Интернет бывает жутким местом. Анонимность способствует безответственности в заявлениях, и некоторые люди приходят в интернет только ради потехи. Однако в интернете также самый широкий круг тестировщиков, какой только можно найти. Если вы работаете над онлайн-игрой, вам рано или поздно придется обратиться к интернету и посмотреть, что из этого выйдет. Но прежде чем сделать это, вам нужно подготовиться к сбору данных о тестировании, как это описано в разделе «Онлайн-тестирование».

Методы тестирования игр

Существует несколько разных методов тестирования игр, каждый из которых лучше подходит для определенного этапа. На следующих страницах описаны методы тестирования, которые оказались наиболее полезными в моей практике.

Неформальное индивидуальное тестирование

Как независимый разработчик, большей частью я тестирую игры сам. В последнее время я много внимания уделяю мобильным играм, поэтому легко могу носить с собой свое устройство и показывать игры людям. Чаще во время перерыва в разговоре я спрашиваю человека, хотел бы он взглянуть на мою игру. Этот подход наиболее полезен на ранних стадиях разработки или когда в вашей игре появляется что-то новенькое, что вам хотелось бы протестировать. Во время такого тестирования желательно следовать определенным правилам, в том числе:

- **Не рассказывайте игроку слишком много:** даже на ранних стадиях полезно выяснить, насколько понятен интерфейс и цели игры. Предложите свою игру тестировщикам и понаблюдайте за их действиями, прежде чем инструктировать их. Такой подход поможет вам узнать, на какие взаимодействия способна ваша игра. В конце концов, вы выслушаете конкретные краткие предложения, которые помогут вашей игре стать понятнее людям и смогут стать основой вашего руководства к ней.
- **Не управляйте тестировщиком:** не задавайте наводящих вопросов, которые могут непреднамеренно направлять игрока. Даже такой простой вопрос, как «Вы заметили предметы, которые повышают здоровье?», информирует тестировщика о существовании предметов, добавляющих здоровья, и подсказывает, насколько важно собирать их. Когда вы выпустите свою игру, вас не будет рядом с игроками и некому будет объяснять им какие-то игровые моменты, поэтому позвольте тестировщикам помочь вам узнать, какие важные аспекты игры остались непонятыми.
- **Не спорьте и не оправдывайтесь:** как и повсюду в проектировании игр, вашему эго не место в тестировании. Слушайте, что думают тестировщики, даже (и особенно) если вы не согласны с ними. Это не время вступаться за свою игру; это шанс узнать от человека, уделившего вам свое время, как еще можно улучшить игру.
- **Делайте заметки:** держите при себе записную книжку и конспектируйте, что говорит вам тестировщик, особенно если вы слышите совсем не то, что ожидали. Позже вы сможете сопоставить заметки и поискать повторяющиеся утверждения. Не слишком акцентируйте внимание на том, что сказал один тестировщик, но обязательно исследуйте отзывы разных людей, содержащие схожие высказывания.

ПРАВИЛА ОФОРМЛЕНИЯ ЗАМЕТОК О ТЕСТИРОВАНИИ

Заметки, сделанные в процессе тестирования, — один из самых ценных инструментов для изучения и реализации улучшений в играх, но вы должны вести заметки так, чтобы обеспечить максимальную эффективность их анализа. Простая запись заметок в случайном порядке без дальнейшего их исследования ничем вам не поможет. На рис. 10.1 показана таблица, которую я обычно использую для записи заметок в ходе тестирования.

Игрок	Где	Отзыв	Причина	Важность	Возможное решение
(Имя и контактная информация)	Boss1	«Я не понял, что делать после сражения с первым боссом. Куда идти дальше?»	Игрок растерялся и не понимает, каким должен быть следующий шаг после боя с первым боссом. До этого игра явно направляла игрока.	Высокая	Наставник мог бы вернуться после победы в бою с боссом и дать игроку второе задание.

Рис. 10.1. Пример одной записи в заметках о тестировании

Как упоминалось в главе 7, важно собрать как можно больше полезной информации из каждой пробной игры, и данная форма поможет вам в этом. Запишите первые три столбца во время тестирования. Для каждого нового комментария, сделанного тестировщиком, добавляйте новую строку.

Завершив пробную игру, организуйте собрание с членами своей команды и заполните три оставшихся столбца. Действуя так, вы выявите проблемы, одни из которых испытывают лишь отдельные тестировщики, а другие — почти все. На этом этапе вы также можете объединить несколько записей в одну, если считаете, что одно решение исправит несколько проблем.

Официальное групповое тестирование

Долгие годы официальное групповое тестирование было единственной формой пробной игры, существовавшей в больших студиях, и когда я работал в Electronic Arts, то принимал участие во многих пробных играх для других команд. В официальном групповом тестировании несколько человек заводят в комнату с отдельными станциями, на которых они играют в игру. Им может даваться или не даваться короткая инструкция и разрешается играть в игру строго определенное время (обычно около 30 минут). Спустя это время тестировщики пишут письменные отчеты, и иногда исследователи беседуют с ними индивидуально. Это отличный способ узнать мнение большого количества людей, и, используя его, можно получить большое количество ответов на некоторые важные вопросы.

Вот некоторые примеры вопросов, на которые предлагается ответить тестировщикам после пробной игры:

- «Перечислите три наиболее понравившиеся части игры».
- «Перечислите три наименее понравившиеся части игры».
- Тестировщику передают список разных мест в игре (или изображений, что даже лучше) и предлагают ответить на вопрос: «Как бы вы описали свои чувства к каждому из этих мест в игре?»
- «Какие чувства вы испытываете к главному герою (или другим персонажам) игры? Изменились ли ваши чувства к главному герою в процессе игры?»
- «Сколько вы готовы заплатить за эту игру?» или «Сколько стоит эта игра, по вашему мнению?»[1]
- «Какие три самых непонятных момента в игре?»

Официальное групповое тестирование чаще производится исследователями, не входящими в основную команду разработчиков, и есть даже компании, предоставляющие такие услуги по проведению тестирования.

Для проведения официального тестирования требуется сценарий

Вы должны писать для исследователей сценарий всякий раз, когда собираетесь провести официальное тестирование, независимо от того, являются ли исследователи членами вашей команды. Сценарий даст гарантию, что все тестировщики будут находиться в одинаковых условиях, и позволит минимизировать влияние внешних факторов на результаты тестирования. Сценарий должен включать следующие сведения:

- Что исследователь должен говорить тестировщику в процессе подготовки к игре? Какие инструкции он должен давать?
- Как должен вести себя исследователь в процессе пробной игры? Должен ли он задавать вопросы тестировщикам, если видит, что те делают что-то интересное или необычное? Должен ли он давать подсказки тестировщикам в процессе игры?
- В каком окружении должно производиться тестирование? Как долго должна длиться пробная игра?
- Какие вопросы должны задаваться тестировщику на собеседовании по окончании пробной игры?
- Какие заметки должен делать исследователь в процессе тестирования?

[1] Эти два интересных вопроса представляют A/B-тест ваших тестировщиков (то есть первый вопрос задается одним тестировщикам, а второй — другим). Когда спрашивают, как много они готовы заплатить, люди обычно занижают цену. Когда спрашивают, сколько это стоит, люди обычно завышают цену. Справедливая цена обычно находится где-то посередине.

Официальное индивидуальное тестирование

В отличие от официального группового тестирования, направленного на сбор небольших фрагментов информации среди множества разных людей и формирование исследователями общей картины реакции тестировщиков на игру, официальное индивидуальное тестирование проводится с целью выявить тонкие детали восприятия отдельного тестировщика. С этой целью исследователи подробно записывают все детали переживаний одного человека, а потом, позднее, сопоставляют свои записи, чтобы убедиться, что ничего не было упущено из виду. При проведении официального индивидуального тестирования одновременно должна вестись запись разных данных:

- **Экрана игры:** вы должны видеть то же, что видел игрок.
- **Действий тестировщика:** вы должны видеть, какие действия предпринимал игрок. Если игра управляется мышью и клавиатурой, поместите камеру над ними. Если игра тестируется на планшетном компьютере с сенсорным экраном, вы должны делать снимки рук игрока, касающихся экрана.
- **Лица тестировщика:** вы должны видеть выражение лица тестировщика, чтобы прочитать его эмоции.
- **Звуков, издаваемых тестировщиком:** даже если тестировщик не произносит осмысленных слов, издаваемые им звуки порой могут дать очень много информации о процессе его мышления.
- **Игровых данных:** игра также должна фиксировать свое внутреннее состояние в файле, снабжая каждую запись меткой времени. Это могут быть сведения о действиях ввода игрока (например, нажатия кнопок на пульте), успех или неудача игрока в разных задачах, местоположение игрока, время пребывания в каждой области игры и т. д. Дополнительную информацию вы найдете во врезке «Автоматизированная регистрация данных» далее в этой главе.

ПОДГОТОВКА ЛАБОРАТОРИИ ДЛЯ ОФИЦИАЛЬНОГО ИНДИВИДУАЛЬНОГО ТЕСТИРОВАНИЯ

На создание лаборатории для официального индивидуального тестирования легко можно потратить десятки тысяч долларов — и многие студии тратят их, но точно так же можно создать вполне приличную лабораторию за меньшие деньги.

Все компьютерные платформы позволяют захватывать все потоки данных, перечисленные выше, для чего достаточно мощного игрового ноутбука и одной дополнительной камеры: современные графические карты способны записывать содержимое экрана прямо в процессе игры, веб-камера ноутбука может записывать выражение лица игрока, а дополнительную отдельную камеру можно настроить на съемку рук игрока. Запись звуков по всем каналам поможет вам синхронизировать их поток. Журнал с игровыми данными также должен содержать метки времени для синхронизации.

Синхронизация данных

Многие современные программные пакеты позволяют синхронизировать несколько видеопотоков, но чаще более старые методы оказываются самыми простыми, и в данном случае можно использовать цифровую версию *синхронизатора*, появившегося на ранних этапах развития звукового кино, когда изображение и звук записывались на разных аппаратах. В кино роль синхронизатора играла маленькая доска-хлопушка, которая показывалась перед началом сцены. Член съемочной команды показывал перед камерой хлопушку, на которой написано название фильма, номер сцены и номер дубля. Произносил эту информацию вслух и затем хлопал хлопушкой. Позднее это помогало монтажеру совместить визуальное изображение со звуком по видимому моменту хлопка с хлопком на аудиопленке.

То же самое можно сделать, реализовав цифровой аналог хлопушки в игре. В начале сеанса тестирования на экране игры может отображаться цифровая хлопушка с уникальным номером сеанса. Исследователь может громко прочитать номер и затем нажать кнопку на контроллере. Одновременно программа может показать закрывающуюся хлопушку, произвести звук хлопка и зафиксировать состояние игры с отметкой времени, соответствующей внутренним часам на машине для тестирования. Все это потом можно использовать для синхронизации разных видеопотоков (с использованием звука хлопушки для синхронизации потоков, на которых не отображается экран), и можно даже синхронизировать журнал с игровыми данными. Большинство даже не самых сложных видеоредакторов позволяет поместить каждый видеопоток в свою четверть экрана и вывести в четвертой четверти данные, время и уникальный номер сеанса тестирования. Затем вы сможете просматривать все эти данные в синхронизированном виде на одном экране.

Конфиденциальность

Многих, по понятным причинам, волнует их личная неприкосновенность. Вы должны быть готовы к этому и предупредить игроков, что процесс тестирования будет сниматься на видеокамеру. Также вы должны пообещать им, что видео будет использоваться только для внутренних целей и никогда не будет передаваться за пределы компании.

Позднее все эти потоки синхронизируются друг с другом, чтобы дизайнеры могли ясно видеть связь между ними. Это позволяет различить восторг или разочарование на лице игрока, видя при этом, что наблюдал игрок на экране в этот момент и какие действия пытался выполнить. Это огромный объем данных, но современные технологии значительно удешевили создание очень неплохой лаборатории для индивидуального тестирования.

Проведение официального индивидуального тестирования

Исследователи должны стремиться к тому, чтобы индивидуальное тестирование своими условиями максимально близко соответствовало опыту, который игрок, купивший игру, получил бы, играя дома. Для игрока должна быть создана комфортная

и непринужденная обстановка. Возможно, вам придется дать игроку закуски или напитки, и, если игра создана для планшета или игровой приставки, может быть, вам даже понадобится усадить игрока на диван или предусмотреть другое удобное место, где тот мог бы с комфортом расположиться. (Для компьютерных игр часто лучше подходят офисные стол и кресло.)

Перед началом пробной игры сообщите игроку, насколько вы цените время, которое он выделил для тестирования вашей игры, и насколько полезными будут его отзывы. Вы должны также попросить игрока произносить все свои мысли вслух. Некоторые тестировщики фактически так и поступают, но попросить никогда не помешает.

Когда тестировщик закончит прохождение раздела игры, запланированного для тестирования, исследователю лучше сесть рядом с ним и обсудить впечатления, полученные от игры. Вопросы можно задавать те же самые, что задаются в конце официального группового тестирования, но формат «с глазу на глаз» позволяет исследователю выстроить вопросы в более осмысленном порядке и получить более качественную информацию. Сеансы собеседования с игроками после пробной игры также записывайте на камеру, хотя для собеседования после игры аудиозапись будет иметь большую ценность, чем видео.

Как при любом официальном тестировании, лучше, если в роли исследователей будут выступать лица, не входящие в состав команды разработчиков игры. Это поможет избавить вопросы исследователя от предвзятости лиц, вложивших свой труд в игру. После того как вы найдете хорошего исследователя, старайтесь пользоваться его услугами на протяжении всего времени разработки, чтобы он мог дать вам информацию о тенденциях изменения реакций тестировщиков.

Онлайн-тестирование

Как отмечалось выше, самый широкий круг тестировщиков можно найти в онлайн-сообществах. Прежде чем пытаться воспользоваться их услугами, необходимо довести игру до состояния бета-версии, поэтому такое тестирование часто называется *бета-тестированием* и имеет несколько форм:

- **Закрытое:** тестирование осуществляется ограниченным кругом специально приглашенных людей. Именно с этого вы должны начинать онлайн-тестирование. Прежде всего у вас должно быть несколько знакомых, пользующихся вашим доверием. Их помощь даст вам шанс найти любые ошибки в архитектуре вашего сервера и неясности в вашей игре, прежде чем их увидит более широкая аудитория.

  Для нашего дипломного проекта *Skyrates*[1] первое онлайн-тестирование бета-версии прошло через восемь недель после начала разработки, и в нем приняли

[1] *Skyrates* (Airship Studios, 2006) — игра, представленная в главе 8 «Цели проектирования». При ее создании использовалась идея спорадической игры, согласно которой один сеанс взаимодействия с игрой длится лишь несколько минут. Теперь же это обычный режим

участие четыре члена команды разработчиков и всего 12 посторонних людей, находившихся в офисах в том же здании, где располагалась наша команда. Через две недели, после исправления ошибок в игре и на сервере и добавления некоторых новых особенностей, мы расширили группу тестировщиков до 25 человек, точно так же находившихся в офисах одного с нами здания. Еще через пару недель мы увеличили количество тестировщиков до 50 человек. До этого момента члены команды разработчиков встречались с каждым игроком и обучали его игре.

В течение следующих двух недель мы разработали онлайн-руководство к игре и вошли в фазу ограниченного бета-тестирования.

- **Ограниченное:** ограниченное бета-тестирование обычно открыто для всех подписавшихся, хотя иногда есть определенные ограничения. Одним из распространенных ограничений является количество игроков.

  Когда игра *Skyrates* вышла на этап ограниченного бета-тестирования, мы установили предел в 125 игроков и сказали им, что они могут пригласить в игру одного друга или члена семьи. Мы получили намного больше одновременно играющих игроков, чем в предыдущих раундах, и хотели убедиться, что сервер справится с такой нагрузкой. Затем, перед этапом открытого бета-тестирования, в следующем раунде мы расширили ограничение до 250 человек.

- **Открытое:** открытое бета-тестирование позволяет любому желающему принять участие в онлайн-игре. Этот этап может дать фантастическую возможность увидеть, как игра обретает популярность по всему миру, но он же может стать тяжелым испытанием из-за большого наплыва игроков, что может вызвать перегрузку сервера. Вообще вы должны убедиться, что игра почти закончена, прежде чем переходить к открытому этапу бета-тестирования в онлайн.

  Игра *Skyrates* вступила в этап открытого бета-тестирования в конце первого семестра разработки. Мы не предполагали продолжать работать над игрой до второго семестра, поэтому оставили игровой сервер работать на целое лето. К нашему удивлению, несмотря на то что *Skyrates* первоначально разрабатывалась как двухнедельная игра, некоторые люди продолжили играть все лето, и, по нашим данным, в течение лета число игроков колебалось между 500 и 1000. Однако все это происходило в 2006 году, еще до того, как Facebook стал игровой платформой, и до появления вездесущих игр для смартфонов и планшетов. Даже при том, что 99 % всех игр для этих платформ вообще не пользуются популярностью, имейте в виду, что игра, выпущенная для любой из них, теоретически может пройти путь от нескольких игроков до миллионов всего за несколько дней. Будьте осторожны, переходя к открытому бета-тестированию на социальных платформах, но знайте, что рано или поздно вам придется открыть свою игру для этого.

взаимодействий с играми в Facebook, но когда мы писали свою игру, эта идея была в диковинку, и для ее уточнения потребовалось провести множество раундов пробной игры.

АВТОМАТИЗИРОВАННАЯ РЕГИСТРАЦИЯ ДАННЫХ

Реализуйте автоматическую регистрацию данных в своей игре как можно раньше. Под автоматической регистрацией подразумевается автоматическая запись информации о действиях игрока и событиях всякий раз, когда кто-то играет в вашу игру. Часто запись выполняется на сервере, но также может сохраняться в локальных файлах и выводиться игрой позднее.

В 2007 году в Electronic Arts я работал над игрой *Crazy Cakes* для Pogo.com. Это была первая игра, реализовавшая автоматическую регистрацию данных, что впоследствии стало стандартной частью производства. Механизм автоматической регистрации для *Crazy Cakes* был по-настоящему прост. Для каждого уровня игры регистрировалось несколько элементов данных:

- Отметка времени: дата и время начала раунда.
- Имя пользователя игрока: это позволяло нам общаться с игроками, заработавшими очень большое количество очков, и узнавать, какие стратегии они применяли, или связываться с ними, когда в игре происходило что-то необычное.
- Уровень сложности и номер раунда: у нас было пять уровней сложности, каждый из которых содержал четыре все более сложных раунда.
- Очки.
- Число и типы энергетических элементов, использованных в каждом раунде.
- Количество заработанных жетонов.
- Количество обслуженных посетителей.
- Количество десертов, поданных посетителям: некоторые посетители запрашивали несколько десертов, и эта информация помогала нам их отследить.

На тот момент у Pogo.com были сотни бета-тестеров, поэтому всего за три дня с момента выпуска *Crazy Cakes* мы записали данные для более чем 25 000 сеансов игры! Из этих данных я отобрал случайным образом 4000 записей и перенес их в приложение электронных таблиц, которое использовал для балансировки игры на основе реальных данных, а не домыслов. Когда я решил, что игра наконец сбалансирована, я отобрал еще 4000 случайных записей и подтвердил достижение баланса.

Другие важные типы тестирования

Помимо пробной игры можно выполнить еще несколько видов тестирования.

Фокус-тестирование

Фокус-тестирование предполагает набор группы людей, представляющих основную демографическую группу вашей игры (*фокус-группу*), и выяснение их реакции на

внешний вид, замысел, музыку и другие эстетические и сюжетные элементы игры. Иногда студии используют фокус-тестирование, чтобы определить, является ли их игра хорошим бизнес-решением.

Выяснение интереса

В настоящее время можно использовать социальные сети, такие как Facebook, или краудфандинговые сайты, такие как Kickstarter, для выяснения уровня интереса, который может вызвать ваша игра в онлайн-сообществе. На этих веб-сайтах вы можете размещать видеоролики, демонстрирующие игру, и получать отзывы в форме лайков на сайте социальной сети или залогов на краудфандинговом сайте. Если вы независимый разработчик с ограниченными ресурсами, выяснение интереса может оказаться одним из источников финансирования вашей игры, но, как вы понимаете, результаты совершенно непредсказуемы.

Тестирование удобства

Многие методы, используемые в официальном индивидуальном тестировании, появились в сфере тестирования удобства использования. Тестирование удобства по своей сути — это попытка выяснить, насколько легко тестировщикам удается понять и использовать пользовательский интерфейс. Поскольку понимание очень важно для удобства, с целью его выяснения широко применяется запись данных с экрана, взаимодействий и выражения лица тестировщика. Кроме опробования процесса игры, не менее важно проводить тестирование удобства использования, которое покажет, насколько легко тестировщику взаимодействовать с интерфейсом и получать важную информацию из игры. Сюда может входить тестирование разных макетов организации информации на экране, несколько конфигураций элементов управления и т. д.

Тестирование качества

Тестирование качества осуществляется с целью выявления ошибок в игре и способов их достоверного воспроизведения. Такому тестированию посвящена целая индустрия. Тестирование качества в целом далеко выходит за рамки этой книги, тем не менее перечислю некоторые основные его элементы:

1. Поиск ошибок (багов) в игре (место, где игра прерывается или реагирует неправильно).
2. Определение и запись шагов, необходимых для надежного воспроизведения ошибки.
3. Назначение приоритета ошибки. Ошибка вызывает крах игры? Может ли простой игрок воспроизвести ее? Насколько она заметна?

4. Если ошибка имеет достаточно высокий приоритет, сообщите о ней команде инженеров, чтобы они могли устранить ее.

Тестирование качества чаще выполняется одновременно командой разработчиков и группой тестировщиков, нанятых на заключительном этапе проекта. Также можно предусмотреть каналы, по которым игроки тоже смогут присылать отчеты о выявленных ошибках, даже при том, что они обычно не имеют навыков создания четких и ясных отчетов об ошибках с подробным описанием шагов, необходимых для воспроизведения ошибки. С этой целью вы можете развернуть на веб-сайте проекта одну из бесплатных систем слежения за ошибками, такую как *Bugzilla*, *Mantis Bug Tracker* или *Trac*.

Автоматизированное тестирование

Под автоматизированным тестированием подразумевается поиск ошибок в игре или на сервере с привлечением другого программного обеспечения и без участия человека. Система автоматизированного тестирования может, например, имитировать быстрый ввод пользователя (скажем, сотни щелчков мышью в секунду в самых разных областях на экране). Для проверки сервера такая система может нагрузить сервер тысячами запросов в секунду, чтобы определить, какой уровень нагрузки может привести к сбою сервера. Несмотря на сложность реализации, автоматизированное тестирование способно эффективно протестировать игру способами, которые сложно воспроизвести вручную. Так же как в случае с другими видами тестирования, существуют компании, предлагающие услуги автоматизированного тестирования.

Итоги

Цель этой главы состояла в том, чтобы познакомить вас с самыми разными формами тестирования игр. Как дизайнер игр, вы должны выбрать те, которые покажутся вам наиболее полезными, и попробовать реализовать их. У меня есть опыт успешного применения нескольких разных форм тестирования, и я считаю, что все формы, описанные в этой главе, смогут дать важную информацию, которая поможет вам улучшить вашу игру.

Следующая глава описывает математику, приводящую в движение механизм развлечения. В ней вы также узнаете, как использовать электронную таблицу для балансирования игры.

11 Математика и баланс игры

В этой главе исследуются разные системы вероятностей и случайности и их взаимосвязь с технологиями настольных игр. Здесь вы также узнаете немного о приложении электронных таблиц Google Sheets, которое поможет вам в исследовании этих возможностей.

После математических исследований (которые, обещаю, будут просты и понятны) я расскажу, как использовать эти системы в настольных и цифровых играх, чтобы улучшить и сбалансировать игровой процесс.

Значение баланса игры

После создания начального прототипа и нескольких пробных игр с ним вам наверняка придется выделить этап в своем итерационном процессе для его балансировки. Вам часто придется слышать термин «баланс», работая над играми, но в разных ситуациях у него разный смысл.

В многопользовательской игре под балансом часто подразумевается *справедливость*: у всех игроков должны быть равные шансы на победу в игре. Этого легче всего достичь в *симметричных* играх, где все игроки начинают в равных условиях и с равными возможностями. Сбалансировать *асимметричную* игру намного сложнее, потому что, казалось бы, равные начальные условия и возможности на практике могут оказаться смещенными в пользу преобладания возможностей одних игроков над другими. Это одна из множества причин, почему пробная игра так важна.

В однопользовательской игре под балансом обычно подразумевается *соответствие уровня сложности игры* способностям игрока и *постепенное увеличение сложности*. Если сложность игры увеличивается большими скачками в какие-то моменты времени, эти моменты становятся местами в игре, где игроки теряют интерес и бросают игру. Это прямо относится к обсуждению потокового состояния как одной из целей игрока в главе 8 «Цели проектирования».

В этой главе вы познакомитесь с некоторыми математическими аспектами, являющимися частью дизайна и баланса игры, в том числе с пониманием вероятности, исследованием разных способов внесения элемента случайности в настольных играх, идеей взвешенного распределения, перестановок, положительной и отрицательной обратной связи. В ходе исследования мы будем использовать Google

Sheets, бесплатную онлайн-программу электронных таблиц, чтобы вам было проще понять представленные понятия.

Важность электронных таблиц

Строго говоря, для определенных исследований, которые вы будете проводить в этой главе, применение электронных таблиц, таких как Google Sheets, не является обязательным — те же результаты можно получить с помощью листка бумаги и калькулятора, но я считаю, что применение электронных таблиц как аспекта балансирования игры важно по нескольким причинам:

- Электронная таблица поможет вам быстро извлечь наглядную информацию из числовых данных. В главе 9 я привел пример двух видов оружия — дробовик и пулемет — с разными тактико-техническими характеристиками. В конце этой главы я покажу вам процесс, который преодолел, чтобы сбалансировать эти два вида оружия друг с другом, а также с тремя другими, сопоставляя статистику оружия, созданную первоначально исходя из интуитивных предпосылок и уточненную с помощью электронной таблицы.
- Данные диаграмм и электронных таблиц можно использовать, чтобы убедить других в обоснованности проектного решения, принятого вами. При разработке игры вам часто придется сотрудничать с другими людьми, и среди них будут попадаться такие, которые предпочитают видеть за решениями цифры, а не инстинкты. Это не означает, что вы всегда должны принимать решения, основываясь только на цифрах; я лишь хочу, чтобы вы умели делать это при необходимости.
- Многие профессиональные дизайнеры игр пользуются электронными таблицами ежедневно, но я видел очень мало программ обучения проектированию игр, в рамках которых студентов учили бы, как ими пользоваться. Кроме того, курсы при университетах, где обучают приемам работы с электронными таблицами, в большей степени ориентированы на бизнес или бухгалтерский учет, чем на балансирование игр, и поэтому основной упор делают на другие особенности электронных таблиц, а не на те, которые помогают мне в работе.

Как и все остальное в разработке игр, процесс конструирования электронной таблицы является итеративным и довольно запутанным. Поэтому я не буду показывать в этой главе безупречные примеры создания электронных таблиц от начала до конца, придуманные заранее, а продемонстрирую шаги создания электронной таблицы, а также реальный итеративный процесс ее конструирования и планирования.

Выбор Google Sheets для этой книги

Для этой книги я решил использовать Google Sheets, потому что это бесплатная, кросс-платформенная и легкодоступная программа. Большинство других программ электронных таблиц обладают схожими свойствами (например, Microsoft Excel,

Open Office Calc и LibreOffice Calc Spreadsheet), но каждая имеет свои отличительные черты, поэтому попытка следовать инструкциям в этой главе, используя другую программу, может привести к разочарованию.

Дополнительную информацию по разным программам вы найдете во врезке «Не все программы электронных таблиц одинаковы».

НЕ ВСЕ ПРОГРАММЫ ЭЛЕКТРОННЫХ ТАБЛИЦ ОДИНАКОВЫ

Электронные таблицы чаще всего используются для управления и анализа больших объемов числовых данных. В числе современных популярных электронных таблиц можно назвать: Microsoft Excel, Apache OpenOffice Calc, LibreOffice Calc, Google Sheets и Apple Numbers.

- **Google Sheets** (http://sheets.google.com) — часть бесплатного набора онлайн-программ, доступного подписчикам Google Drive (Google Диск). Написанная на HTML5, она совместима с большинством современных веб-браузеров, но для эффективного ее использования у вас должно быть хорошее подключение к интернету. Качество Google Sheets значительно улучшилось с момента выхода первого издания этой книги, и теперь я предпочитаю эту программу. Еще одно преимущество этой программы — возможность синхронной работы сразу нескольких членов команды. Существуют также бесплатные версии для iOS и Android, которые позволяют работать без подключения к интернету.
- **Microsoft Excel** (http://office.microsoft.com) когда-то была одной из самых распространенных программ электронных таблиц, несмотря на дороговизну. Кроме того, между версиями для PC и macOS существуют некоторые отличия из-за разных графиков выпуска. В Excel используется тот же синтаксис определения формул, что и в Google Sheets, который еще считается индустриальным стандартом в деловых кругах, хотя на практике я выяснил, что эта программа медленнее и не такая удобная, как Google Sheets.
- **Apache OpenOffice Calc** (http://openoffice.org) — бесплатная программа с открытым исходным кодом, предназначавшаяся для предоставления тех же возможностей, что и Excel, но бесплатно для пользователя. Совместима с PC, macOS и Linux. Excel и OpenOffice Calc немного отличаются друг от друга, включая различия в пользовательском интерфейсе, но функционально они во многом идентичны. Одно из больших отличий: для разделения аргументов в формулах в OpenOffice используется точка с запятой (;), а в Excel и Google Sheets — запятая (,)[1]. В первом издании этой книги я использовал OpenOffice Calc, но теперь Google Sheets достигла достаточно высокого уровня совершенства, и я больше не пользуюсь Calc.

[1] Важно отметить, что если в настройках Google Sheets выбраны региональные настройки для России, тогда роль разделителя будет играть точка с запятой (;) и применение запятой (,) будет вызывать ошибку. — *Примеч. пер.*

- **Apple Numbers** (http://www.apple.com/numbers/) — включается в состав офисного пакета, распространяемого вместе с новыми компьютерами Mac, но также доступна для загрузки за 20 долларов США. Numbers работает только в macOS и включает некоторые интересные визуальные эффекты, недоступные в других программах, хотя я считаю, что они только мешают. Своими основными функциональными возможностями она сопоставима с другими программами, но на мой взгляд, это не самый удачный выбор.
- **LibreOffice Calc** (http://libreoffice.org) — бесплатная программа с открытым исходным кодом, предназначенная для предоставления тех же возможностей, что и Excel, но бесплатно для пользователя. Первоначально пакет LibreOffice был основан на исходном коде OpenOffice, поэтому у этих двух пакетов много общего. Одно небольшое преимущество LibreOffice перед OpenOffice для тех, у кого есть опыт использования Excel, заключается в использовании запятой в качестве разделителя параметров в формулах (как в Excel) вместо точки с запятой в OpenOffice.

Все эти программы могут открывать и экспортировать файлы Microsoft Excel, но при этом каждая поддерживает свой формат. Даже если у вас уже есть одна из перечисленных программ и вы умеете ею пользоваться, мне хотелось бы, чтобы вы дали шанс Google Sheets в этой главе.

Исследование вероятности с игральной костью в Sheets

Большая часть игровой математики сводится к вероятности, поэтому очень важно знать и понимать, как работает вероятность и случайность. Воспользуемся Google Sheets, чтобы исследовать распределение вероятности выпадения различных чисел для игральной кости 2d6 (два шестигранных кубика).

В одном броске кости 1d6 (одинарного шестигранного кубика) вы можете получить 1, 2, 3, 4, 5 или 6. Это достаточно очевидно. Однако ситуация становится намного интереснее, когда в игру вступает второй такой же кубик. Бросок кости 2d6 дает один из 36 разных вариантов:

```
Кубик А: 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6
Кубик Б: 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 5 5 5 5 5 5 6 6 6 6 6 6
```

Вы можете записать их вручную, но я хотел бы, чтобы вы начали учиться пользоваться электронными таблицами Google Sheets, которые потом помогут вам сбалансировать игру. Взгляните на рис. 11.5, чтобы увидеть, что у вас должно получиться.

Начало работы с Google Sheets

Для использования Google Sheets необходимо постоянное подключение к интернету. Это одна из отрицательных сторон программы, но она быстро превращается

в стандартный инструмент, которым пользуются многие дизайнеры, которых я знаю, поэтому вы тоже должны познакомиться с ней. Итак, чтобы начать исследовать математику игры, выполните следующее:

1. Откройте в веб-браузере страницу с адресом http://sheets.google.com.

   В результате вы попадете на главную страницу Sheets. Я настоятельно рекомендую использовать для работы браузер Google Chrome или Mozilla Firefox. Chrome обладает небольшим преимуществом, позволяя редактировать электронные таблицы Google Sheets в автономном режиме, но работа в этом режиме имеет ограничения и иногда может вызывать путаницу. Даже при использовании Chrome хорошо, если у вас будет постоянное подключение к интернету.

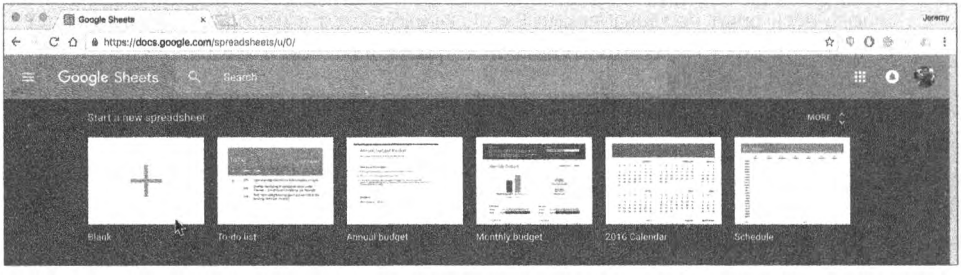


Рис. 11.1. Создание новой электронной таблицы на странице http://sheets.google.com

2. В разделе Start a new spreadsheet (Создать таблицу) щелкните на кнопке Blank (Пустой), как показано на рис. 11.1. В результате будет создана таблица Untitled spreadsheet (Новая таблица), как показано на рис. 11.2.

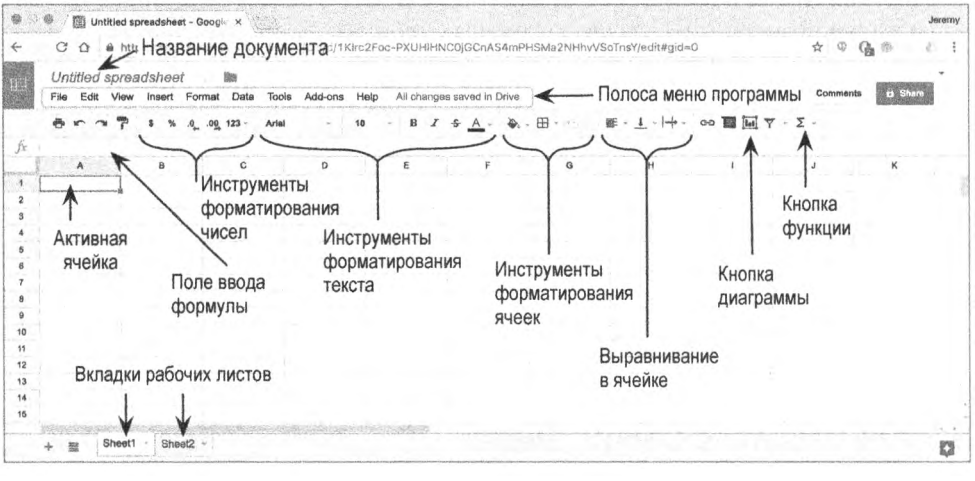


Рис. 11.2. Новая электронная таблица Google Sheets и некоторые важные элементы интерфейса

Начало работы с листами таблицы

У каждой ячейки электронной таблицы есть имя, состоящее из буквы, обозначающей столбец, и номера строки. Левая верхняя ячейка в электронной таблице названа A1. На рис. 11.2 ячейка A1 выделена синей рамкой и маленьким синим квадратиком в правом нижнем углу рамки, которые показывают, что эта ячейка является *активной*. Выполните следующие шаги, чтобы начать использовать таблицу:

1. Щелкните на ячейке A1, чтобы выбрать ее, и убедитесь, что она стала активной.
2. Введите число 1 с клавиатуры и нажмите Return. В ячейке A1 появится значение 1.
3. Щелкните на ячейке B1, введите =A1+1 и нажмите Return. В результате в ячейке B2 будет создана формула, вычисляющая значение на основе значения в ячейке A1. Все формулы начинаются с символа =. В ячейке B1 теперь должно появиться значение 2 (результат сложения значения в ячейке A1 с числом 1). Если теперь изменить число в ячейке A1, значение в ячейке B1 автоматически изменится.
4. Щелкните на ячейке B1 и скопируйте ее содержимое в буфер обмена (выберите пункт меню Edit > Copy (Правка > Копировать) или нажмите комбинацию Command-C в macOS или Ctrl+C на PC).
5. Нажмите клавишу Shift и, удерживая ее, щелкните на ячейке K1. В результате будут выделены ячейки B1:K1 (то есть все ячейки от B1 до K1; двоеточие в данном случае означает диапазон между двумя указанными ячейками). Вам может понадобиться использовать полосу прокрутки в нижней части окна Sheets, чтобы переместить ячейку K1 в видимую область.
6. Вставьте формулу из B1 в выделенные ячейки (выберите пункт меню Edit > Paste (Правка > Вставить) или нажмите комбинацию Command+V в macOS или Ctrl+V на PC). В результате формула из B1 будет скопирована в ячейки B1:K1 (то есть формула =A1+1). Поскольку ссылка на ячейку A1 в формуле является *относительной*, она изменится в зависимости от позиции новой ячейки, куда происходит вставка. Иными словами, в ячейку K1 будет вставлена формула =J1+1, потому что J1 — это ячейка, находящаяся слева от K1, так же как A1 — это ячейка слева от B1.

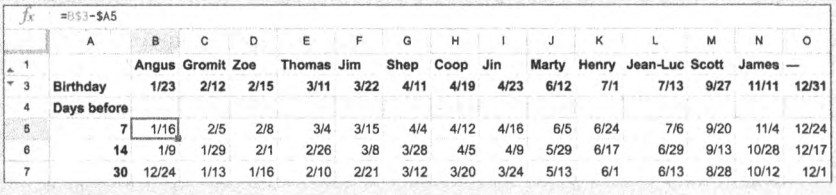
Дополнительную информацию об относительных и абсолютных ссылках вы найдете во врезке «Относительные и абсолютные ссылки».

ОТНОСИТЕЛЬНЫЕ И АБСОЛЮТНЫЕ ССЫЛКИ

Формула =A1+1 в ячейке B1 содержит *относительную ссылку* A1, то есть формула хранит *местоположение* ячейки, упомянутой в ссылке, *относительно* ячейки B1, а не абсолютную ссылку на ячейку A1. Иными словами, элемент A1 в этой формуле ссылается на ячейку, находящуюся непосредственно слева от ячейки с формулой (B1), и будет изменяться при копировании формулы в другие ячейки. Это важный аспект электронных таблиц, повышающий удобство работы с ними, в чем вы могли убедиться на шаге 6 в разделе «Начало работы с листами таблицы».

В Google Sheets также можно использовать *абсолютные ссылки* на ячейки, то есть ссылки, которые не изменяются при перемещении или копировании формулы. Чтобы сделать ссылку абсолютной, добавьте $ (знак доллара) перед именем столбца и номером строки. Например, чтобы сделать ссылку A1 абсолютной, ее нужно записать так: A1. Абсолютным можно сделать только имя столбца или номер строки, добавив $ только перед буквой столбца или номером строки.

На рис. 11.3 можно видеть пример частично абсолютных ссылок. Здесь я написал функцию, вычитающую разные числа из дат дня рождения моих друзей, чтобы знать, когда надо начинать искать подарки для них. Вы можете видеть эту формулу в ячейке B5, она имеет вид: =B$3-$A5. Я скопировал ее в ячейки B5:O7. Частичная абсолютная ссылка B$3 указывает, что имя столбца должно изменяться, а номер строки — нет, а частичная абсолютная ссылка $A5 сообщает, что номер строки должен изменяться, а имя столбца — нет.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		Angus	Gromit	Zoe	Thomas	Jim	Shep	Coop	Jin	Marty	Henry	Jean-Luc	Scott	James	—
3	Birthday	1/23	2/12	2/15	3/11	3/22	4/11	4/19	4/23	6/12	7/1	7/13	9/27	11/11	12/31
4	Days before														
5	7	1/16	2/5	2/8	3/4	3/15	4/4	4/12	4/16	6/5	6/24	7/6	9/20	11/4	12/24
6	14	1/9	1/29	2/1	2/26	3/8	3/28	4/5	4/9	5/29	6/17	6/29	9/13	10/28	12/17
7	30	12/24	1/13	1/16	2/10	2/21	3/12	3/20	3/24	5/13	6/1	6/13	8/28	10/12	12/1

Рис. 11.3. Пример частично абсолютных ссылок

Вот как изменится формула на рис. 11.3 после копирования в другие ячейки:

B5: =B$3-$ O5: =O$3-$A5

H6: =H$3-$A6

B7: =B$3-$A7 O7: =O$3-$A7

Выбор названия документа

Чтобы дать документу определенное название, выполните следующие действия:

1. Щелкните на словах Untitled spreadsheet (Новая таблица) в области «Название документа», как показано на рис. 11.2.
2. Измените название таблицы на «2d6 Dice Probability» и нажмите Return.

Создание строки чисел от 1 до 36

В результате выполнения предыдущих инструкций у вас должны появиться числа от 1 до 11 в ячейках A1:K1. Далее мы расширим строку, чтобы она содержала числа от 1 до 36 (36 возможных вариантов выпадения двух кубиков).

Добавление дополнительных столбцов

Сначала добавим несколько столбцов, чтобы уместить все ячейки. Прямо сейчас ячейки довольно широкие, и если вы попробуете прокрутить лист вправо, то увидите, что последний доступный столбец — Z. Прежде всего добавим дополнительные столбцы, чтобы можно было сохранить 36 разных чисел в одной строке, а затем сузим их.

1. Щелкните непосредственно на *заголовке столбца* A (то есть на поле A в верхней части столбца A).
2. Прокрутите лист вправо (воспользовавшись полосой прокрутки внизу окна с листом) и, удерживая нажатой клавишу Shift, щелкните на *заголовке столбца* Z. В результате будут выбраны все столбцы A:Z.
3. Если навести указатель мыши на заголовок столбца Z, появится кнопка со стрелочкой, направленной вниз. Щелкните на этой кнопке и в появившемся контекстном меню выберите пункт Insert 26 Right (Вставить справа: 26), как показано слева на рис. 11.4A. В результате справа будет добавлено еще 26 столбцов с именами AA:AZ.

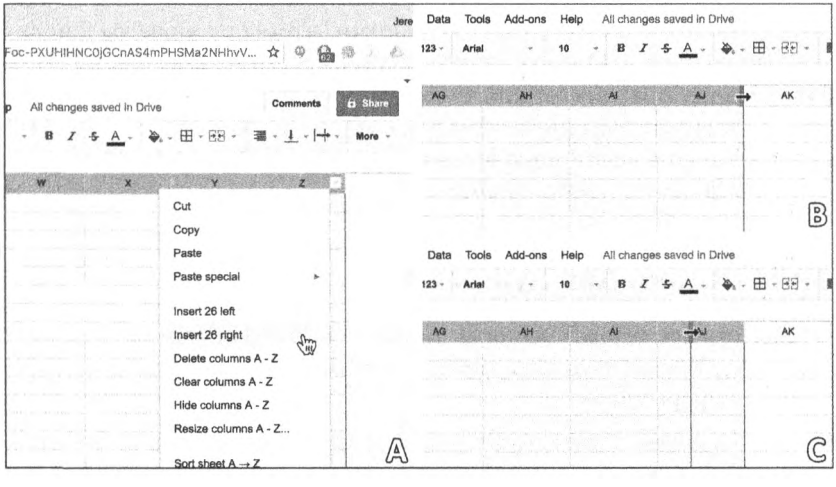


Рис. 11.4. Добавление 26 столбцов справа от столбца Z

Настройка ширины столбцов

Настроим ширину столбцов так, чтобы видеть на экране сразу 36 столбцов (A:AJ). Для этого выполните следующие шаги:

1. Щелкните на *заголовке столбца* A.
2. Прокрутите окно вправо и, удерживая нажатой клавишу Shift, щелкните на *заголовке столбца* AJ. В результате будут выбраны все 36 столбцов A:AJ.

3. Наведите указатель мыши на правую границу заголовка столбца AJ; вы увидите, как указатель мыши изменится и под ним появится утолщенная граница, окрашенная в синий цвет (как показано на рис. 11.4B справа).
4. Нажмите левую кнопку мыши и, удерживая ее, переместите влево, уменьшив ширину столбца AJ примерно до 1/3 первоначальной ширины (как показано справа на рис. 11.4C). В результате изменится ширина всех столбцов A:AJ. Если столбцы все еще не умещаются по ширине окна, можете сузить их еще больше.

Заполнение строки 1 числами от 1 до 36

Чтобы заполнить строку 1, выполните следующие шаги:

1. Щелкните на ячейке B1, чтобы выбрать ее. Другой способ скопировать содержимое одной ячейки в большой диапазон ячеек — использовать маленький синий квадратик в правом нижнем углу выбранной ячейки (который можно видеть в правом нижнем углу ячейки A1 на рис. 11.2).
2. Щелкните на синем квадратике и, удерживая нажатой кнопку мыши, перетащите его вправо, пока не будут выбраны ячейки B1:AJ1. Когда вы отпустите кнопку мыши, ячейки A1:AJ1 заполнятся последовательными числами от 1 до 36.
3. Если столбцы оказались слишком узкими, чтобы отобразить числа, снова выберите столбцы A:AJ и увеличьте их ширину до нужного размера. Вместо перетаскивания правой границы столбцов можно просто дважды щелкнуть на утолщенной синей границе между любыми двумя столбцами, после чего они примут оптимальную ширину; но в этом случае столбцы с одной цифрой будут у́же, чем столбцы с двумя цифрами.

Создание строки для кубика A

Теперь у вас есть последовательность чисел от 1 до 36, но нам нужны две строки для кубиков A и B, как было показано выше в этой главе. Сделать их можно с помощью простой формулы:

1. Щелкните на кнопке Function (Функция) (рис. 11.2) и в открывшемся контекстном меню выберите пункт More Functions... (Дополнительные функции...).
2. В текстовом поле Filter with a few keywords... (Фильтровать по ключевым словам) введите текст MOD (ОСТА), в результате список сократится и в нем останется меньше десятка функций. Найдите функцию с именем MOD (ОСТАТ) среди перечисленных с типом Math (Математические функции).
3. С правой стороны в строке с функцией MOD (ОСТАТ) найдите ссылку Learn more (Подробнее...) и щелкните на ней. В браузере откроется новая вкладка с описанием функции. Согласно описанию, функция MOD (ОСТАТ) делит одно число на другое и возвращает остаток. Например, две формулы, `=MOD(1,6)` и `=MOD(7,6)`, вернут число 1, потому что обе операции деления, 1/6 и 7/6, дают в остатке 1.

4. Вернитесь в таблицу, щелкнув на вкладке 2d6 Dice Probability в окне браузера.
5. Щелкните на ячейке A2, чтобы выбрать ее.
6. Введите `=MOD(A1,6)`[1] и нажмите Return. По мере ввода текста формула будет отображаться в ячейке A2 и в поле ввода формулы (см. рис. 11.2). По завершении ввода в ячейке A2 появится число 1.
7. Щелкните на ячейке A2 и, удерживая нажатой клавишу Shift, щелкните на ячейке AJ2 (чтобы выбрать ячейки A2:AJ2).
8. Нажмите комбинацию Command-R на клавиатуре (Ctrl+R на PC). В результате формула из самой левой ячейки (A2) будет скопирована во все остальные выбранные ячейки (B2:AJ2).

Теперь вы можете убедиться, что функция `MOD` возвращает верный результат для всех 36 ячеек; но хотелось бы получить числа в диапазоне от 1 до 6, а не от 0 до 5, поэтому выполним еще одну итерацию и скорректируем числа во второй строке.

Итерация корректировки строки для кубика A

Нам нужно исправить две проблемы: во-первых, наименьшее число должно быть в столбцах A, F, L и т. д.; во-вторых, числа должны быть в диапазоне от 1 до 6, а не от 0 до 5. Исправить обе проблемы можно следующими простыми изменениями:

1. Выберите ячейку A1, измените ее значение с 1 на 0 и нажмите Return (Enter на PC). В результате благодаря формулам изменения распространятся на ячейки B1:AJ1 и в первой строке появится последовательность чисел от 0 до 35. Теперь формула в ячейке A2 вернет 0 (остаток деления 0 на 6), а в ячейках A2:AJ2 появятся последовательности чисел 0 1 2 3 4 5, то есть первая проблема решена.
2. Чтобы исправить вторую проблему, выберите ячейку A2 и измените в ней формулу так: `=MOD(A1,6)+1`. Это изменение просто добавит 1 к результату предыдущей формулы, благодаря чему число 0 в ячейке A2 заменит число 1. Похоже, нас ждет утомительная работа в цикле, но не спешите: выполнив шаг 3, вы поймете, почему я так поступил.
3. Выберите ячейку A2, нажмите клавиши Shift-Command (Shift+Ctrl на PC) и, удерживая их, нажмите клавишу со стрелкой вправо. В результате будут выделены все непустые ячейки справа от A2, то есть A2:AJ2. Нажмите комбинацию Command-R в macOS (Ctrl+R на PC), чтобы снова *скопировать формулу из ячейки слева в ячейки справа*.

---

[1] Важно отметить, что если в настройках таблицы File > Spreadsheet Settings... > Locale (Файл > Настройки таблицы... > Региональные настройки) выбраны региональные настройки для России, тогда роль разделителя будет играть точка с запятой (;) и применение запятой (,) будет вызывать ошибку. Соответственно, с региональными настройками для России формула должна иметь вид `=MOD(A1;6)`. — *Примеч. пер.*

Теперь строка для кубика A готова, и в ней отображаются шесть серий чисел 1 2 3 4 5 6. Остатки от делений все еще находятся в диапазоне от 0 до 5, но теперь они следуют в нужном нам порядке, а благодаря добавлению `+1` в формулу получаются числа, соответствующие количеству очков на гранях кубика A.

Создание строки для кубика B

В строке для кубика B каждое число от 1 до 6 должно повторяться по шесть раз подряд. Чтобы получить желаемый результат, воспользуемся функциями деления и округления вниз. Функция деления работает в точном соответствии с нашими ожиданиями (например, `=3/2` вернет результат `1.5`); а вот функцию округления вниз кое-кто из вас мог не встречать прежде.

1. Выберите ячейку A3.
2. Введите в нее формулу `=FLOOR`, сразу после этого под ячейкой появится подсказка с текстом «FLOOR Rounds number down to nearest multiple of a factor» («Округляет число в меньшую сторону до ближайшего кратного...»). (По умолчанию «кратное» равно 1.)

   Функция `FLOOR` используется для округления вещественных чисел до целых, при этом округление всегда выполняется вниз. Например, `=FLOOR(5.1)` вернет 5, и `=FLOOR(5.999)` тоже вернет 5.
3. Введите `=FLOOR(A1/6)` в ячейку A3. После этого ячейка обновится и в ней появится результат 0.
4. Так же как мы делали это в строке для кубика A, нужно добавить 1 к результату формулы. Измените формулу в ячейке A3 так: `=FLOOR(A1/6)+1`; теперь в ячейке появится результат 1.
5. Скопируйте содержимое ячейки A3. Выделите ячейки A3:AJ3 и выполните вставку (Command-V в macOS, Ctrl+V на PC или выберите пункт меню Edit > Paste (Правка > Вставить)). В результате формула из ячейки A3 будет скопирована в ячейки A3:AJ3.

Теперь ваша таблица должна выглядеть так, как показано на верхней части рис. 11.5. Однако было бы понятнее, если бы строки и столбцы были подписаны так, как показано на нижней части рис. 11.5.

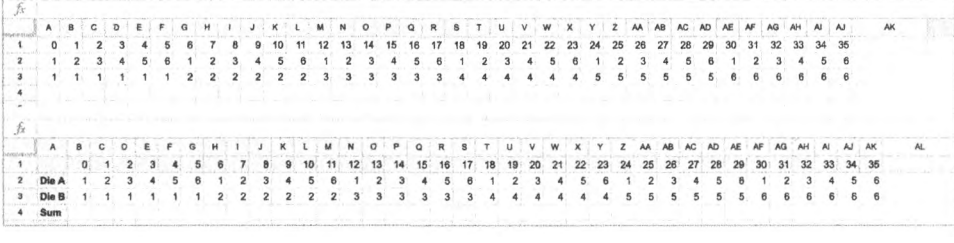


Рис. 11.5. Добавление ясности с помощью надписей

Добавление надписей

Чтобы добавить надписи, как показано на нижней части рис. 11.5, нужно вставить один столбец слева от столбца A:

1. Щелкните правой кнопкой мыши на *заголовке столбца* A и выберите в контекстном меню пункт Insert 1 left (Вставить слева: 1). В результате слева от текущего столбца A будет вставлен новый столбец. Новый столбец получит имя A, а старый столбец A теперь станет столбцом B. Если вы пользуетесь macOS и у вашей мышки нет правой кнопки, можете нажать клавишу Ctrl и, удерживая ее, щелкнуть левой кнопкой. В разделе «Щелчок правой кнопкой в macOS» в приложении Б «Полезные идеи» вы найдете еще несколько способов сымитировать щелчок правой кнопкой мыши в macOS.
2. Щелкните на новой пустой ячейке A2, чтобы выбрать ее, и введите текст `Die A`.
3. Чтобы расширить столбец и увидеть в нем всю надпись целиком, можно или дважды щелкнуть на границе справа в *заголовке столбца* A, или передвинуть эту границу вправо мышкой.
4. Введите текст `Die B` в ячейку A3.
5. Введите текст `Sum` в ячейку A4.
6. Чтобы отобразить текст в столбце A жирным шрифтом, щелкните на *заголовке столбца* A (чтобы выбрать столбец целиком) и нажмите комбинацию Command-B (Ctrl+B на PC) или щелкните на кнопке форматирования с изображением буквы «B» в области с инструментами форматирования текста (см. рис. 11.2).
7. Чтобы сделать фон столбца A серым, убедитесь, что столбец A все еще выбран целиком, и щелкните на кнопке выбора цвета заливки над полем ввода формулы (слева в блоке инструментов *форматирования ячеек*, см. рис. 11.2). В открывшейся палитре цветов выберите светло-серый. После этого фон в выделенных ячейках окрасится в светло-серый цвет.
8. Чтобы сделать фон строки 1 серым, щелкните на *заголовке строки 1* (с цифрой 1 слева в строке) и выберите строку целиком, а затем выберите тот же светло-серый цвет в меню выбора цвета заливки.

Теперь ваша таблица должна выглядеть так, как показано внизу на рис. 11.5.

В GOOGLE SHEETS НЕТ НЕОБХОДИМОСТИ СОХРАНЯТЬ ТАБЛИЦЫ! На протяжении этой книги — и особенно в учебных примерах в конце — я постоянно напоминаю, что вы должны сохранять файлы. Мне приходилось терять проделанную работу во многих программах из-за их краха и других компьютерных ошибок. Но я никогда не терял работу в Google Sheets, потому что эта программа постоянно сохраняет работу, которую я выполняю онлайн в облаке. Единственная проблема: если вы работаете в Google Sheets в браузере Chrome в автономном режиме и закроете окно до подключения к интернету, ваши изменения могут не сохраниться, хотя, как показали мои эксперименты, даже в этом случае таблицы сохранялись автоматически.

Суммирование результатов двух кубиков

Добавим еще одну формулу, которая суммирует результаты двух кубиков.

1. Щелкните на ячейке B4 и введите формулу `=SUM(B2,B3)`, которая складывает значения ячеек B2 и B3 (также можно ввести формулу `=B2+B3`). Она вставит значение 2 в ячейку B4.
2. Скопируйте ячейку B4 и вставьте в ячейки B4:AK4. Теперь строка 4 должна отображать результаты всех 36 возможных вариантов выпадения кости 2d6.
3. Чтобы выделить результаты на общем фоне, выделите всю строку 4, щелкнув на ее заголовке, и оформите отображение жирным шрифтом.

Подсчет сумм выпадений кубиков

Теперь строка 4 отображает результаты всех 36 возможных вариантов выпадения очков на двух шестигранных кубиках. Но несмотря на присутствие всех данных перед глазами, интерпретировать их все еще непросто. Это как раз та область, где можно найти реальное применение возможностям электронных таблиц. Чтобы приступить к интерпретации данных, создадим формулу подсчета количества выпадений каждой суммы (то есть подсчитаем, например, сколькими разными способами выпадает сумма 7). Сначала создадим вертикальную серию всех возможных сумм от 2 до 12:

1. Введите 2 в ячейку A7.
2. Введите 3 в ячейку A8.
3. Выберите ячейки A7 и A8.
4. Перетащите маленький синий квадратик справа внизу в ячейке A8 вниз, чтобы выделить ячейки A7:A17, и отпустите кнопку мыши.

   Ячейки A7:A17 заполнятся числами от 2 до 12. Программа Google Sheets поймет, что вы начали последовательность чисел, введя 2 и 3 в смежные ячейки, и при попытке перетащить эту серию она продолжит ее.

   Далее создадим формулу подсчета количества появлений значения 2 в строке 4:

5. Выберите ячейку B7 и введите `=COUNTIF` (но пока не нажимайте Return (Enter на PC)).
6. Щелкните на ячейке B4 и, не отпуская кнопку мыши, переместите указатель на ячейку AK4. В результате будут выделены ячейки B4:AK4 и в набираемую формулу добавится аргумент `B4:AK4`.
7. Введите запятую (`,`).
8. Щелкните на ячейке A7. В результате в формулу добавится аргумент `A7`. В этот момент введенная формула должна выглядеть так: `=COUNTIF(B4:AK4,A7`.
9. Введите закрывающую круглую скобку) и нажмите Return (или Enter в Windows). Теперь формула в ячейке B7 будет такой: `=COUNTIF(B4:AK4,A7)`.

Функция COUNTIF подсчитывает количество выполнения определенного критерия в серии. Первый параметр COUNTIF — это диапазон ячеек для поиска (в данном случае B4:AK4), а второй (после запятой) — искомое значение (значение ячейки A7, которое равно 2). В ячейке B7 функция COUNTIF просматривает все ячейки B4:AK4 и подсчитывает количество появлений числа 2 (встречается только один раз).

Подсчет вхождений всех возможных сумм

Далее распространим подсчет количества двоек на подсчет всех других сумм от 2 до 12 в вертикальной серии A7:A17:

1. Скопируйте формулу из B7 и вставьте ее в B7:B17.

   Вы заметите, что получившийся результат работает неправильно. Счетчики всех сумм, кроме 2, содержат 0. Давайте выясним, почему так получилось.

2. Выберите ячейку B7 и щелкните один раз в поле ввода формулы. В результате будут выделены все ячейки, которые просматривает формула в ячейке B7.

3. Нажмите клавишу Esc. Это важный шаг, потому что он производит выход из режима редактирования ячейки. Если не нажать Esc, щелчок на другой ячейке добавит ссылку на нее в формулу. Подробнее об этом рассказывается в предупреждении, следующем ниже.

> **ВЫХОД ИЗ РЕЖИМА РЕДАКТИРОВАНИЯ ФОРМУЛЫ.** При работе в Sheets вы должны нажимать Return, Tab или Esc (Enter, Tab или Esc на PC) для выхода из режима редактирования формулы. Нажимайте Return или Tab, чтобы принять изменения, или Esc, чтобы отменить их. Если не выйти из режима редактирования формулы, щелчок на любой ячейке добавит ссылку на нее в формулу (что для вас наверняка станет нежелательной случайностью). Если такое случится, нажмите Esc, чтобы выйти из режима формулы без фактического ее изменения.

4. Выберите ячейку B8 и щелкните один раз на поле ввода формулы.

   Теперь вы должны увидеть, в чем заключается проблема с формулой в B8. Вместо подсчета троек в B4:AK4 она ищет тройки в ячейках B5:AK5. Это результат автоматического обновления относительных ссылок, о котором рассказывалось выше в этой главе. Так как B8 находится на одну ячейку ниже, чем B7, все ссылки в B8 также изменились, сместившись на одну ячейку вниз. Это верно для второго аргумента в формуле (то есть формула в B8 должна искать число из A8, а не из A7), но в первом аргументе ссылки нужно сделать абсолютными, чтобы заставить функцию всегда просматривать строку 4, в какой бы ячейке она ни использовалась.

5. Нажмите Esc, чтобы выйти из режима редактирования B8.

6. Выберите B7 и измените формулу на `=COUNTIF(B$4:AK$4,A7)`. Знак доллара ($) в формуле создаст абсолютную ссылку на строку 4 в первом параметре функции COUNTIF.

7. Скопируйте формулу из B7 и вставьте ее в B7:B17. После этого вы увидите, как появятся правильные числа и все формулы в B7:B17 будут правильно просматривать ячейки B$4:AK$4.

Графическое представление результатов

Теперь ячейки B7:B17 отображают верные данные. В шести вариантах из 36 для игральной кости 2d6 выпадает 7, но на 2 и 12 приходится только по одному варианту. Эту информацию можно прочитать из чисел в ячейках, но представление в виде графической диаграммы было бы проще и понятнее. Выполните следующие шаги, чтобы создать диаграмму, отображающую частоту выпадения разных сумм. В процессе создания обращайтесь к изображениям окна Chart editor (Редактор диаграмм), показанным на рис. 11.6, на которых буквами в кружочках изображена последовательность этапов процесса. В нижнем ряду на рис. 11.6 показано, как изменяется диаграмма на каждом этапе.

1. Выберите ячейки A7:B17.
2. Щелкните на кнопке *вставки диаграммы*, как показано на рис. 11.2. (Если вы не видите кнопку диаграммы у себя на экране, значит, ваше окно слишком узкое; в этом случае щелкните на кнопке с изображением многоточия справа на панели инструментов, и в дополнительной всплывающей панели вы увидите кнопку диаграммы.) В результате откроется панель Chart editor (Редактор диаграмм), изображенная на рис. 11.6.
3. Щелкните на раскрывающемся списке Chart type (Тип диаграммы) — рис. 11.6A — и выберите первую пиктограмму в разделе тип Column (Столбчатые) — рис. 11.6B.
4. Внизу на вкладке DATA (ДАННЫЕ) в панели Chart editor (Редактор диаграмм) установите флажок Use column A as labels (Ярлыки — значения столбца A) — рис. 11.6C. Это превратит значения в столбце A из данных, отображаемых на диаграмме, в метки на оси X.
5. Щелкните на вкладке CUSTOMIZE (ДОПОЛНИТЕЛЬНЫЕ) — рис. 11.6D — и откройте область Chart & axis titles (Названия диаграмм и осей) — рис. 11.6E. Введите в поле Title text (Введите название) название `2d6 Dice Roll Probability` — рис. 11.6F.
6. Откройте область Horizontal axis (Горизонтальная ось) — рис. 11.6G. Внутри этой области установите флажок Treat labels as text (Читать ярлыки как текст) — рис. 11.6H. Благодаря этому ярлыки на горизонтальной оси будут отображаться под всеми столбиками.
7. Щелкните на кнопке закрытия панели Chart editor (Редактор диаграмм) — рис. 11.6I. Если хотите, можете переместить диаграмму и изменить ее размер.

Я знаю, что это был довольно утомительный путь к получению данных, но я хотел познакомить вас с электронными таблицами, потому что они могут быть чрезвычайно полезным инструментом для балансировки игры.

Рис. 11.6. Диаграмма распределения вероятностей для игральной кости 2d6

Математика вероятности

В данный момент вы могли бы подумать, что должен быть более простой путь выяснить вероятность выпадения разных значений, чем перебор всех вариантов. К счастью, есть целая область математики, занимающаяся вопросами вероятности, и в этом разделе мы рассмотрим некоторые правила из этой области.

Во-первых, определим количество возможных комбинаций для двух шестигранных кубиков (2d6). Для каждого кубика имеется 6 вариантов выпадения, соответственно, для двух кубиков таких вариантов $6 \times 6 = 36$. Для трех кубиков (3d6) число вариантов составляет $6 \times 6 \times 6 = 216$, или 6^3. Для восьми кубиков (8d6) число вариантов составит $6^8 = 1\,679\,616$! Это означает, что для вычисления параметров распределения вероятностей для игральной кости 8d6 понадобится электронная таблица умопомрачительных размеров, если вы решите использовать метод, описанный выше в этой главе.

В своей книге «The Art of Game Design» Джесси Шелл представил «Десять правил определения вероятностей, которые должен знать каждый геймдизайнер»[1], я повторю их здесь.

[1] Schell, *The Art of Game Design*, 155–163.

- **Правило 1: доли — это десятичные дроби и проценты.** Понятия «доли», «десятичные дроби» и «проценты» используются взаимозаменяемо, и вы часто будете ловить себя на мысли, что свободно перескакиваете между ними, рассуждая о вероятностях. Например, вероятность выпадения 1 при броске игральной кости 1d20 равна 1/20, или 0,05, или 5 %. Для преобразования одного в другое следуйте правилам ниже:
  - **Доли в десятичные дроби:** вычислите долю на калькуляторе. (Введите `1 ÷ 20 =`, и вы получите результат `0.05`.) Обратите внимание, что некоторые доли не имеют точного представления в виде десятичной дроби (например, 2/3 — точное значение доли, а 0,666666667 — ее приближенное значение в виде десятичной дроби).
  - **Проценты в десятичные дроби:** разделите на 100 (например, 5 % = 5 ÷ ÷ 100 = 0,05).
  - **Десятичные дроби в проценты**: умножьте на 100 (например, 0,05 = (0,05 × × 100) % = 5 %).
  - **Что угодно в доли**: иногда сделать это очень сложно из-за отсутствия простого пути преобразования десятичных дробей или процентов в доли, кроме некоторых широко известных случаев (например, 0,5 = 50 % = 1/2, 0,25 = 1/4).
- **Правило 2: вероятности изменяются в диапазоне от 0 до 1 (что эквивалентно диапазону от 0 до 100 % и от 0/1 до 1/1).** Вероятность чего бы то ни было никогда не может быть меньше 0 % и больше 100 %.
- **Правило 3: вероятность — это отношение числа «искомых исходов» к числу «возможных исходов».** Если вы бросаете шестигранный кубик и хотите получить 6, в этом случае имеется 1 искомый исход (6) и 6 возможных исходов (1, 2, 3, 4, 5 или 6). Вероятность выпадения 6 равна 1/6 (что примерно составляет 0,16666, или примерно 17 %). В обычной колоде с 52 игральными картами имеется 13 карт масти пик, поэтому если выбирать из колоды случайную карту, вероятность вытянуть карту пиковой масти равна 13/52 (то есть 1/4, или 0,25, или 25 %).
- **Правило 4: сложные математические задачи можно решать методом перебора.** Если количество возможных исходов невелико, для решения задачи с успехом можно использовать простой метод перебора, как было показано выше на примере составления таблицы для двух шестигранных кубиков. Если число возможных исходов велико (как, например, при использовании игральной кости 10d6, имеющей 60 466 176 возможных вариантов), можно написать компьютерную программу, осуществляющую их перебор. Приобретя некоторый опыт программирования, загляните в раздел «Вероятность игральной кости» приложения Б «Полезные идеи».
- **Правило 5: когда искомые исходы являются взаимоисключающими, их вероятности складываются.** В качестве примера Шелл приводит вероятность вытянуть из колоды карту с картинкой **ИЛИ** туза. В колоде имеется 12 карт с картинками (3 карты каждой масти) и 4 туза. Тузы и карты с картинками явля-

ются взаимоисключающими в том смысле, что нет карт, которые одновременно являлись бы картинками и тузами. Вопрос: какова вероятность вытянуть из колоды карту с картинкой **ИЛИ** туза? Ответ: 12/52 + 4/52 = 16/52 (0,3077 ≈ 31 %).

Какова вероятность выпадения 1, 2 **ИЛИ** 3 для кости 1d6? 1/6 + 1/6 + 1/6 = 3/6 (0,5 = 50 %). Всякий раз, когда вы используете союз **ИЛИ** для объединения взаимоисключающих искомых исходов, складывайте их вероятности.

- **Правило 6: когда искомые исходы не являются взаимоисключающими, их вероятности перемножаются.** Если вы хотите знать вероятность вытянуть карту, которая является картинкой **И** имеет масть пик, умножьте их вероятности. Поскольку вероятность всегда меньше или равна 1, произведение вероятностей почти всегда оказывается меньше вероятности каждого из вариантов. В колоде имеется 13 карт масти пик (13/52) и 12 карт с картинками (12/52). Их перемножение дает следующий результат:

$$13/52 \times 12/52 = (13 \times 12) / (52 \times 52)$$
$$= 156/2704 \qquad \textit{Числитель и знаменатель делятся на 52.}$$
$$= 3/52\ (0{,}0577 \approx 6\ \%)$$

Это действительно так, потому что, как известно, в колоде всего 3 карты-картинки масти пик (то есть 3 из 52).

Другим примером может служить вероятность выпадения 1 на одном кубике **И** 1 на другом кубике. Перемножая их, получаем 1/6 × 1/6 = 1/36 (0,0278 ≈ 3 %), и, как вы могли убедиться на примере использования электронной таблицы Google Sheets, вероятность выпадения двух единиц на двух кубиках действительно равна 1/36.

Запомните, если для объединения двух не взаимоисключающих искомых исходов используется союз **И**, их вероятности *перемножаются*.

Следствие: когда искомые исходы независимы, их вероятности перемножаются. Если два действия совершенно не зависят друг от друга (то есть не являются взаимоисключающими), вероятность их одновременного появления равна произведению вероятностей их появления по отдельности.

Вероятность выпадения шестерки на кубике (1/6) **И** вероятность выпадения решки при броске монеты (1/2) **И** вероятность вытянуть туз из колоды карт (4/52) равна 1/156 (1/6 × 1/2 × 4/52 = 6/624 = 1/156).

- **Правило 7: единица минус вероятность, что событие «случится», равна вероятности, что событие «не случится».** Вероятность, что какое-то событие произойдет, равна разности единицы и вероятности, что это событие не произойдет. Например, вероятность выпадения 1 на кубике равна 1/6. Это означает, что вероятность невыпадения 1 на кубике составляет 1 – 1/6 = 5/6 (0,8333 ≈ 83 %). Это очень полезное правило, потому что иногда проще узнать вероятность ненаступления события, чем вероятность его наступления.

Например, представьте, что вам нужно узнать вероятность выпадения 6 хотя бы на одном кубике из двух. Методом перебора вы без труда найдете ответ: 11/36 (искомыми исходами являются 6_x, x_6 и 6_6, где x — любое число, отличное от шести). Можно также подсчитать в таблице Google Sheets количество столбцов, в которых имеется хотя бы одно число 6. Но еще один способ — воспользоваться правилами 5, 6 и 7.

Вероятность выпадения 6 на одном кубике равна 1/6. Вероятность выпадения любого другого числа, кроме 6, равна 5/6, поэтому вероятность выпадения 6 на одном кубике **И** вероятность выпадения другого числа, кроме 6, на другом (то есть 6_x) составляет 1/6 × 5/6 = 5/36. (Как вы помните из правила 6, союз **И** означает умножение.) Поскольку искомый исход случается при выпадении 6_x **ИЛИ** x_6, сложим эти две вероятности: 5/36 + 5/36 = 10/36. (Правило 5: **ИЛИ** означает сложение.)

Выпадение 6 на одном кубике **И**, а выпадение 6 на другом кубике (6_6) составляет 1/6 × 1/6 = 1/36.

Поскольку все три случая (6_x, x_6 **ИЛИ** 6_6) являются взаимоисключающими, их вероятности нужно сложить: 5/36 + 5/36 + 1/36 = 11/36 (0,3055 ≈ 31 %).

Цепь рассуждений выше выглядит довольно сложной, но можно воспользоваться правилом 7, чтобы упростить ее. Например, можно решить обратную задачу поиска вероятности невыпадения 6 на двух кубиках, которая формулируется так: «Какова вероятность получить числа, отличные от 6, на первом **И** на втором кубике (то есть x_x)?» Эти два искомых исхода не являются взаимоисключающими, поэтому их можно перемножить! То есть вероятность получить числа, отличные от 6, на двух кубиках равна 5/6 × 5/6 = 25/36. Используя правило 7, получаем 1 – 25/36 = 11/36. Как видите, эта цепь рассуждений выглядит намного проще!

А теперь определим вероятность выпадения хотя бы одной 6 на четырех кубиках. Воспользовавшись правилом 7, мы легко найдем ответ:

$$
\begin{aligned}
&1 - (5/6 \times 5/6 \times 5/6 \times 5/6) \\
&= 1 - (5^4 / 6^4) \\
&= 1 - (625 / 1{,}296) \\
&= (1{,}296 / 1{,}296) - (625 / 1{,}296) && \textit{1,296/1,296 равно 1.} \\
&= (1{,}296 - 625) / 1{,}296 && \textit{Оба кратны 1,296.} \\
&= 671 / 1{,}296 \; (0{,}5177 \approx 52\,\%)
\end{aligned}
$$

В 52 % случаев хотя бы на одном кубике выпадет 6.

- **Правило 8: сумма очков, выпадающих на нескольких кубиках, распределяется нелинейно.** Как вы могли убедиться на примере исследования распределения вероятностей в Google Sheets, несмотря на то что для каждого отдельного кубика вероятность выпадения очков имеет линейное распределение, то есть у всех чисел 1–6 вероятность выпадения одинакова, когда в игру вступают несколько кубиков, вы получаете взвешенное распределение вероятности. Кривая рас-

пределения становится все более сложной с увеличением числа кубиков, как показано на рис. 11.7.

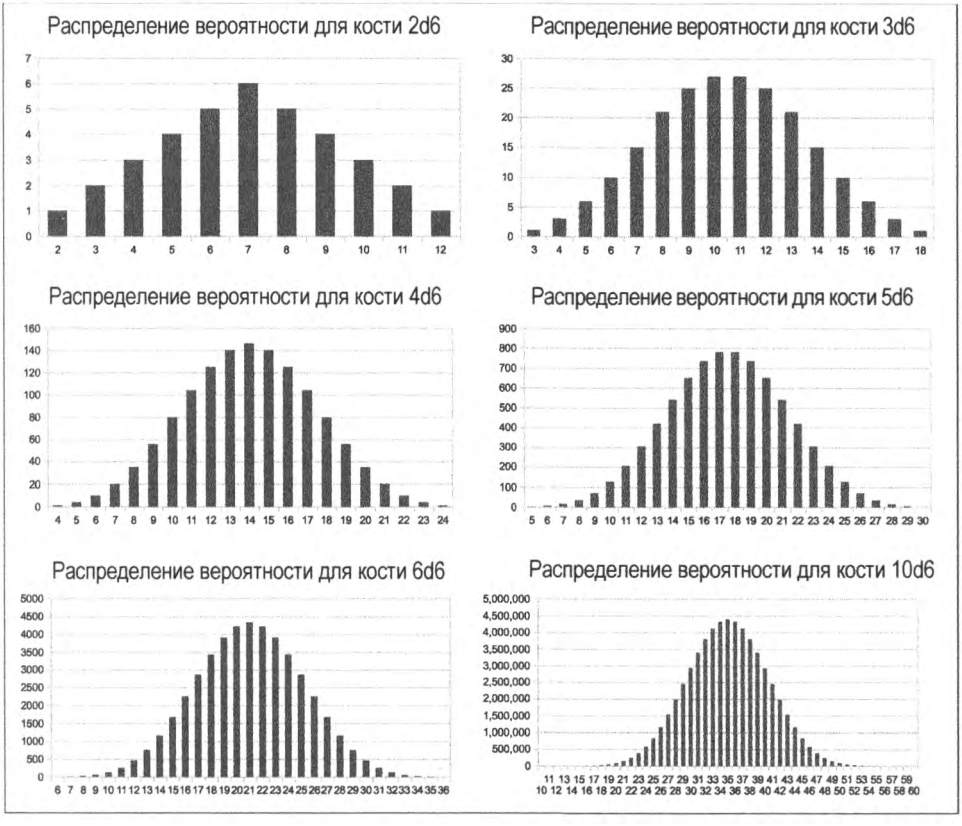


Рис. 11.7. Распределение вероятностей для игральных костей 2d6, 3d6, 4d6, 5d6 и 10d6

Как можно заметить на рис. 11.7, с увеличением числа кубиков растет вероятность выпадения средней суммы. Фактически для 10 кубиков вероятность выпадения всех 6 составляет 1/60 466 176, но вероятность выпадения суммы 35 составляет 4 395 456/60 466 176 (0,0727 ≈ 7 %), а вероятность выпадения чисел в диапазоне от 30 до 40 составляет 41 539 796/60 466 176 (0,6869922781 ≈ 69 %). При желании можно найти специальные математические статьи, описывающие вычисление этих значений по формулам, но я предпочел воспользоваться правилом 4 и написал программу, делающую эти вычисления (см. приложение Б).

Дизайнеру игр не так важно знать точные параметры этих распределений вероятности. Намного важнее помнить следующее: *чем больше кубиков должен бросать игрок, тем выше вероятность получить сумму очков, близкую к средней.*

- **Правило 9: теоретическая и практическая вероятности.** Вместо теоретических приемов определения вероятностей, о которых мы говорили выше, иногда проще использовать практические подходы. Существуют цифровые и аналоговые пути к этому.

  Следуя цифровым путем, можно написать простую компьютерную программу, выполняющую миллионы испытаний и определяющую исход. Этот подход часто называют методом Монте-Карло, и он используется некоторыми из лучших реализаций искусственного интеллекта, разработанных для игры в шахматы и Го. Игра Го настолько сложна, что до недавнего времени лучшим решением для компьютера было вычисление результатов миллионов случайных ходов компьютера и его соперника-человека и выявление статистически наиболее выгодного хода. Этот подход также можно использовать для поиска ответов к очень сложным для теоретического решения задачам. В качестве примера реализации этого правила Шелл приводит компьютерную программу, способную быстро смоделировать миллионы бросков кубика в игре *Monopoly* и позволить узнать программисту, в какие клетки на доске игроки попадут вероятнее всего.

  Другой аспект этого правила состоит в том, что не все игровые кубики одинаковы. Например, если вы хотите выпустить настольную игру и ищете производителя для игровых кубиков, будет полезно приобрести сначала по паре кубиков каждого потенциального производителя и бросить каждую пару несколько сотен раз, записывая результаты бросков. Для этого может потребоваться час или даже больше, но результаты помогут вам выяснить, насколько хорошо сбалансированы кубики и не выпадает ли на них какое-то число чаще других.

- **Правило 10: звонок другу.** Почти все студенты колледжей, занимающиеся изучением информатики и математики, должны пройти курс изучения теории вероятностей. Если вы столкнулись со сложной задачей, основанной на вероятности, которую вы не можете решить самостоятельно, попробуйте обратиться за помощью к такому другу. Фактически, как утверждает Шелл, исследование теории вероятностей началось еще в 1654-м, когда шевалье де Мере (Chevalier de Méré)[1] не смог определить, почему вероятность выпадения 6 в четырех бросках одного кубика оказалась выше, чем вероятность выпадения суммы 12 в 24 бросках двух кубиков, хотя, казалось бы, все должно быть наоборот. Шевалье де Мере обратился за помощью к своему другу Блезу Паскалю. Паскаль написал другу своего отца — Пьеру де Ферма, и их переписка стала основой для исследования вероятностей[2].

В приложение Б «Полезные идеи» я добавил программу для Unity, которая вычисляет распределение выпадения очков для кубиков с любым количеством граней (правда, она требует некоторого времени для вычислений).

---

[1] https://ru.wikipedia.org/wiki/Гомбо,_Антуан. — *Примеч. пер.*

[2] Schell, *The Art of Game Design*, 154.

Технологии внесения элемента случайности в настольные игры

К числу наиболее типичных источников случайности в настольных играх относятся кубики, вертушки и колоды карт.

Кубики

В этой главе уже было представлено много информации о кубиках. Вот наиболее важные выводы из обсуждения выше:

- Единственный кубик генерирует последовательность случайных чисел с линейным распределением.
- С увеличением числа кубиков растет вероятность выпадения средней суммы (и распределение уходит все дальше от линейного), и форма кривой распределения становится все более колоколообразной.
- К стандартным размерам кубиков относятся: d4, d6, d8, d10, d12 и d20. Обычно в наборы кубиков для игр входят: 1d4, 2d6, 1d8, 2d10, 1d12 и 1d20.
- Кости 2d10 иногда называют *процентильными*, потому что одна кость из этой пары определяет единицы (размечена числами от 0 до 9), а другая десятки (размечена числами 00 до 90 с шагом 10) процентов. Эта пара дает равномерное распределение чисел в диапазоне от 00 до 99 (выпадение значений 0 и 00 обычно принимается за 100 %).

Вертушки

Доступно несколько видов вертушек, но все они имеют вращающийся и неподвижный элементы. В большинстве настольных игр вертушка состоит из картонной основы, разделенной на сегменты, и смонтированной на ней вращающейся пластиковой стрелки (рис. 11.8A). Большие вертушки (например, барабан в телешоу «Поле чудес») часто состоят из вращающегося колеса, разбитого на сегменты, и неподвижной стрелки над ним (рис. 11.8B). Если игроки вращают вертушку с достаточной силой, она дает примерно равномерное распределение вероятностей, подобно кубику.

Вертушки часто используются в детских играх по двум основным причинам:

- У маленьких детей пока недостаточный контроль над моторикой, чтобы вбросить кубики в пределах небольшой области, поэтому они часто бросают кости так, что те выкатываются за пределы игрового стола.
- Маленькие дети намного реже пытаются проглотить вертушки.

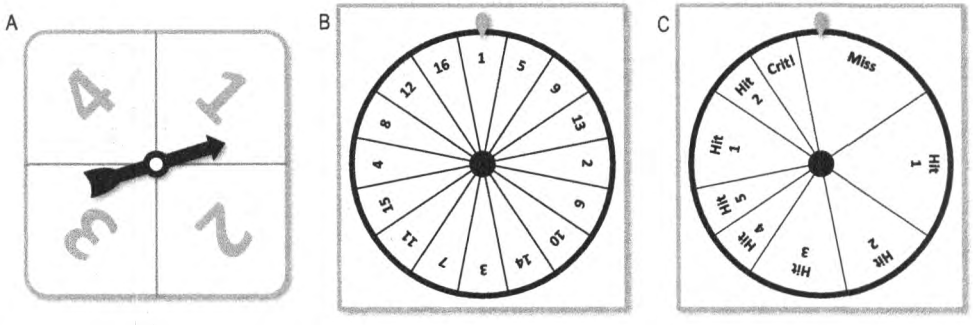


Рис. 11.8. Разные вертушки. На всех изображениях серые элементы — неподвижные, а черные — вращающиеся

Несмотря на меньшую распространенность в играх для взрослых, вертушки обладают некоторыми интересными возможностями, отсутствующими у кубиков:

- Поверхность вертушки можно разделить на любое число сегментов, тогда как создать кубик с 3, 7, 13 или 200 гранями очень сложно[1].
- Вертушки очень легко взвешивать, чтобы обеспечить неравные шансы выпадения разных секторов. На рис. 11.8C изображена гипотетическая вертушка для использования атакующим игроком. Эта вертушка дает игроку следующие вероятности:
  - вероятность 3/16 промазать (Miss);
  - вероятность 1/16 попасть 4 снарядами (Hit 4);
  - вероятность 5/16 попасть 1 снарядом (Hit 1);
  - вероятность 1/16 попасть 5 снарядами (Hit 5);
  - вероятность 3/16 попасть 2 снарядами (Hit 2);
  - вероятность 1/16 нанести критическое поражение (Crit!);
  - вероятность 2/16 попасть 3 снарядами (Hit 3).

Колоды карт

Стандартная колода включает по 13 игральных карт четырех разных мастей и иногда еще два джокера (рис. 11.9). В том числе карты ценностью от 1 (также иногда называются тузами) до 10, валет, дама и король каждой из четырех мастей: трефы, бубны, червы и пики.

[1] Часто для этого создаются кости в форме мяча для американского футбола с нужным количеством плоских граней. Поищите в интернете по фразе «oblong dice», «crystal dice» или «d7 dice», чтобы увидеть примеры.

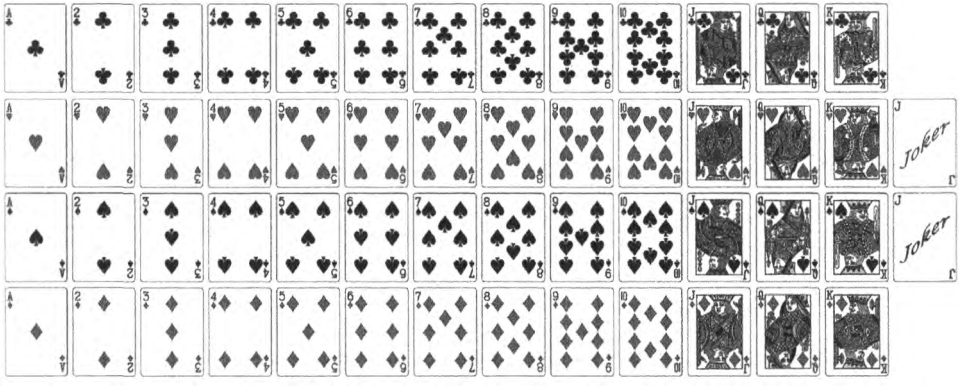


Рис. 11.9. Стандартная колода игральных карт с двумя джокерами[1]

Игральные карты пользуются большой популярностью благодаря компактности и множеству разных способов их раздачи.

С колодой без джокеров вы имеете следующие вероятности:

- вероятность вытянуть конкретную карту: 1/52 (0,0192 ≈ 2 %);
- вероятность вытянуть карту определенной масти: 13/52 = 1/4 (0,25 = 25 %);
- вероятность вытянуть карту с картинкой (В, Д или К): 12/52 = 3/13 (0,2308 ≈ 23 %).

Нестандартные колоды карт

Колода карт — один из самых простых и настраиваемых источников случайности, который можно добавить в настольную игру. Вы можете добавлять или удалять конкретные карты, чтобы изменить вероятность вытянуть эту карту из колоды. Более подробно об этом рассказывается в разделе, посвященном взвешенному распределению, далее в этой главе.

СОВЕТЫ ПО СОЗДАНИЮ НЕСТАНДАРТНЫХ КОЛОД КАРТ

Одна из сложностей на пути создания нестандартных карт — подобрать для их изготовления материал, позволяющий легко тасовать колоду таких карт. Бумага для заметок 3 x 5 плохо подходит на эту роль, но есть пара неплохих вариантов.

- Использовать маркер или наклейки для изменения имеющихся карт. Метки маркером — хорошая идея, и они не изменяют толщину карт, как наклейки.

[1] Комплект векторных графических изображений карт Vectorized Playing Cards 1.3 (http://sourceforge.net/projects/vector-cards/ — был доступен в апреле 2018) ©2011, Крис Агиляр (Chris Aguilar), распространяется на условиях лицензии LGPLv3. www.gnu.org/copyleft/lesser.html. (Текст лицензии на русском языке можно найти по адресу http://licenseit.ru/wiki/index.php/GNU_Lesser_General_Public_License_version_3. — *Примеч. пер.*)

- Купите колоду протекторов для карт и вставьте в них листы бумаги вместе с обычными картами, как описывалось в главе 9 «Прототипирование на бумаге».

Главное, чего не нужно делать при создании колоды (и любого другого элемента бумажного прототипа), — уделять слишком много времени любой конкретной части. Посвятив время созданию большого количества хороших карт (например), вам будет жалко убрать любую из них из прототипа или отказаться от них полностью и начать все сначала.

Конструирование цифровой колоды

Недавно для создания своих прототипов я начал пользоваться инструментами для создания цифровых колод карт, такими как *nanDECK* (http://www.nand.it/nandeck/). *nanDECK* — это программное обеспечение для Windows, которое позволяет создавать колоды карт на простом языке разметки (немного напоминающем HTML). Моя любимая фишка *nanDECK* — возможность извлекать данные карты из онлайн-файла Google Sheets и превращать их в полную колоду карт. В этой книге не так много места, чтобы можно было во всех деталях описать работу с этим инструментом, поэтому я рекомендую заглянуть на сайт *nanDECK*. Кроме руководства в формате PDF на веб-сайте проекта, можно найти несколько интересных видеоуроков на YouTube, если поискать по слову «nanDECK».

Когда тасовать колоду

Перемешивание колоды карт перед каждым вытягиванием очередной карты уравнивает вероятности вытягивания карт (как в случае, когда вы бросаете кубик или вращаете вертушку). Однако большинство использует колоду карт совсем иначе. Обычно люди вытаскивают карты из колоды, пока не вынут последнюю, после чего перетасовывают колоду. При таком подходе колода карт работает совершенно иначе, чем кубик. Если у вас есть колода с шестью картами, пронумерованными от 1 до 6, и вы последовательно вытягиваете все карты из колоды до перемешивания, вы гарантированно увидите каждое из шести чисел 1–6 в каждой серии вытягивания шести карт. Шесть бросков кубика не дают такой гарантии. Еще одно отличие заключается в том, что игроки могут считать карты и определять, какие карты остались в колоде, что позволяет им судить о вероятности вытягивания следующей карты. Например, если из колоды с шестью картами были вытянуты карты 1, 3, 4 и 5, с 50 % вероятностью следующей картой будет 2 или 6.

Это различие между колодами и кубиками было подмечено в игре *Settlers of Catan*. Некоторые игроки были настолько огорчены разницей между теоретическим и фактическим распределениями вероятностей выпадения очков на двух шестигранных кубиках, что производитель решил включить в комплект игры колоду из 36 карт (размеченных вариантами выпадения очков на двух кубиках), которую можно ис-

пользовать взамен кубиков. Это гарантировало полное соответствие практического и теоретического распределений вероятностей.

Взвешенные распределения

Взвешенным называют такое распределение, где некоторые варианты имеют большую вероятность появления, чем другие. Большинство примеров источников случайности, которые вы видели до сих пор, обеспечивают равномерное распределение вероятностей, но дизайнерам часто требуется придать каким-то вариантам больший вес. Например, дизайнеры настольной игры *Small World* хотели в половине случаев дать атакующему игроку дополнительные случайные очки от +1 до +3 за финальную атаку в каждом ходе. Для этого они создали шестигранный кубик, изображенный на рис. 11.10.

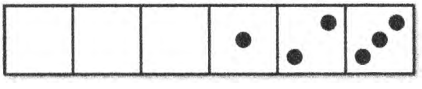


Рис. 11.10. Кубик для начисления дополнительных очков атакующему в игре Small World со взвешенным распределением вероятностей

Вероятность не получить дополнительных очков с таким кубиком равна 3/6 = = 1/2 (0,5 = 50 %), а вероятность получить 2 очка равна 1/6 (0,1666 ≈ 17 %), то есть вероятность получить 0 очков намного выше вероятностей трех других вариантов.

А как быть, если желательно, чтобы игрок все так же получал дополнительные очки только в половине случаев, но при этом вероятность выпадения 1 была в три раза выше, чем 3, а вероятность выпадения 2 — в два раза выше, чем 3? Обеспечить такое взвешенное распределение может кубик, изображенный на рис. 11.11.

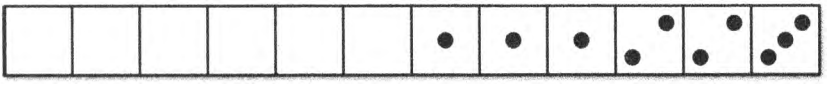


Рис. 11.11. Кубик с вероятностью 1/2 выпадения 0, 1/4 выпадения 1, 1/6 выпадения 2 и 1/12 выпадения 3

К счастью, такую игральную кость можно получить из обычного додекаэдра. В случаях, когда подобрать распространенную геометрическую фигуру невозможно, всегда можно создать вертушку или колоду карт с тем же распределением вероятностей (правда, колоду карт пришлось бы перетасовывать перед вытягиванием каждой карты). Смоделировать взвешенное распределение случайных исходов можно также в Sheets. Процесс очень напоминает работу со случайными числами, которую мы будем выполнять позднее в Unity и C#.

Взвешенное распределение вероятностей в Sheets

Взвешенное распределение — обычное дело в цифровых играх. Например, чтобы при столкновении с игроком в 40 % случаев враг нападал на него, в 40 % случаев выбирал оборонительную тактику и в 20 % пытался убежать, можно создать массив значений [Атака, Атака, Оборона, Оборона, Побег][1] и заставить код искусственного интеллекта врага выбирать из этого массива случайное значение при обнаружении игрока.

Выполните следующие шаги, чтобы создать лист в Sheets, который можно использовать для случайного выбора значений из последовательности. Изначально этот пример реализует выбор случайного числа в диапазоне от 1 до 12. Завершив работу над листом, вы сможете заменить варианты в столбце A на любые по своему выбору.

	A	B	C
1		# Choices:	12
2		Random:	0.4843701814
3		Index:	6
4		Result:	6
5			
6			
7			
8			
9			
10			
11			
12			

Рис. 11.12. Таблица Google Sheets для выбора случайных чисел со взвешенной вероятностью

1. Добавьте новый лист в существующий документ Sheets, щелкнув на кнопке с символом «плюс» слева от вкладки Sheet 1 (Лист1) в нижней части окна (рис. 11.2 выше в этой главе).
2. В новом листе заполните столбцы A и B, как показано на рис. 11.12, но столбец C пока оставьте пустым. Чтобы выровнять текст по правому краю в столбце B, выделите ячейки B1:B4 и выберите пункт Format > Align > Right (Формат > Выровнять > По правому краю) в полосе меню в окне браузера, которая на рис. 11.2 выделена серой рамкой.

[1] Квадратные скобки (то есть [ ]) используются в языке C# для определения массивов (группы значений), поэтому я также использую их для группировки пяти возможных вариантов действий.

3. Выберите ячейку C1 и введите формулу `=COUNTIF(A1:A100,"<>")`. Она подсчитывает количество непустых ячеек в диапазоне A1:A100 (оператор <> в формулах Sheets означает «отличное от», а с отсутствующим значением справа от него означает «отличное от ничего», то есть «непустая ячейка»). Эта формула вернет количество доступных вариантов, перечисленных в столбце A (в настоящее время их 12).
4. В ячейку C2 введите формулу `=RAND()`, которая генерирует числа в диапазоне от 0 (включительно) до 1 (не включая его)[1].
5. Выберите ячейку C3 и введите формулу `=FLOOR(C2*C1)+1`. Она округляет вниз произведение случайного числа в диапазоне от 0 до ≈ 0,9999 на число возможных вариантов, которое в данном случае равно 12. Это означает, что после округления чисел от 0 до ≈ 11,9999 получится целое число в диапазоне от 0 до 11. Затем прибавляется 1, в результате получается случайное число в диапазоне от 1 до 12.
6. В ячейку C4 введите формулу `=INDEX(A1:A100,C3)`. Функция INDEX() принимает диапазон значений (например, A1:A100) и выбирает элемент с указанным индексом (в данном случае с индексом в C3, который может быть числом от 1 до 12). Теперь в ячейке C4 отображается случайное значение из списка в столбце A.

Чтобы получить другое случайное значение, скопируйте ячейку C2 и вставьте обратно в ячейку C2. Это заставит функцию RAND выполниться повторно. Точно так же пересчет запускается при любом изменении в таблице (например, вы можете ввести 1 в ячейку E1 и нажать Return, чтобы инициировать пересчет). Всякий раз, когда изменяется какая-нибудь ячейка в таблице, повторно вызывается функция RAND.

В ячейки столбца A можно поместить числа или текст, главное, чтобы не было пустых строк. Попробуйте заменить числа в ячейках A1:A12 взвешенными значениями, изображенными на рис. 11.11 (то есть [0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 3]). Если вы сделаете это и попробуете пересчитать случайное значение в C2 несколько раз, то заметите, что в половине случаев в ячейке C4 выпадает 0. Можете также заполнить ячейки A1:A5 словами [Атака, Атака, Оборона, Оборона, Побег] и посмотреть на взвешенный выбор искусственного интеллекта врага, как описывалось в примере в начале этого раздела.

Перестановки

Есть одна старая игра с названием *Bulls and Cows* («Быки и коровы», рис. 11.13), которая стала основой для популярной настольной игры *Master Mind* (создана Мордехаем Меировичем (Mordecai Meirowitz) в 1970 году). В этой игре каждый игрок записывает секретный четырехзначный числовой код (в котором все цифры

[1] «0 (включительно)» означает, что формула может вернуть число 0, а «1 (не включая его)» означает, что возвращаемое число никогда не будет равно 1 (хотя оно может быть равно 0,99999999).

разные). Затем игроки по очереди пытаются угадать коды своих соперников, и первый угадавший считается победителем. Когда игрок высказывает предполагаемый код, его соперник, чей код пытаются угадать, сообщает количество *быков* и *коров*. Бык дается за каждую угаданную цифру в коде, если она стоит на своем месте, а корова — за каждую угаданную цифру, но не на своем месте. На рис. 11.13 серые кружки представляют быков, а белые — коров.

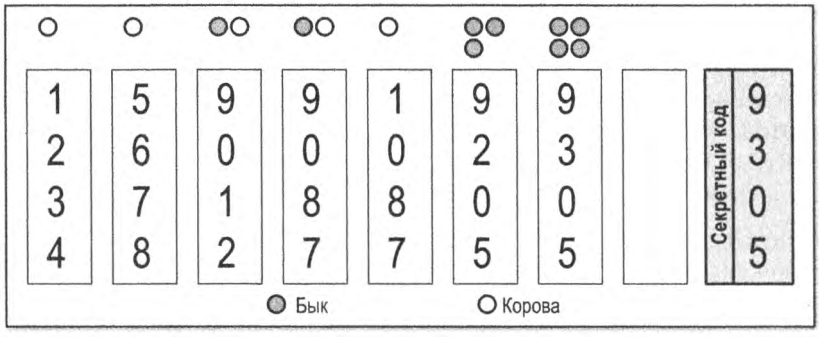


Рис. 11.13. Пример игры «Быки и коровы»

С точки зрения угадывающего, секретный код фактически является последовательностью случайных выборов. Математики называют такие серии *перестановками*. Секретный код в игре *Bulls and Cows* — это перестановка десяти цифр 0–9, из которой выбираются четыре неповторяющиеся цифры. В игре *Master Mind* имеется восемь возможных цветов, из которых выбираются четыре неповторяющихся. В обоих случаях код является перестановкой, а не *комбинацией*, потому что позиции элементов имеют значение (9305 — это не то же самое, что 3905). Комбинация — это набор вариантов, позиция которых не имеет значения. Например, 1234, 2341, 3421, 2431 и т. д. — все это одна и та же комбинация. Хорошим примером комбинации может служить комбинация трех видов наполнителя в мороженом: совершенно неважно, в каком порядке они добавлялись, главное, чтобы они присутствовали.

Перестановки с повторяющимися элементами

Математические основы для работы с перестановками, допускающими повторяющиеся элементы, немного проще, поэтому начнем с них. В случае с четырьмя цифрами, когда допускаются повторения, возможны 10 000 комбинаций (числа от 0000 до 9999). Это легко увидеть, глядя на числа, но нам нужен более обобщенный способ представления (на случай, если возможное число вариантов в каждой позиции не равно 10). Поскольку каждая цифра независима от других и имеется 10 возможных вариантов, согласно правилу 6 определения вероятностей, вероятность получить любое конкретное число равна $1/10 \times 1/10 \times 1/10 \times 1/10 = 1/10\,000$.

Получившийся результат также сообщает, что имеется 10 000 возможных вариантов секретного кода (если повторения допустимы).

Обобщенный расчет перестановок с повторениями заключается в перемножении количества вариантов в каждой позиции. Для четырех позиций и десяти вариантов получается: 10 × 10 × 10 × 10 = 10 000. Если вы хотите составить код из очков на шести гранях кубика, тогда для каждой позиции будут возможны шесть вариантов: 6 × 6 × 6 × 6 = 1296 возможных вариантов.

Перестановки без повторяющихся элементов

А что получается в случае с игрой *Bulls and Cows*, где не разрешается повторять цифры? В действительности все намного проще, чем вы могли бы подумать. После использования цифры она становится недоступной. То есть для первой позиции имеется выбор из цифр 0–9, но после выбора цифры (например, 9) для второй позиции остается на выбор уже 9 вариантов (0–8). Так продолжается для всех остальных позиций, то есть число возможных кодов в игре *Bulls and Cows* составляет: 10 × 9 × 8 × 7 = 5040. Недопустимость повторений исключает почти половину возможных вариантов.

Использование Google Sheets для балансировки оружия

Еще одно применение математики и программ, таких как Google Sheets, в проектировании игр заключается в балансировке разных видов оружия и возможностей. В этом разделе вы увидите процесс балансировки оружия для игры, похожей на *Valkyria Chronicles* компании Sega. В этой игре у каждого оружия есть три важные характеристики:

- Количество выстрелов, производимых одновременно.
- Степень повреждений, производимых каждым выстрелом.
- Вероятность попадания в цель каждой выпущенной пули на данной дистанции.

При балансировке разных видов оружия обычно преследуется цель сделать их примерно равными по мощности, но при этом придать отличительные черты. Например, такие отличительные черты могут быть у некоторых видов оружия:

- **Пистолет**: основное оружие; позволяет справляться с большинством ситуаций, но не самое лучшее.
- **Винтовка:** хороший выбор для поражения целей на средних и дальних дистанциях.
- **Дробовик:** убойное оружие на коротких дистанциях, но его мощность быстро падает с увеличением расстояния; имеет низкую скорострельность, поэтому точность имеет большое значение.

- **Снайперская винтовка:** крайне неудачный выбор для ближнего боя, но фантастическое оружие для стрельбы на дальние дистанции.
- **Пулемет:** обладает высокой скорострельностью, поэтому даже при не самой высокой точности обладает высокой убойной силой, благодаря этому кажется самым надежным оружием, хотя и не самым мощным.

На рис. 11.14 показаны значения характеристик оружия, которые мне показались удачными на первый взгляд. Значение ToHit (Для попадания) — это минимальное количество очков, выпавшее при броске шестигранного кубика, которое означает попадание в цель на данном расстоянии. Например, для пистолета в ячейке K3, соответствующей расстоянию 7 единиц, указано число 4, то есть если игрок делает выстрел на расстоянии 7 единиц, будет считаться, что он попал в цель, если на кубике выпадет число 4 или больше. Это дает 50 % вероятность попадания (потому что попадание засчитывается, если выпадет 4, 5 или 6 очков).

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	Weapon	Shots	D/Shot		ToHit											Percent Chance										
2	Original				1	2	3	4	5	6	7	8	9	10		1	2	3	4	5	6	7	8	9	10	
3	Pistol	4	2		2	2	2	3	3	4	4	5	5	6												
4	Rifle	3	3		4	3	2	2	2	3	3	3	4	4												
5	Shotgun	1	10		2	2	3	3	4	5	6															
6	Sniper Rifle	1	8		6	5	4	4	3	3	2	2	3	4												
7	Machine Gun	6	1		3	3	4	4	5	5	6	6														

Рис. 11.14. Начальные значения характеристик оружия в электронной таблице для балансировки

Определение вероятности попадания для каждой пули

В ячейках под заголовком Percent Chance (Процент вероятности) нам нужно рассчитать вероятность попадания в цель каждой пули, выпущенной из данного оружия на данной дистанции. Для этого выполните следующие шаги.

1. Создайте новый документ в Google Sheets и введите все данные, как показано на рис. 11.14. Чтобы изменить цвет фона ячеек, используйте кнопку Cell Color (Цвет заливки) в инструментах форматирования ячеек (рис. 11.2).

Судя по значению в ячейке E3, выстрел из пистолета на расстоянии в 1 единицу достигает цели, если при броске кубика выпадет 2 очка или больше. Это означает, что промах происходит, только когда выпадает 1 очко, а при выпадении 2, 3, 4, 5 или 6 очков выстрел достигает цели. Значит, доля попаданий составляет 5/6 (или ≈ 83 %), и теперь нам нужна формула для ее вычисления. Согласно правилу 7 определения вероятностей, это соответствует вероятности промаха, равной 1/6 (то есть число в ToHit минус 1 и деленное на 6).

2. Выберите ячейку P3 и введите формулу `=(E3-1)/6`. В результате в ячейке P3 отобразится вероятность промаха из пистолета на расстоянии в 1 единицу. Арифметические операции в Sheets выполняются в соответствии с правилами

математики, поэтому деление выполнится перед вычитанием, если не заключить выражение `E3-1` в круглые скобки.

3. Снова воспользуемся правилом 7. Согласно ему, 1 — вероятность промаха = вероятность поражения, поэтому замените формулу в ячейке P3 на `=1-((E3-1)/6)`. Теперь в ней должно появиться число 0,8333333.
4. Чтобы преобразовать десятичную дробь в ячейке P3 в проценты, выберите ячейку P3 и щелкните на кнопке с символом % в области инструментов форматирования чисел, как показано на рис. 11.2. Можно также пару раз щелкнуть на кнопке, находящейся правее кнопки с символом % (с надписью .0 и стрелкой влево). Она удалит десятичные знаки, и в ячейке будет отображаться только число 83% вместо более точного, но запутывающего %83.33. В этом случае изменится только отображаемое значение — фактическое число останется прежним, — и ячейка сохранит точность для дальнейших вычислений.
5. Скопируйте формулу из ячейки P3 и вставьте ее во все ячейки P3:Y7. Вы увидите, что все расчеты выполнились благополучно, за исключением пустых ячеек в разделе ToHit (Для попадания), для которых процент попаданий составляет 117%! Изменим формулу, чтобы она игнорировала пустые ячейки.
6. Снова выберите ячейку P3 и замените в ней формулу на `=IF(E3="", "", 1-((E3-1)/6))`[1]. Функция `IF` в Sheets имеет три аргумента, разделенных запятыми.
   - `E3=""`: аргумент 1 — условие: равно ли содержимое ячейки E3 пустому значению `""`? (то есть E3 — пустая ячейка?)
   - `""`: аргумент 2 — что поместить в ячейку, если условие в первом аргументе истинно. То есть если E3 — пустая ячейка, очистить ячейку P3.
   - `1-((E3-1)/6)`: аргумент 3 — что поместить в ячейку, если условие в первом аргументе ложно. То есть если E3 — не пустая ячейка, использовать данную формулу.
7. Скопируйте новую формулу из ячейки P3 и вставьте ее в ячейки P3:Y7. Теперь пустым ячейкам в разделе ToHit (Для попадания) будут соответствовать пустые ячейки в разделе Percent Chance (Процент вероятности). (Например, L5:N5 — пустые ячейки, поэтому соответствующие им ячейки W5:Y5 также будут пустыми.)
8. Далее, добавим немного цвета в эту диаграмму. Выберите ячейки P3:Y7. В меню Sheets выберите пункт Format > Conditional formatting (Формат > Условное форматирование...). Справа в окне появится новая панель Conditional format rules

[1] Это новая таблица, поэтому отмечу еще раз, что если в настройках таблицы File > Spreadsheet Settings... > Locale (Файл > Настройки таблицы... > Региональные настройки) выбраны региональные настройки для России, тогда роль разделителя будет играть точка с запятой (;) и применение запятой (,) в формулах будет вызывать ошибку. Соответственно, с региональными настройками для России формула должна иметь вид `=IF(E3=""; ""; 1-((E3-1)/6))`. — *Примеч. пер.*

(Правила условного форматирования). Условное форматирование позволяет определить формат отображения ячеек в зависимости от их содержимого.

9. Щелкните на вкладке Color scale (Градиент) в верхней части панели Conditional format rules (Правила условного форматирования).
10. В разделе Preview (Предварительный просмотр) на вкладке Color scale (Градиент) вы увидите слово Default (По умолчанию) на изображении градиента зеленого цвета. Щелкните на слове Default (По умолчанию) и выберите вариант внизу в центре: Green to yellow to red (От зеленого к красному через желтый).
11. Щелкните на кнопке Done (Готово), и раздел Percent Chance (Процент вероятности) приобретет вид, как на рис. 11.15.

	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD	AE	AF	AG	AH	AI	AJ	AK
1		Percent Chance											Average Damage										
2		1	2	3	4	5	6	7	8	9	10		1	2	3	4	5	6	7	8	9	10	
3		83%	83%	83%	67%	67%	50%	50%	33%	33%	17%		6.67	6.67	6.67	5.33	5.33	4.00	4.00	2.67	2.67	1.33	
4		50%	67%	83%	83%	83%	67%	67%	67%	50%	50%		4.50	6.00	7.50	7.50	7.50	6.00	6.00	6.00	4.50	4.50	
5		83%	83%	67%	67%	50%	33%	17%					8.33	8.33	6.67	6.67	5.00	3.33	1.67				
6		17%	33%	50%	50%	67%	67%	83%	83%	67%	50%		1.33	2.67	4.00	4.00	5.33	5.33	6.67	6.67	5.33	4.00	
7		67%	67%	50%	50%	33%	33%	17%	17%				4.00	4.00	3.00	3.00	2.00	2.00	1.00	1.00			

Рис. 11.15. Разделы Percent Chance (Процент вероятности) и Average Damage (Средний урон) в таблице балансировки оружия. Разделом Average Damage (Средний урон) мы займемся далее. (Обратите внимание, что я прокрутил таблицу вправо, чтобы также был виден раздел Average Damage (Средний урон))

Вычисление среднего урона

Следующий шаг в процессе балансировки — определение среднего урона, наносимого каждым видом оружия на определенном расстоянии. Поскольку некоторые виды оружия делают сразу несколько выстрелов и каждый выстрел наносит определенный урон, средний урон будет равен произведению числа выстрелов на урон одного выстрела и на вероятность попадания каждой пули:

1. Выделите столбцы O:Z и скопируйте их (нажмите комбинацию Command-C или Ctrl+C на PC или выберите пункт меню Edit > Copy (Правка > Копировать)).
2. Выберите ячейку Z1 и выполните вставку (Command-V, Ctrl+V на PC или выберите пункт меню Edit > Paste (Правка > Вставить)). В результате в таблицу добавятся дополнительные столбцы AA:AK и заполнятся данными из только что скопированных столбцов.
3. Введите текст `Average Damage` в ячейку AA1.
4. Выберите ячейку AA3 и введите в нее формулу `=IF(P3="", "", $B3*$C3*P3)`. Так же как в формуле для ячейки P3, функция `IF` гарантирует, что вычисления будут выполняться только для непустых ячеек. Формула включает абсолютные ссылки на столбцы `$B` и `$C`, потому что столбец B хранит число выстрелов, а столбец C — урон, наносимый одним выстрелом независимо от

расстояния до врага. В данном случае смещаться при копировании должны только номера строк, но не столбцов (поэтому только столбцы описываются абсолютными ссылками).

5. Выберите ячейку AA3 и щелкните на самой правой кнопке в области с инструментами форматирования чисел (с надписью 123▾). Выберите пункт Number (Число) в открывшемся меню.
6. Скопируйте ячейку AA3 и вставьте ее в ячейки AA3:AJ7. Теперь в них будут отображаться точные значения, но условное форматирование все еще привязано к разделу Percent Chance (Процент вероятности), из-за чего числа выше 1 вынуждают проценты в диапазоне от 0 до 100 % в разделе Percent Chance (Процент вероятности) отображаться на зеленом фоне.
7. Выберите ячейки AA3:AJ7. Если панель Conditional format rules (Правила условного форматирования) больше не отображается, выберите в главном меню пункт Format > Conditional formatting (Формат > Условное форматирование...), чтобы вновь открыть ее.
8. С выбранными ячейками AA3:AJ7 вы должны увидеть в панели Conditional format rules (Правила условного форматирования) правило с заглавием Color scale (Градиент) и со ссылками на ячейки P3:Y7,AA3:AJ7 под ним. Щелкните на этом правиле.
9. В открывшихся настройках, в разделе Apply to range (Применить к диапазону) оставьте только P3:Y7 и щелкните на кнопке Done (Готово). В результате будет восстановлено прежнее оформление раздела Percent Chance (Процент вероятности).
10. Выберите ячейки AA3:AJ7 еще раз. В панели Conditional format rules (Правила условного форматирования) щелкните на кнопке Add new rule (Добавить правило).
11. Выберите в поле Preview (Предварительный просмотр) на вкладке Color scale (Градиент) вариант Green to yellow to red (От зеленого к красному через желтый), как прежде. Затем щелкните на кнопке Done (Готово).

В результате к разделам Percent Chance (Процент вероятности) и Average Damage (Средний урон) будут применены разные правила форматирования. Отделение правил для разных областей гарантирует их раздельное применение, что важно, потому что диапазоны чисел в разных разделах слишком отличаются друг от друга. Теперь раздел Average Damage (Средний урон) должен выглядеть так, как показано на рис. 11.15.

Вывод графиков среднего урона

Следующий важный шаг — вывод графиков среднего урона. Даже при том, что внимательное изучение цифр уже позволяет делать некоторые выводы, графики, поддерживаемые в Sheets, упростят эту работу и помогут визуально оценить происходящее. Для этого выполните следующие шаги.

1. Выберите ячейки A2:A7.
2. Прокрутите лист так, чтобы можно было видеть раздел Average Damage (Средний урон). Со все еще выделенными ячейками A2:A7 нажмите и удерживайте нажатой клавишу Command (или Ctrl на PC), нажмите левую кнопку мыши на ячейке AA2 и, удерживая нажатой клавишу и кнопку мыши, переместите указатель на ячейку AJ7, чтобы выделить область AA2:AJ7. Теперь у вас должны быть выделены две области: A2:A7 и AA2:AJ7.
3. Щелкните на кнопке диаграммы (см. рис. 11.2), чтобы открыть редактор диаграмм.
4. Щелкните на раскрывающемся списке Chart type (Тип диаграммы, в котором сейчас отображается текст «Column chart» («Столбчатая диаграмма»), см. рис. 11.6A) и выберите самый левый тип в разделе Line (Графики) — это левая верхняя пиктограмма с изображениями синей и красной ломаных линий.
5. В нижней части на вкладке DATA (ДАННЫЕ) установите флажок Switch rows / columns (Строки/столбцы).
6. Установите флажок Use column A as headers (Заголовки — значения столбца A). Убедитесь также, что установлен флажок Use row 2 as labels (Ярлыки — значения строки 2).
7. Щелкните на кнопке закрытия панели Chart editor (Редактор диаграмм), чтобы завершить создание графиков.

В результате у вас получится диаграмма как на рис. 11.16. Как видите, с оружием не все гладко. Некоторые виды, такие как снайперская винтовка (Sniper Rifle) и дробовик (Shotgun), имеют ярко выраженные отличительные черты, как мы и надеялись (дробовик обладает большой убойной силой на близких дистанциях, а снайперская

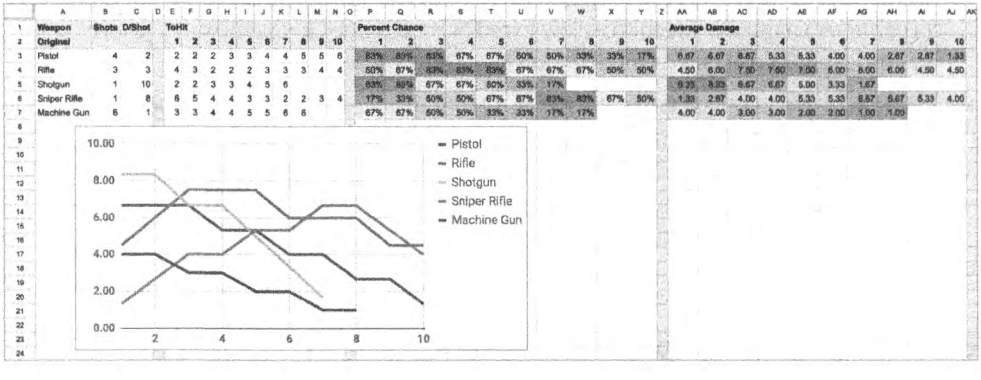


Рис. 11.16. Как показывают графики, оружие с исходными параметрами сбалансировано недостаточно хорошо. (Я внес некоторые настройки на вкладке CUSTOMIZE (ДОПОЛНИТЕЛЬНЫЕ) редактора диаграмм, чтобы сделать текст более разборчивым на рисунке)

винтовка лучше подходит для поражения целей на дальних дистанциях), но среди других явно наблюдаются проблемы:

- Пулемет (Machine gun) оказался до смешного слаб.
- Пистолет (Pistol) получился, пожалуй, избыточно мощным.
- Винтовка (Rifle) также получилась слишком мощной на фоне других видов оружия.

Проще говоря, разные виды оружия плохо сбалансированы между собой.

Дублирование параметров оружия

В процессе балансировки оружия удобно иметь перед глазами информацию до и после балансировки:

1. Для начала переместите диаграмму вниз, ниже строки 16.
2. Дважды щелкните в пределах диаграммы, после чего откроется панель Chart editor (Редактор диаграмм). Выберите вкладку CUSTOMIZE (ДОПОЛНИТЕЛЬНЫЕ) в верхней части панели (рис. 11.6D). Распахните раздел Chart axis & titles (Названия диаграмм и осей, рис. 11.6E) и введите в поле Title (Название) текст `Original` (Исходный баланс, рис. 11.6F).
3. Теперь нужно скопировать уже добавленные данные и формулы. Выберите ячейки A1:AK8 и скопируйте их.
4. Щелкните на ячейке A9 и выполните вставку. В результате должны появиться новые ячейки A9:AK16 — полная копия всех прежде созданных данных.
5. Замените текст в ячейке A10 на `Rebalanced` (После балансировки).

   В этом наборе мы будем вносить изменения и пробовать новые значения.
6. Чтобы на основе новых данных создать диаграмму, идентичную той, что отображает исходные данные, выберите диапазоны ячеек A10:A15,AA10:AJ15, в точности как вы выбирали ячейки A2:A7,AA2:AJ7 в шагах 1 и 2 в разделе «Вывод графиков среднего урона». Следуйте дальнейшим инструкциям в том разделе и создайте вторую диаграмму, отображающую балансируемые значения.
7. Расположите новую диаграмму правее предыдущей, чтобы можно было видеть обе диаграммы и данные над ними.
8. Замените название новой диаграммы на `Rebalanced` (После балансировки).

Отображение общего урона

Добавим еще одну статистику, которую тоже полезно видеть, — общий урон. Это сумма среднего урона, наносимого оружием во всех диапазонах, которая позволит вам получить представление об общей мощности оружия. Для этого можно воспользоваться трюком, который я часто использую для создания простых гистограмм

внутри ячеек в электронной таблице (то есть не в виде отдельных диаграмм). Результат показан на рис. 11.17.

1. Щелкните правой кнопкой мыши на заголовке столбца AK и в открывшемся меню выберите пункт Insert 1 right (Вставить справа: 1).
2. Щелкните правой кнопкой мыши на заголовке столбца B и выберите пункт Copy (Копировать). В результате в буфер обмена будет скопирован весь столбец B.
3. Щелкните правой кнопкой мыши на заголовке столбца AL и выберите пункт Paste (Вставить). В результате в столбец AL будет вставлено содержимое столбца B, включая оформление фона и шрифта.
4. Щелкните правой кнопкой мыши на заголовке столбца AL и выберите пункт Insert 1 right (Вставить справа: 1).
5. Введите в ячейки AL1 и AL9 текст `Overall Damage` (Общий урон).
6. Выберите ячейку AL3 и введите формулу `=SUM(AA3:AJ3)`. Она вычисляет средний урон, наносимый пистолетом во всех диапазонах (он должен быть равен `45.33`).
7. Чтобы выполнить трюк с гистограммой, нужно преобразовать это вещественное число в целое, то есть результат функции SUM требуется округлить. Замените формулу в AL3 на: `=ROUND( SUM(AA3:AJ3) )`. Теперь в ячейке должно отобразиться число `45.00`. Чтобы удалить лишние нули, выберите ячейку AL3 и щелкните на кнопке удаления десятичных знаков, как вы уже делали это прежде (третья кнопка в инструментах *форматирования чисел*).
8. Выберите ячейку AM3 и введите формулу `=REPT("|", AL3)`. Функция REPT повторяет текст в первом аргументе заданное число раз во втором аргументе. Текстом в данном случае служит символ вертикальной черты (чтобы ввести его, нужно нажать клавишу Shift и, удерживая ее, нажать клавишу с обратным слешем (\), которая на большинстве клавиатур находится над клавишей Return/Enter), и он будет повторен 45 раз, потому что ячейка AL3 содержит значение 45. В ячейке AM3 появится небольшая колонка, простирающаяся вправо. Щелкните дважды на правой границе в заголовке столбца AM, чтобы раскрыть его до требуемой ширины.
9. Выберите ячейки AL3:AM3 и скопируйте их. Вставьте содержимое буфера обмена в ячейки AL3:AM7 и AL11:AM15. Теперь у вас перед глазами будет текстовая гистограмма, отображающая общий урон всех видов оружия, исходный и после балансировки. Наконец, снова отрегулируйте ширину столбца AM, чтобы видеть все символы вертикальной черты.

Балансировка оружия

Теперь у нас есть два набора данных и две диаграммы, и можно пробовать балансировать оружие. Как увеличить мощность пулемета? Что следует увеличить — количество выстрелов, вероятность попадания в цель или убойную силу одного выстрела? При балансировке следует учитывать некоторые дополнительные правила игры.

- В этом примере игры персонажи имеют только 6 единиц здоровья, поэтому они теряют сознание при нанесении им урона, равного 6 единицам.
- В *Valkyria Chronicles*, если враг не был убит атакующим солдатом, он автоматически переходит в контратаку. Поэтому нанесение урона в 6 единиц предпочтительнее, чем нанесение урона в 5 единиц, так как это также защищает атакующего от контратаки.
- Оружие с большим количеством выстрелов (например, пулемет) дает более высокую вероятность нанести средний урон за один ход, тогда как оружие с единственным выстрелом (например, дробовик и снайперская винтовка) кажется менее надежным. На рис. 11.7 показано, как распределение вероятности все больше смещается к среднему с увеличением числа кубиков, бросаемых, чтобы определить вероятность поражения цели несколькими выстрелами.
- Даже с учетом всей имеющейся информации эта диаграмма не отражает некоторых аспектов баланса оружия, в том числе замечание о более высокой вероятности нанести средний урон, сделанное выше, в отношении оружия с несколькими выстрелами, а также преимущество снайперской винтовки для поражения врагов, находящихся слишком далеко для эффективной контратаки.

Попробуйте сами сбалансировать эти параметры оружия, изменяя только значения в ячейках B11:N15. Не трогайте исходные параметры и ничего не меняйте в разделах Percent Chance (Процент вероятности) и Average Damage (Средний урон); они оба, так же как диаграмма Rebalanced (После балансировки), будут обновляться автоматически и отражать изменения, произведенные вами в ячейках Shots (Выстрелов), D/Shot (Урон/Выстрел) и ToHit (Для попадания). Поиграв с числами, продолжайте чтение.

Один пример сбалансированных значений

На рис. 11.17 можно видеть параметры оружия, которые я подобрал для спроектированного прототипа[1]. Конечно, это не единственный способ балансировки оружия, и даже не самый лучший, но он позволяет достичь многих целей дизайна.

- Каждое оружие имеет свои отличительные характеристики, и никакое из них не является избыточно или недостаточно мощным.
- Даже при том, что на этой диаграмме дробовик (Shotgun) может показаться очень похожим на пулемет (Machine gun), эти два вида оружия существенно отличаются двумя факторами: 1) попадание с 6-очковым уроном из дробовика немедленно выводит противника из строя; и 2) пулемет выпускает множество пуль, поэтому он будет наносить средний урон гораздо чаще.
- Пистолет имеет приличные характеристики для близкого расстояния и более универсальный, чем дробовик или пулемет благодаря его способности поражать врагов на более длинных дистанциях.

[1] Я использовал игру, описанную здесь, и эти сбалансированные значения как пример бумажного прототипа для главы 9 «Прототипирование на бумаге» в первом издании книги.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	**Weapon**	**Shots**	**D/Shot**		**ToHit**										
10	**Rebalanced**				**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
11	Pistol	3	2		2	2	2	3	3	4	4	5	6	6	
12	Rifle	2	4		5	4	4	4	3	3	3	4	5	5	
13	Shotgun	1	6		2	2	2	2	3	4	5	6			
14	Sniper Rifle	1	6		6	6	5	5	4	3	3	2	2	3	
15	Machine Gun	10	1		4	4	4	4	5	5	6	6			

	AL	AM
9	**Overall Damage**	
10		
11	33	
12	40	
13	30	
14	31	
15	30	

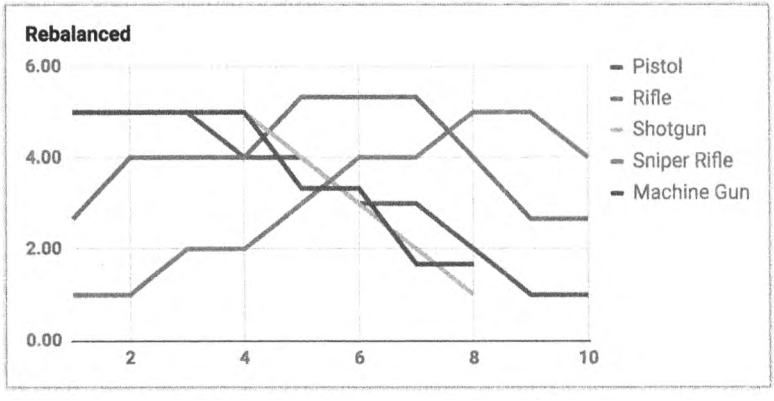


Рис. 11.17. Параметры оружия, выбранные для главы 9

- Винтовка (Rifle) по-настоящему блистает на средних дистанциях.
- Снайперская винтовка (Sniper rifle) никуда не годится на близком расстоянии, но является лучшим оружием на дальней дистанции. Попадание из снайперской винтовки наносит урон в 6 очков подобно дробовику, то есть она так же способна уничтожить врага одним выстрелом.

Несмотря на то что такая балансировка с применением электронной таблицы не охватывает всех возможных последствий дизайна оружия, она все еще остается важным инструментом в арсенале дизайнера, потому что помогает быстро проанализировать большие объемы данных. Некоторые дизайнеры бесплатных игр большую часть своего времени проводят за корректировкой данных в электронных таблицах, стараясь сделать свои игры чуть более сбалансированными, поэтому если вы собираетесь работать в сфере дизайна игр, владение электронными таблицами и проектированием на основе данных (что мы только что сделали) вам очень пригодится.

Положительная и отрицательная обратная связь

Одним из последних ключевых элементов поддержки баланса игры, который мы обсудим, является понятие положительной и отрицательной обратной связи. В игре с положительной обратной связью игрок, захвативший лидерство с самого

начала, получает дополнительное преимущество и, скорее всего, победит. В игре с отрицательной обратной связью преимущество получают проигрывающие игроки.

Покер — отличный пример игры с положительной обратной связью. Если игрок выигрывает большой банк и денег у него больше, чем у других игроков, отдельные ставки значат для него меньше и у него больше свободы делать такие шаги, как блеф (потому что он может позволить себе потерять что-то). Напротив, у игрока, потерявшего деньги в начале игры, меньше свободы действий и нет возможности рисковать. Сильная положительная обратная связь есть в игре *Monopoly*, где игрок, обладающий лучшими объектами, постоянно получает больше денег и способен заставить других игроков продавать свои объекты, если они оказываются не в состоянии платить арендную плату, когда попадают на свои участки. Для большинства игр положительная обратная связь — нежелательное явление, но она хорошо подходит для случаев, когда требуется, чтобы игра завершалась быстро (хотя в *Monopoly* это преимущество не используется, потому что дизайнер этой игры хотел показать несчастья бедных в капиталистическом обществе). В играх с одним игроком также часто есть механизмы положительной обратной связи, цель которых — дать игроку почувствовать себя всесильным в игре.

Mario Kart — прекрасный пример игры с отрицательной обратной связью в форме случайных предметов, присуждаемых игроку, когда тот наезжает на коробки. Обычно игрок получает банан (по сути, оружие защиты), связку из трех бананов или зеленую ракушку (один из самых слабых видов оружия). Игрок на последнем месте часто получает более мощные предметы, такие как молния, которая замедляет всех других игроков, участвующих в гонке. Отрицательная обратная связь делает игры более справедливыми по отношению к игрокам, не сумевшим пробиться в лидеры, и более длинными, а также дает всем игрокам возможность почувствовать, что у них все еще есть шанс выиграть, даже если они сильно отстают.

Итоги

В этой главе было много математики, но, я надеюсь, вы убедились, что знание ее очень пригодится вам как дизайнеру игр. Большинство тем, затронутых в этой главе, заслуживают отдельной книги, поэтому я призываю продолжить их изучение, если они вам интересны.

12 Руководство игроком

Как отмечалось в предыдущих главах, ваша главная задача как дизайнера — создание приятных впечатлений у игрока. И чем дальше вы будете углубляться в свой проект и свой дизайн, тем очевиднее и понятнее будет казаться вам игра. Это обусловлено вашим близким знакомством с игрой и совершенно естественно.

Однако это также означает, что вы должны с пристрастием следить за своей игрой и стараться сделать ее такой же понятной для игроков, которые прежде не видели ее, чтобы они могли получить весь комплекс впечатлений, который вы задумывали. Это требует ненавязчивого и порой незримого руководства игроком.

В этой главе рассматриваются два стиля руководства игроком: *прямой*, когда игрок замечает, что им руководят; и *косвенный*, когда руководство осуществляется настолько тонко, что игрок часто не замечает этого. В конце главы приводится информация об *обучающих последовательностях* — способе обучения игроков новым понятиям или ознакомления их с новыми механиками.

Прямое руководство

Методы прямого руководства непосредственно заметны игроку. Прямое руководство может принимать самые разные формы, но качественно все они определяются релевантностью, ограниченностью, лаконичностью и ясностью:

- **Релевантность:** сообщение должно посылаться игроку, когда оно непосредственно относится к происходящему. Некоторые игры пытаются описать все элементы управления игрой с самого начала (иногда даже отображается схема пульта с подписанными кнопками), но глупо думать, что игрок запомнит их и вовремя применит. Прямая информация об элементах управления должна предоставляться непосредственно в первый раз, когда они нужны игроку. В игре *Kya: Dark Lineage* для PlayStation 2 перед персонажем падает дерево, которое тот должен перепрыгнуть, и когда это происходит, в нужный момент

на экране появляется надпись «Press X to jump» (Нажмите X, чтобы перепрыгнуть).

- **Ограниченность:** многие современные игры предусматривают множество элементов управления и множество целей. Не перегружайте игрока, подавая слишком большой объем информации за раз. Чем реже будут появляться инструкции и другие элементы прямого управления, тем ценнее они будут для игрока и тем выше вероятность, что они не останутся незамеченными. Это также относится к заданиям. Игрок может сосредоточиться только на каком-то одном задании, а некоторые, в общем, неплохие игры с открытым миром, такие как *Skyrim*, нагружают игрока разными заданиями до такой степени, что спустя несколько часов игры игрок может оказаться занятым выполнением десятков разных заданий, многие из которых он просто проигнорирует.
- **Лаконичность:** никогда не используйте больше слов, чем нужно, и не давайте игроку слишком много информации за раз. В тактической боевой игре *Valkyria Chronicles*, чтобы научить игрока прятаться за мешками с песком нажатием клавиши O, выводится простой текст: «When near sandbags, press O to take cover and reduce damage from enemy attacks» («Находясь рядом с мешками с песком, нажмите O, чтобы укрыться за ними и уменьшить урон от вражеских атак»).
- **Ясность:** старайтесь быть предельно ясными в том, что собираетесь донести. В предыдущем примере можно поддаться соблазну и просто сообщить игроку: «When standing near sandbags, press O to take cover» («Находясь рядом с мешками с песком, нажмите O, чтобы укрыться»), потому что можно предположить, что игрок понимает — укрывшись, он защитится от вражеских пуль. Однако в игре *Valkyria Chronicles*, укрывшись за мешками, можно не только защититься, но и значительно уменьшить урон от попадания пуль (в сравнении с тем, когда между врагом и жертвой нет защиты). Чтобы игрок понимал все, что нужно, о защите, ему также нужно сообщить о снижении урона.

Четыре метода прямого руководства

Есть несколько методов прямого руководства, которые можно обнаружить в играх.

Инструкции

Игра явно сообщает игроку, что нужно сделать. Инструкции могут принимать форму текстовых сообщений, диалога с авторитетным компьютерным персонажем (то есть персонажем, которым управляет компьютер) или визуальных диаграмм и часто являются комбинацией всех трех форм. Инструкции — одна из самых ясных форм прямого руководства, но они также страдают наибольшей вероятностью нагрузить игрока слишком большим объемом информации или вызвать раздражение из-за педантичного повторения информации, которую игрок уже знает.

Призыв к действию

Игра явно призывает игрока выполнить действие и объясняет причину для этого. Этот метод часто принимает форму заданий (миссий), которые даются игроку компьютерным персонажем. Призывая к действию, желательно поставить перед игроком ясную долгосрочную цель и по мере продвижения вперед ставить средне- и краткосрочные цели, которых нужно достигнуть на пути к долгосрочной.

Игра *Legend of Zelda: Ocarina of Time* начинается с пробуждения Линка волшебницей Нави, которая сообщает ему, что его вызывает Великое Дерево Деку. Это сообщение затем повторяется первым персонажем, с которым встречается Линк, покинув свой дом. Этот персонаж сообщает Линку, что приглашение — это большая честь и тот должен поспешить. В результате Линк получает долгосрочную цель — найти Великое Дерево Деку (и беседа Великого Дерева Деку с Нави перед пробуждением Линка подсказывает игроку, что по прибытии перед ним будет поставлена еще более долгосрочная цель). На пути к Великому Дереву Деку Линку встречается Мидо, который сообщает ему, что он должен найти щит и меч, прежде чем вступить в лес. Теперь перед игроком стоят две среднесрочных цели, которых он должен достичь, прежде чем отправиться к долгосрочной цели. Чтобы получить щит и меч, Линк должен пересечь небольшой лабиринт, поговорить с несколькими персонажами и заработать хотя бы 40 рупий. Все эти краткосрочные цели ясно и тесно связаны с долгосрочной целью прибытия на встречу с Великим Деревом Деку.

Карта или система навигации

Многие игры включают карту или систему навигации в стиле GPS, которая направляет игрока к цели или к следующему этапу в задании. Например, в *Grand Theft Auto V* имеется маленькая карта/радар в углу экрана, где обозначен маршрут игрока к следующей цели. Мир *GTA V* настолько огромен, что задания часто приводят игрока в незнакомые области, где игрок вынужден полагаться на GPS. Но имейте в виду, что такого рода руководство может заставить игрока большую часть времени следовать указаниям виртуального GPS, вместо того чтобы оценивать свое местоположение и самостоятельно выбирать маршрут, что может увеличить время, необходимое для знакомства игрока с игровым миром.

Всплывающие подсказки

В некоторых играх присутствуют контекстные элементы управления, меняющие назначение в зависимости от присутствия тех или иных объектов рядом с игроком. В игре *Assassin's Creed IV: Black Flag* одна и та же кнопка управляет такими разными действиями, как открытие двери, поджигание бочек с порохом и стрельба из выбранного оружия. Чтобы помочь игроку понять все имеющиеся возможности, всякий раз, когда это необходимо, над кнопкой появляются пиктограмма и очень короткое текстовое описание действия, выполняемого кнопкой.

Косвенное руководство

Косвенное руководство — это искусство так влиять на игрока и руководить его действиями, что тот даже не будет подозревать, что им управляют. Вам, как дизайнеру, могут пригодиться несколько методов косвенного руководства. Качество косвенного руководства можно оценить с помощью тех же критериев, что и прямого (релевантность, ограниченность, лаконичность и ясность), плюс незаметность и надежность:

- **Незаметность:** насколько игрок осознает, что им управляют? Будет ли это осознание отрицательно отражаться на восприятии игры? Ответив на второй вопрос, вы сможете определить, насколько незаметным должно быть руководство. Иногда требуется, чтобы руководство оставалось полностью незамеченным; иногда игроку пойдет на пользу, если он заметит руководство им. В любом случае, качество косвенного руководства зависит от того, как влияет осознание игроком, что им руководят, на его восприятие игры.
- **Надежность:** как часто косвенное руководство должно направлять игрока в соответствии с вашими желаниями? Косвенное руководство — тонкая материя и может быть несколько ненадежным. Даже при том, что, оказавшись в темном помещении, большинство игроков будет стремиться к освещенному дверному проему, найдутся такие, кто предпочтет остаться в темноте. Используя приемы косвенного руководства в своих играх, тщательно проверяйте их, чтобы убедиться в их влиянии на достаточно большой процент игроков. Если этого не происходит, значит, ваше косвенное влияние оказалось недостаточно стимулирующим.

Семь методов косвенного руководства

Впервые с идеей косвенного руководства меня познакомил Джесси Шелл, который изложил ее под названием «Косвенное управление» в своей книге «The Art of Game Design: A Book of Lenses» в главе 16. Это целый список из шести методов косвенного управления[1].

Ограничения

Если дать игроку ограниченный набор вариантов для выбора, он выберет один из них. Кажется, что здесь все просто, но подумайте о разнице между предложением заполнить пробел и предложением выбрать из четырех вариантов. Без наложения ограничений есть риск поставить игрока в ситуацию паралича выбора, когда человеку предоставляется настолько широкий выбор вариантов, что он оказывается не в состоянии принять взвешенное решение и просто отказывается от выбора.

[1] Jesse Schell, *The Art of Game Design: A Book of Lenses* (Boca Raton, FL: CRC Press, 2008), 283–298.

Именно по этой причине в меню ресторана может иметься 100 разных блюд, но только для 20 приводятся изображения. Владельцы ресторанов хотят облегчить вам задачу выбора блюд на ужин.

Цели

Как указывает Шелл, если перед игроком стоит цель собрать бананы и есть две двери, через которые он сможет пройти, — поместив бананы за одной из дверей так, чтобы они были отчетливо видны, вы побудите игрока направиться к этой двери.

Игроки часто желают определять себе цели сами, но вы можете подталкивать их к выбору нужных вам целей, давая игрокам материалы, необходимые для их достижения. В игре *Minecraft* (название которой составлено из двух слов прямых инструкций: «mine» (добывай) и «craft» (конструируй)) дизайнеры определили, какие конструкции могут создавать игроки и из каких материалов, и эти дизайнерские решения, в свою очередь, подразумевают цели, которые игроки могут ставить перед собой. Поскольку большинство простейших рецептов позволяют игроку создавать строительные материалы, простые инструменты и оружие, они фактически направляют игрока на путь строительства своего дома и защитных сооружений. Эта цель заставляет игрока заняться исследованием материалов. Например, знание, что алмазы позволяют создавать лучшие инструменты, заставляет игрока исследовать все более и более глубокие туннели в поисках алмазов (которые встречаются редко и только на глубине около 50–55 метров) и расширять границы известного мира.

Физический интерфейс

В книге Шелла рассказывается, как дизайнеры могут использовать форму физического интерфейса для косвенного руководства игроком: если игроку в *Guitar Hero* или *Rock Band* дать пульт в форме гитары, он попытается с его помощью воспроизводить музыку. Если дать игроку в *Guitar Hero* обычный игровой пульт, он подумает, что с его помощью сможет управлять движением своего персонажа (потому что стандартный игровой пульт обычно используется именно для управления движениями персонажей), но держа в руках пульт в форме гитары, он будет думать только о воспроизведении музыки.

Тактильные ощущения также можно использовать для косвенного руководства. Одним из примеров может служить функция вибрации, присутствующая во многих игровых пультах, которая заставляет пульт вибрировать в руках игрока с разной интенсивностью. Настоящие трассы для автомобильных гонок по внутреннему радиусу поворотов оборудованы поребриками, которые имеют ребристую форму и раскрашены в красные и белые полосы. Ребристая форма позволяют водителю почувствовать тряску на рулевом колесе, если он входит в поворот по слишком малому радиусу и оказывается на поребрике. Это полезно, потому что гонщики часто стараются пройти поворот по наименьшему радиусу — по оптимальной траектории — и могут точно почувствовать момент, когда колеса оказываются на

поребрике, что очень трудно увидеть из кокпита гоночного автомобиля. Тот же метод — вибрация пульта, когда автомобиль игрока оказывается на границе внутренней окружности поворота, — широко используется во многих гоночных играх. Этот прием можно расширить и заставить пульт беспорядочно вибрировать, если игрок съезжает с ровной трассы и мчится по траве. Тактильные ощущения помогут игроку понять, что он должен вернуться на трассу.

Визуальный дизайн

Для косвенного руководства игроком можно также использовать визуальные эффекты и некоторые другие способы:

- **Свет:** свет привлекает внимание людей. Если поместить игрока в темную комнату с кругом света в одном из концов, он пойдет сначала в направлении освещенной области, прежде чем заняться исследованием других участков.
- **Сходство:** после того как игрок увидел что-то хорошее в игровом мире (полезное, исцеляющее, ценное и т. д.), он будет обращать внимание на похожие предметы.
- **Тропинки:** сходство может создавать эффект «следа из хлебных крошек», когда игрок подбирает какой-то предмет и затем следует за похожими предметами к точке, куда дизайнер хочет привести его.
- **Ориентиры:** визуально привлекательные и необычные объекты можно использовать в роли ориентиров. В начале игры *Journey*, созданной студией thatgamecompany, игрок оказывается среди пустыни рядом с песчаной дюной. Окружающий ландшафт видится ему как череда песчаных дюн, кроме нескольких черных каменных столбов на вершине самой высокой соседней дюны (рис. 12.1 слева). Поскольку эти столбы — единственное, что выделяется в окружающем ландшафте, игрок направляется к ним вверх по дюне. Когда он достигает вершины, камера поднимается над ним, открывая вид на возвышающуюся гору с сияющей вершиной (рис. 12.1 справа). Движение камеры заставляет гору появиться прямо за каменными столбами, подсказывая игроку, что гора — следующая его цель. Движение камеры прямо переносит цель со столбов на гору.

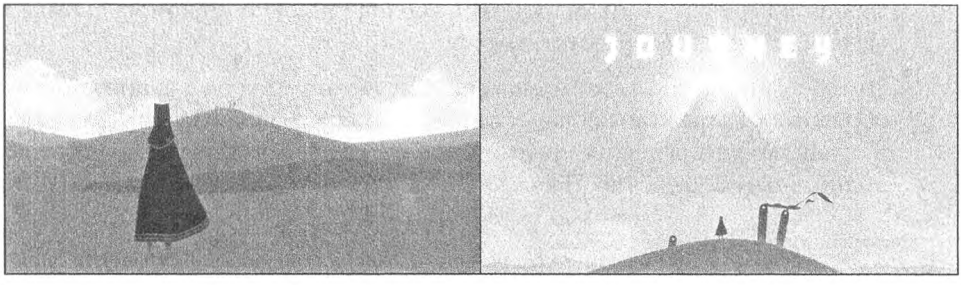


Рис. 12.1. Ориентиры в игре Journey

Проектируя Диснейленд, в компании Walt Disney Imagineering (которая в то время называлась WED Enterprises) предусмотрели разные ориентиры, направляющие гостей вокруг парка и предотвращающие их скапливание в центре. Гости, впервые приходившие в парк, оказывались на Главной улице США, которая выглядела как идеализированный американский городок начала XX века. Однако пройдя немного вдоль по Главной улице, они замечали в ее конце Замок спящей красавицы и немедленно направлялись туда. Дойдя до замка, гости замечали, что он намного меньше, чем казался сначала, и что там действительно нечего делать. Теперь, оказавшись в центре Диснейленда, они могли увидеть гору Маттерхорн, возвышающуюся перед ними, статую космического века перед входом в Страну будущего (Tomorrowland) справа и форт Приграничной страны (Frontierland) — слева. Из центра эти новые ориентиры кажутся намного интереснее, чем маленький замок, и гости вскоре покидали территорию замка, направляясь к новым достопримечательностям[1].

Ориентиры также присутствуют в серии игр *Assassin's Creed*. Когда игрок впервые попадает в область на карте, он должен заметить несколько строений, возвышающихся над всем остальным в этой области. Помимо естественной заметности этих ориентиров, каждый из них также является контрольной точкой в игре, с которой игрок может *синхронизироваться*, чтобы обновить игровую карту с подробной информацией о районе. Поскольку дизайнеры дали игроку и ориентир, и цель (заполнить свою карту), можно смело предположить, что основной своей задачей игроки будут считать поиск контрольной точки в новой части мира.

- **Указатели:** изображения на рис. 12.2 демонстрируют примеры малозаметных указателей, используемых для руководства игроком в игре *Uncharted 3: Drake's Deception*, созданной в студии Naughty Dog. На этих изображениях игрок (точнее, его персонаж Дрейк) преследует противника по имени Талбот.

  A. Когда игрок поднимается на крышу здания, многочисленные линии, образуемые физическими границами, и контрастное освещение направляют внимание игрока влево. К линиям относятся: выступ, край перед игроком, доски слева и даже край серого пластикового стула.

  B. Как только игрок оказывается на крыше, камера поворачивается, и теперь уступ, край стены и доски указывают, куда должен двигаться игрок (крыша здания в верхней части кадра). Шлакоблок рядом со стеной на снимке B даже образует своеобразную стрелку.

     Это особенно важно в этот момент погони, потому что область приземления обрушится, когда игрок попадет на нее, что может заставить его усомниться в правильности решения прыгнуть на эту крышу. Указатели придают дополнительную уверенность.

[1] На это мне первым указал Скотт Роджерс, который более подробно описал этот прием на уровне 9 (то есть в главе 9) своей книги *Level Up!: The Guide to Great Video Game Design* (Chichester, UK: Wiley, 2010).

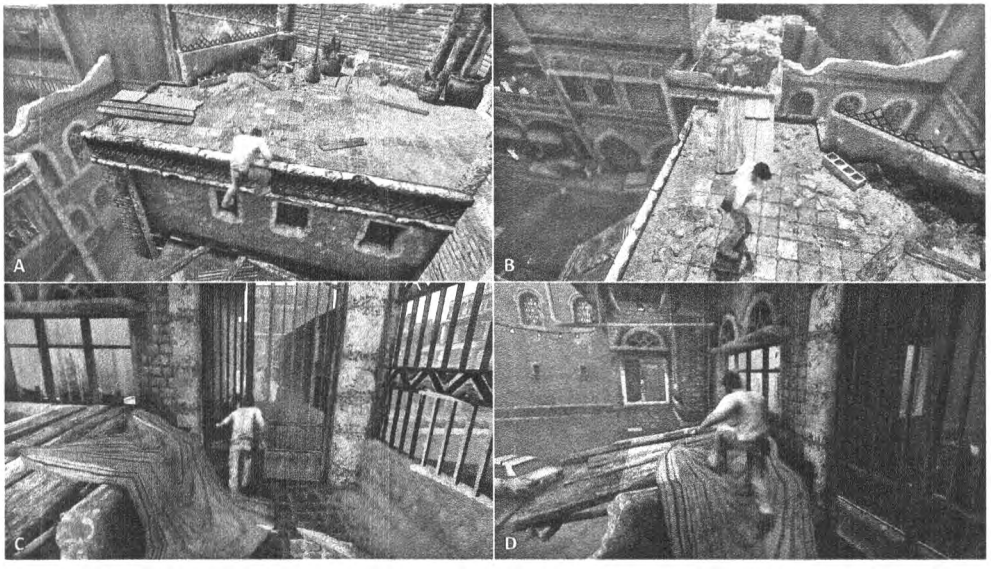


Рис. 12.2. Указатели, создаваемые в игре *Uncharted 3* линиями и контрастными перепадами, направляют игрока к цели

Команда разработчиков *Uncharted 3* называет деревянные доски, как на этом изображении, *трамплинами* и использует их повсюду в игре, чтобы побудить игрока прыгнуть в определенном направлении. Еще один трамплин можно видеть на изображении A на рис. 12.3.

C. В этой части той же погони Талбот пробежал через ворота, створка которых захлопнулась перед игроком. Синяя ткань слева направляет взгляд игрока влево, и левый угол ткани также образует указывающую стрелку.

D. Теперь камера направлена влево, и с этой точки зрения синяя ткань видится как стрелка, указывающая на окно с желтой рамой (следующую цель игрока). Яркие синий и желтый цвета, как на изображении, используются на протяжении всей игры, чтобы подсказать игроку правильное направление движения, поэтому их присутствие укрепляет игрока в решении направиться в сторону желтого окна.

- **Камера:** во многих играх, где возникает трудный выбор направления дальнейшего движения, для руководства игроком используется камера. Показывая игроку следующую цель или направление движения, камера руководит им там, где могут возникнуть затруднения с выбором. Этот прием демонстрируют снимки из игры *Uncharted 3*, изображенные на рис. 12.3.

  На снимке A камера находится непосредственно позади игрока; но когда игрок прыгает и цепляется за перекладину, камера смещается и показывает панораму слева, направляя игрока влево (снимок B). Камера продолжает

смотреть влево (снимок C), пока игрок не достигнет крайней левой лестницы, после чего камера выравнивается и поворачивается вниз, показывая желтые ступени (снимок D).

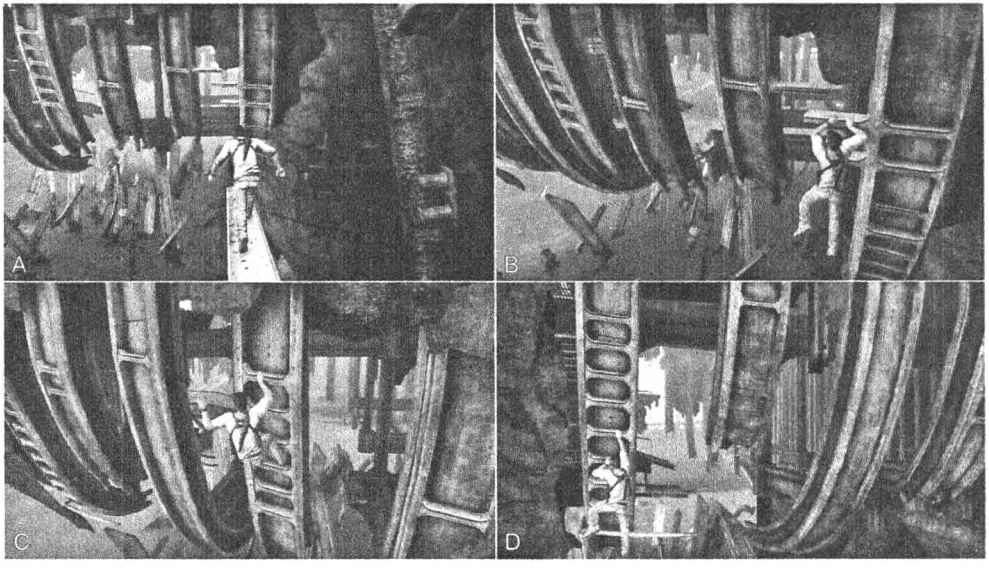


Рис. 12.3. Руководство игроком с помощью камеры в игре *Uncharted 3*

- **Контраст:** снимки на рис. 12.2 и 12.3 также демонстрируют использование контраста для привлечения внимания игрока. На рис. 12.2 и 12.3 показано несколько видов контраста, вносящих свой вклад в руководство игроком:
  - **Яркость:** на снимках A и B рис. 12.2 выступ и стена, которые образуют стрелку, имеют самую высокую яркость. Темные области рядом со светлыми выделяют линии на общем фоне.
  - **Текстура:** на снимках A и B рис. 12.2 деревянные доски гладкие, а у окружающей каменной кладки шероховатый вид. На снимках C и D рис. 12.2 ткань, струящаяся мягкими складками, контрастирует с окружающей твердой каменной кладкой. Размещение ткани на каменной кромке также смягчает ее и подсказывает игроку, что тот может через нее перепрыгнуть.
  - **Цвет:** на снимках C и D рис. 12.2 синяя ткань, желтая рама и желтые полосы контрастируют с приглушенностью других цветов в сцене. На снимке D рис. 12.3 желтая ступенька выделяется на сине-сером фоне окружающей сцены.
  - **Направленность:** хотя этот прием используется реже, его также можно применять для направления взгляда игрока. На снимке A рис. 12.3 горизонтальные ступени выделяются на общем фоне вертикальных линий.

Звуковое оформление

Как отмечает Шелл, музыку можно использовать для влияния на настроение игрока, а значит, на его поведение[1]. Разная музыка ассоциируется с разными действиями. Медленная, тихая, немного джазовая музыка часто ассоциируется с такими действиями, как подкрадывание или поиск подсказок (как можно видеть и слышать в мультсериале *Scooby Doo*), тогда как громкая, быстрая, энергичная музыка (как в приключенческих фильмах) лучше подходит для сцен, где игрок должен отважно сражаться с врагами, чувствовать себя непобедимым.

Звуковые эффекты также можно использовать для влияния на поведение игрока, побуждая его к возможным действиям. В серии игр *Assassin's Creed* всякий раз, когда игрок оказывается рядом с сундуком, наполненным сокровищами, воспроизводится переливающийся звонок. Он информирует игрока, что тот может заняться поиском сундука, а поскольку звонок раздается, только когда сундук находится рядом, игроку сообщается, что сундук нужно искать поблизости. При гарантированной награде в непосредственной близости игрок обычно переключается на поиск сундука, если в этот момент не занят чем-то более важным.

Персонаж

Модель игрока (то есть управляемый им персонаж) может сильно влиять на поведение игрока. Если персонаж игрока выглядит как рок-звезда и держит гитару в руках, игрок почти наверняка будет ожидать, что его персонаж умеет играть. Если в руках персонаж держит меч, игрок будет полагать, что сможет наносить им удары и сражаться с врагами. Если персонаж носит колпак волшебника, одет в длинный халат и держит в руках книгу, игрок будет воспринимать это как предложение отказаться от прямой схватки и сосредоточиться на заклинаниях.

Неигровые персонажи

Неигровые персонажи — одна из самых сложных и гибких разновидностей косвенного руководства игроком, и это руководство может принимать множество форм.

Моделирование поведения

Неигровые персонажи могут проявлять несколько разных типов поведения. В играх под моделированием поведения понимается акт демонстрации определенного поведения, позволяющий игроку увидеть последствия такого поведения. На рис. 12.4 изображены разные примеры моделирования поведения в игре *Kya: Dark Lineage* компании Atari, в том числе:

[1] Schell, *Art of Game Design*, 292–293.

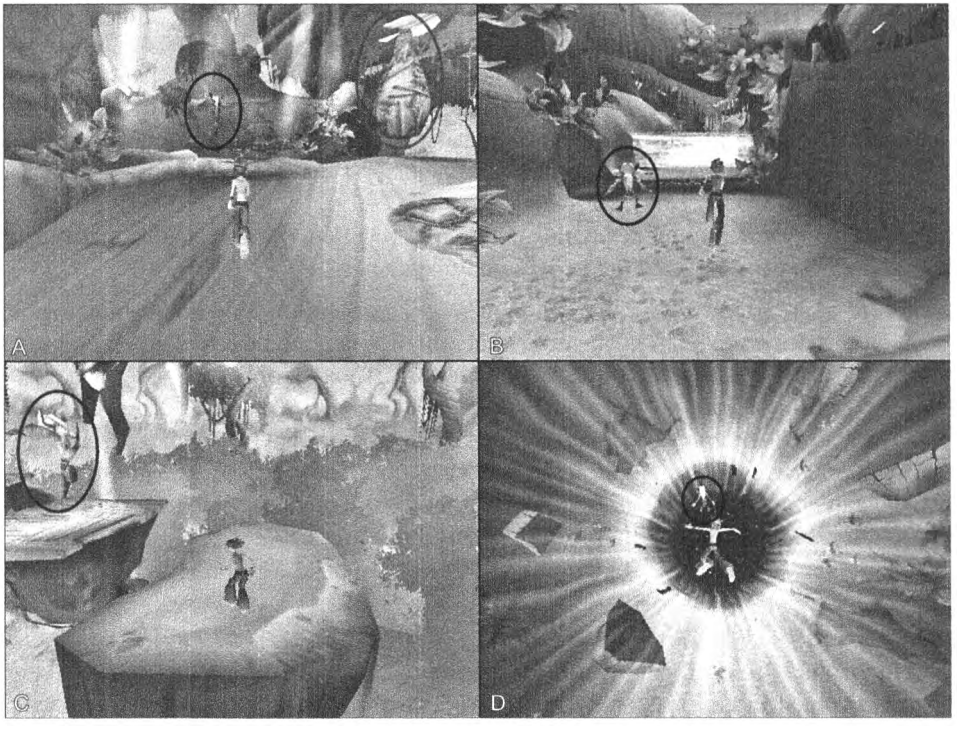


Рис. 12.4. Нативы, управляемые компьютером, демонстрируют модели поведения в игре *Kya: Dark Lineage*

- **Неправильное поведение:** в модели неправильного поведения неигровой персонаж делает нечто, чего игрок должен избегать, и демонстрируются последствия. На снимке A рис. 12.4 в красном овале один из Нативов (аборигенов) наступил на одну из круговых ловушек на земле и был пойман (ловушка затем подняла Натива в воздух и перенесла к врагам, преследующим Нативов и игрока).
- **Правильное поведение:** другой Натив на снимке A (в овале) перепрыгнул через ловушку, показывая, как их преодолевать. Это модель правильного поведения, демонстрирующая, как должен действовать игрок в игровом мире. На снимке B показан другой пример — Натив остановился непосредственно перед местом, где тропу периодически пересекает мощный воздушный поток, дующий слева направо, хотя в этот момент никакого потока еще нет. Натив ждет, пока появится поток, дожидается его прекращения и потом продолжает бег вперед. Таким способом игроку демонстрируется, что тот должен остановиться перед местом, где появляется воздушный поток, дождаться нужного момента и только потом продолжить путь.
- **Безопасность:** на снимках C и D Натив прыгает на что-то, что на первый взгляд кажется опасным. Как бы то ни было, наблюдая решимость Натива прыгнуть, игрок понимает, что это безопасно, и следует за ним.

Использование эмоциональных привязанностей

Другой способ влияния неигровых персонажей на поведение игрока — использование эмоциональных привязанностей, которые возникают между игроком и персонажами.

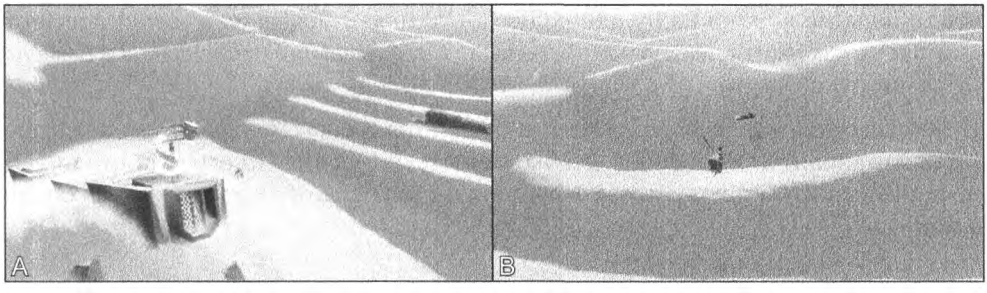


Рис. 12.5. Эмоциональная привязанность в игре *Journey*

На снимках из игры *Journey*, изображенных на рис. 12.5, игрок следует за неигровым персонажем, потому что между ними установилась эмоциональная связь. В начале игры *Journey* игрок чувствует себя одиноким, и неигровой персонаж — это первое эмоциональное существо, которое игрок встречает на своем пути через пустыню.

В момент встречи существо радостно облетает игрока (снимок A) и затем направляется вперед (снимок B). В этой ситуации игрок почти всегда направляется вслед за ним.

Также можно заставить игрока двигаться за неигровым персонажем, создав отрицательную эмоциональную связь. Например, неигровой персонаж может стащить что-нибудь у игрока и пуститься в бегство, заставляя того броситься в погоню, чтобы вернуть украденное имущество. В любом случае игрок стремится последовать за неигровым персонажем, и это отличный способ направить игрока в другое место.

Обучение новым навыкам и понятиям

Несмотря на то что косвенное руководство обычно применяется, чтобы побудить игрока перемещаться через виртуальный мир игры, в этом заключительном разделе мы рассмотрим методы, помогающие игроку лучше понять, как играть в игру.

Когда игры были проще, дизайнеры могли показать игроку простую схему элементов управления или даже просто дать им возможность экспериментировать. В игре *Super Mario Bros.* для Nintendo Entertainment System (NES) одна кнопка заставляла Марио подпрыгнуть, а другая заставляла его бежать (и стрелять огненными шара-

ми после того, как он подберет огненный цветок). Немного поэкспериментировав, игрок легко осваивал работу с кнопками A и B на пульте NES. Современные пульты обычно имеют два аналоговых джойстика (которые также могут действовать как кнопки), блок из восьми кнопок направления, шесть лицевых кнопок, две плечевые кнопки и два триггера. Даже с таким набором элементов управления многие современные игры позволяют игроку выполнять настолько широкий диапазон действий, что отдельные кнопки на пульте получают разное назначение в зависимости от контекста, как уже отмечалось в подразделе «Всплывающие подсказки» в разделе «Прямое руководство».

Вследствие такой большой сложности в некоторых современных играх обучение игрока становится жизненной необходимостью. Буклет с инструкциями ничем не поможет в этом; теперь дизайнеры должны руководить игроком, обучая его *по шагам*.

Обучающая последовательность

Обучающая последовательность — это искусство ненавязчивого представления игроку новой информации, и в большинстве случаев используется стиль, изображенный на рис. 12.6. На этом рисунке показано несколько шагов обучения из игры *Kya: Dark Lineage*, знакомящих игрока с механикой парения в воздухе, которая много раз используется в игре:

- **Изолированное знакомство:** вниманию игрока представляется новая механика, которую он должен освоить, чтобы продолжить игру. На снимке A рис. 12.6 присутствует непрерывный поток воздуха, направленный вверх, и игрок должен нажать и удерживать клавишу X, чтобы пролететь на достаточном расстоянии под стеной, нависающей сверху. Пока игрок удерживает кнопку X и пытается пролететь под стеной, ничего не происходит, поэтому он не ощущает давления времени, приобретая новый навык.
- **Расширенное знакомство:** на снимке B рис. 12.6 изображен следующий шаг в этой последовательности. Здесь перед игроком возникают уже две стены, сверху и снизу, и теперь он должен научиться пролетать посередине туннеля, «качая» (то есть нажимая и отпуская) кнопку X. На этом этапе игрок также не штрафуется за неспособность выполнить маневр правильно.
- **Добавление опасности:** на снимке C рис. 12.6 добавлены некоторые опасности. Слишком приблизившись к красному полу, игрок может причинить себе вред; однако потолок все еще полностью безопасен, поэтому, не нажимая кнопку X, игрок остается в безопасности. Далее, на снимке D, пол становится безопасным, а потолок — опасным, поэтому, если игрок все еще оттачивает свои навыки, он может просто удерживать кнопку X нажатой и скользить вперед вдоль пола.
- **Увеличение сложности:** на снимках E и F рис. 12.6 показаны последние этапы обучающей последовательности. На снимке E потолок пока остается безопасным, но игрок должен двигаться по узкому каналу. На снимке F изображена сцена,

где игрок также должен двигаться по узкому каналу, но теперь опасными являются и пол, и потолок. Игрок должен продемонстрировать мастерство владения механикой нажимания кнопки X, чтобы безопасно преодолеть туннель[1].

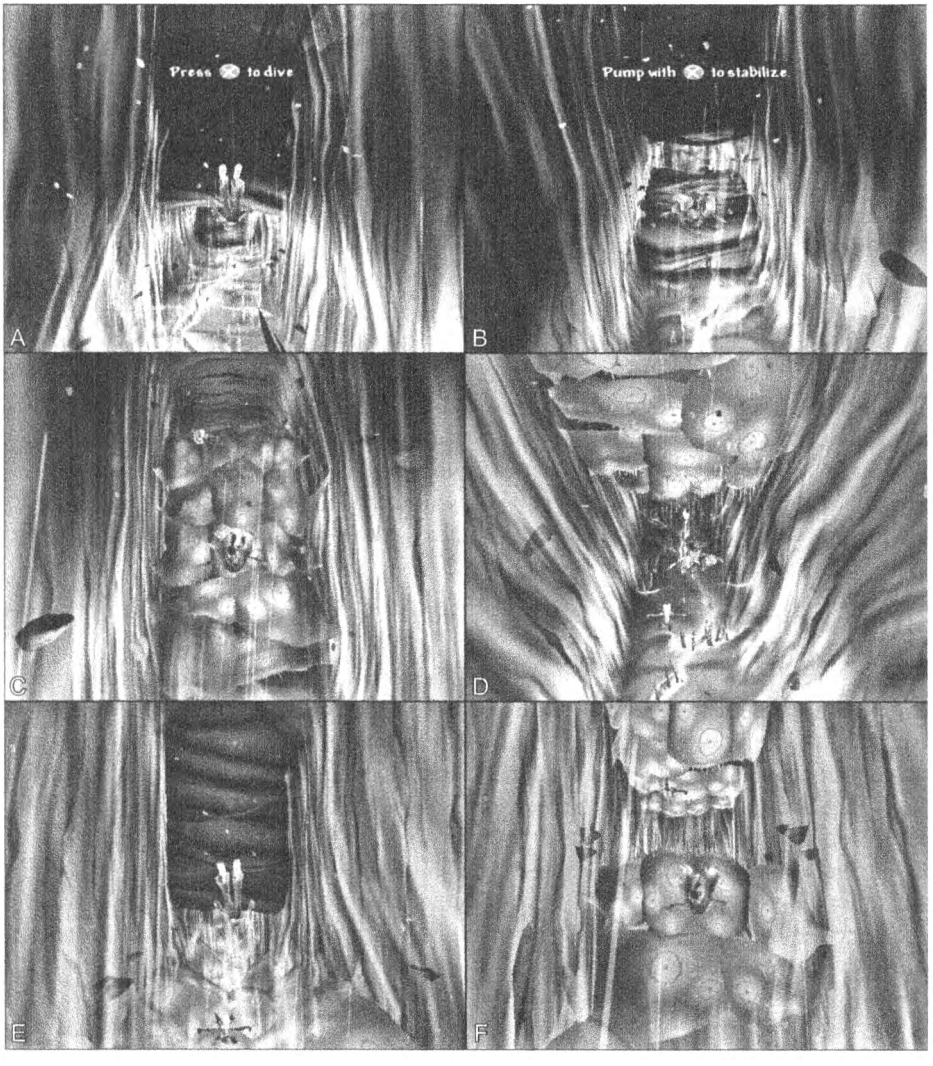


Рис. 12.6. Пошаговое обучение парению в воздухе в игре *Kya: Dark Lineage*

[1] На рис. 12.6 также показано, как использовать цветовой контраст для передачи информации о безопасности. Цвет туннеля переходит из зеленого в красный, чтобы показать увеличение опасности, а фиолетовый цвет в конце туннеля на снимке F сообщает игроку, что испытание скоро закончится.

В этой главе я использовал несколько скриншотов, сделанных в игре *Kya: Dark Lineage*, потому что в ней обучающая последовательность реализована лучше, чем во всех других играх, которые я видел. В первые шесть минут игрок обучается перемещаться, прыгать, обходить ловушки, беречься шипов, катать и пинать ногами шарообразных животных для обезвреживания ловушек, избегать горизонтальных потоков воздуха, перепрыгивать с площадки на площадку, парить в воздухе (рис. 12.6), прятаться и еще десяткам других механик. Для обучения всему этому используются обучающие последовательности, и к концу вводной игры я запомнил их все.

Многие игры используют обучающие последовательности. В серии игр *God of War* каждый раз, когда Кратос получает новое оружие или заклинание, с помощью текстовых сообщений игроку объясняется, как им пользоваться, а затем сразу же демонстрируется обучающая последовательность. Для заклинаний, таких как удар молнией, которые игрок может использовать для питания электрических устройств или против врагов, ему предлагается сначала использовать их для небоевых целей (например, игрок может применить заклинание, вызывающее молнию, в комнате с закрытыми дверьми и должен воспользоваться им, чтобы привести в действие электрическое устройство, отпирающее двери). Затем игроку дается возможность поучаствовать в бою и легко победить, воспользовавшись новым заклинанием. Этот прием не только дает игроку опыт применения заклинания в бою, но и демонстрирует, насколько мощным оно является и насколько могущественным делает игрока.

Интеграция

Когда игрок поймет, как пользоваться новой игровой механикой в изоляции (как было описано выше), наступает момент научить его пользоваться ею в сочетании с другими механиками. Это можно сделать явно (например, сообщить игроку, что молния в воде имеет более широкий диапазон действия — поражает врагов не только в радиусе 6 футов, как на воздухе, но вообще всех находящихся в воде) или неявно (например, заставить игрока принять бой в воде, чтобы он сам заметил, что молния убивает всех врагов, находящихся в воде, а не только врагов в радиусе 6 футов). Когда позднее игрок получит заклинание, позволяющее обливать врагов водой и создавать лужи, он сразу же поймет, что эту механику можно использовать для расширения зоны действия молнии.

Итоги

Методов руководства игроками намного больше, чем уместилось в этой главе, но я надеюсь, что она послужила вам неплохим введением в некоторые конкретные методы, а также объяснила причины их использования. Проектируя свои игры, постоянно помните о необходимости руководства игроком. Это может быть самой сложной задачей для вас, потому что вам (как дизайнеру) все игровые механики

будут казаться очевидными. Встать на чужую точку зрения настолько сложно, что большинство компаний, занимающихся разработкой игр, нанимают десятки и сотни одноразовых тестировщиков в течение периода разработки. Очень важно находить новых людей, которые могли бы попробовать вашу игру и поделиться мнением о качестве обучающих последовательностей с точки зрения того, кто никогда прежде не видел ее. Игры, разработанные в изоляции, без привлечения наивных тестировщиков, часто или оказываются слишком сложными для новых игроков, или страдают неравномерным, ступенчатым увеличением сложности, вызывая разочарование. Как отмечалось в главе 10 «Тестирование игры», тестирование должно начинаться как можно раньше, производиться как можно чаще и, если это возможно, все новыми людьми.

13 Проектирование головоломок

Головоломки — важная часть многих цифровых игр, а также интереснейшая задача проектирования сама по себе. Эта глава начинается изучением проектирования головоломок глазами одного из величайших дизайнеров головоломок — Скотта Кима.

В последней части главы рассматриваются разные типы головоломок, широко используемые в современных играх, часть из которых может не соответствовать вашим ожиданиям.

Как вы узнаете в этой главе, в большинстве однопользовательских игр имеется какая-нибудь головоломка, тогда как для многопользовательских игр это нехарактерно. Главная причина в том, что в однопользовательских играх и головоломках главным соперником игрока является сама игровая система, тогда как в многопользовательских цифровых играх (где не предусмотрены совместные действия) соперниками друг другу являются сами игроки. Из-за этого сходства однопользовательских игр и головоломок изучение особенностей проектирования головоломок может помочь в освоении проектирования любых игр, имеющих однопользовательский или кооперативный режим.

Скотт Ким о проектировании головоломок

Скотт Ким — один из ведущих современных дизайнеров головоломок. С 1990-го он придумывал головоломки для таких журналов, как Discover, Scientific American и Games, и разрабатывал режимы головоломок для некоторых игр, включая Bejeweled 2. Он читал лекции о проектировании головоломок на конференциях TED и Game Developers Conference. Его влиятельный однодневный семинар «The Art of Puzzle Design»[1], который он проводил вместе с Алексеем Пажитновым (создателем игры Тетрис) в 1999 и 2000 годах на конференциях Game Developers Conference

[1] Scott Kim and Alexey Pajitnov, «The Art of Puzzle Game Design» (представлен на Game Developers Conference, Сан-Хосе, Калифорния, 15 марта 1999): https://web.archive.org/web/20030219140548/http://scottkim.com/thinkinggames/GDC99/gdc1999.ppt. Статья была доступна в апреле 2018 года.

помог многим дизайнерам игр сформировать представления о головоломках более чем на десятилетие. В этой главе мы познакомимся с содержимым этого семинара.

Определение понятия «головоломка»

Как утверждает Ким, его определение понятия «головоломка» самое простое:

> «Головоломка — это интерес и наличие правильного ответа»[1].

Это отличает головоломки от игрушек — интересных, но не имеющих правильного ответа — и от игр — интересных, но имеющих цель, а не конкретный правильный ответ. Ким отделяет головоломки от игр, хотя я лично считаю их более высокоразвитым подмножеством игр. Несмотря на простоту этого определения, в нем скрываются некоторые важные детали.

Головоломка — это интерес...

Как отмечает Ким, головоломки обладают тремя элементами, обеспечивающими интерес:

- **Новизна:** для решения многих головоломок необходимо обладать некоторой специфической информацией, и когда игрок получает эту информацию, он легко находит решение. Наибольшее удовольствие от решения головоломки доставляет озарение, радость создания нового решения. Если головоломка не отличается новизной, игрок часто обладает информацией, необходимой для решения, еще до того, как приступит к ее решению, и из-за этого теряется интерес.
- **Адекватная сложность:** так же как игры должны давать игроку адекватные задачи, головоломки должны соответствовать навыкам, опыту и творческим способностям игрока. Каждый игрок, приступающий к решению головоломки, имеет определенный опыт решения головоломок этого типа и готов испытать определенную степень разочарования, прежде чем сдаться. Некоторые из лучших головоломок имеют два решения: адекватное, средней сложности, и экспертное, требующее передовых навыков для его обнаружения. Еще одна замечательная стратегия заключается в создании головоломок, которые кажутся простыми, хотя на самом деле являются очень сложными. Если игрок воспринимает головоломку как простую, он будет менее склонен сдаться.
- **Хитрость:** многие замечательные головоломки заставляют игрока менять свой взгляд или образ мышления для их решения. Но даже после этого игрок все еще должен чувствовать, что для решения головоломки необходимы умения и остроумие. Этот элемент наглядно демонстрирует игра *Mark of the Ninja* студии Klei Entertainment, где игрок должен решать головоломные задачи для

[1] Scott Kim, «What Is a Puzzle?» https://web.archive.org/web/20070820000322/http://www.scottkim.com/thinkinggames/whatisapuzzle/. Статья была доступна в апреле 2018 года.

проникновения в комнаты, наполненные врагами, и, составив план решения, в точности приводить его в исполнение[1].

...и наличие правильного ответа

У каждой головоломки должен быть правильный ответ, но у многих таких верных ответов несколько. Один из главных элементов хорошей головоломки — ясное ощущение правоты после того, как игрок находит правильный ответ. Если решение не создает безоговорочного ощущения правильности, головоломка может показаться слишком запутанной и непривлекательной.

Жанры головоломок

Ким определяет четыре жанра головоломок (рис. 13.1)[2], каждый из которых заставляет игрока использовать разные подходы и навыки. Эти жанры находятся на пересечениях головоломок и других видов деятельности. Например, сюжетная головоломка являет собой смесь повествования и серии головоломок.

- **Активные действия:** головоломки, требующие активных действий, такие как *Tetris*, ограничивают время для принятия решения и дают игрокам возможность исправлять ошибки. Они представляют собой комбинацию головоломки и игры с активными действиями.
- **Рассказ:** сюжетные головоломки, такие как *Myst*, серия *Professor Layton* и большинство игр, суть которых заключается в поиске скрытых предметов, включают головоломки, которые игроки должны решать, чтобы двигаться вдоль сюжетной линии и исследовать окружающее пространство. Они объединяют повествование и головоломки.
- **Конструирование:** игры-конструкторы предлагают игроку создать объект из деталей, чтобы решить некоторую задачу. Одной из наиболее успешных таких игр была *The Incredible Machine*, в которой игрок должен создать машину Руба Голдберга, чтобы прогнать кошек в каждой сцене. Некоторые игры-конструкторы даже включают наборы деталей, позволяющих игрокам создавать и распространять свои головоломки. Они лежат на пересечении головоломок и конструирования, проектирования и пространственного мышления.

---

[1] Nels Anderson, «Of Choice and Breaking New Ground: Designing *Mark of the Ninja*» (выступление на конференции Game Developers Conference, Сан Франциско, Калифорния, 29 марта 2013 года). Нельс Андерсон, ведущий геймдизайнер *Mark of the Ninja*, говорил в этом выступлении о сужении разрыва между намерением и исполнением. Команда обнаружила, что с увеличением простоты воплощения планов игровые навыки игрока все больше смещаются из области физического исполнения в область умственного планирования, что делает игру больше похожей на головоломку и более интересной для игроков. Он опубликовал ссылку на слайды и сценарий доклада в своем блоге: http://www.above49.ca/2013/04/gdc-13-slides-text.html, статья была доступна в апреле 2018 года. Его выступление также можно найти на сайте GDC Vault: http://gdcvault.com.

[2] Scott Kim and Alexey Pajitnov, «The Art of Puzzle Game Design», слайд 7.

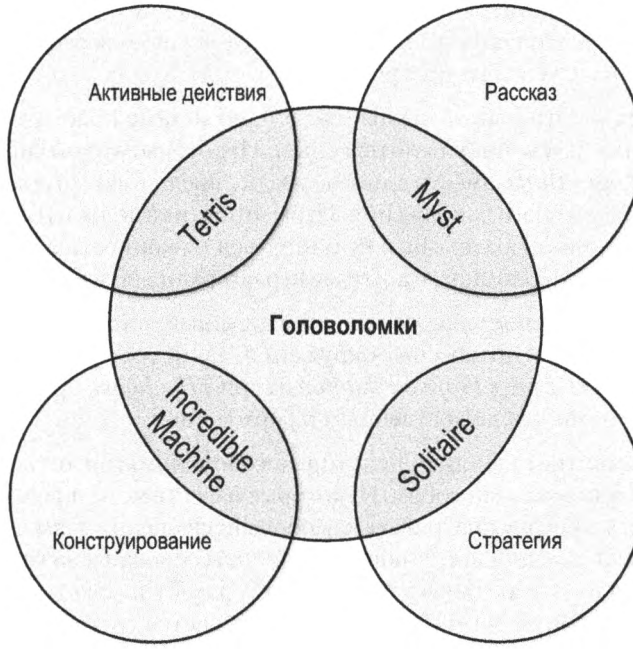


Рис. 13.1. Четыре жанра головоломок, выделяемых Кимом[1]

- **Стратегия:** многие стратегические головоломки являются аналогами пасьянса и часто встречаются в играх, обычно многопользовательских. К ним относятся, например, головоломные ситуации при игре в бридж (когда игроку дается набор карт в руках и предлагается решить, как продолжить игру) и шахматные задачи (когда игроку дается определенная комбинация фигур на доске и предлагается объявить мат сопернику за определенное число ходов). Они сочетают мышление, необходимое в многопользовательских играх, с умением конструировать головоломки, чтобы помочь игрокам тренировать в себе навыки для многопользовательской игры.

Ким также считает, что некоторые чистые головоломки не вписываются ни в один из четырех жанров. К ним относятся такие игры, как *Sudoku* или кроссворды.

Четыре основные причины, почему люди играют в головоломки

Опираясь на свои исследования и опыт, Ким пришел к выводу, что люди играют в головоломки в основном по следующим причинам[2]:

[1] Scott Kim and Alexey Pajitnov, «The Art of Puzzle Game Design», слайд 7.
[2] Ibid, слайд 8.

- **Вызов:** людям свойственно откликаться на брошенные вызовы и испытывать радость от их преодоления. Головоломки дают игрокам возможность испытать чувство гордости, успеха и прогресса.
- **Развлечение:** кто-то ищет сложные задачи, но многие просто хотят скоротать время, занимаясь чем-нибудь интересным. Некоторые головоломки, такие как *Bejeweled* и *Angry Birds*, вместо сложных задач предлагают игрокам интересное развлечение без особого напряжения. Игры этого типа должны быть относительно простыми и однообразными, а не опираться на конкретные знания (как это часто бывает в головоломках, в которые играют ради преодоления сложностей).
- **Персонаж и окружение:** людям нравятся длинные истории и персонажи, красивые изображения и интересные окружения. Такие головоломки, как *Myst*, *The Journeyman Project*, серия *Professor Layton* и серия *The Room*, привлекают игроков своими историями и художественным оформлением.
- **Духовное развитие:** наконец, некоторые головоломки имитируют духовное развитие тем или иным способом. Некоторые известные головоломки, такие как *кубик Рубика*, можно рассматривать как обряд посвящения — вы или соберете его хотя бы раз в жизни, или нет. Многие лабиринты основаны на том же принципе. Кроме того, головоломки могут имитировать развитие архетипического героя: игрок начинает в игре обычную жизнь, сталкивается с головоломкой, которая погружает его в царство борьбы, бьется над ее решением, получает озарение и затем легко побеждает головоломку, которая поставила его в тупик всего несколькими мгновениями раньше.

Образы мышления для решения головоломок

Разные типы головоломок требуют разных подходов к их решению, и большинство игроков имеют свой привычный образ мышления (и, соответственно, свой любимый класс головоломок). Эти понятия иллюстрирует рис. 13.2.

Головоломки для одного образа мышления

Ниже приводятся описания головоломок, требующих одного образа мышления (рис. 13.2):

- **Слово:** существует множество разных головоломок со словами, большинство которых требует от игрока богатого словарного запаса. Головоломки со словами часто особенно хорошо подходят для игроков в возрасте, потому что словарный запас накапливается лишь с годами.
- **Образ:** к числу головоломок с образами относятся пазлы, поиск скрытых предметов, а также 2- и 3-мерные пространственные головоломки. Головоломки этого типа, как правило, решаются с привлечением области головного мозга, связанной с обработкой визуальной/пространственной информации и распознаванием образов.

Рис. 13.2. Диаграмма Венна с тремя основными способами мышления для решения головоломок (Слово, Образ и Логика), а также примеры головоломок для каждого способа и некоторые головоломки, требующие двух способов[1]

- **Логика:** логические головоломки, такие как *Bulls & Cows* (описывалась в главе 11 «Математика и баланс игры»), загадки и дедуктивные задачи заставляют игроков прибегать к логическим рассуждениям. Многие игры основаны на дедуктивных рассуждениях: последовательном отбрасывании ложных вариантов, пока не останется один истинный (например, игрок мог бы рассуждать так: «Я знаю, что все другие подозреваемые невиновны, поэтому мистера Бодди, скорее всего, убил полковник Мастард»). К подобным играм относятся головоломки *Clue*, *Bulls & Cows* и *Logic Grid*. Гораздо реже используются *индуктивные* рассуждения: экстраполяция от частного к общему (примером может служить такое рассуждение: «Последние пять раз, когда Джон блефовал в покер, он неосознанно почесывал нос; сейчас он тоже почесывает нос, поэтому, скорее всего, он блефует»). Дедуктивная логика ведет к определенности, тогда как индуктивная логика — к предположениям, основанным на высокой вероятности. Определенность ответов традиционно делает дедуктивную логику более привлекательной для дизайнеров головоломок.

Головоломки сразу для двух образов мышления

В следующем списке перечислены смешанные типы головоломок, находящиеся на рис. 13.2 в областях пересечения. Их примеры представлены на рис. 13.3.

- **Слово/Образ:** многие игры, такие как *Scrabble*, ребусы (как на рис. 13.3) и те, что основаны на поиске слов, требуют богатого словарного запаса и хорошего

[1] Scott Kim and Alexey Pajitnov, «The Art of Puzzle Game Design», слайд 9.

образного мышления. *Scrabble* — это головоломка смешанного типа, а кроссворд — нет, потому что, играя в *Scrabble*, игрок должен подобрать слово и расположить его так, чтобы получить наибольшее количество очков. Кроссворд не требует ни одного из этих двух действий, связанных с принятием решения о пространственном расположении[1].

- **Образ/Логика:** скользящие блоки, лазерные лабиринты и другие головоломки, изображенные во второй категории на рис. 13.3, требуют от игроков использовать логику для решения образных задач.
- **Логика/Слово:** в категорию Логика/Слово попадает большинство головоломок, включая классическую игру «Riddle of the Sphinx», которая является первой загадкой на рис. 13.3. Эту загадку задал сфинкс Эдипу в классической трагедии Софокла «Царь Эдип».

Рис. 13.3. Головоломки разных смешанных типов (решения приводятся в конце главы)

[1] Занимаясь подготовкой этого второго издания книги, я параллельно работал над игрой-головоломкой *Ledbetter* смешанного типа Слово/Образ для мобильных устройств. Вы можете найти ее по адресу http://exninja.com/ledbetter.

Восемь шагов Кима в проектировании цифровых головоломок

Скотт Ким описал восемь шагов, которые он обычно выполняет, проектируя головоломки[1].

1. **Вдохновение:** источником вдохновения для создания головоломки, как и игры, может стать все что угодно. По словам Алексея Пажитнова, на создание игры *Тетрис* его вдохновила идея пентамино (12 пятиклеточных фигур, которыми можно оптимально заполнить пространство, укладывая их определенным образом) математика Соломона Голомба (Solomon Golomb) и желание использовать их в деятельной игре. Однако пятиклеточных фигур пентамино оказалось слишком много, поэтому он ограничился семью четырехклеточными фигурами тетрамино и использовал их в игре *Тетрис*.
2. **Упрощение:** чтобы из вдохновения родилась хорошая головоломка, часто требуется пройти через этап упрощения.
   a) Определение основной механики головоломки: важнейшего необходимого навыка.
   b) Устранение любых излишних деталей и сужение фокуса.
   c) Добавление единообразия. Например, создавая головоломку-конструктор, поместите элементы на однородную сетку, чтобы игроку было проще манипулировать ими.
   d) Простота управления. Управление головоломкой должно соответствовать интерфейсу. Ким рассказывал, насколько просто манипулировать *кубиком Рубика* в реальной жизни и насколько плохо — его цифровой версией с использованием мыши и клавиатуры.
3. **Конструктор:** создайте инструмент, позволяющий легко и быстро сконструировать головоломку. Многие головоломки можно сконструировать и проверить на бумаге, но если этот путь не подходит для вашей головоломки, тогда конструктор — это первое, что нужно воплотить в программный код. Эффективный конструктор, бумажный или цифровой, может существенно упростить создание новых уровней. Выясните повторяющиеся задачи, которые приходится выполнять в процессе конструирования, и посмотрите, можно ли создать многократно используемые элементы или автоматизировать процессы их создания.
4. **Правила:** определите четкие правила, в том числе устройство игрового поля, элементы, способы их перемещения и конечную цель головоломки или уровня.
5. **Головоломки:** создайте несколько уровней головоломки. Разные уровни должны предлагать разные элементы и игровую механику.

[1] Scott Kim and Alexey Pajitnov, «The Art of Puzzle Game Design», слайд 97.

6. **Тестирование:** так же как в случае с обычными играми, нельзя заранее сказать, как игроки будут реагировать на головоломку, пока она не попадет им в руки. Даже Ким, обладая многолетним опытом, часто обнаруживает, что некоторые его головоломки, простые, по его мнению, оказываются на удивление сложными, тогда как кажущиеся сложными в действительности решаются с легкостью. Тестирование — важный этап для всех форм проектирования. Обычно шаг 6 заставляет дизайнера вернуться к шагам 4 и 5 и уточнить предыдущие решения.
7. **Определение последовательности:** после уточнения правил и создания нескольких уровней самое время подумать о расположении их в определенной последовательности. Каждый раз, вводя новую идею, делайте это в изоляции, требуя от игрока использовать только ее и наиболее простым способом. Затем можно постепенно усложнять головоломку, требуя использовать эту же идею для ее решения. Наконец, можно создать головоломки, для решения которых необходимо применить эту идею вместе с другими, уже знакомыми игроку. Это очень напоминает прием обучающей последовательности, представленный в главе 12 «Руководство игроком», который рекомендуется применять для обучения игроков любым новым идеям.
8. **Представление:** после определения правил, создания уровней и расположения их в правильном порядке можно заняться внешним видом головоломки. Этот этап также предполагает доработку интерфейса и способов отображения информации для игрока.

Семь целей эффективного проектирования головоломок

Проектируя головоломки, имейте в виду цели, перечисленные ниже. Чем большего числа из них вы достигнете, тем лучше получится ваша головоломка:

- **Дружественное отношение к пользователю:** головоломки должны быть понятными и познавательными для игроков. Они могут включать некоторые хитрости, но не должны обманывать игрока или заставлять его чувствовать себя глупцом.
- **Доступность:** в течение минуты игрок должен понять, как играть с головоломкой. В течение нескольких минут его должен охватить интерес.
- **Немедленная обратная связь:** головоломка должна быть «сочной», как говорит Кайл Геблер (Kyle Gabler), один из создателей игр *World of Goo* и *Little Inferno*: она должна реагировать на ввод игрока так, чтобы у него возникало чувство физического осязания, активной работы и энергетической заряженности.
- **Безостановочное движение:** игра должна постоянно подталкивать игрока к следующему шагу, не должно быть четкой остановки. Когда я работал в Pogo.

com, все наши игры заканчивались кнопкой Play Again (Сыграть еще) вместо заставки с сообщением о завершении игры. Даже такой нехитрый прием способен заставить игрока продолжить играть.

- **Предельно четкие цели:** игрок всегда должен предельно четко понимать главную цель головоломки. Но точно так же полезно иметь дополнительные цели, которые игрок мог бы обнаруживать со временем. Игры *Hexic* и *Bookworm* — вот примеры головоломок с четкими начальными и некоторыми дополнительными целями, которые наиболее опытные игроки с радостью обнаруживают по прошествии времени.
- **Уровни сложности:** у игрока должна быть возможность решать головоломку на уровне сложности, соответствующем его умениям. Так же как в любых играх, сложность головоломки — непременное условие появления интереса у игроков.
- **Нечто особенное:** в большинстве хороших головоломок есть что-то, что делает их уникальными и интересными. Игра *Tempuc* Алексея Пажитнова сочетает очевидную простоту с возможностью стратегического мышления и неуклонно возрастающей интенсивностью. Обе игры, *World of Goo* и *Angry Birds*, имеют невероятно сочный, реактивный игровой процесс.

Примеры головоломок в активных играх

Современные AAA-игры часто включают несколько головоломок, большинство которых относится к одной из следующих категорий.

Скользящие блоки/позиционные головоломки

Скользящие блоки, или позиционные головоломки, обычно встречаются в активных играх от третьего лица и требуют от игрока перемещать большие блоки по решетчатому полу, чтобы воспроизвести определенный узор. Как вариант, может понадобиться расположить зеркала особым образом, чтобы луч света или лазера мог пройти от источника до цели. Одним из распространенных вариантов является скользкий пол, заставляющий блоки двигаться, пока те не упрутся в стену или какое-то другое препятствие.

- **Примеры игр:** *Soul Reaver*, *Uncharted*, *Prince of Persia: The Sands of Time*, *Tomb Raider*, некоторые игры в серии *The Legend of Zelda*.

Физические головоломки

Все физически головоломки включают имитацию действия законов физики для перемещения объектов вокруг сцены или поражения разных целей игроком или

другими объектами. Это основная механика серии *Portal*, и она приобретает все большую популярность благодаря развитию и распространению современных физических движков, таких как Havok и Nvidia PhysX (встроен в Unity).

- **Примеры игр:** *Portal, Half-Life 2, Super Mario Galaxy, Rochard, Angry Birds.*

Поиск пути

Головоломки этого вида показывают место на уровне, которого вы должны достичь, но как это сделать — часто неочевидно. Игрок часто должен отклоняться от маршрута, чтобы отпереть ворота или опустить мосты, находящиеся на пути к цели. Игры-гонки, такие как *Gran Turismo*, тоже являются головоломками поиска пути; игрок должен найти идеальную траекторию, чтобы проходить каждый круг максимально эффективно и быстро. Это очень важно в головоломках «Burning Lap» из серии *Burnout*, где игроки должны избегать ошибок на треке, где имеются участки со встречным и поперечным движением и крутые повороты.

- **Примеры игр:** *Uncharted, Tomb Raider, Assassin's Creed, Oddworld: Abe's Oddyssee, Gran Turismo, Burnout, Portal.*

Скрытность

Разновидность головоломок поиска пути, ставшая настолько важной, что была выделена в отдельный жанр. В головоломках, основанных на скрытности (стелс-игры, stealth puzzles), игрок должен пересечь уровень, избегая обнаружения врагами, которые обычно выполняют обход уровня по заданному маршруту или следуют определенному графику. Часто игрокам дается возможность обезвреживать врагов, хотя при плохом исполнении это может привести к обнаружению.

- **Примеры игр:** *Metal Gear Solid, Uncharted, Oddworld: Abe's Oddyssee, Mark of the Ninja, Beyond Good and Evil, The Elder Scrolls V: Skyrim, Assassin's Creed.*

Цепная реакция

Игры типа «цепная реакция» включают физические системы, в которых могут взаимодействовать разные компоненты, часто создавая взрывы и причиняя разрушения. Игроки используют свои инструменты для установки ловушек или порождения серий событий, чтобы решить головоломку или получить преимущество перед врагом. Серия игр-гонок *Burnout* включает режим аварий, в котором игрок должен вести машину так, чтобы причинить максимальный урон в денежном выражении, вызывая фантастические аварии с участием множества автомобилей.

- **Примеры игр:** *Pixel Junk Shooter, Tomb Raider (2013), Half-Life 2, The Incredible Machine, Magicka, Red Faction: Guerilla, Just Cause 3, Bioshock, Burnout.*

Схватки с боссами

Многие схватки с боссами[1], особенно в классических играх, требуют решения головоломки некоторого вида, когда игрок должен изучить характер реакций и атак босса и определить последовательность действий, учитывающих этот характер и ведущих к победе. Это особенно верно для активных игр от третьего лица компании Nintendo, таких как серии *Zelda*, *Metroid* и *Super Mario*. Одним из распространенных элементов в головоломках этого вида является *правило трех*[2]:

1. Когда игрок в первый раз совершает правильное действие, наносящее урон боссу, оно часто оказывается неожиданным для него.
2. Во второй раз игрок совершает это же действие намеренно, чтобы проверить, действительно ли найден способ победить босса.
3. В третий раз он демонстрирует свою способность решить головоломку и побеждает босса.

Большинство боссов в серии *Legend of Zelda*, начиная с *The Ocarina of Time*, можно победить за три атаки — при условии, что игрок разгадал головоломку босса.

- **Примеры игр:** *The Legend of Zelda*, *God of War*, *Metal Gear Solid*, *Metroid*, *Super Mario 64/Sunshine/Galaxy*, *Guacamelee*, *Shadow of the Colossus*, многопользовательские групповые рейды в *World of Warcraft*.

Итоги

Как было показано в этой главе, головоломки — важный аспект многих однопользовательских игр или многопользовательских игр с поддержкой совместных действий. Проектирование головоломок не требует особых навыков, сильно отличающихся от тех, что вы уже приобрели как дизайнер игр, хотя есть некоторые тонкие отличия. При проектировании игр наиболее важным аспектом является течение игрового процесса, тогда как при проектировании головоломок первостепенное значение имеют решение и момент озарения. (Впрочем, в активных головоломках, таких как *Тетрис*, озарение и решение наступают с падением и размещением каждой фигуры.) Кроме того, когда игрок находит решение головоломки, для него важна уверенность в правильности решения, тогда как в играх интерес во многом зависит от неопределенности в уме игрока относительно результата или правильности принятых решений.

Независимо от различий в проектировании для головоломок, столь же важно использовать итеративный процесс проектирования, как для всех других видов интерактивного опыта. Как дизайнер головоломок, вы точно так же должны создавать

[1] Босс (от *англ.* boss) — персонаж в играх, особенно сильный или редкий противник. — *Примеч. пер.*

[2] Насколько я помню, впервые о «правиле трех» я услышал от Джесси Шелла.

прототипы и проводить пробные игры (тестирование); однако для головоломок особенно важно, чтобы ваши тестировщики не встречали их прежде (потому что иначе они не смогут повторно испытать чувство озарения).

В заключение на рис. 13.4 показаны решения головоломок, изображенных на рис. 13.3. Первоначально я не хотел говорить этого, но для решения головоломки со спичками фактически требуется использовать все три образа мышления: логика, образ и слово.

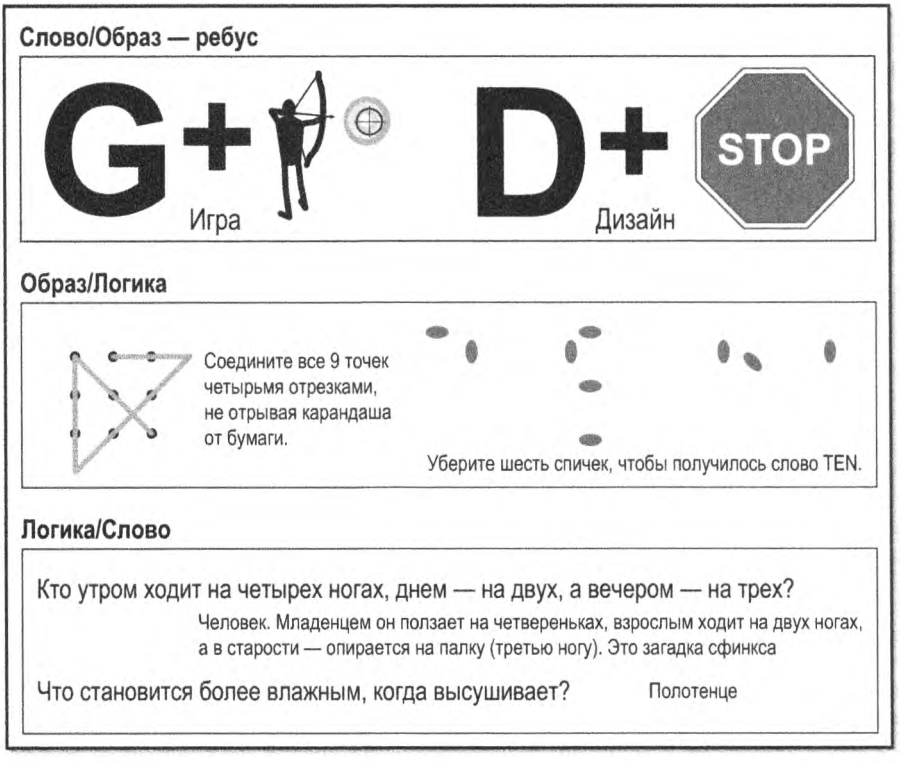


Рис. 13.4. Решения головоломок смешанных типов, показанных на рис. 13.3

14 Agile-мышление

В этой главе вы узнаете, как рассуждать о проектах с точки зрения гибкого прототипирования и как взвешивать варианты, приступая к реализации проекта. Эта глава познакомит вас с гибким мышлением и методологией Скрам. В ней также подробно описываются графики выполнения работ, которые я рекомендую использовать в проектах любых игр.

Прочитав эту главу, вы лучше будете понимать, как подходить к реализации проектов, как разбивать их на спринты, которые можно выполнить за определенное время, и как расставить приоритеты задач в этих спринтах.

Манифест гибкой разработки программного обеспечения

Много лет большинство программных продуктов, включая игры, разрабатывалось с использованием *каскадного*, или *водопадного*, метода. Согласно каскадному методу, небольшая команда проектировщиков определяла проект игры в виде массивного документа, описывающего ее дизайн, следуя букве которого команда разработчиков должна была создать игру. Слишком строгое следование каскадному методу приводило к тому, что игры не тестировались, пока не оказывались на одной из заключительных стадий, а члены таких команд разработчиков чувствовали себя скорее винтиками большой машины, чем настоящими разработчиками игр.

Благодаря опыту, полученному в главах, посвященных прототипированию, вы, я уверен, сразу заметите некоторые проблемы этого подхода. В 2001 году разработчики, сформировавшие альянс Agile Alliance, также заметили эти проблемы и выпустили «Манифест гибкой разработки программного обеспечения»[1], который гласит:

> Мы постоянно открываем для себя более совершенные методы разработки программного обеспечения, занимаясь разработкой непосредственно и помогая в этом другим. Благодаря проделанной работе мы смогли осознать, что:

[1] Kent Beck at al. «Manifesto for Agile Software Development», Agile Alliance (2001). (Текст манифеста на русском языке доступен по адресу http://agilemanifesto.org/iso/ru/manifesto.html. — *Примеч. пер.*)

- **Люди и взаимодействие** важнее процессов и инструментов
- **Работающий продукт** важнее исчерпывающей документации
- **Сотрудничество с заказчиком** важнее согласования условий контракта
- **Готовность к изменениям** важнее следования первоначальному плану

То есть, не отрицая важности того, что справа, мы все-таки больше ценим то, что слева.

В этих четырех ценностях можно увидеть многие принципы, которые я пытался донести до вас в этой книге:

- Следовать личному пониманию дизайна, постоянно задавать вопросы и развивать процедурное мышление намного важнее следования предопределенным правилам или использования конкретных инструментов разработки.
- Создание простого действующего прототипа и его итеративное совершенствование дает более высокую вероятность успеха, чем многомесячное ожидание появления идеи идеальной игры или решения ваших проблем.
- Позитивное обсуждение ваших идей с другими творческими личностями в обстановке сотрудничества намного важнее беспокойства о принадлежности интеллектуальной собственности.[1]
- Прислушиваться к отзывам тестировщиков и реагировать на них намного важнее, чем следовать первоначальному проекту. Вы должны позволить своей игре развиваться.

До того как я ввел методологию гибкой разработки в свои курсы, студенты часто катастрофически отставали от графика разработки своих игр. Фактически они даже не осознавали отставания, потому что им недоставало инструментов управления процессом разработки. Кроме того, тестирование игр студентами было затруднительным почти до самого окончания разработки проекта.

После внедрения методов гибкой разработки и соответствующих инструментов я заметил, что:

- Студенты начали лучше понимать, как движется разработка их проектов, и точнее придерживаться графика.
- Игры стали получаться намного лучше — во многом благодаря тому, что студенты постоянно уделяли внимание наличию действующей сборки, позволяющей начать тестирование как можно раньше и производить его чаще.
- Улучшилось понимание C# и Unity студентами, а также возросла их уверенность в своих технических навыках.

---

[1] Конечно, вы должны уважать права других людей на интеллектуальную собственность. Я лишь хочу сказать, что сотрудничество важнее споров о том, кому какой процент принадлежит.

Первые два пункта были вполне ожидаемы. Но третий в первый момент меня удивил, однако я обнаружил, что он проявляется в каждом курсе, который я преподавал с использованием методов гибкой разработки. В результате я продолжил использовать их во всех моих курсах, и в личной практике разработки игр, и даже когда писал эту книгу. Я надеюсь, что вы поступите так же[1].

Методология Скрам

За годы, прошедшие после 2001 года, было разработано множество инструментов и методик, помогающих коллективам легче адаптироваться к гибкой разработке. Одна из моих самых любимых — методология Скрам.

В действительности методология Скрам появилась задолго до манифеста гибкой разработки и разрабатывалась разными людьми, но ее связь с гибкой разработкой была закреплена в 2002 году, в книге Кена Швабера и Майка Бидла «Agile Software Development with Scrum»[2]. В ней они описали многие типичные элементы методологии Скрам, по-прежнему пользующиеся популярностью.

Цель Скрама, как и большинства методологий гибкой разработки, — создать действующий продукт или игру как можно быстрее, чтобы получить возможность адаптировать его с учетом отзывов тестировщиков и дизайнеров. В оставшейся части этой главы я познакомлю вас с некоторыми терминами и приемами, используемыми в методологии Скрам, и покажу, как составлять графики выполнения работ в электронной таблице на примере графика, созданного мной для этой книги.

Команда Скрам

Команда Скрам, если говорить о прототипировании игр, состоит из одного *владельца продукта*, одного *скрам-мастера* и *команды разработчиков* численностью до 10 человек, обладающих знаниями в разных областях, включая программирование, проектирование игр, моделирование, графическое оформление, звуковое оформление и т. д.

- **Владелец продукта:** голос клиента или будущих игроков вашей игры.[3] Задача владельца продукта — убедиться, что все интересные особенности включены в игру, и донести правильное понимание сути игры.

---

[1] Спасибо моему другу и коллеге Тому Фризину, который познакомил меня с методологией гибкой разработки Скрам.

[2] Ken Schwaber and Mike Beedle, Agile Software Development with Scrum (Upper Saddle River, NJ: Prentice Hall, 2002).

[3] В редких случаях владелец продукта может быть фактическим клиентом, но чаще это сотрудник компании, защищающий интересы клиента.

- **Скрам-мастер:** голос разума. Скрам-мастер проводит ежедневные совещания (скрам-митинги), следит, чтобы все были заняты делом и нагрузка не была избыточной. Скрам-мастер выступает в роли связующего звена с владельцем продукта, предоставляя реалистичную информацию об оставшемся объеме работ над проектом и о том, насколько эффективно члены команды разработчиков справляются со своими задачами. Если проект отстает от графика или требуется урезать какие-то особенности, скрам-мастер отвечает за возвращение проекта в график и обеспечение всех необходимых изменений.
- **Команда разработчиков:** люди на передовой. Команда разработчиков включает всех, кто работает над проектом, и может включать владельца продукта и скрам-мастера, которые за пределами ежедневных совещаний часто действуют как обычные члены команды. На одном ежедневном совещании члены команды получают задания, на следующем отчитываются об их выполнении. В Скраме у отдельных членов команды больше свободы, чем в других методологиях разработки, но за эту свободу приходится платить ежедневной подотчетностью перед всей командой.

Бэклог продукта/список возможностей

Скрам-проект начинается с *журнала требований продукта* (или бэклога), также известного как *список возможностей*, в котором перечисляются все возможности, элементы механики, художественные элементы и т. д., которые команда должна реализовать в окончательной версии игры. Некоторые из них имеют расплывчатые формулировки и в процессе разработки разбиваются на несколько более конкретных подпунктов.

Выпуски и спринты

Продукт разбивается на *выпуски* и *спринты*. Под выпуском подразумевается известный момент времени, когда вы покажете игру другим (например, на встрече с инвесторами, в день выпуска общедоступной бета-версии или на раунде официального тестирования), а спринт — это шаг на пути к выпуску. В начале спринта создается *список требований спринта*, включающий все, что должно быть завершено к его концу. Обычно спринт длится от 1 до 4 недель, и какие бы задачи вы ни взялись решить в ходе спринта, важно к его окончанию иметь действующую игру (или часть игры). Но еще лучше, если, начиная с момента создания первого действующего прототипа, каждый рабочий день будет завершаться действующей игрой (хотя иногда это трудная задача).

Ежедневные совещания

Ежедневные совещания (*скрам-митинг*) — это короткие 15-минутные собрания на ходу (когда все участники проводят встречу на ногах), которые не дают команде

выбиться из графика. Совещание ведет скрам-мастер, и каждому члену команды задаются три вопроса:

1. Что достигнуто за вчерашний день?
2. Чего предполагается достичь сегодня?
3. Какие препятствия могут встретиться на пути?

Вот и все. Цель ежедневной встречи — держать всех в курсе текущих дел. Ответы на вопросы 1 и 2 сверяются с *диаграммой сгорания задач*, чтобы увидеть, как движется реализация проекта. Встречи должны быть как можно короче, чтобы творческие личности в вашей команде не тратили много времени на совещания. Например, любые проблемы, обнаруживающиеся при ответе на вопрос 3, не должны обсуждаться на ежедневных совещаниях. Вместо этого скрам-мастер должен определить добровольца, чтобы помочь решить потенциальную проблему, и двигаться дальше. Потом, когда совещание закончится, человек, столкнувшийся с проблемой и добровольный помощник обсуждают пути решения без участия остальной команды.

Благодаря скрам-митингам все члены команды знают свои обязанности, то, над чем работают их коллеги, и кого они могут попросить о помощи, если потребуется. Так как встречи проводятся каждый день, проблемы решаются сразу после появления и не накапливаются как снежный ком.

Диаграмма сгорания задач

Я считаю диаграмму сгорания задач одним из самых полезных инструментов в моем процессе разработки игр и в моих курсах. Диаграмма сгорания задач начинается с составления списка задач, которые должны быть выполнены в ходе спринта (список требований спринта), и определения времени, необходимого для выполнения каждой задачи (в часах, днях, неделях и т. д.). На протяжении всего проекта диаграмма сгорания задач позволяет следить за продвижением каждого члена к поставленной ему цели и определить не только общее количество часов, оставшееся до завершения проекта, но и понять, успеет ли команда закончить его к назначенному сроку.

Вся прелесть диаграммы сгорания задач состоит в том, что она преобразует огромный объем данных в простую диаграмму, отвечающую на три важных вопроса:

1. Команда успеет закончить спринт вовремя?
2. Какие задачи возложены на каждого члена команды?
3. Все ли в команде работают с полной отдачей сил (все ли справляются с нагрузкой)?

На эти вопросы часто трудно ответить, в какой бы команде вы ни работали, но диаграмма сгорания задач эффективно помогает получить эти ответы. Графики выполнения работ настолько важны, что остальная часть главы будет посвящена описанию использования шаблона диаграммы сгорания задач, который я подготовил для вас.

Пример диаграммы сгорания задач

Я создал шаблон диаграммы сгорания задач, доступный онлайн в виде документа Google Sheets. Приложение электронных таблиц Google Sheets — бесплатный конкурент Microsoft Excel, который был представлен в главе 11 «Математика и баланс игры». Объяснение формул для электронных таблиц, используемых в этом графике выполнения работ, выходит далеко за рамки книги, зато с основами электронных таблиц и особенностями их использования можно познакомиться на примере балансирования игр в главе 11.

Пример диаграммы сгорания задач в электронной таблице можно найти:

- по ссылке http://bit.ly/IGDPD_BDC_Example

  или

- на веб-сайте книги http://book.prototools.net в разделе Chapter 14.

Этот график много раз будет упоминаться на следующих страницах, поэтому, пожалуйста, перейдите по указанной ссылке прямо сейчас. Чтобы получить возможность внести исправления в таблицу, вы должны создать свою копию. Для этого выберите пункт меню File > Make a copy... (Файл > Создать копию), как показано на рис. 14.1 слева.

После создания копии откройте ее в Google Sheets и продолжайте чтение главы.

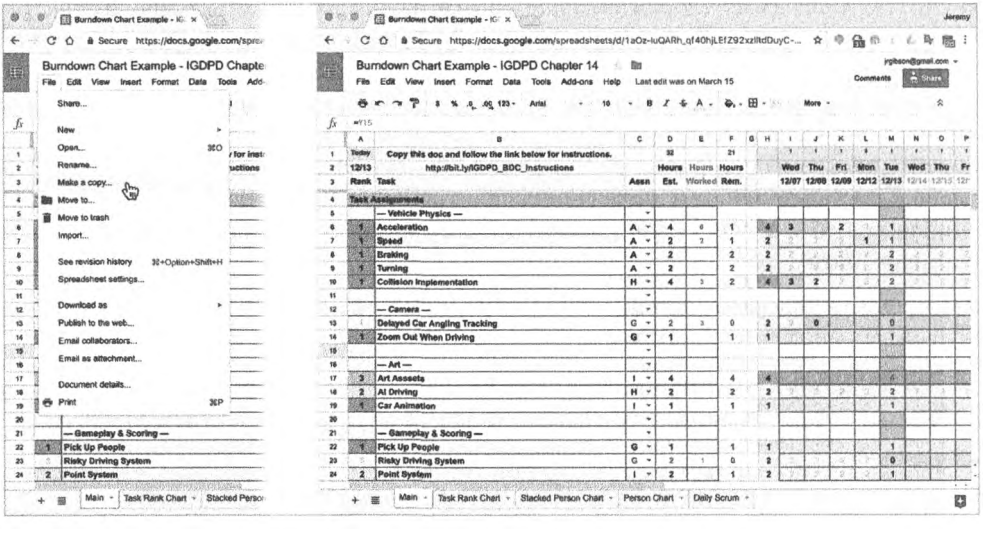


Рис. 14.1. Пункт в главном меню для создания копии и лист Main из примера в Google Sheets

Пример диаграммы сгорания задач: листы

Современные электронные таблицы часто разбиваются на несколько листов, между которыми можно переключаться, выбирая вкладки внизу окна (с надписями Main, Task Rank Chart и др., как можно видеть на рис. 14.1). Знакомясь с описанием листов, что приводится ниже, щелкайте на вкладках в нижней части окна электронной таблицы, чтобы увидеть каждый лист.

Все листы в этой таблице имеют определенное назначение:

- Main (Главный): на этом листе выполняется слежение за ходом выполнения заданий и оставшимся временем в часах. Здесь вводится больше всего данных.
- Task Rank Chart (Диаграмма по рангам заданий): эта диаграмма отражает текущий прогресс и сроки выполнения заданий в проекте, отсортированных по рангу (или важности).
- Stacked Person Chart (Диаграмма с накоплением по участникам): эта диаграмма отражает текущий прогресс отдельных членов команды, которым поручены задания.
- Person Chart (Диаграмма по участникам): эта диаграмма отражает прогресс выполнения заданий отдельным участником проекта.
- Daily Scrum (Ежедневные встречи): этот лист позволяет членам команды принимать виртуальное участие в ежедневных встречах, даже если они не могут встретиться друг с другом лично.

Далее мы исследуем каждый лист подробнее.

МЕНЯЙТЕ ЗНАЧЕНИЯ ТОЛЬКО В ЯЧЕЙКАХ С ТЕМНО-СЕРЫМИ РАМКАМИ!
В обеих таблицах, Burndown Chart Example и Burndown Chart Template, изменяйте значения только в ячейках с темно-серыми границами. Во всех других ячейках данные или не подлежат изменению, или (что более вероятно) вычисляются с использованием формул. Например, даты в ячейках I3:Z3 вычисляются по формуле на основе начальной даты (F102), конечной даты (F103) и рабочих дней (J102:J108), указанных в ячейках с темно-серыми рамками в области Sprint Settings (Параметры спринта). То есть вы ни при каких условиях не должны вручную править содержимое ячеек I3:Z3.

Лист Main

Лист Main (Главный) — это место для внесения большинства изменений в диаграмму сгорания задач. Верхняя часть этого листа включает примерно 100 строк и служит для слежения за заданиями, их назначением и количеством оставшегося рабочего времени для их выполнения. В нижней половине находятся ячейки для ввода имен членов команды, начальной и конечной даты проекта и определения рабочих дней недели. Также в нижней части листа сосредоточены расчетные данные, используемые в диаграммах на других вкладках.

Параметры спринта

Прокрутите лист вниз до строки 101, чтобы увидеть раздел Sprint Settings (Параметры спринта), как показано на рис. 14.2. Как уже упоминалось выше в этой главе, обычно *спринт* длится пару недель, в течение которых предполагается выполнить определенный круг заданий (*список требований спринта*). Этот раздел листа Main (Главный) включает информацию, которую необходимо настроить при создании электронной таблицы ГВР для каждого спринта.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Today	Copy this doc and follow the link below for instructions.		32		21			1	1	1	3	1	1
2	12/13	http://bit.ly/IGDPD_BDC_Instructions		Hours	Hours	Hours			Wed	Thu	Fri	Mon	Tue	Wed
3	Rank	Task	Assn	Est.	Worked	Rem.			12/07	12/08	12/09	12/12	12/13	12/14
100														
101	Sprint Settings – Only change cells with dark gray borders								Workdays					
102		Archon	A	Start Date		12/07			Sun	0				
103		Henri	H	End Date		12/21			Mon	1				
104		Icarus	I						Tue	1				
105		Gilbert	G	Total Days		14			Wed	1				
106				Work Days		10			Thu	1				
107									Fri	1				
108		All	ALL						Sat	0				
109		Unassigned												
110		Days to Look Back for Burndown Velocity	2											

Рис. 14.2. Раздел Sprint Settings (Параметры спринта) на листе Main (Главный)

- **Члены команды:** список до шести членов команды, принимающих участие в этом спринте (B102:B107), а также их одно- или двухбуквенные идентификаторы (C102:C107), которые будут использоваться для назначения заданий в разделе выше.
- **Даты спринта:** здесь определяются начальная (F102) и конечная (F103) даты спринта.
- **Рабочие дни недели:** в ячейках J102:J107 вводится 1 для дней недели, которые в вашей команде считаются рабочими, и 0 — для дней, в которые не планируется работать. Эта информация используется для вычисления количества рабочих дней в ячейке F106.

Эта информация используется многими другими ячейками в таблице для расчетов. Она находится ниже всего остального на листе Main (Главный), потому что определяется только один раз, в начале спринта.

РАСЧЕТНОЕ КОЛИЧЕСТВО ЧАСОВ. В этом примере диаграммы сгорания задач текущим днем (ячейка A2) всегда считается вторник 13 декабря. Однако в шаблоне Burndown Chart Template в ячейке с текущим днем (A2) всегда отражается фактическая текущая дата.

Назначение заданий и времени для выполнения

Прокрутите лист Main (Главный) обратно вверх (рис. 14.3).

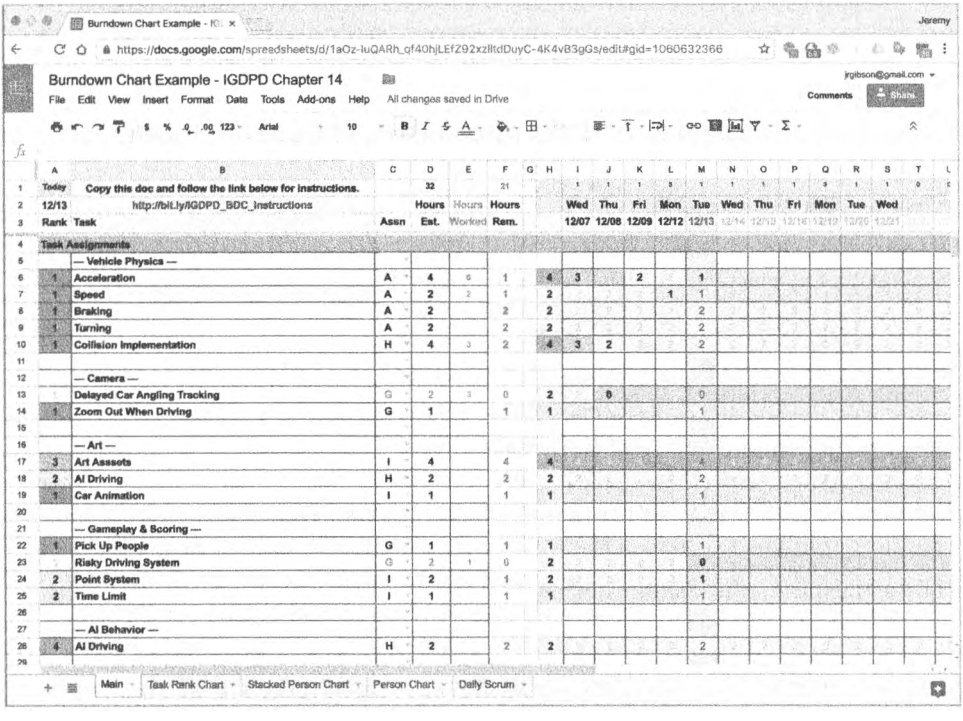


Рис. 14.3. Раздел Task Assignments (Назначение заданий) на листе Main (Главный)

Перед началом спринта в столбцы A:D каждой строки необходимо ввести важную информацию:

A. Rank (Ранг): Степень важности задания от 1 (критически важное) до 5 (маловажное).

B. Task (Задание): Краткое описание задания.

C. Assignment (Поручено): Одно- или двухбуквенный идентификатор члена команды, которому поручено это задание.

D. Hours Estimate (Расчетное время): Предполагаемое количество часов на выполнение этого задания.

Предполагаемое количество часов на выполнение каждого задания является основой всей диаграммы сгорания задач. На протяжении всего проекта вы будете обращаться к этому числу, поэтому важно выбирать это число как можно более точным и адекватным. См. совет «Расчетное время».

РАСЧЕТНОЕ ВРЕМЯ. Оценка времени, необходимого для выполнения задания, — одна из самых больших сложностей для программистов, художников и других творческих работников. На выполнение работы почти всегда уходит больше времени, чем предполагалось, и очень редко случаются ситуации, когда задание планировалось выполнить за 20 часов, а удалось сделать всего за два. Пока просто выбирайте наиболее вероятное значение, следуя каким-нибудь простым правилам, учитывая при этом, что с увеличением задания точность неизбежно уменьшается.

- Если вы оцениваете время в часах, придерживайтесь значений 1, 2, 4 или 8.
- В днях: 1, 2, 3 или 5.
- В неделях: 1, 2, 4 или 8.

Впрочем, если время на выполнение каких-то заданий для спринта вы начинаете оценивать в неделях, вам лучше разбить их на меньшие задания!

Прогресс спринта

В правой половине листа Main (Главный) расположен раздел слежения за выполнением заданий в спринте. Столбец H отражает начальное расчетное время в часах для каждого задания, а столбцы правее — прогресс команды в их выполнении. Столбец с текущей датой выделен синим цветом фона, а цифры в нем отображаются красным (в данном примере — столбец M).

Работая над разными заданиями, члены команды сообщают расчетное количество часов до завершения задания в столбцах I:Z. В крайнем случае вы сами должны заполнять ГВР в конце каждого рабочего дня и вводить оставшиеся часы только в столбец, соответствующий текущему дню (с синим фоном). В столбцах, соответствующих текущему и предшествующим дням, числа отображаются жирным черным шрифтом, если в этот день член команды работал над заданием и уменьшил расчетное количество оставшихся часов.

Расчетное и фактическое время

Одним из важнейших понятий в графике выполнения работ является разница между расчетным и фактическим количеством времени. После того как вы определите расчетное количество часов для задания, оставшееся время на его выполнение должно вычисляться не в фактических рабочих часах, а *в процентах незавершенной доли*. Например, рассмотрим задание Acceleration (Ускорение) в строке 6 (рис. 14.4).

Первоначально на выполнение задания Acceleration (Ускорение) планировалось потратить 4 часа.

- 12/07 (7 декабря): Архонт (Archon, A) работал над заданием Acceleration (Ускорение) 2 часа, но выполнил его только на 25 %. Незавершенная доля составила 75 %, поэтому он ввел в таблицу на эту дату (ячейка I6) число 3, потому что 3 — это 75 % от исходных расчетных 4 часов. Он также ввел число 2 в столбец Hours Worked (Потрачено часов, столбец E) для учета фактически потраченного времени.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Today	Copy this doc and follow the link below for instructions.		32		21			1	1	1	3	1
2	12/13	http://bit.ly/IGDPD_BDC_Instructions		Hours	Hours	Hours			Wed	Thu	Fri	Mon	Tue
3	Rank	Task	Assn	Est.	Worked	Rem.			12/07	12/08	12/09	12/12	12/13
4	Task Assignments												
5		— Vehicle Physics —											
6	1	Acceleration	A	4	6	1		4	3		2		1
7	1	Speed	A	2	2	1		2				1	1
8	1	Braking	A	2		2		2					2

Рис. 14.4. Выполнение задания Acceleration (Ускорение) за первые 5 дней

- 12/09: Он проработал еще 3 часа, доведя степень выполнения задания до 50 % и оставив 2 часа от расчетных 4. Поэтому он ввел 2 в столбец за 12/09 (K6) и добавил 3 часа к фактически потраченному времени в столбце Hours Worked (Потрачено часов), в результате общее потраченное время получилось равным 5 часам.
- 12/13 (сегодня): За один час задание было выполнено еще на 25 % (в этот раз он работал немного интенсивнее), и теперь, на сегодняшний день, ему осталось сделать 25 % — или 1 расчетный час — в задании Acceleration (Ускорение). Он ввел число 1 в столбец 12/13 (L6) и увеличил число в столбце Hours Worked (Потрачено часов, E6) до 6.

Как видите, наиболее важное значение — это *процент оставшейся доли задания*, выраженный в часах от первоначального расчетного времени. Но Архонт (Archon) также записал 6 часов, фактически потраченных на задание Acceleration (Ускорение), в столбец Hours Worked (Потрачено часов, E). В будущем это поможет ему точнее определять расчетное время (сейчас, похоже, ему требуется в два раза больше времени, чем он прогнозировал).

Данные, указанные в листе Main (Главный), используются в трех диаграммах, помогающих точнее оценить движение вашей команды вперед и вклад каждого члена команды.

Лист Task Rank Chart

Лист Task Rank Chart (Диаграмма по рангам заданий) (рис. 14.5) отражает текущий прогресс с накоплением по рангам заданий. Здесь по двум признакам видно, что команда отстает от графика:

- Черная линия On-Track Line (План) показывает, какой средний объем работы должен выполняться каждый рабочий день, чтобы завершить задания вовремя. Если сумма всех рангов находится выше этой линии, значит, команда отстает от графика, если ниже — команда идет с опережением. Это дает членам команды возможность понять, насколько успешно они продвигаются вперед.
- Красная линия Burndown Velocity (Скорость выполнения) отражает недавнюю скорость выполнения, показывая, чего может достичь команда в ближайшие дни. Если она касается базовой линии (то есть нулевого уровня по оси Estimated Work Hours (Расчетное количество рабочих часов)) до конечной даты, значит,

команда хорошо справляется со своими задачами. Напротив, если она нигде не касается базовой линии, команда рискует не успеть завершить проект вовремя.

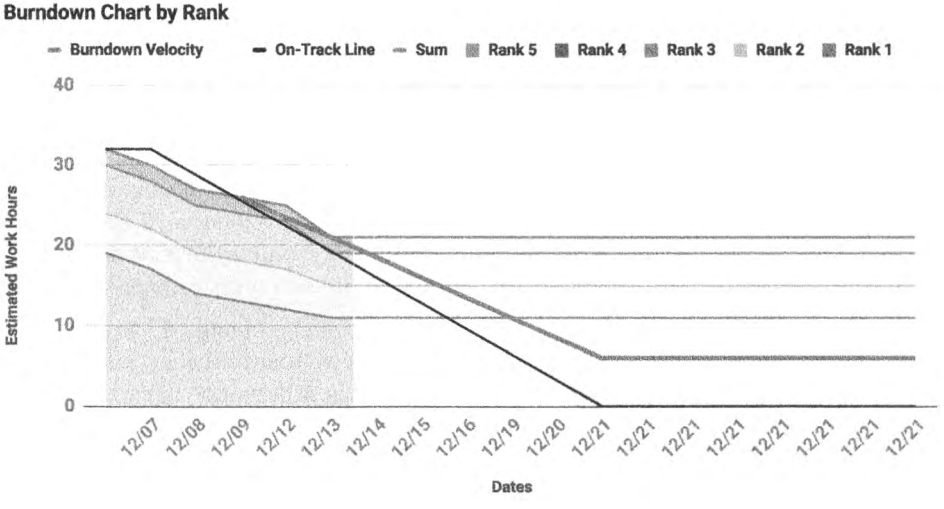


Рис. 14.5. Диаграмма выполнения заданий по рангам

Скорость выполнения (Burndown Velocity, BDV) — это текущая скорость выполнения работ, вычисленная по количеству расчетных часов, выполненных за день. Ячейка C110 (см. рис. 14.2) на листе Main (Главный) определяет период в днях для вычисления скорости. В данном случае — два дня, именно поэтому красная линия простирается назад от текущей даты только на два рабочих дня (от 13 декабря до 9 декабря, без учета выходных 10 и 11 декабря).

Также обратите внимание, как на графике Task Rank Chart (Диаграмма по рангам заданий) видно, что команда в первую очередь уделяет внимание заданиям с рангом 1 (красная область внизу) и оставляет менее приоритетные задания (с рангами 4 и 5) на потом. В этой части команда определенно преуспевает. Область заданий с рангом 1 (красная) постепенно сужается, а область заданий с рангом 4 (фиолетовая) остается неизменной по ширине.

Лист Stacked Person Chart

Лист Stacked Person Chart (Диаграмма с накоплением по участникам) (рис. 14.6) отражает текущий прогресс отдельных членов команды. Эта диаграмма поможет увидеть вклад каждого члена команды в продвижение проекта. В идеале желательно, чтобы вклады всех членов команды сужались на графике одинаково во все рабочие дни. Также желательно, чтобы разделы всех участников были примерно одинаковы в любые дни. В данном случае можно сказать, что Архонт (Archon, красная область)

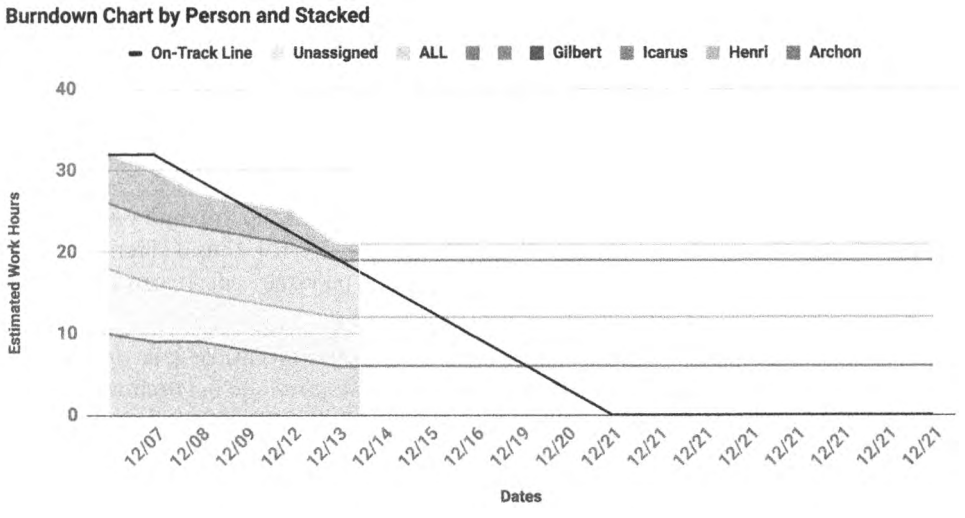


Рис. 14.6. Диаграмма выполнения по участникам с накоплением

получил большую нагрузку, чем другие, а Икар (Icarus, зеленая область) выполнил заданий меньше других, потому что зеленая область сузилась лишь ненамного.

Лист Person Chart

Диаграмма (без накопления) на листе Person Chart (Диаграмма по участникам), изображенная на рис. 14.7, отражает прогресс выполнения заданий отдельными

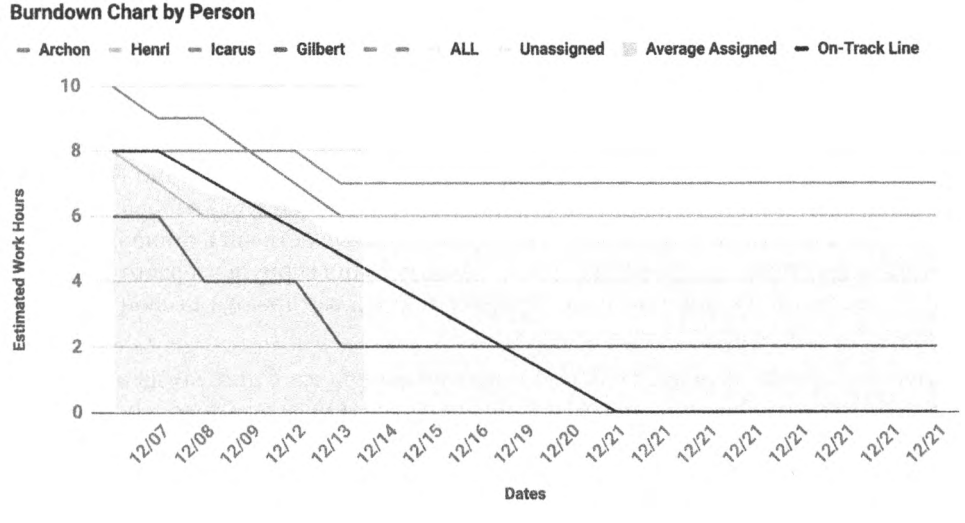


Рис. 14.7. Диаграмма выполнения по участникам

участниками проекта относительно друг друга. Область, залитая серым фоном, соответствует среднему объему работы, назначенному каждому члену команды, при условии равенства рабочей нагрузки. Черная линия On-Track Line (План) показывает, какой объем должен выполнять каждый участник ежедневно, чтобы оставаться в графике (если им назначить одинаковый объем работы). Разные линии показывают фактический объем работы, выполняемой каждым участником.

В данном случае видно, что Архонт (Archon) действительно хорошо справляется с заданиями, хотя ему поручено работы больше, чем другим. Генри (Henri) и Икар (Icarus) не отличаются успехами, а Гилберт (Gilbert) работает урывками, но каждый раз вносит довольно весомый вклад.

Эта диаграмма дает отличную возможность увидеть, что на самом деле происходит с проектом, кто вносит наибольший вклад, а кому желательно бы поднажать.

Лист Daily Scrum

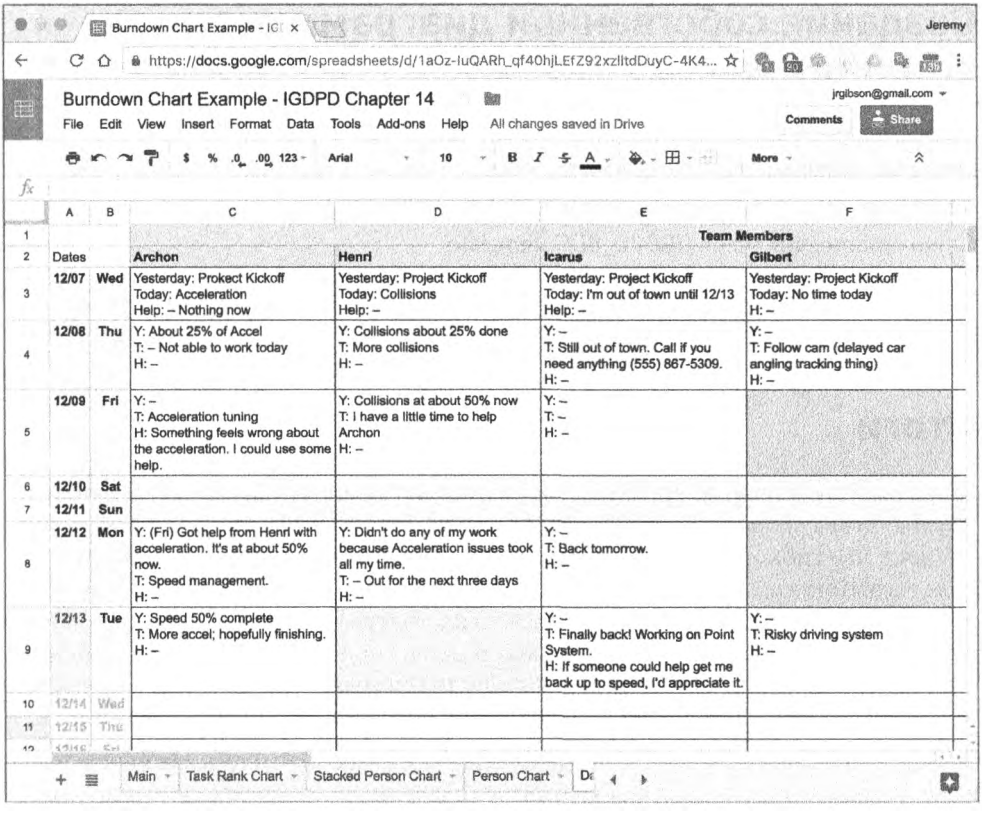
Всегда предпочтительнее личное участие членов команды в ежедневных встречах, но, если это невозможно, данный лист (рис. 14.8) поможет участникам оставаться на связи. Так же как на обычных скрам-встречах, каждый член команды должен каждый день отвечать на три вопроса:

- **Вчера (Yesterday, Y):** Каждый участник сообщает, что достигнуто за вчерашний день (или с момента последней скрам-встречи).
- **Сегодня (Today, T):** Каждый участник сообщает, что он планирует сделать сегодня (или до следующей скрам-встречи).
- **Помощь (Help, H):** Каждый участник просит помощи, если необходимо.

Вся команда должна каждый день заполнять скрам-отчет к определенному времени. В данном примере команда выбрала таким временем 10 часов утра. Если бы временем отчета было выбрано 18 часов вечера, тогда терминология изменилась бы на: Сегодня (Today, T), Далее (Next, N) и Помощь (Help, H).

Лист Daily Scrum (Ежедневные встречи) помогает взглянуть на команду с другой стороны, в отличие от других листов графика. Например, глядя на лист Person Chart (Диаграмма по участникам), можно подумать, что Икар (Icarus) вообще ничего не делает; но если открыть лист Daily Scrum (Ежедневные встречи), становится ясно, что с начала спринта он был в отъезде и только сегодня вернулся к работе. Но даже будучи в отъезде, он заполнял отчет каждый день и даже предлагал свои контакты, если кому-то понадобится его помощь.

С другой стороны, Гилберт (Gilbert) никак не отмечался в дни, когда не работал, и весьма скупо сообщает о сделанном в рабочие дни. Он делает довольно много, но не общается с командой, и эту проблему нужно решить как можно скорее.



Burndown Chart Example - IGDPD Chapter 14

	A	B	C	D	E	F
1					Team Members	
2	Dates		Archon	Henri	Icarus	Gilbert
3	12/07	Wed	Yesterday: Prokect Kickoff Today: Acceleration Help: – Nothing now	Yesterday: Project Kickoff Today: Collisions Help: –	Yesterday: Project Kickoff Today: I'm out of town until 12/13 Help: –	Yesterday: Project Kickoff Today: No time today H: –
4	12/08	Thu	Y: About 25% of Accel T: – Not able to work today H: –	Y: Collisions about 25% done T: More collisions H: –	Y: – T: Still out of town. Call if you need anything (555) 867-5309. H: –	Y: – T: Follow cam (delayed car angling tracking thing) H: –
5	12/09	Fri	Y: – T: Acceleration tuning H: Something feels wrong about the acceleration. I could use some help.	Y: Collisions at about 50% now T: I have a little time to help Archon H: –	Y: – T: – H: –	
6	12/10	Sat				
7	12/11	Sun				
8	12/12	Mon	Y: (Fri) Got help from Henri with acceleration. It's at about 50% now. T: Speed management. H: –	Y: Didn't do any of my work because Acceleration issues took all my time. T: – Out for the next three days H: –	Y: – T: Back tomorrow. H: –	
9	12/13	Tue	Y: Speed 50% complete T: More accel; hopefully finishing. H: –		Y: – T: Finally back! Working on Point System. H: If someone could help get me back up to speed, I'd appreciate it.	Y: – T: Risky driving system H: –
10	12/14	Wed				
11	12/15	Thu				

Рис. 14.8. Лист Daily Scrum (Ежедневные встречи) в примере диаграммы сгорания задач

Взглянув на строку, соответствующую сегодняшнему дню (12/13), можно увидеть, что сегодня Генри (Henri) еще не заполнил свой отчет (возможно, потому, что куда-то уехал). Лист Daily Scrum (Ежедневные встречи) подсвечивает пустые клетки: зеленым — соответствующие сегодняшнему дню и красным — соответствующие прошедшим дням.

В коллективе жизненно важное значение имеет плотное взаимодействие, и этот лист поможет вам достичь его. Ко мне не раз подходили студенты, обеспокоенные тем, что не знали, где находятся члены их команды и чем они занимаются. После внедрения этой электронной таблицы в моих классах студенты, использовавшие ее, обнаружили, что работать над проектом стало намного спокойнее. Даже если кто-то не мог заниматься работой один или два дня, проект все равно оставался управляемым, если остальные члены команды знали, чего ожидать.

Создание собственной диаграммы сгорания задач

Теперь, познакомившись с особенностями примера диаграммы сгорания задач, вы сможете создать свои диаграммы на основе шаблона, который я использовал для этого примера. Как и сам пример, вы можете найти шаблон в двух местах:

- по ссылке http://bit.ly/IGDPD_BDC_Template

  или

- на веб-сайте книги http://book.prototools.net, в разделе Chapter 14.

Итоги

Когда вы начнете проектировать и создавать свои игры, вы узнаете, что порой трудно уследить за процессом разработки. За годы работы разработчиком и профессором я понял, что гибкое мышление и методология Скрам — два лучших инструмента для этого. Конечно, я не могу гарантировать, что вам эти инструменты подойдут так же хорошо, как мне и моим студентам, но я настоятельно рекомендую попробовать их. В конце концов, неважно, насколько хорошо гибкое мышление и методология Скрам подходят вам. Важно, чтобы вы нашли инструменты, которые помогут вам сохранить интерес и высокую производительность при работе над играми.

В следующей главе я расскажу об индустрии цифровых игр и о том, как можно войти в нее. Я также расскажу, как познакомиться с людьми на конференциях разработчиков игр и что можно найти в университетской программе обучения разработке игр[1].

[1] Возможно, вам интересно, почему я выбрал имена Архонт (Archon), Генри (Henri), Икар (Icarus) и Гилберт (Gilbert) для своего примера диаграммы сгорания задач. Это имена персонажей, которые я и три моих товарища использовали, когда мы вместе работали над игрой *Skyrates* в аспирантуре. Я использовал эти имена как дань уважения им. Независимо от того, что могла бы сказать эта вымышленная диаграмма, они были тремя лучшими товарищами, с которыми я когда-либо имел удовольствие работать, и я был бы рад вновь поработать с любым из них.

15 Индустрия цифровых игр

Если вы нашли время для чтения этой книги и обучения разработке прототипов игр, можно смело предположить, что вы определенно заинтересованы в присоединении к игровой индустрии.

В этой главе я немного расскажу о текущем состоянии отрасли и затем поведаю об университетских образовательных программах обучения разработке игр. Также я дам несколько советов о том, как знакомиться с людьми, как работать в Сети и как искать работу. Наконец, я расскажу, как подготовиться к разработке собственных проектов.

Об индустрии игр

О современной индустрии игр можно с полной определенностью сказать, что она меняется. Многие известные компании, такие как Electronic Arts и Activision, существовавшие три последних десятилетия, существуют и по сию пору, но при этом появилось множество стартапов, таких как Riot Games — разработчик *League of Legends*, который вырос из небольшого коллектива, организованного в 2008 году, до одной из самых популярных в мире онлайн-игр. Еще совсем недавно никто бы не поверил, что сотовый телефон может быть одной из самых успешных игровых платформ, но теперь объем продаж игр для устройств Apple с операционной системой iOS составляет миллиарды долларов. Поскольку все меняется слишком быстро, я не собираюсь давать вам какие-либо конкретные цифры. Вместо этого я подскажу, на какие ресурсы (которые обновляются ежегодно) можно обратиться за ними.

Отчет «Essential Facts» ассоциации Entertainment Software Association

ESA (http://theesa.com) — это ассоциация производителей развлекательного программного обеспечения и лоббистская организация для большинства компаний, производящих игры. Именно ESA выступила в Верховном суде США с требованием признать игры произведениями искусства, подпадающими под действие Первой поправки к Конституции. Ассоциация ESA выпускает ежегодный отчет о состоянии

индустрии игр под названием «Essential Facts» (Основные факты), который можно найти, набрав в Google «ESA essential facts». Конечно, в отчетах присутствует некоторая тенденциозность (работа ESA как раз и состоит в том, чтобы показать индустрию игр в розовом свете), тем не менее они позволяют получить общее представление о положении дел в игровой индустрии. Далее перечисляются десять фактов из отчета «Essential Facts» за 2016 год:[1]

1. В **63 % домохозяйств в США** проживал хотя бы один человек, игравший в игры не менее 3 часов в неделю. В среднем **на каждое домохозяйство приходилось 1,7 игрока**. С учетом текущего количества домохозяйств в США **общее число игроков составляет примерно 125 миллионов**[2].
2. В 2015 году потребители потратили **23,5 миллиарда долларов** на видеоигры, оборудование и аксессуары.
3. Доля цифрового контента, включая игры, дополнительный контент, мобильные приложения, подписки и игры социальных сетей, составила **56 % от общего объема продаж игр** в 2015 году (против 40 % в 2012 году).
4. Средний возраст игроков мужского пола — 35 лет, а продолжительность увлечения играми — 13 лет. Средний возраст игроков женского пола — 44 года.
5. **26 % игроков старше 50 лет**. Исходя из цифр, приводившихся выше, это означает, что в США **32 миллиона игроков старше 50 лет**! Это **гигантский** неосвоенный рынок!
6. **41 % всех игроков — женщины** (хотя, к сожалению, этот показатель снизился после пика в 48 % в 2014 году). Женщины старше 18 все еще составляют большую часть игроков (31 %), чем юноши 17 лет и младше (17 %).
7. Наиболее распространенным устройством, на котором люди играют в игры, является персональный компьютер (56 %). Среди других распространенных платформ специализированные игровые приставки (53 %), смартфоны (36 %), беспроводные устройства, такие как iPad (31 %), и специализированные портативные системы (17 %).
8. Игры-головоломки занимают доминирующее положение на рынке беспроводных и мобильных устройств. Наибольшей популярностью на беспроводных и мобильных устройствах пользуются головоломки/настольные/карточные игры (38 %), активные игры (6 %) и стратегии (6 %).
9. В целом 89 % игр, оцененных организацией ESRB (Entertainment Software Ratings Board — негосударственная рейтинговая организация по оценке развлекательного программного обеспечения) в 2012 году, получили рейтинг «E» — Everyone (для всех), «E10+» — Everyone 10+ (для всех от 10 и старше) или

[1] http://www.theesa.com/wp-content/uploads/2016/04/Essential-Facts-2016.pdf. Выделение мое. — *Авт.*

[2] Дополнительная информация из Бюро переписи населения США: https://www.census.gov/quickfacts/. 116 926 305 домохозяйств в США * 63 % * 1,7 игроков на домохозяйство ≈ 125 миллионов.

«T» — Teen (для подростков). (Более подробную информацию о рейтингах игр ищите на сайтах www.esrb.org и www.pegi.info.)

10. Из наиболее увлеченных игроков 55 % знакомы с понятием «виртуальная реальность», а 22 % планируют приобрести оборудование виртуальной реальности в следующем году.

Что меняется

Изменения в индустрии связаны с условиями работы, стоимостью производства игр, появлением условно-бесплатных игр и независимых разработчиков.

Условия работы в игровых компаниях

Ничего не знающие об индустрии игр могли бы подумать, что работать в игровой компании легко и весело. Если вы немного знакомы с этой сферой, то, наверное, слышали, что сотрудники игровых компаний обычно работают по 60 часов в неделю с обязательными сверхурочными без дополнительной оплаты. Хотя в настоящее время положение дел в большинстве компаний лучше, те истории, что вы могли слышать, основаны на реальных фактах. Более того, у меня есть друзья, которые в периоды кранчей имеют 70-часовую обязательную рабочую неделю (работая по 10 часов в день без выходных). К счастью, эта тенденция пошла на спад в последнее десятилетие. В большинстве компаний, особенно в крупных, по-прежнему иногда будут просить вас работать сверхурочно, но от разработчиков уже все реже можно услышать, что они неделями не видят своих детей или друзей (хотя, к сожалению, такое еще случается). Тем не менее на собеседовании при поступлении на работу в любую игровую компанию обязательно спрашивайте об их политике в отношении сверхурочной работы и кранчей.

Стоимость производства игр категории AAA

Каждое поколение игровых приставок пережило рост стоимости разработки топовых игр (также известных как AAA-игры, произносится как «трипл-а»). Это было особенно верно для PlayStation 3 и Xbox 360 против PlayStation 2 и Xbox, и эта тенденция продолжилась также для PlayStation 4 и Xbox One. Команды разработчиков AAA-игр в настоящее время насчитывают более 100 или 200 человек, и некоторые, казалось бы, небольшие команды фактически передают разработку некоторых аспектов другим студиям, в каждой из которых сотни своих работников. Все еще необычным, но уже не заоблачным для AAA-игр считается бюджет, превышающий 100 миллионов долларов, и участие в разработке более 1000 человек, работающих в нескольких разных студиях.

Все это влияет на игровую индустрию точно так же, как повлияло раздувание бюджетов фильмов в киноиндустрии: чем больше денег компания тратит на проект, тем меньше у нее желания рисковать. Вот почему на рис. 15.1, где изображен список из

отчета ESA с 20 наиболее продаваемыми играми в 2015 году, только одна (*Dying Light*) не является продолжением серии (эта версия *Minecraft* является ремейком PC-версии для игровой приставки).

1	CALL OF DUTY: BLACK OPS III (M)	11	BATMAN: ARKHAM KNIGHT (M)
2	MADDEN NFL 16 (E)	12	LEGO: JURASSIC WORLD (E)
3	FALLOUT 4 (M)	13	BATTLEFIELD HARDLINE (M)
4	STAR WARS BATTLEFRONT 2015 (T)	14	HALO 5: GUARDIANS (T)
5	NBA 2K16 (E)	15	SUPER SMASH BROS. (E)
6	GRAND THEFT AUTO V (M)	16	THE WITCHER 3: WILD HUNT (M)
7	MINECRAFT (E 10+)	17	DYING LIGHT (M)
8	MORTAL KOMBAT X (M)	18	DESTINY: THE TAKEN KING (T)
9	FIFA 16 (E)	19	NBA 2K15 (E)
10	CALL OF DUTY: ADVANCED WARFARE (M)	20	METAL GEAR SOLID V: THE PHANTOM PAIN (M)

Рис. 15.1. 20 самых продаваемых видеоигр в 2015 году (из отчета «ESA Essential Facts 2016»)

Взлет (и падение) условно-бесплатных игр

По данным Flurry Analytics, за первые шесть месяцев 2011 года условно-бесплатные игры для iOS быстро обогнали платные по уровню доходов[1]. В январе 2011 года доля платных игр (которые приобретаются непосредственно) составила 61 % от общего объема доходов от игр в магазине приложений iOS App Store. К июню эта доля рухнула до 35 %, а 65 % доходов принесли *условно-бесплатные* игры. Благодаря условно-бесплатной модели, в которой игрок получает игру бесплатно, но должен платить небольшие суммы за дополнительные преимущества или настройки игрового процесса, всего за несколько лет Zynga выросла из стартапа с двумя участниками до компании с более чем 2000 сотрудников. Однако эта модель лучше показала себя в отношении казуальных игр, чем традиционных, и некоторые разработчики игр традиционных жанров, создающие ныне мобильные игры, предпочли вернуться к платной модели, потому что считают, что их рынок менее склонен к условно-бесплатной модели.

Несколько условно-бесплатных игр прекрасно справились с привлечением более постоянных (то есть менее случайных) игроков. Главное отличие между этими и казуальными условно-бесплатными играми заключается в том, что многие казуальные игры позволяют и поощряют игроков приобретать конкурентные преимущества (то есть предлагают платить больше, чтобы выиграть больше), тогда как полноценные игры, такие как *Team Fortress 2* (TF2), позволяют приобретать только эстетические вещи (например, шляпы) или игровые предметы, меняющие механику игры, но не нарушающие ее сбалансированности (например, ракетомет

[1] Согласно статье «Free-to-play Revenue Overtakes Premium Revenue in the App Store» Джеферсона Валадареса (Jeferson Valadares, от 7 июля 2011), http://web.archive.org/web/20140108025130/http://blog.flurry.com/bid/65656/Free-to-play-Revenue-Overtakes-Premium-Revenue-in-the-App-Store. Была доступна в апреле 2018 года.

Black Box (Черный ящик) для Солдата имеет обойму с емкостью на 25 % меньше, но дает 15 очков здоровья при каждом попадании во врага). Кроме того, почти все игровые предметы, которые можно приобрести в игре TF2, можно также добыть самому в игровом процессе. Критически важным в данном случае является нежелание постоянных игроков чувствовать, что кто-то *купил* преимущество перед ними.

Выбирая между условно-бесплатной и традиционной платной моделями, учитывайте жанр игры, которую вы предполагаете развивать, особенности рынка и игроков. Посмотрите на другие игры на рынке, выясните, какие стандарты используются, а затем решайте, пойдете ли вы вперед, опираясь на них, или выберете свой путь.

Появление независимых разработчиков

В то время как разработка игр категории AAA продолжает дорожать, повсеместное распространение бесплатных или недорогих инструментов разработки игр, таких как *Unity*, *GameMaker* и *Unreal Engine*, привело к появлению всемирного сообщества независимых разработчиков, которого прежде никогда не было. Как вы увидите далее в этой книге, почти каждый может научиться программированию, и десятки разработчиков доказали, что для создания игры хватит отличной идеи, немного таланта и много времени. Многие известные игровые проекты начинались как увлечение одного человека, включая *Minecraft*, *Spelunky* и *The Stanley Parable*. В 2005 году начала работу IndieCade — конференция, посвященная исключительно независимым играм. Кроме того, существуют десятки других конференций — или посвященных независимой разработке, или следящих за этим направлением, или включающих конкурс для независимых разработчиков[1]. Создавать видеоигры теперь проще, чем когда-либо, и этому вас научит оставшаяся часть книги.

Обучение созданию игр

За последние десять лет обучение проектированию и разработке игр на университетском уровне превратилось из курьеза в серьезную область. В *Princeton Review* теперь ежегодно составляют список лучших программ бакалавриата и магистратуры обучения созданию игр, и есть даже программы для обучения аспирантов в области создания и изучения игр.

Обычно у людей возникает два вопроса в отношении этих программ:

- Так ли необходимо пройти курс обучения созданию игр?
- Какие программы следует выбрать?

[1] В духе полной открытости заявляю, что, начиная с 2013 года, я занимаю пост председателя IndieCade по образованию и развитию и занимался программированием для секций IndieXchange и Game U конференции (2013–2015). Я горд, что имею отношение к этой замечательной организации и конференции.

Так ли необходимо пройти курс обучения созданию игр?

Как профессор, посвятивший несколько последних лет своей жизни обучению по этим программам, в ответ на этот вопрос я определенно говорю «да». Программы обучения созданию игр дают несколько очевидных преимуществ:

- У вас будет время и место для развития навыков проектирования и разработки организованным способом.
- Вас будут окружать преподаватели, которые смогут дать вам честные и содержательные отзывы о вашей работе, и сверстники, которые смогут стать хорошими компаньонами. Кроме того, многие преподаватели на этих программах работают в индустрии игр и имеют связи с различными игровыми компаниями.
- Многие игровые компании активно набирают работников из числа студентов лучших школ. Оказаться в одной из них означает шанс попасть стажером в одну из игровых студий, которые вам нравятся.
- Когда новые сотрудники нанимаются из числа студентов университетских программ, особенно магистратуры, они часто входят в компанию на более высоком уровне, чем другие. Обычно, попадая в игровые компании, люди первое время работают в отделах контроля качества и тестирования игр. Если они хорошо зарекомендовали себя там, их могут заметить и предложить перейти на другие должности. Хотя это довольно эффективный способ входа в индустрию, я видел, как выходцы из университетских программ часто оказывались в более высоком положении, чем талантливые люди с опытом работы в несколько лет и прошедшие через службу контроля качества.
- Высшее образование в целом поможет вам расти и просто сделает вас хорошим человеком.

Здесь следует сделать несколько оговорок. На обучение необходимы время и деньги. Если у вас нет степени бакалавра, я лично считаю, что ее обязательно нужно получить. Она откроет перед вами больше дверей и даст вам больше возможностей. Степень магистра менее необходима в индустрии, но программы на степень магистра способны предложить более целенаправленное обучение и действительно могут перевернуть вашу жизнь. Обучение на степень магистра обычно занимает два-три года и стоит примерно 60 000 долларов или больше. Однако, как говорил мой профессор, доктор Рэнди Пауш, ваше время должно быть для вас важнее стоимости обучения. Долгосрочная задолженность и хищническая практика кредитования являются настоящими проблемами и повышают конечную стоимость продвинутого обучения. Но даже в этих условиях самым важным вопросом, который вы должны задать себе, является вопрос необходимости дополнительного обучения в течение от двух до шести лет, когда вы, возможно, уже работаете в индустрии. Когда я решил поступить в университет Карнеги—Меллона, я руководствовался желанием изменить направление своей карьеры. Я решил потратить два года своей жизни, чтобы платить деньги, а не зарабатывать, и в моем случае это решение полностью окупилось. Конечно же, вы должны поступать так, как считаете правильным.

Какие программы обучения следует выбрать?

В наши дни сотни университетов предлагают программы обучения созданию игр, и каждый год появляются новые. Школы, перечисленные в списке *Princeton Review*, пользуются большим уважением, но вы должны выбрать такую школу, чтобы она подходила именно вам, и обязательно с учетом пути, по которому вы собираетесь пойти. Уделите время изучению программ и узнайте об изучаемых предметах и факультетах. Оцените, как много внимания в них уделяется разным аспектам разработки игр: проектированию, оформлению, программированию, управлению проектами и т. д. Выполняются ли факультетом какие-то работы в игровой индустрии или он полностью сосредоточен на обучении? Каждая школа имеет свои сильные стороны.

Я в свое время обучался в центре развлекательных технологий университета Карнеги—Меллона на степень магистра в области развлекательных технологий. Главной целью центра было обучение совместной работе и работе с клиентами. В первом семестре (ставшем для меня лучшим семестром за все студенческие годы) при изучении предмета «Создание виртуальных миров» (Building Virtual Worlds, BVW) каждый учащийся участвовал в пяти двухнедельных проектах в составе команд, случайно подбираемых из сокурсников. Численность группы обычно составляла более 60 студентов, и это давало возможность поработать с разными людьми. В этом семестре каждый работал по 80 часов в неделю над заданиями для своей команды и при этом обучался еще двум-трем другим предметам, дополняющим BVW. В оставшиеся три семестра каждый студент закреплялся за одной командой на полный семестр и изучал только один дополнительный предмет. Для большинства проектов длительностью в один семестр имелся настоящий клиент, поэтому студенты центра обучались в первую очередь управлению ожиданиями клиентов, работе с коллегами и решению внутренних споров. Целью этих проектов было дать студентам полноценный опыт работы в индустрии всего за два года. Перед центром стояла задача подготовить дизайнеров игр, продюсеров, программистов и технических художников к работе в производственных командах.

Программа обучения на степень магистра изящных искусств на факультете «Игры и интерактивная среда» Южно-Калифорнийского университета (где я преподавал четыре года), напротив, была организована совсем иначе. Численность группы обычно составляла 15 человек или меньше, и в первый год все студенты обучались вместе нескольким предметам. Кроме групповых проектов, студенты выполняли также несколько индивидуальных заданий. Во второй год обучения студентам предлагалось разделиться и продолжить обучение в соответствии с их личными интересами. В этот год примерно половину предметов студенты изучали вместе всей группой, а вторую половину предметов каждый выбирал для себя сам. Третий год почти целиком был посвящен работе студентов над дипломными проектами. Но несмотря на то что каждый студент писал свой дипломный проект, они редко работали в одиночку. Большинство команд насчитывало от 6 до 10 человек, и в них могли принять участие другие заинтересованные студенты с других факультетов.

За каждым дипломным проектом закреплялись кураторы из числа наставников, работающих в индустрии и академических кругах, кому был интересен этот проект, а общее руководство проектом осуществлял представитель факультета «Игры и интерактивная среда». Целью факультета было «воспитание идейных лидеров». Для этой программы более важным был рост отдельных студентов и создание ими чего-то инновационного, а не подготовка их к работе в индустрии.

В настоящее время я преподаю своим студентам предмет «Проектирование и разработка игр» на факультете «Медиа и информация» в Университете штата Мичиган. Эта программа считается лучшей в мире среди программ обучения созданию игр. Наша задача состоит в том, чтобы подготовить студентов к профессиональной работе в индустрии. Одним из главных преимуществ программы является получение каждым студентом хорошего опыта по специальностям «Медиа и информация», «Информационные технологии» или «Художественное искусство» (три основные специальности в области разработки игр), а также лучшее образование в области проектирования и разработки игр, какое только возможно в нашем университете. Этим она очень отличается от программ в других университетах, где студенты получают сокращенный объем знаний в области создания игр.

Очевидно, что каждая из этих трех программ приносит пользу студентам по-своему. Я выбрал для иллюстрации эти программы, потому что наиболее знаком с ними, но, вообще говоря, все школы разные, и вы ради себя должны узнать, какие цели преследует каждая в отношении своих студентов и как она предполагает достичь их через предлагаемые предметы.

Вхождение в индустрию

Содержимое для этого раздела я взял из своего доклада «Общение с профессионалами», который сделал на конференции Game Developers Conference Online в 2010 году. Если вам интересно ознакомиться с полной версией, вы найдете ее на веб-сайте книги[1].

Знакомство с людьми, занятыми в индустрии

Лучший способ познакомиться с людьми, работающими в индустрии, — пойти туда, где они собираются. Если вам интересны настольные игры, отправляйтесь на конференцию Gen Con, интересующимся разработкой AAA-игр лучше отправиться на конференцию Game Developers Conference в Сан-Франциско, а если вы интересуетесь разработкой независимых игр, добро пожаловать на IndieCade. Другие конференции тоже в целом неплохи, но эти три пользуются особым вниманием разработчиков каждой из групп[2].

[1] Полный комплект слайдов из доклада доступен по адресу http://book.prototools.net.

[2] К числу известных игровых конференций относятся также E3 и PAX, но на них ниже вероятность встретиться с разработчиками.

Однако присутствие на конференции означает только, что вы будете находиться в одном месте с разработчиками игр. Чтобы познакомится с ними, вы должны найти повод подойти и поздороваться. Для этого хорошо подходят встречи после докладов или встречи с разработчиками во время работы в выставочных залах. Однако в любом случае вы должны быть вежливы, лаконичны и уважительны в отношении разработчиков и особенно других людей, которые также хотели бы поговорить с ними. Разработчики игр — люди занятые, и у каждого есть своя веская причина присутствовать на конференции. Они тоже хотят встречаться с людьми, расширять свои контакты и общаться с другими разработчиками. Поэтому не отнимайте у них слишком много времени, не заставляйте их чувствовать себя в ловушке, общаясь с вами, и всегда держите при себе интересное предложение или вопрос. У вас должна быть заготовлена тема для разговора, которая заинтересовала бы их.

Встречаясь с людьми впервые, избегайте подобострастного тона. Каждый дизайнер игр, от Уилла Райта до Дженовы Чена, — это обычные люди, и очень немногие из них хотели бы оказаться в роли идола. Избегайте таких слов, как: «Я обожаю вас! Я самый большой ваш поклонник!» Правда, это звучит чертовски глупо. Вместо этого лучше сказать что-нибудь вроде: «Мне понравилась игра *Journey*». Говоря это, вы хвалите игру — игру, над которой работало несколько человек, а не отдельного человека, о котором вы практически ничего не знаете.

Конечно, самая лучшая ситуация для знакомства — когда вас знакомят. Это дает вам и вашему новому знакомому почву для разговора (например, о вашем общем друге). Однако в этом случае вы несете ответственность перед другом, который представил вас, — ответственность не дать повода думать о нем плохо. Когда кто-то представляет вас, он ручается за вас, и если вы допустите какую-то неловкость или бестактность, это может плохо отразиться на вашем друге.

Кроме того, не старайтесь сосредоточиться на знакомствах именно с известными разработчиками. Все, кто посещает такие конференции, обожают игры, а студенты и волонтеры на них — самые увлеченные и творческие люди, и с ними вы тоже можете побеседовать. Плюс, кто знает, может быть, сегодняшний студент завтра станет великим дизайнером и окажется замечательным другом, который будет находить время, чтобы взглянуть на ваши игры.

Как подготовиться к игровой конференции

Если вы собираетесь знакомиться с людьми, заготовьте свои визитные карточки. Вы можете написать на лицевой стороне все что захотите, главное, чтобы текст был разборчивым. Я обычно рекомендую оставлять заднюю сторону пустой, чтобы человек, которому вы дадите карточку, мог сделать заметки, напоминающие о вашем разговоре.

Также я обычно беру с собой:

- **Мятные таблетки для свежести дыхания и зубочистки.** Я серьезно.

- **Карманные инструменты, например маленький многофункциональный инструмент фирмы Leatherman.** Быть человеком, способным исправить мелкие поломки, очень приятно.
- **Резюме.** Я больше не ношу их с собой, потому что вполне доволен своей текущей работой, но если вы ищете работу, полезно иметь при себе несколько копий.

Порядок дальнейших действий

Итак, вы познакомились с кем-то на конференции и взяли его визитку. Что дальше? Примерно через две недели напишите человеку электронное письмо. Две недели нужны по той простой причине, что участникам конференции по возвращении домой необходимо время, чтобы разобрать накопившуюся электронную почту. Желательно, чтобы ваше письмо следовало формату, представленному на рис. 15.2.

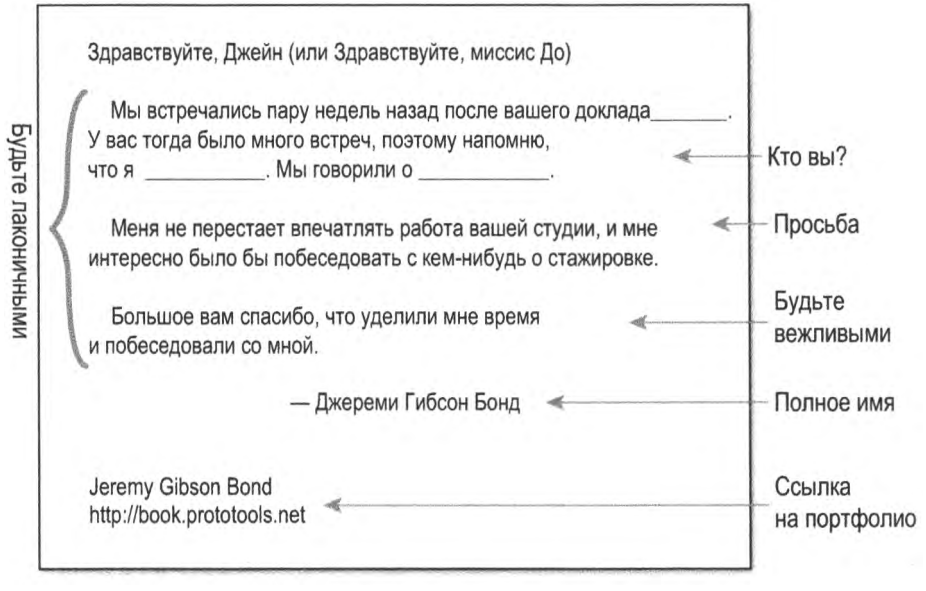


Рис. 15.2. Пример письма

Отправьте письмо и выждите еще пару недель. Если ответа не последовало, напишите еще одно, начав его примерно с таких слов: «Вероятно, вы были очень заняты после конференции, поэтому я решил написать вам еще раз, чтобы убедиться, что мое письмо дошло до вас». Если на второе письмо тоже не последовало ответа, не пишите больше. Вы встретите много новых людей в игровой индустрии, и не нужно никого беспокоить или подталкивать.

Собеседование

Если все пройдет хорошо, вас могут пригласить в студию на собеседование. Как подготовиться к нему?

Вопросы, которые необходимо задать до собеседования

На собеседовании вы будете разговаривать с людьми, которые действительно входят в состав команд, занимающихся разработкой. Но перед собеседованием вы, скорее всего, будете общаться со специалистом по подбору кадров. Его работа отчасти заключается в том, чтобы подготовить кандидата к собеседованию, и оценка его труда в конце года в какой-то мере будет зависеть от качества кандидатов, которых он подберет. То есть этот человек заинтересован в вашей хорошей подготовке и почти наверняка будет готов ответить на любые вопросы, которые помогут вам лучше подготовиться к собеседованию.

Вот какие вопросы вы могли бы задать этому специалисту:

- **Какая работа меня ожидает?** В ваших интересах получить как можно более конкретный ответ на этот вопрос, чтобы вы могли подготовиться.
- **В каком проекте я буду участвовать?** Это также поможет вам понять, кого ищет компания — специалиста на конкретное место или просто хороших людей, независимо от того, в каком проекте они будут работать.
- **Отличительные черты культуры в компании?** Каждая компания отличается своей культурой, особенно в игровой индустрии. Подобный вопрос может также привести к обсуждению таких аспектов, как сверхурочная работа и практика авралов. В действительности вам необязательно знать особенности культуры в компании, но о них обязательно нужно знать перед подписанием контракта.
- **Что было бы уместно надеть на собеседование?** Многие, очень многие упускают этот простой, но важный вопрос. Вообще я склонен одеваться на собеседование более формально, чем в обычные рабочие дни, но для большинства компаний это не значит носить костюм (и тем более галстук). Помните, вы собираетесь не на ужин, не на вечеринку, не на свидание и не на религиозную церемонию. Моя жена, профессиональный художник по костюмам и профессор, рекомендует: оденьтесь красиво, но неброско, чтобы основное внимание уделялось вашим навыкам и уму, а не внешнему виду.

  Еще одно важное обстоятельство, которое нужно учитывать: безусловно, вы должны одеваться так, чтобы чувствовать себя комфортно, но те, кто будет беседовать с вами, тоже должны чувствовать себя комфортно. Каждая студия так или иначе общается с инвесторами, с прессой, с издателями и другими людьми, склонными работать в более формальной культуре, чем игровая студия. Поэтому один из вопросов, который захотят в студии узнать о вас, — сможете ли вы участвовать в таких дискуссиях или вас придется прятать в задней комнате,

чтобы им не пришлось краснеть за вас перед гостями. Постарайтесь произвести впечатление, чтобы вас отнесли к первой категории.

В интернете ходит много разных мнений, как правильно одеваться на собеседование, поэтому самое лучшее, что можно сделать, — задать прямой вопрос специалисту по подбору кадров. Он встречается со всеми кандидатами и сможет подсказать, что подойдет вам, а что нет.

Вдобавок к одежде подумайте также о том, как выглядят ваши волосы (в том числе усов и бороды), они не должны выглядеть запущенными.

Какие игры других студий желательно попробовать перед собеседованием? Безусловно, вы должны попробовать игры, создаваемые студией, куда идете на собеседование, а если ваша цель — получить работу и принять участие в создании конкретной игры, не сыграть в нее или в предшествующие версии будет непростительным упущением. Этот вопрос поможет понять, в какие другие игры, по их мнению, вы должны сыграть, чтобы оставаться в курсе положения дел в жанре.

- **Можете ли вы сказать мне, кто будет проводить собеседование?** Зная это, можно провести небольшое исследование опыта этого человека. Знание других проектов, в которых работали те, кто будет вести собеседование перед приходом в эту студию, или знание других студий, где они работали прежде, может дать вам дополнительную информацию об их опыте и об обсуждаемых темах.

Есть также вопросы, которые вы ни в коем случае не должны задавать:

- **Какие игры создает студия? / Как давно существует студия?** Ответы на эти вопросы легко найти в интернете. Подобные вопросы показывают, что вы не провели свое исследование перед интервью (а значит, вам не особенно интересно это собеседование или работа).
- **Сколько я буду получать?** Конечно, это очень важный вопрос, но задавать его специалисту по подбору кадров или проводящему собеседование неуместно. Этот вопрос следует отложить до этапа переговоров после получения приглашения на работу. Для информации о средних показателях по индустрии можете заглянуть в «Обзор зарплат разработчиков игр» на сайте GameCareerGuide.com[1] или на таких сайтах, как glassdoor.com.

После собеседования

После собеседования отправьте рукописные благодарственные записки людям, с которыми вы поговорили особенно хорошо. Постарайтесь в течение дня делать заметки, чтобы потом упомянуть что-то особенное для каждого. «Большое спасибо,

[1] Исследование размера заработной платы традиционно печаталось каждый год в журнале *Game Developer Magazine*, имевшем тех же владельцев, что и сайт GameCareerGuide.com. Издание журнала прекратилось в 2013 году. Но вы все еще можете увидеть обзор заработной платы, опубликованный в 2013 году: http://gamecareerguide.com/features/1279/game_developer_salary_survey_.php.

что устроили мне экскурсию по студии и, особенно, что представили меня команде X», — выглядит намного лучше, чем: «Был рад встретиться с вами и поговорить о том о сем». Так же как находки в играх, рукописные письма ценны, потому что редки. Каждый месяц я получаю тысячи электронных писем, сотню отпечатанных писем по обычной почте и очень редко, реже одного раза в месяц, рукописные записки с благодарностями. Рукописные записки никогда не считались и не считаются спамом.

Не ждите, чтобы начать делать игры!

То, что вы не являетесь сотрудником игровой компании, еще не означает, что вы не можете создавать игры. Закончив читать эту книгу и получив некоторый опыт разработки прототипов, вы, возможно, начнете искать над чем поработать. На этот случай у меня есть для вас несколько советов.

Присоединитесь к проекту

Уверен, в вашей голове роятся десятки идей будущих игр, но на начальном этапе лучше присоединиться к команде с опытом разработки. Работа с командой других разработчиков — даже если они все еще учатся, как и вы, — это один из лучших способов быстро развить свои навыки.

Начните свой проект

Поработав в команде и получив некоторый опыт или не найдя команду, к которой можно присоединиться, попробуйте начать свой проект игры. Для этого нужны пять важных составляющих: правильная *идея*, правильная *оценка*, правильная *команда*, правильный *график* и *желание добиться результата*.

Правильная идея

В умах людей рождаются миллионы разных идей игр. Вы должны выбрать такую, которая будет воспринята положительно. Это должно быть что-то, к чему вы не потеряете интереса, что не просто копирует любимую игру, что другие сочтут интересным, и главное — что вы сможете сделать. Это ведет нас к...

Правильная оценка

Первое, что мешает командам завершить создание игры, — переоценка своих сил, то есть попытка откусить больше, чем можно проглотить. Многие начинающие разработчики плохо понимают, сколько времени может занять создание игры, поэтому они часто переоценивают свои силы и возможности. Оценка по нижней границе — это процесс оценки минимальной игры без излишеств. Чтобы верно оценить игру, вы должны четко понимать, сколько усилий нужно приложить для

ее реализации, и вы должны быть совершенно уверены, что у вас есть команда и достаточно времени для ее завершения.

Создать маленькую игру и потом расширять ее — намного лучше, чем сразу начать с попытки создать что-то огромное. Вы должны понимать, что большинство игр, в которые вы играли, создавались большими командами профессионалов в течение пары лет, при бюджете в несколько миллионов долларов. Даже разработка независимых игр часто занимает годы работы команды опытных и талантливых разработчиков. В начале пути подумайте о чем-нибудь небольшом. Позднее вы всегда сможете расширить игру.

Правильная команда

Работа над игрой с кем-то — это долгосрочные отношения, и вы должны воспринимать это именно так. Кроме того, как это ни печально, приходится признать, что многое, на чем держится ваша крепкая дружба с кем-то, часто оказывается совсем не тем, что необходимо, чтобы сделать вас членами сплоченной команды. Прежде чем приглашать кого-то к сотрудничеству, убедитесь, что они имеют привычки, схожие с вашими, а еще лучше, если их часы трудовой активности совпадают с вашими. Даже работая в команде удаленно, можно с успехом использовать текстовые и видеосообщения для общения с коллегами.

При создании команды также следует обсудить вопросы, связанные с интеллектуальной собственностью на игру. Если соглашение не заключено явно, по умолчанию все участники проекта имеют равную долю[1]. Проблемы с интеллектуальной собственностью часто трудно разрешимы, и может показаться, что преждевременно говорить о правах до появления игры, но это очень важная тема, и ее обязательно нужно обговорить. Кроме того, я действительно видел, как не удавалось создать команду из-за споров об интеллектуальной собственности на несуществующую игру. Вам определенно нужно постараться избежать этой ловушки.

БАЛЛЫ НАЧИСЛЕНИЙ

В своей компании я использую *балльную систему*, кажущуюся мне справедливым способом распределения доходов, заработанных от распространения независимых игр, которые мы создаем. Основная идея состоит в том, что каждый получает свое количество баллов за время, уделенное проекту на всем его протяжении. Вот как это работает в моих командах:

- 50 % от любых поступлений в проект идут непосредственно в компанию. Это помогает накапливать суммы для оплаты труда людей в будущем (прямо сейчас люди работают на свои будущие гонорары).

[1] Я не юрист и не даю юридических советов. Я лишь выражаю свое понимание ситуации. Если у вас есть знакомый юрист, расспросите его об этих тонкостях или поищите информацию в интернете.

- Другие 50 % распределяются между работниками на основе процентного соотношения от общего числа баллов.
- За каждые 10 часов работы в проекте люди получают по 1 баллу.
- Баллы начисляются и за время разработки, и за время поддержки проекта.
- Баллы суммируются в электронной таблице, доступной для просмотра всем членам команды (но править ее могу только я).

С такой системой баллов заработок членов команды напрямую зависит от их вклада в проект, и чем больше и успешнее они работают, тем больше баллов они получают. Если кто-то плохо работает, он исключается из команды, и хотя продолжает получать начисления на уже заработанные баллы, его доля становится все меньше и меньше по сравнению с другими членами команды, которые продолжают вносить свой вклад.

Это также означает, что члены команды поддержки могут заработать больше баллов, чем члены команды разработчиков; собственно, так и было задумано. Большинство небольших независимых проектов в прошлом присваивали определенное количество баллов каждому участнику в начале проекта, что сильно осложняло учет трудозатрат в будущем и адаптацию к изменениям рабочей ситуации. Я считаю, что система баллов обеспечивает определенную гибкость принятия решений в студии и при этом остается ясной и справедливой в отношении участников.

Правильный график

В главе 14 «Гибкое мышление» я рассказывал о методах гибкой разработки и графиках выполнения работ. Обязательно прочитайте ее перед началом проекта. Ваш опыт может отличаться от моего, тем не менее я заметил, что для моих команд и подавляющего большинства команд моих студентов графики выполнения работ оказались фантастическим инструментом, помогающим видеть, как продвигается работа каждого члена команды. Кроме того, графики выполнения работ оказывают неоценимую услугу, помогая понять разницу между прогнозируемым и фактическим временем выполнения заданий. Анализируя эту разницу от начала проекта до текущего момента, можно получить более реалистичный прогноз времени, необходимого на завершение оставшихся заданий.

Желание добиться результата

По мере продвижения проекта вы достигнете момента, когда четко увидите какие-то аспекты, которые могли бы реализовать лучше. Вы заметите беспорядок в коде и то, что художественное оформление могло бы быть лучше, а дизайн имеет досадные пробелы. Многие команды достигают этого момента на удивление близко к завершению проекта. Если вы близки к завершению, идите вперед. Главным стремлением для вас должно быть стремление завершить игру. Если убийца игр номер один — это плохая оценка, то убийца номер два — это последние 10 % проекта, которые всегда

оказываются самыми тяжелыми. Продолжайте идти вперед, потому что даже если игра не идеальна — и поверьте мне, идеальных игр не бывает, — даже если в ней не удалось воплотить все, что хотелось, даже если она получилась намного меньше, чем хотелось, ее обязательно нужно завершить. Вы станете разработчиком с завершенной игрой, а это много значит для всех, с кем вы надеетесь работать в будущем.

Итоги

Я мог бы продолжать и продолжать рассказывать об игровой индустрии, но, увы, ограничен рамками одной главы. К счастью, существует большое количество вебсайтов и публикаций, охватывающих игровую индустрию, и на конференциях часто рассказывается, как присоединиться к индустрии и создать свою компанию. Простой поиск в интернете поможет вам с ходу найти несколько таких публикаций, и среди них — сайт GDC Vault (http://gdcvault.com), где можно найти видеоматериалы и доклады по теме.

Если вы решите создать свою компанию, обязательно отыщите юриста и бухгалтера, которым вы сможете доверять, прежде чем столкнетесь с проблемами на этом пути. Юристы и бухгалтеры имеют многолетнюю подготовку в области создания и защиты компаний, и их доступность для консультаций сделает ваш путь в индустрию намного проще.

ЧАСТЬ II

Цифровое прототипирование

Глава 16. Цифровое мышление
Глава 17. Введение в среду разработки Unity
Глава 18. Знакомство с нашим языком: C#
Глава 19. Hello World: ваша первая программа
Глава 20. Переменные и компоненты
Глава 21. Логические операции и условия
Глава 22. Циклы
Глава 23. Коллекции в C#
Глава 24. Функции и параметры
Глава 25. Отладка
Глава 26. Классы
Глава 27. Объектно-ориентированное мышление

16 Цифровое мышление

Если прежде вы никогда не программировали, эта глава послужит вам введением в новый мир: в мир, полный новых возможностей в создании цифровых прототипов игр, какие только можно себе представить.

Эта глава описывает образ мыслей, который вы должны воспитать в себе, приступая к созданию программных объектов. Она даст вам упражнения для развития такого мышления и научит воспринимать мир с точки зрения систем отношений, наполненных смыслом. К концу главы вы приобретете правильный настрой для освоения части «Цифровое прототипирование» этой книги.

Системный подход к настольным играм

В первой части книги вы узнали, что игры создаются из взаимосвязанных систем. В играх эти системы кодируются в правилах игры и в самих игроках — в том смысле, что все игроки привносят в игру свои ожидания, способности, знания и социальные нормы. Например, увидев пару стандартных шестигранных кубиков в коробке с новой настольной игрой, вы сразу же готовы предположить, как они будут использоваться в игре:

- **Распространенные предположения об использовании двух шестигранных кубиков в настольных играх.**
  1. Каждый кубик используется, чтобы сгенерировать случайное число в диапазоне от 1 до 6 (включительно).
  2. Кубики часто бросаются вместе, особенно если они одного размера и цвета.
  3. Когда кубики бросаются вместе, выпавшие на них очки обычно суммируются. Например, 3 на одном кубике и 4 на другом складываются и дают в сумме 7.
  4. Иногда, когда выпадают «дубли» (то есть на обоих кубиках выпадает одинаковое число очков), игрок получает какую-то дополнительную выгоду.

Также вы наверняка предположите, чего нельзя сделать с помощью кубиков:

- **Распространенные предположения об ограничениях на использование кубиков в настольных играх.**
  1. Игрок не может просто повернуть кубики нужными ему гранями вверх.
  2. Чтобы бросок был признан действительным, кубики должны остаться после броска на столе и успокоиться на плоских гранях.
  3. Бросок кубиков обычно не влияет на остальную часть хода игрока.
  4. Вообще кубики не принято бросать в других игроков (или есть их).

Подробное исследование таких простых, часто неписаных, правил может показаться излишне педантичным, но оно помогает показать, как много правил в настольных играх фактически отсутствует в инструкциях к ним; часто они основаны на общепринятом понимании *честной игры*. Эта идея заложена в понятие волшебного круга и в значительной мере помогает группам детей так легко создавать спонтанные игры, понятные всем участникам. Большинство игроков привносит в игры массу предубеждений о том, как правильно играть в них.

В отличие от настольных, компьютерные игры опираются на конкретные инструкции абсолютно во всем. По сути, независимо от того, насколько мощными они стали за последние десятилетия, компьютеры являются бездушными машинами, четко выполняющими конкретные инструкции миллиарды раз в секунду. Наделение компьютера некоторым подобием интеллекта путем воплощения идей в очень простые инструкции — это уже ваша задача как программиста.

Упражнение в простых инструкциях

Чтобы помочь начинающим студентам-информатикам понять, что значит мыслить в простых инструкциях, часто используется несложное упражнение, заключающееся в том, чтобы описать другому человеку, как встать из положения лежа и принять вертикальное положение. Для этого упражнения вам понадобится друг.

Попросите друга лечь на пол на спину и в точности следовать вашим инструкциям. Ваша задача: отдать подробные команды, выполнив которые тот сможет встать на ноги; при этом нельзя использовать сложные команды, такие как «встать». Команды должны быть очень простыми: представьте, что вы отдаете их роботу. Вот пример подобных команд:

- Согнуть левый локоть под углом, близким к 90 градусам.
- Вытянуть правую ногу по направлению к двери.
- Положить левую руку на пол ладонью вниз.
- Вытянуть правую руку в направлении телевизора.

На самом деле даже эти простые команды намного сложнее, чем те, что можно отдать роботу, и могут интерпретироваться в очень широких пределах. Однако для нашего упражнения этого уровня вполне достаточно.

Попробуйте.

Сколько времени вам понадобилось, чтобы дать другу правильные инструкции и поставить его на ноги? Если вы попытаетесь в точности следовать духу и букве упражнения, вам понадобится очень много времени. Если вы попробуете проделать то же самое с другими людьми, вам потребуется еще больше времени, намного больше даже, чем в случае, если бы ваш друг не знал вашей конечной цели — поставить его вертикально.

Сколько вам было лет, когда домашние впервые попросили вас накрыть на стол? Мне было около четырех лет, когда мои родители решили, что я справлюсь с этой сложной задачей, получив единственную инструкцию: «Пожалуйста, накрой стол к ужину». А теперь, опираясь на опыт, полученный в упражнении, представьте, сколько простых инструкций нужно дать человеку, чтобы тот справился с этой сложной задачей и накрыл на стол; и все же дети часто справляются с ней задолго до того, как пойдут в школу.

Что это значит для цифрового программирования

Разумеется, это упражнение я дал не для того, чтобы отговорить вас. На самом деле следующие две главы призваны вас воодушевить! Я дал это упражнение, чтобы помочь вам понять, как думают компьютеры, и подготовить несколько метафор об аспектах компьютерного программирования. Рассмотрим их.

Компьютерный язык

Когда я дал вам список из четырех примеров команд, я обозначил параметры языка, который вы могли бы использовать для передачи команд другу. Понятно, что это было очень вольное определение языка. На протяжении всей этой книги мы будем использовать язык программирования C# (произносится как «си шарп»), который, к счастью, имеет намного более строгое определение. Вы будете изучать C# на протяжении всех последующих глав в этой части книги, и я хотел бы заметить, что, основываясь на моем личном опыте и более чем десятилетнем опыте преподавания разных языков программирования тысячам студентов, язык C# является одним из лучших для людей, изучающих свой первый язык программирования. Несмотря на то что для изучения C# требуется больше усердия, чем для изучения более простых языков, таких как Processing или JavaScript, он дает обучающимся гораздо лучшее понимание основных идей разработки, знание которых будет помогать им в прототипировании игр и в карьере разработчика, и прививает правильные навыки программирования, которые в конечном итоге упростят и ускорят разработку кода.

Библиотеки кода

В предыдущем упражнении вы могли заметить, что было бы намного проще иметь возможность дать команду «встать» вместо перечисления множества элементарных команд. В данном случае «встать» — это универсальная высокоуровневая инструкция, которую вы могли бы дать своему другу, независимо от его начального положения. Аналогично, «накрой на стол» — типичная высокоуровневая инструкция, которая генерирует желаемый результат, независимо от того, какие блюда готовятся, сколько людей будет за столом и даже от того, в какой семье это происходит. В C# коллекции высокоуровневых инструкций, описывающих типичные варианты поведения, называют *библиотеками кода*, и вам, как разработчику на C# и Unity, доступны сотни таких библиотек.

Чаще всего в своей практике вы будете использовать библиотеку, которая адаптирует C# для работы в среде разработки Unity. В ваш код эта чрезвычайно мощная библиотека будет импортироваться под именем `UnityEngine`. Библиотека `UnityEngine` включает код, обеспечивающий:

- потрясающие эффекты освещения, такие как туман и зеркальное отражение;
- имитацию действия гравитации, моделирование упругих столкновений и даже свободно развевающейся ткани;
- ввод с мыши, клавиатуры, игрового пульта и сенсорного экрана;

и многое, многое другое.

Кроме того, имеются тысячи бесплатных (и платных) библиотек, упрощающих программирование. Если вы собираетесь реализовать что-то очень распространенное (например, плавное перемещение объекта по экрану в течение одной секунды), вполне возможно, что кто-то уже написал превосходную библиотеку для этого (для данного примера есть библиотека iTween Боба Беркбайла (Bob Berkebile), http://itween.pixelplacement.com/).

Доступность отличных библиотек кода для Unity и C# означает, что *вы* сможете сосредоточиться на реализации новых, уникальных аспектов вашей игры, не тратя времени на изобретение колеса в каждом новом проекте. Со временем вы начнете собирать свои библиотеки кода и использовать их в разных проектах. В этой книге мы так и поступим, начав собирать свою библиотеку под названием `ProtoTools`, которая будет расти от проекта к проекту.

Среда разработки

Среда разработки игр Unity — это ваш главный инструмент. Правильнее рассматривать приложение Unity как среду, в которой можно собирать и конструировать ресурсы, из которых состоят игры. В Unity вы будете объединять трехмерные модели, музыку и аудиоклипы, двумерную графику и текстуры и, наконец, сценарии на C#, которые будете писать сами. Ни один из этих ресурсов не создается непосредственно в Unity; Unity — это место, где они собираются воедино, в полноценную

компьютерную игру. Вы также будете использовать Unity для размещения игровых объектов в трехмерном пространстве, обработки ввода пользователя, настройки виртуальной камеры, обозревающей сцену, и, наконец, для компоновки всех этих ресурсов в действующую игру. В главе 17 «Введение в среду разработки Unity» мы подробно обсудим возможности Unity.

Деление сложных задач на простые подзадачи

Одно из важнейших условий, которое вы должны были заметить в упражнении выше, состоит в исключении сложных команд, таких как «встать». Это должно было заставить вас задуматься о разделении сложных команд на более простые, дискретные команды. Это было непросто, но, начав программировать, вы заметите, что навык деления сложных задач на более простые является одним из самых мощных инструментов, помогающих преодолевать сложности и создавать игры небольшими фрагментами. Я пользуюсь этим навыком каждый день, разрабатывая свои игры, и обещаю, что вам он также сослужит хорошую службу. В качестве примера разложим на простые команды игру Apple Picker, которую мы напишем в главе 28 «Прототип 1: Apple Picker».

Анализ игры: Apple Picker

Apple Picker — это первый прототип, который мы создадим в этой книге. Он основан на идее классической игры *Kaboom!*, которую создал Ларри Каплан и выпустила компания Activision в 1981 году[1]. На протяжении многих лет создано множество клонов игры *Kaboom!*, и наша версия является наименее жестокой из них. В оригинальной версии игрок перемещает емкости влево и вправо по экрану, пытаясь поймать бомбы, сбрасываемые «Безумным бомбардировщиком». В нашей версии игрок будет ловить яблоки, падающие с дерева (рис. 16.1).

В этом анализе мы рассмотрим каждый из игровых объектов `GameObject`[2] в Apple Picker, проанализируем их поведение и разложим поведение каждого на простые команды в форме блок-схем. Так мы увидим, как из простых команд можно конструировать сложное поведение и забавное течение игры.

Я рекомендую поискать в интернете по фразе «play Kaboom!» (играть в Kaboom!), чтобы найти онлайн-версию игры и поиграть в нее, прежде чем продолжить исследовать ее. Впрочем, игра достаточно простая, поэтому можно этого не делать.

---

[1] https://ru.wikipedia.org/wiki/Kaboom. — *Примеч. пер.*

[2] Под именем `GameObject` в Unity подразумеваются активные игровые объекты. Каждый игровой объект `GameObject` может содержать компоненты, такие как трехмерные модели, текстуры, данные для расчетов при столкновениях, код на C# и т. д.

Кроме того, готовый прототип игры Apple Picker можно найти на сайте книги http://book.prototools.net в разделе Chapter 16, но имейте в виду, что Apple Picker состоит из одного бесконечного уровня, тогда как *Kaboom!* имеет восемь разных уровней сложности.

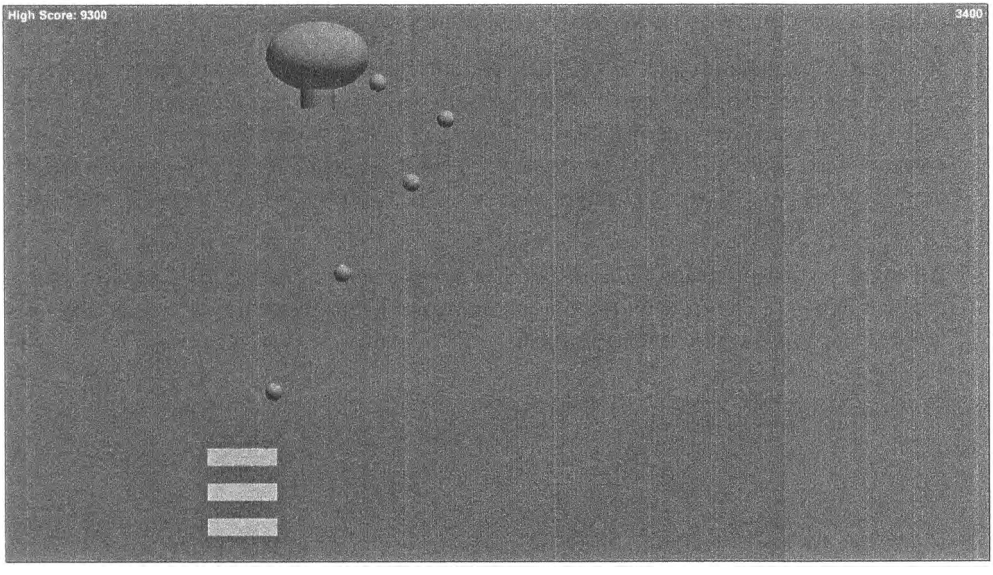


Рис. 16.1. Игра Apple Picker, которую мы создадим в главе 28

Сценарий игры Apple Picker

Игрок управляет тремя корзинами в нижней части экрана и может перемещать их мышью влево и вправо. Яблоня в случайном порядке раскачивается взад-вперед и сбрасывает яблоки, и игрок должен ловить их в корзины, не давая упасть на землю. За каждое пойманное яблоко игроку начисляются очки, но если хотя бы одно яблоко упадет на землю, все другие яблоки исчезают, и вместе с ними исчезает одна корзина. Когда игрок теряет все три корзины, игра завершается. (В оригинальной игре *Kaboom!* действуют немного другие правила в отношении количества очков за каждую пойманную бомбу (яблоко), и игрок перемещается с уровня на уровень, но это неважно для нашего анализа.)

Игровые объекты GameObject в Apple Picker

В терминологии Unity любой объект, участвующий в игре и изображающий что-то на экране, называется *игровым объектом*, или `GameObject`. Мы можем использовать этот термин в обсуждении элементов, изображенных на скриншоте (рис. 16.2). Для

единообразия с более поздними проектами Unity я буду давать игровым объектам имена, начинающиеся с заглавных букв (например, Яблоко, Корзина и Яблоня), как в следующем списке.

A. **Корзины:** управляются игроком. Корзины могут двигаться влево и вправо, следуя за движениями мыши. Когда Корзина сталкивается с Яблоком, Яблоко считается пойманным, и игроку начисляются очки.

B. **Яблоки:** сбрасываются с Яблони и падают вниз. Если Яблоко сталкивается с одной из трех Корзин, оно считается пойманным и исчезает с экрана (а игрок получает несколько очков). Если Яблоко достигает нижней границы экрана, оно исчезает, и вместе с ним исчезают все Яблоки, имеющиеся на экране. Также исчезает одна Корзина (самая верхняя). После этого Яблоня вновь начинает сбрасывать Яблоки.

C. **Яблоня:** раскачивается влево и вправо и сбрасывает Яблоки. Яблоки сбрасываются через регулярные интервалы, поэтому единственный элемент случайности — это движение влево-вправо.

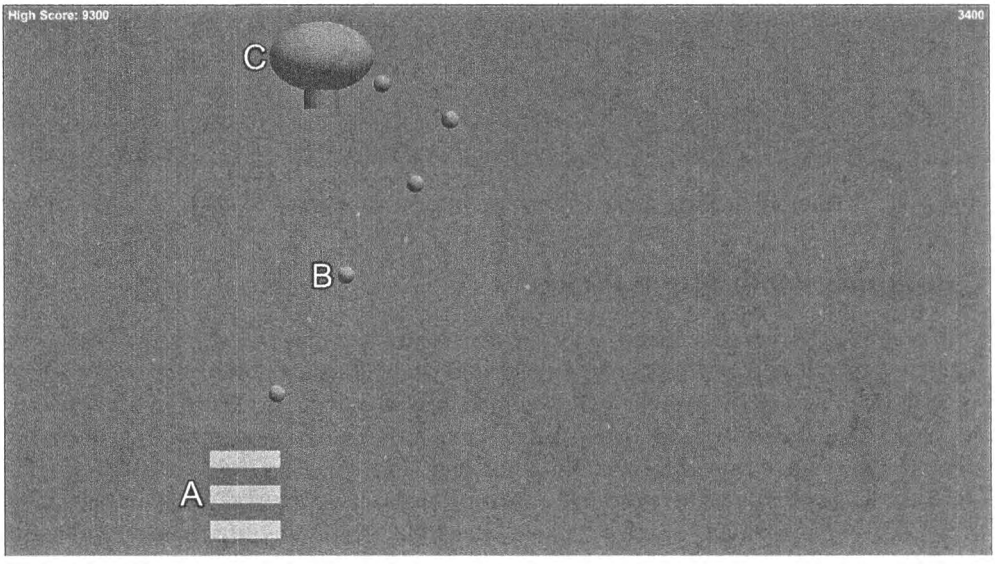


Рис. 16.2. Apple Picker с игровыми объектами

Список действий игровых объектов в Apple Picker

В этом анализе не рассматривается возрастающая сложность уровней в оригинальной игре *Kaboom!*. Вместо этого мы сосредоточимся на действиях, выполняемых каждым игровым объектом.

Действия Корзины

Корзина выполняет следующие действия:

- Перемещается влево и вправо, следуя за движениями мыши.
- Если Корзина сталкивается с Яблоком, это Яблоко считается пойманным[1].

Вот и все! Корзина — очень простой игровой объект.

Действия Яблока

Яблоко выполняет следующие действия:

- Падает вниз.
- Если достигает нижнего края экрана, завершает текущий раунд игры[2].

Яблоки — тоже очень простые игровые объекты.

Действия Яблони

Яблоня выполняет следующие действия:

- Раскачивается по экрану влево и вправо.
- Сбрасывает Яблоки через каждые 0,5 секунды.

И Яблоня — очень простой игровой объект.

Блок-схемы игровых объектов Apple Picker

Блок-схемы часто дают отличную возможность изобразить поток действий и решений в игре. Давайте рассмотрим несколько блок-схем для игры Apple Picker. В следующих блок-схемах упоминаются такие действия, как добавление очков и завершение раунда, но прямо сейчас просто посмотрите, какие действия выполняются в одном раунде, и не задумывайтесь о том, как в действительности происходит начисление очков или завершение раунда.

Блок-схема Корзины

На рис. 16.3 изображена блок-схема, описывающая поведение Корзины. Операции на этой блок-схеме игра выполняет в каждом *кадре* (по меньшей мере 30 раз в се-

[1] Проверка столкновения и реакция на него также может выполняться Яблоком, но я решил присвоить это действие Корзине.

[2] В конце раунда с экрана исчезают все Яблоки, а также удаляется одна из Корзин, после чего начинается следующий раунд, но это действие необязательно должно выполняться Яблоком. В действительности оно будет осуществляться сценарием ApplePicker, управляющим всеми элементами игры.

кунду). Этот факт отмечен в овале, в верхней части блок-схемы. Прямоугольники обозначают действия (например, *Перемещение влево/вправо за движениями мыши*), а ромбы — решения. Что такое кадр, вы узнаете во врезке «Кадры в компьютерных играх».

Рис. 16.3. Блок-схема Корзины

КАДРЫ В КОМПЬЮТЕРНЫХ ИГРАХ

Термин *кадр* пришел из мира кино. Исторически фильмы выпускались в виде длинных целлулоидных лент, содержащих тысячи отдельных картинок (которые называют кадрами). Когда эти картинки отображаются в быстрой последовательности (со скоростью 16 или 24 кадра в секунду), создается иллюзия движения. Позднее, когда появилось телевидение, движение конструировалось быстрой сменой электронных изображений, проецируемых на экран, которые также называются кадрами (в США была принята скорость 30 кадров в секунду).

Когда компьютеры стали достаточно мощными, чтобы отображать анимацию и другие движущиеся изображения, каждое отдельное изображение, отображаемое на экране компьютера, тоже стали называть кадром. Кроме того, все вычисления, необходимые для создания изображения на экране, тоже являются частью кадра. Когда среда Unity выполняет игру со скоростью 60 кадров в секунду, она не только создает и отображает новое изображение 60 раз в секунду. В это время она

также производит огромный объем вычислений, необходимых для правильного перемещения объектов между кадрами.

На рис. 16.3 изображена блок-схема, представляющая вычисления, которые связаны с перемещением Корзины между кадрами.

Блок-схема Яблока

Яблоко тоже имеет очень простую блок-схему (рис. 16.4). Не забывайте, что проверка и обработка столкновения Яблока с Корзиной является частью поведения Корзины, поэтому соответствующие операции отсутствуют в блок-схеме Яблока.

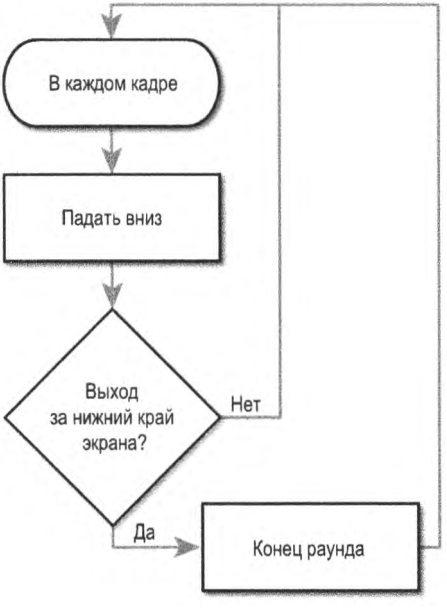


Рис. 16.4. Блок-схема Яблока

Блок-схема Яблони

Блок-схема Яблони немного сложнее (рис. 16.5), потому что в каждом кадре Яблоня принимает два решения:

- Изменить направление движения?
- Сбросить Яблоко?

Решение о смене направления движения можно принимать до или после фактического перемещения. Для наших целей подойдет любой вариант.

Рис. 16.5. Блок-схема Яблони

Итоги

Как видите, цифровые игры можно разбить на множество очень простых решений и команд. Эта задача вытекает из моего подхода к созданию прототипов для этой книги, и вы сами будете решать ее, когда начнете проектировать и разрабатывать свои игровые проекты.

В главе 28 «Прототип 1: Apple Picker» будет представлен расширенный вариант этого анализа и там же будет показано, как превратить все эти списки действий в строки кода, заставляющие Корзины двигаться вслед за мышью, Яблоки — падать, а Яблоню в случайном порядке перемещаться по экрану и сбрасывать Яблоки, подобно «Безумному бомбардировщику».

17 Введение в среду разработки Unity

Здесь начинается ваше приключение в мире программирования.

В этой главе вы загрузите Unity, среду разработки игр, которую будете использовать на протяжении оставшейся части книги. Глава рассказывает, почему Unity считается фантастическим инструментом создания игр для любого начинающего дизайнера или разработчика игр и почему я выбрал C# в качестве языка для обучения.

Вы также увидите пример проекта, входящего в состав дистрибутива Unity, познакомитесь с разными панелями в интерфейсе Unity и разместите эти панели в том логическом порядке, в каком они демонстрируются в оставшейся части книги.

Загрузка Unity

Прежде всего запустим загрузку Unity. Размер устанавливаемого дистрибутива превышает 1 Гбайт, поэтому, в зависимости от скорости вашего подключения к интернету, процесс загрузки может занять от нескольких минут до пары часов. Запустив загрузку, мы сможем перейти к разговору о Unity.

На момент написания этих строк самой свежей версией Unity была Unity 2017. Согласно текущему плану выпуска Unity, новые версии должны выходить каждые 90 дней. Но, независимо от версии, дистрибутив Unity всегда доступен для бесплатной загрузки в магазине Unity:

http://store.unity.com

После ввода этого адреса в браузере перед вами должна открыться страница со ссылками на разные версии Unity (рис. 17.1). Есть версии Unity для Windows и для macOS, обе они почти идентичны. В бесплатной версии Personal вы сможете повторить почти все примеры, что приводятся в этой книге. Щелкните на зеленой кнопке Try Personal (Попробуйте Personal) в колонке с заголовком *Personal*, чтобы запустить процесс. Затем щелкните на кнопке Download Installer (Загрузить установщик) на странице, которая появится вслед. Команда Unity относительно

часто меняет интерфейс страницы своего магазина, но сам процесс остается почти неизменным.

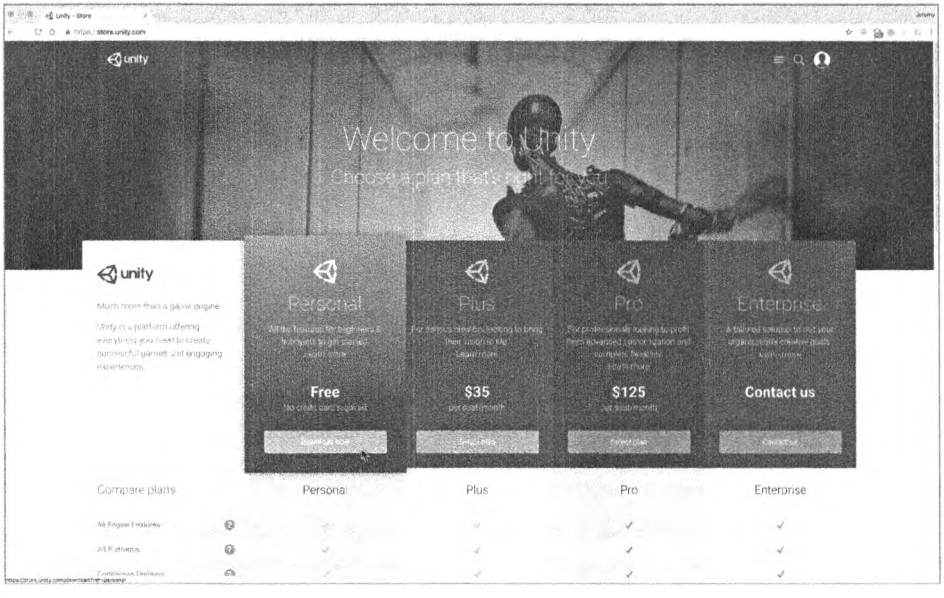


Рис. 17.1. Веб-страница для загрузки Unity

После этого ваш компьютер приступит к загрузке *Unity Download Assistant*, маленькой программы (с размером меньше 1 Мбайт), которая загрузит остальную часть дистрибутива Unity, когда вы запустите ее. Программу Unity Download Assistant вы найдете в своей папке Downloads (*Загрузки*).

В macOS

Чтобы установить Unity в macOS, выполните следующие шаги:

1. Откройте загруженный файл UnityDownloadAssistant-x.x.x.dmg (где x.x.x обозначают номер версии Unity). В результате откроется папка.
2. Щелкните дважды на *Unity Download Assistant.app* внутри папки, чтобы запустить приложение (рис. 17.2A).
3. macOS спросит, уверены ли вы в своем желании запустить это приложение, потому что оно загружено из интернета. Щелкните на кнопке Open (Открыть), чтобы подтвердить свое желание (рис. 17.2B).
4. На появившейся странице Install, activate and get creating with Unity (Установить, активировать и приступить к работе с Unity) щелкните на кнопке Continue (Продолжить).

5. Чтобы установить Unity, вы должны согласиться с условиями предоставления услуг, щелкнув на кнопке **Agree** (Согласен).
6. На странице, изображенной на рис. 17.2C, установите следующие флажки:
   - **Unity x.x.x** — текущая версия Unity.
   - **Documentation** (Документация) — уверяю вас, это совершенно необходимо!
   - **Standard Assets** (Стандартные ресурсы) — набор полезных ресурсов, распространяемых в составе Unity. Включает несколько красивых эффектов частиц, элементы ландшафта и т. п.
   - **Example Project** (Пример проекта) — мы исследуем его далее в этой главе.
   - **WebGL Build Support** (Поддержка сборки с WebGL) — в настоящее время это единственная возможность создавать в Unity онлайн-игры, и мы будем использовать этот инструмент далее в книге.

   Вам также может понадобиться ввести пароль своей учетной записи в macOS или Windows для установки.
7. Программа Download Assistant спросит, куда установить Unity. Я рекомендую устанавливать на главный жесткий диск, чтобы обеспечить быстрый доступ к Unity. Щелкните на кнопке **Continue** (Продолжить).

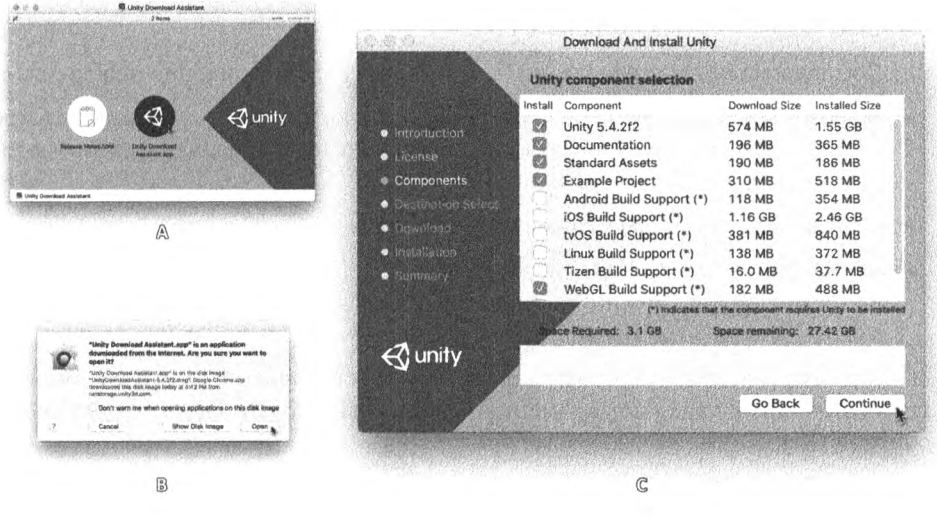


Рис 17.2. Этапы установки в macOS

Download Assistant запустит продолжительный процесс загрузки. С параметрами, которые я рекомендовал, у меня было загружено около 3 Гбайт, поэтому вам придется подождать.

В Windows

Чтобы установить Unity в Windows, выполните следующие шаги:

1. Откройте загруженное приложение UnityDownloadAssistant-x.x.x.exe (где x.x.x обозначают номер версии Unity).
2. Windows спросит, доверяете ли вы приложению выполнить изменения на вашем PC. Щелкните на кнопке Yes (Да), чтобы подтвердить свое желание (рис. 17.3A).
3. На первой странице мастера установки щелкните на кнопке Next > (Далее >).
4. Чтобы установить Unity, вы должны установить флажок I accept the terms of the License Agreement (Я принимаю условия лицензионного соглашения) и щелкнуть на кнопке Next > (Далее >).
5. Если вы пользуетесь 64-разрядной версией Windows, выберите вариант 64-bit (64-разрядная) на следующей странице. Но если у вас 32-разрядная версия Windows, выберите 32-bit (32-разрядная). Для верности откройте диалог Windows Settings (Параметры Windows) и щелкните на ярлыке System (Система, с изображением компьютера). Затем щелкните на пункте About (О системе) в списке слева. Рядом с заголовком System type (Тип системы) вы увидите 64-bit (64-разрядная) или 32-bit (32-разрядная) (рис. 17.3B). После выбора версии щелкните на кнопке Next > (Далее >).
6. На странице, изображенной на рис. 17.3C, установите следующие флажки:[1]
   - Unity x.x.x — текущая версия Unity.
   - Documentation (Документация) — уверяю вас, это совершенно необходимо!
   - Standard Assets (Стандартные ресурсы) — набор полезных ресурсов, распространяемых в составе Unity. Включает несколько красивых эффектов частиц, элементы ландшафта и т. п.
   - Example Project (Пример проекта) — мы исследуем его далее в этой главе.
   - WebGL Build Support (Поддержка сборки с WebGL) — в настоящее время это единственная возможность создавать в Unity онлайн-игры, и мы будем использовать этот инструмент далее в книге.
7. Программа Download Assistant спросит, куда установить Unity. Я рекомендую устанавливать в папку по умолчанию C:\Program Files\Unity. Щелкните на кнопке Next > (Далее >).

[1] У вас может возникнуть соблазн установить заодно Microsoft Visual Studio Community, но я не рекомендую делать это сейчас. Visual Studio — более мощный редактор кода, чем MonoDevelop (входящий в состав Unity), и в настоящее время есть возможность установить его вместе с Unity. Но я не советую делать это, потому что в книге демонстрируются примеры использования MonoDevelop. Но если вы уже имеете опыт работы в Visual Studio, тогда, может быть, вам стоит попробовать.

Download Assistant запустит продолжительный процесс загрузки. С параметрами, которые я рекомендовал, объем загрузки составит около 3 Гбайт, поэтому вам придется подождать. А пока вы ждете, прочитайте следующий раздел «Введение в нашу среду разработки».

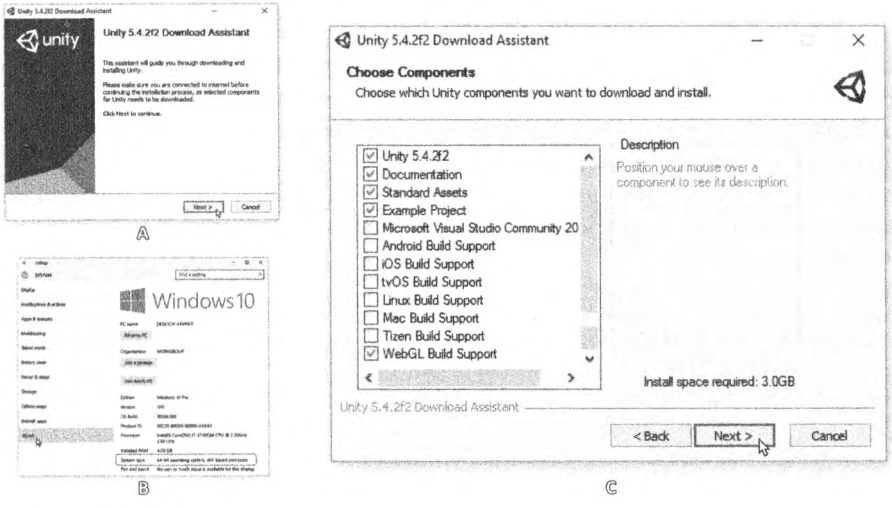


Рис. 17.3. Этапы установки в Windows

Введение в нашу среду разработки

Прежде чем приступать к прототипированию всерьез, нужно познакомиться с Unity, выбранной нами средой разработки. Unity можно считать программой-компоновщиком; в ней вы будете объединять элементы своих прототипов игр, но фактические ресурсы обычно создаются в других программах. Программировать вы будете в MonoDevelop; модели и текстуры — создавать в программах трехмерного моделирования, таких как Maya, Autodesk 3ds Max или Blender; править изображения — в фоторедакторе, таком как Affinity Photo, Photoshop или GIMP; а редактировать звуки — в аудиоредакторах, таких как Pro Tools или Audacity. Большую часть времени при исследовании учебных примеров в книге вы будете проводить за написанием кода на C# (произносится как «си шарп») в MonoDevelop, а управлять своими проектами и сценами — в Unity. Поскольку среда Unity играет важную роль в нашем процессе, внимательно читайте и тщательно выполняйте инструкции в разделе «Настройка размещения окон Unity» далее.

Почему выбор пал на Unity?

В мире есть много движков для разработки игр, но я выбрал Unity по следующим причинам.

- **Есть бесплатная версия Unity:** используя бесплатную версию Unity Personal, вы сможете создавать и продавать игры, работающие на разных платформах. На момент написания этого издания книги версии Unity Plus и Unity Pro имели не так много особенностей, отсутствующих в Unity Personal. Единственный нюанс: если вы работаете в компании или организации, заработавшей больше 100 000 долларов за последний год, вы должны приобрести Unity Plus (35 долларов в месяц), а если за последний год ваша организация заработала больше 200 000 долларов, вы должны приобрести Unity Pro (125 долларов в месяц). Версии Plus и Pro включают дополнительные аналитические инструменты, возможность изменить экран-заставку, отображаемую во время запуска игры, поддержку большего количества игроков в многопользовательских играх и темную тему оформления для редактора — вот, собственно, и все. Для дизайнера игр, обучающегося созданию прототипов, бесплатной версии более чем достаточно.

СТОИМОСТЬ UNITY. Порядок лицензирования и ценообразования Unity несколько изменился с момента выхода первого издания книги, поэтому я советую изучить эти аспекты на сайте http://store.unity.com.

- **Напиши один раз, применяй где угодно:** бесплатная версия Unity позволяет собирать приложения для macOS, PC, интернета (с использованием WebGL), Linux, iOS, Apple tvOS, Android, Samsung TV, Tizen, Windows Store и многих других платформ — из одного и того же кода и файлов. Такая гибкость составляет основу Unity; фактически именно поэтому компания и ее продукт получили такое название[1]. Профессионалы могут создавать в Unity игры даже для PlayStation 4, Xbox One и некоторых других игровых приставок.
- **Великолепная поддержка:** кроме превосходной документации, Unity поддерживает невероятно активное и доброжелательное сообщество разработчиков. Миллионы разработчиков используют Unity, и многие из них участвуют в дискуссиях на интернет-форумах Unity. Официальный форум Unity находится по адресу https://forum.unity3d.com/.
- **Она превосходна!** Мои студенты и я часто шутим, что в Unity есть кнопка «сделать круто». Даже при том, что это не совсем так, в Unity есть несколько феноменальных особенностей, способных улучшить игровой процесс и внешний вид игры простой установкой флажков в настройках. Инженеры Unity уже решили за вас множество сложнейших задач, связанных с программированием игр. Определение столкновений, имитация действия законов физики, поиск путей, системы частиц, пакетная обработка операций рисования, шей-

[1] Unity переводится как «единство». — *Примеч. пер.*

деры, игровой цикл и многие другие сложнейшие задачи программирования уже решены. Вам остается только написать игру, которая использует все эти преимущества!

Почему выбор пал на C#?

В Unity у вас на выбор есть два языка программирования: JavaScript и C#.

JavaScript

JavaScript часто считают языком для начинающих: он прост в изучении, имеет гибкий синтаксис, прощающий оплошности, а кроме того, он используется при создании сценариев для веб-страниц. Первоначально JavaScript был создан компанией Netscape в середине 1990-х как «облегченная» версия языка программирования Java. Он использовался как язык сценариев для веб-страниц, хотя часто разные функции JavaScript хорошо работали в одном веб-браузере и не работали вовсе — в другом. Синтаксис JavaScript стал основой для HTML5 и очень похож на синтаксис Adobe Flash ActionScript 3. Несмотря на все это, гибкость и всепрощающая природа JavaScript делают его малопригодным для этой книги. Например, JavaScript использует *слабую типизацию*, то есть, создав переменную (или контейнер) с именем `bob`, вы сможете сохранить в ней все что угодно: число, слово, целую повесть и даже главного персонажа вашей игры. Так как переменная `bob` в JavaScript не имеет типа, Unity никогда не сможет точно знать, что в ней действительно хранится, а кроме того, содержимое этой переменной может измениться в любой момент. Такая гибкость JavaScript делает написание сценариев более утомительным занятием и не позволяет программистам пользоваться некоторыми мощными и интересными возможностями современных языков.

C#

Язык C# был создан компанией Microsoft в 2000-м в ответ на появление Java. Он заимствовал множество современных возможностей из Java и облек их в синтаксис, более знакомый и удобный для программистов на традиционном C++. Это означает, что C# обладает всеми особенностями, свойственными современным языкам программирования. Для опытных программистов отмечу, что к числу этих особенностей среди прочих относятся: поддержка виртуальных функций и делегатов, динамическое связывание, перегрузка операторов, лямбда-выражения и мощный язык запросов LINQ. Начинающим изучать программирование достаточно знать, что использование C# с самого начала сделает вас лучшим программистом и разработчиком прототипов. Преподавая прототипирование в Южно-Калифорнийском университете, я обучал своих студентов программированию на двух языках — JavaScript и C# и заметил, что те, кто писал на C#,

неизменно создавали более удачные прототипы игр благодаря более строгим правилам программирования и чувствовали себя увереннее, чем их сокурсники, прежде изучавшие JavaScript.

СКОРОСТЬ ВЫПОЛНЕНИЯ ПРОГРАММ НА КАЖДОМ ЯЗЫКЕ

Имеющие опыт программирования могут предположить, что код на C# в Unity будет выполняться быстрее, чем код на JavaScript. Это предположение основано на том факте, что код на C# обычно *компилируется*, тогда как код на JavaScript — *интерпретируется* (в том смысле, что во время компиляции исходный код преобразуется компилятором в машинный код до запуска программы, а интерпретируемый код транслируется на лету, в процессе игры, из-за чего интерпретируемый код выполняется медленнее; более подробно этот вопрос обсуждается в главе 18 «Знакомство с нашим языком: C#»). Однако в Unity все обстоит иначе: всякий раз, когда вы сохраняете файл с исходным кодом на C# или JavaScript, Unity импортирует его, преобразует в код на одном и том же *обобщенном промежуточном языке* (Common Intermediate Language, CIL) и затем компилирует код на CIL в код на машинном языке. То есть независимо от выбранного языка, ваши прототипы игр в Unity будут выполняться с одинаковой скоростью.

О сложной природе изучения языка

Изучение нового языка — сложная задача, и нет никакого способа обойти ее. Я уверен, что это одна из причин, по которым вы купили эту книгу вместо того, чтобы попробовать справиться с этой задачей самостоятельно. Так же как в испанском, японском, мандаринском, французском или любом другом языке общения между людьми, некоторые аспекты C# первое время будут казаться лишенными смысла, и далее я собираюсь рассказать о его особенностях, которые в первый момент покажутся вам непонятными. Вероятно также, что наступит период, когда кое-что для вас начнет проясняться, но в целом язык C# будет оставаться для вас непонятным (то же чувство возникает после изучения испанского языка в течение семестра, если попытаться посмотреть мыльную оперу на испанском, которые во множестве транслирует компания Telemundo[1]). Это чувство возникало почти у всех моих студентов ближе к середине первого семестра, но к его концу все они чувствовали себя намного увереннее и в C#, и в прототипировании игр.

Уверяю вас, эта книга поможет вам, и, прочитав до конца, вы оставите ее не только с полноценным пониманием C#, но и с несколькими прототипами простых игр, которые сможете использовать как основу для своих будущих проектов. Подход,

[1] Испанская телевещательная компания. — *Примеч. пер.*

предпринятый в этой книге, основан на многолетнем опыте обучения «непрограммистов» развитию в себе скрытых способностей к программированию и, в более широком смысле, к воплощению идей в действующие прототипы. Как вы увидите далее, этот подход включает три этапа.

1. **Введение в понятие:** перед тем, как попросить вас запрограммировать что-то в том или ином проекте, я расскажу, что мы делаем и зачем. Такое общее знакомство с идеей каждого учебного примера даст вам каркас, к которому вы сможете добавить разные программные элементы, описываемые в главе.
2. **Пошаговое руководство:** я шаг за шагом проведу вас через каждый учебный пример, демонстрируя новые понятия в форме действующих игр. В отличие от других руководств, которые вы, возможно, видели, на протяжении всего процесса я буду компилировать и тестировать игры, чтобы вы могли выявлять и исправлять ошибки (в коде) постепенно, а не все сразу в самом конце. Более того, я преднамеренно буду вносить некоторые ошибки, чтобы вы могли познакомиться с ними и с проблемами, которые они порождают; позднее это облегчит вам поиск своих ошибок.
3. **Намылить, смыть, повторить**[1]: во многих учебных примерах я буду просить вас кое-что повторить. Например, глава 30 «Прототип 3: Space SHMUP» — шутер, напоминающий игру *Galaga*, — проведет вас сквозь процесс создания врага одного типа, а глава 31 «Прототип 3.5: Space SHMUP Plus» — еще трех. Не пропустите ее! Такое повторение поможет вам надежнее усваивать понятия.

90 % ОШИБОК — ПРОСТЫЕ ОПЕЧАТКИ. Я так много времени потратил, помогая студентам исправлять ошибки, что с ходу замечаю опечатки в коде. Вот наиболее распространенные из них:

- **Орфографические ошибки:** даже если вы ошибетесь в единственной букве, компьютер не поймет, что вы хотели сказать ему.
- **Ошибки в регистре символов:** для компилятора C# буквы `A` и `a` — это два разных символа, то есть с его точки зрения слова `variable`, `Variable` и `variAble` — это три совершенно разных слова.
- **Отсутствие точки с запятой:** так же как в русском языке почти все предложения должны заканчиваться точкой, почти все инструкции в C# должны заканчиваться точкой с запятой (`;`). Отсутствие точки с запятой часто заставляет компилятор сообщать об ошибке в следующей строке. К вашему сведению: эта роль возложена на точку с запятой по той простой причине, что точка уже используется для отделения целой и дробной частей в вещественных числах, а также в *точечном синтаксисе* ссылок на переменные (например, `varName.x`).

[1] Отсылка к анекдоту про программиста, надолго застрявшего в душе, потому что следовал инструкции на флаконе с шампунем: «намыльте, смойте, повторите». — *Примеч. пер.*

Выше я упоминал, что большинство моих студентов путались и терялись в языке C# примерно до середины семестра, и именно в этот момент я давал им задание под названием «Проект классической игры». В этом задании предлагается в течение четырех недель точно воссоздать механику и атмосферу классической игры. Прекрасными примерами таких игр могут служить *Super Mario Bros.*, *Metroid*, *Castlevania*, *Pokemon* и даже *Crazy Taxi* [1]. Вынужденные работать самостоятельно, планировать свое время и глубоко погружаться во внутреннюю кухню этих, казалось бы, простых игр, студенты начинали осознавать, что понимают C# намного лучше, чем думали, и в этот момент для них многое вставало на свои места. Ключевым компонентом этого процесса является смена парадигмы с «я следую за руководством» на «я должен сделать это — как это организовать?». К концу этой книги вы будете готовы заняться самостоятельной разработкой своих игровых проектов (или своего проекта классической игры, если хотите). Учебные руководства в этой книге могут стать для вас превосходной отправной точкой при создании своих игр.

Первый запуск Unity

Запустив Unity в первый раз, вы должны кое-что настроить[2].

1. В Windows вам может быть предложено разрешить Unity взаимодействовать с интернетом через брандмауэр. Дайте такое разрешение.
2. Вам будет предложено выполнить вход в свою учетную запись Unity. Если вы ее еще не создали, сделайте это прямо сейчас.
3. На следующем экране отметьте флажок Unity Personal и щелкните на кнопке Next (Далее). После этого появится экран с лицензионным соглашением.
4. По условиям нельзя устанавливать версию Unity Personal, если компания или организация, которую вы представляете, заработала больше 100 000 долларов США за предыдущий финансовый год. Как читатель этой книги, установите флажок I don't use Unity in a professional capacity (Я не использую Unity в профессиональной деятельности) и щелкните на кнопке Next (Далее).
5. Щелкните на вкладке Getting started (Введение) в верхней части экрана Unity и просмотрите видеоурок. В нем содержится немного вводной информации. Не волнуйтесь, если вы не успеете что-то рассмотреть. Мы еще вернемся к этому вопросу.

---

[1] Одной из моих любимых классических игр, которые я предлагаю воссоздать, является *The Legend of Zelda*, и именно ею мы займемся в главе 35 «Прототип 7: Dungeon Delver».

[2] Разумеется, с учетом появления новых версий Unity каждые 90 дней, эти шаги могут измениться. Столкнувшись с существенными отличиями, вы всегда найдете самую свежую информацию на сайте книги: http://book.prototools.net.

Пример проекта

Для доступа к примеру проекта выполните следующие шаги[1].

1. Щелкните на вкладке **Projects** (Проекты) в начальном окне Unity, и вы увидите проект **Example Project** в списке стандартных ресурсов. Щелкните на названии проекта, чтобы открыть его.
2. Когда проект откроется, вы должны увидеть картину, изображенную на рис. 17.4. Щелкните на кнопке **Play** (Играть), чтобы запустить сцену.

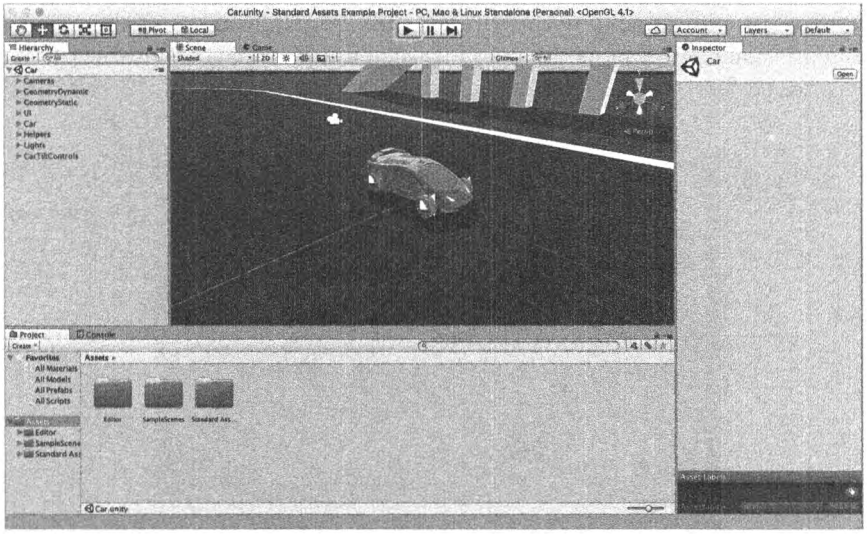


Рис. 17.4. Пример проекта, открытый в Unity и кнопка Play (Играть)

Во время игры в любой момент можно нажать клавишу **Esc**, чтобы открыть меню с доступными сценами. Когда я опробовал этот проект, в сцене **Characters > First Person Character** (Персонажи > Персонаж первого лица) имелась ошибка: указатель мыши исчезал во время воспроизведения этой сцены (вероятно, преднамеренно), но не появлялся после переключения на другую сцену (скорее всего, это ошибка). Поэтому я советую опробовать сцену **First Person Character** (Персонаж первого лица) в последнюю очередь. В этом случае вы сможете нажать клавишу **Esc**, чтобы вернуть указатель мыши на экран, и щелкнуть на кнопке **Play** (Играть) еще раз, чтобы остановить воспроизведение.

Честно говоря, я считаю, что пример проекта *Angry Bots* в Unity 4 лучше справлялся с задачей демонстрации возможностей движка, тем не менее и этот проект показывает некоторую широту возможностей Unity.

[1] Эти инструкции писались для примера проекта в Unity 5.6. Надеюсь, что в скором времени Unity выпустит что-то получше.

Чтобы увидеть видеоролики с более захватывающими играми, сделанными в Unity, я советую зайти на канал Unity в YouTube, или поискать по фразе «YouTube Unity», или сразу перейти по ссылке https://www.youtube.com/user/Unity3D и поискать видеоролики по фразе «Made with Unity» (Сделано в Unity).

Настройка размещения окон Unity

Последнее, что вам следует сделать перед тем, как начать работу в Unity, — настроить размещение окон. Unity — очень гибкая среда и позволяет вам разместить ее панели по вашему вкусу. Вы можете увидеть несколько вариантов размещения панелей в меню Layout (Размещение), в правом верхнем углу окна Unity (рис. 17.5).

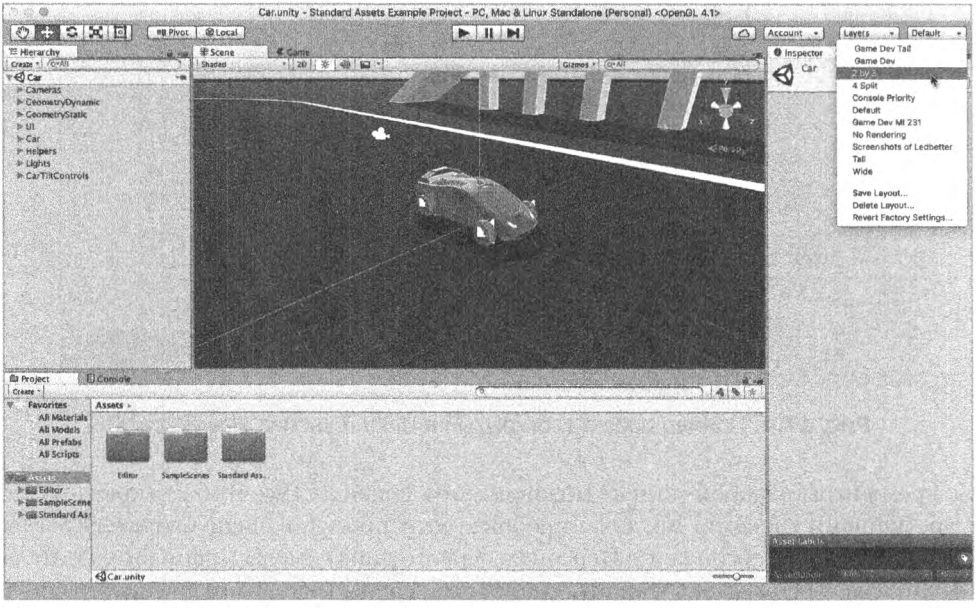


Рис. 17.5. Положение меню Layout (Размещение) с выделенным макетом *2 by 3* (2 по 3)

1. Выберите в меню Layout (Размещение) пункт 2 by 3 (2 по 3), как показано на рис. 17.5. Это станет для нас отправной точкой на пути к созданию своего макета.
2. Прежде чем продолжить, избавьте панель Project (Проект) от лишней информации. Для этого выберите в меню панели Project (Проект) (соответствующая кнопка выделена на рис. 17.6 рамкой) пункт One Column Layout (В одну колонку). В результате панель Project (Проект) превратится в представление с иерархическим списком, которое будет использоваться на протяжении всей книги.

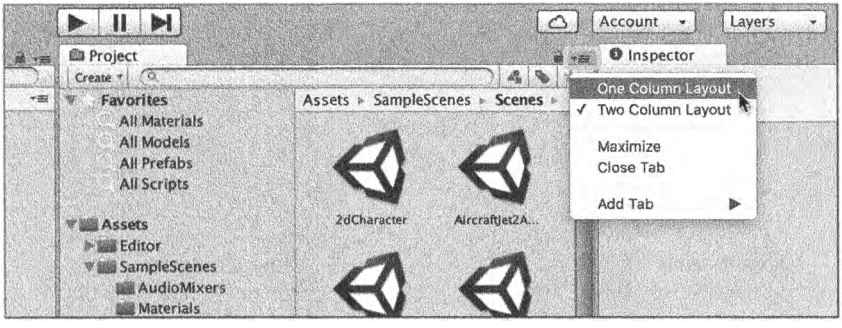


Рис. 17.6. Выбор *размещения в одну колонку* для панели Project (Проект)

Unity позволяет перемещать панели по экрану и изменять положение границ между ними. Как показано на рис. 17.7, вы можете переместить панель, ухватив ее мышью за вкладку (указатель мыши в виде стрелки), или изменить положение границ (указатель мыши в виде двух стрелок влево и вправо).

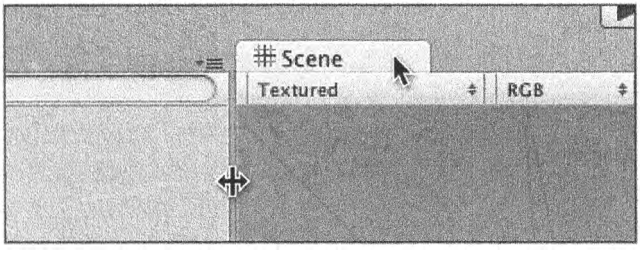


Рис. 17.7. Две формы указателя мыши для перемещения панелей и изменения их размеров

Если ухватить панель за вкладку указателем мыши, на экране появится ее уменьшенная полупрозрачная копия (рис. 17.8). В интерфейсе имеется несколько позиций, где панель может быть закреплена. Когда перемещаемая копия оказывается поблизости от такой позиции, она «прилипает» к ней.

3. Попробуйте разместить панели в окне — передвигая и изменяя их размеры, — пока не получите интерфейс, изображенный на рис. 17.9.
4. Наконец, добавьте панель **Console** (Консоль). В полосе меню выберите пункт **Window > Console** (Окно > Консоль) и передвиньте панель **Console** (Консоль) так, чтобы она оказалась ниже панели **Hierarchy** (Иерархия). В результате она разместится под панелью **Hierarchy** (Иерархия), но не под панелью **Project** (Проект).
5. Ухватите указателем мыши вкладку панели **Project** (Проект) и сдвиньте ее чуть вправо. Вы должны увидеть, что она зафиксировалась в позиции над левой половиной панели **Console** (Консоль). Отпустите кнопку мыши, и вы увидите окончательное размещение панелей, как показано на рис. 17.10.

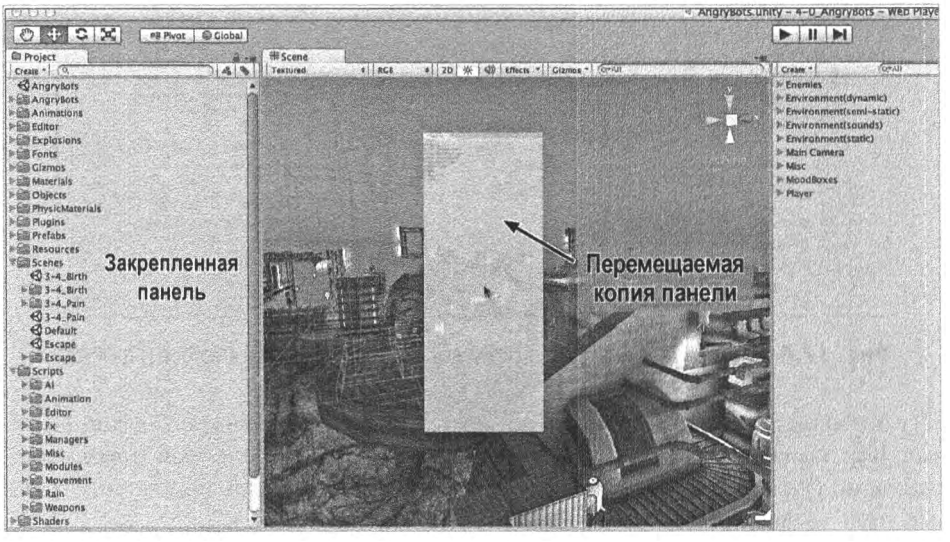


Рис. 17.8. Перемещаемая копия панели и закрепленная панель в окне Unity

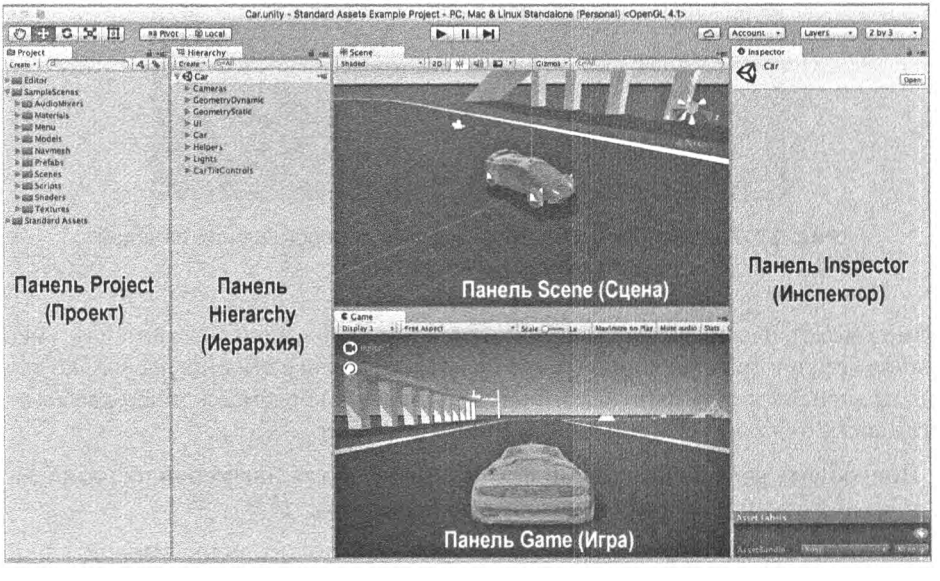


Рис. 17.9. Правильное размещение панелей в окне Unity — но здесь кое-чего не хватает

6. Теперь нужно закрепить это размещение в меню **Layout** (Размещение), чтобы потом не проделывать все описанные действия снова. Щелкните на кнопке вызова меню **Layout** (Размещение) и выберите пункт **Save Layout...** (Сохранить размещение), как показано на рис. 17.11.

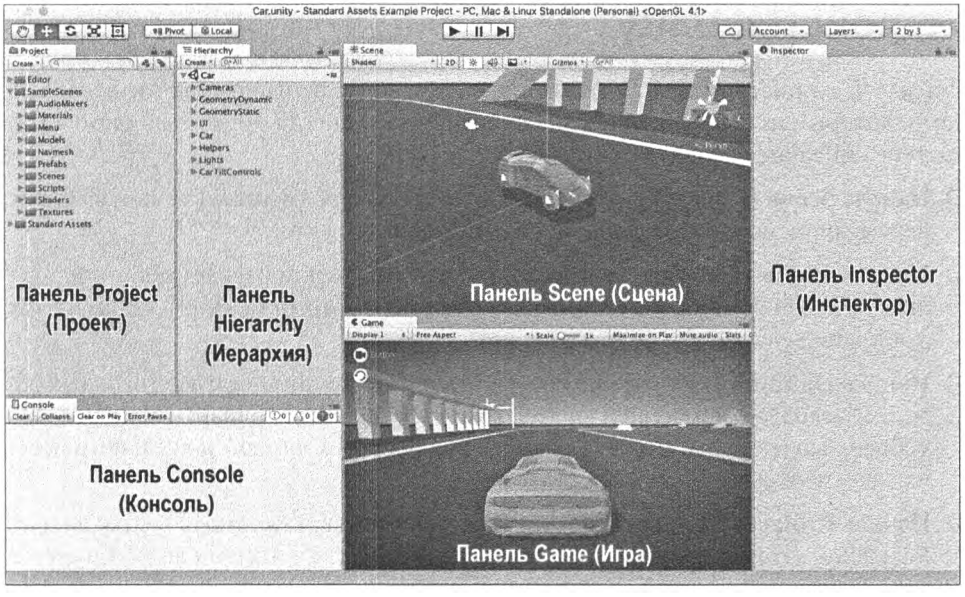


Рис. 17.10. Окончательное размещение панелей в окне Unity, включая панель Console (Консоль)

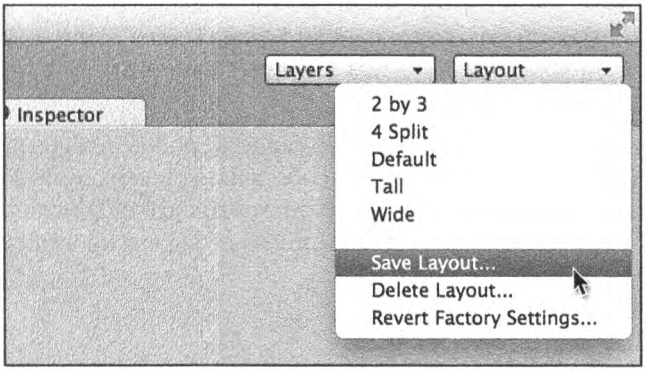


Рис. 17.11. Сохранение размещения

7. Сохраните это размещение с именем *Game Dev*, используя метод, принятый для вашей платформы, в результате имя размещения окажется в меню. В macOS можно перед буквой *G* добавить пробел, а в Windows — символ подчеркивания. Благодаря добавлению пробела или подчеркивания, имя нового размещения окажется в первой строчке меню в соответствии с правилами сортировки. Теперь в любой момент, когда появится желание вернуться к этому размещению, вы сможете просто выбрать его в меню.

Устройство Unity

Прежде чем приступить к программированию, вы должны познакомиться с панелями, которые только что двигали. Обращайтесь к рис. 17.10, когда будете читать краткое описание каждой панели:

- **Панель Scene (Сцена)** позволяет просматривать трехмерные сцены и выбирать, перемещать, поворачивать, изменять размеры объектов.
- **Панель Game (Игра)** — это место, где фактически протекает действие игры; именно в ней вы пробовали поиграть с примером проекта. Здесь также отображается вид с главной камеры в сцене.
- **Панель Hierarchy (Иерархия)** отображает информацию обо всех игровых объектах `GameObject` в текущей сцене. Пока просто рассматривайте эту панель как уровень вашей игры. Все, что находится в сцене, от камеры до главного персонажа, — игровые объекты `GameObject`.
- **Панель Project (Проект)** содержит все ресурсы, включенные в состав проекта. Ресурсы — это файлы, являющиеся частью проекта, в том числе изображения, трехмерные модели, код на C#, текстовые файлы, звуки и шрифты. Панель Project (Проект) является отражением папки *Assets*, находящейся в папке проекта, на жестком диске компьютера. Отображаемые здесь ресурсы необязательно задействованы в текущей сцене.
- **Панель Inspector (Инспектор)**: если щелкнуть на ресурсе в панели Project (Проект), или на игровом объекте в панели Scene (Сцена), или в панели Hierarchy (Иерархия), информация об этом элементе отобразится в панели Inspector (Инспектор).
- **Панель Console (Консоль)** позволяет увидеть сообщения среды Unity, описывающие ошибки в коде, а также ваши собственные сообщения, помогающие понять внутреннюю работу вашего кода[1]. Мы активно будем использовать панель Console (Консоль) в главе 19 «Hello World: ваша первая программа» и в главе 20 «Переменные и компоненты».

Итоги

Вот мы и разобрались с установкой. Теперь перейдем непосредственно к разработке! Как вы могли убедиться в этой главе, среда Unity способна создавать довольно впечатляющие визуальные эффекты и увлекательный игровой процесс. В следующей главе вы напишете вашу первую программу для Unity.

[1] Функции `print()` и `Debug.Log()`, поддерживаемые средой Unity, позволяют выводить сообщения в панель Console (Консоль).

18 Знакомство с нашим языком: C#

Эта глава познакомит вас с ключевыми особенностями языка C# и перечислит некоторые важные причины, почему я выбрал именно этот язык для данной книги. В главе также исследуется базовый синтаксис C# и объясняется, что скрывается под структурой некоторых простых инструкций на C#.

К концу этой главы вы будете лучше понимать C# и подготовитесь заняться более глубокими исследованиями в последующих главах.

Знакомство с особенностями C#

Как рассказывалось в главе 16 «Цифровое мышление», программирование заключается в составлении последовательности простых команд для компьютера, а C# — это язык, на котором пишутся эти команды. В мире существует много разных языков программирования, каждый из которых имеет свои достоинства и недостатки. Вот некоторые особенности, которыми обладает C#:

- компилируемый язык;
- управляемый код;
- строгая поддержка типов;
- основан на функциях;
- объектно-ориентированный.

Все эти особенности описываются в следующих разделах, и каждая из них пригодится вам — так или иначе.

C# — компилируемый язык

Обычно люди пишут компьютерные программы на языке, непонятном компьютеру. В действительности все компьютерные микропроцессоры, имеющиеся на рынке, используют свои, понятные им наборы простых команд, известные как *машинный язык*. Команды на этом языке выполняются микропроцессором очень, очень быстро, но для человека этот язык чрезвычайно сложен. Например, строка на машинном языке

```
000000 00001 00010 00110 00000 100000
```

определенно имеет какое-то значение для соответствующего микропроцессора, но почти ничего не значит для человека. Однако обратите внимание, что в машинном коде используются только два символа — 0 и 1. Поэтому все сложные типы данных — числа, буквы и т. д. — должны преобразовываться в последовательности битов (то есть нули и единицы). Если вы слышали о программировании компьютеров с применением перфокарт, это как раз и есть способ программирования на машинном языке: в большинстве форматов пробивки перфокарт отверстие в перфокарте соответствовало единице, а его отсутствие — нулю.

Чтобы людям было легче писать программы, были придуманы понятные человеку языки программирования — их иногда называют *языками разработки*. Язык разработки можно считать промежуточным, выполняющим роль посредника между вами и компьютером. Языки разработки, такие как C#, имеют логичную структуру и легко интерпретируются компьютером, но при этом достаточно близки к письменным человеческим языкам, чтобы программисты могли легко читать и понимать программы, написанные на них.

Языки разработки также делятся на *компилируемые*, такие как BASIC, C++, C# и Java, и *интерпретируемые*, такие как JavaScript, Perl, PHP и Python (рис. 18.1).

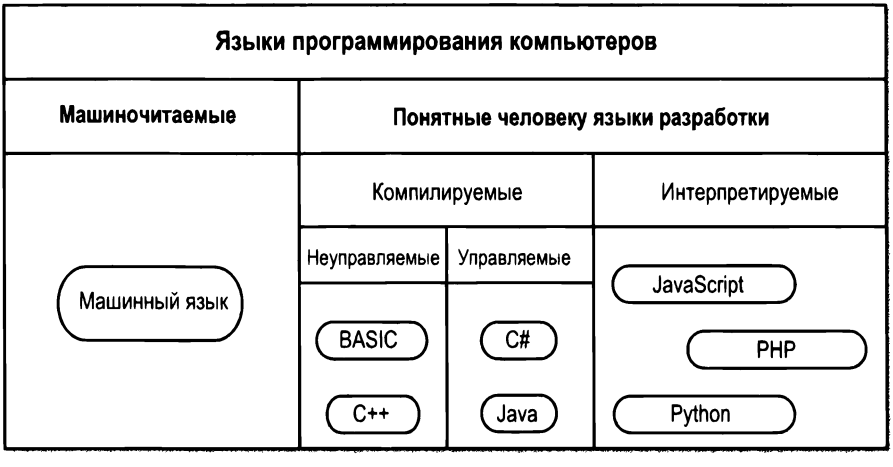


Рис. 18.1. Простая классификация языков программирования

В *интерпретируемых* языках разработка и выполнение кода включает два этапа:

- программист пишет программу;
- затем каждый раз, когда кто-то из игроков играет в игру, код на языке разработки переводится на машинный язык прямо во время выполнения.

Интерпретируемые языки обладают высокой переносимостью кода, потому что код на языке разработки может интерпретироваться в машинные команды для кон-

кретного компьютера, на котором он выполняется. Например, код на JavaScript для веб-страницы будет выполняться практически на любом современном компьютере, независимо от операционной системы, будь то macOS, Windows, Linux или одна из множества операционных систем для мобильных устройств, такая как iOS, Android, Windows Phone и т. д. Однако за такую гибкость приходится платить снижением скорости выполнения из-за необходимости интерпретировать код на языке разработки, не оптимизированном для устройства, на котором он выполняется, а также по ряду других причин. Так как интерпретируемый код должен выполняться на всех устройствах, оптимизация языка для конкретного устройства просто невозможна. Именно по этой причине трехмерные игры, написанные на интерпретируемом языке, таком как JavaScript, обычно выполняются намного медленнее, чем такие же игры, написанные на компилируемом языке, даже на том же самом компьютере.

В *компилируемом* языке, таком как C#, процесс программирования включает три отдельных этапа:

- программист пишет программу на языке разработки, таком как C#;
- компилятор преобразует код на языке разработки в код на машинном языке для компьютера конкретного типа;
- компьютер выполняет скомпилированную программу.

Здесь добавляется промежуточный этап *компиляции*, в ходе которого код на языке разработки преобразуется в выполняемый код (то есть в приложение), который компьютер может выполнять непосредственно, без промежуточной интерпретации. Поскольку компилятор имеет полное представление о программе и платформе, на которой она должна выполняться, он может применять множество разных оптимизаций. В играх эти оптимизации позволяют получить более высокую частоту кадров, использовать более детальную графику и обеспечить более быструю реакцию на действия пользователя. Большинство высокобюджетных игр пишется на компилируемом языке, дающем высокую скорость за счет оптимизации, но это также означает, что для каждой платформы приходится компилировать разный выполняемый код.

Во многих случаях компилируемые языки разработки подходят только для конкретной платформы. Например, *Objective-C* — это собственный язык разработки компании Apple Computer, предназначенный для создания приложений, выполняющихся под управлением macOS и iOS. Этот язык основан на C (предшественнике C++), но включает некоторые особенности, уникальные для macOS и iOS. Аналогично, язык XNA — диалект языка C# — разрабатывался корпорацией Microsoft специально для студентов, обучающихся созданию игр для персональных компьютеров с Windows и игровых приставок Xbox 360.

Как упоминалось в главе 17 «Введение в среду разработки Unity», игры для Unity можно писать на C# или на диалекте JavaScript с названием UnityScript. Исходный код на обоих языках компилируется в код на обобщенном промежуточном языке (Common Intermediate Language, CIL). А уже потом код на CIL компилируется

в целевой код для любой платформы, от iOS и Android до macOS, Windows, игровых приставок, таких как PlayStation и Xbox, и даже в интерпретируемый код на WebGL (особый диалект JavaScript, используемый в веб-страницах). Этот дополнительный этап компиляции в код на CIL гарантирует возможность компиляции программ для Unity на множестве платформ, на каком бы языке они ни были написаны — на C# или UnityScript.

Возможность писать код один раз и компилировать везде не уникальна для Unity, но является одной из основных целей Unity Technologies, развивающей игровой движок Unity. И она лучше интегрирована в Unity, чем в любых других программных продуктах для разработки игр, которые я видел. Тем не менее, как дизайнер, вы должны помнить о различиях между играми для мобильных устройств, управляемых с помощью сенсорного экрана, персональных компьютеров, управляемых мышью и клавиатурой, и виртуальной, или дополненной, реальности, потому что для разных платформ приходится писать немного отличающийся код.

C# — управляемый язык

Более традиционные компилируемые языки, такие как BASIC, C++ и Objective-C, требуют от программиста явно управлять распределением памяти компьютера, вынуждая его вручную выделять и освобождать память, когда требуется создать или уничтожить переменную[1]. В таких языках если программист вручную не освободит ранее выделенный фрагмент ОЗУ, в его программе возникнет «утечка памяти». Он будет продолжать выделять память в ОЗУ, пока не исчерпает ее всю, что вызовет сбой в работе компьютера.

К счастью для нас, C# — *управляемый язык*, то есть выделение и освобождение памяти в нем выполняется автоматически[2]. В управляемом коде все еще можно создать утечки памяти, но сделать это непреднамеренно намного сложнее.

C# — строго типизированный язык

Переменные рассматриваются в последующих главах подробнее, но некоторые их черты вы должны знать уже сейчас. Во-первых, *переменная* — это просто имя

[1] Выделение памяти — это процедура резервирования некоторого объема в оперативном запоминающем устройстве (ОЗУ) компьютера для хранения данных. Даже при том, что современные компьютеры оборудуются жесткими дисками, вмещающими сотни гигабайтов (Гбайт), они до сих пор имеют объем ОЗУ не более 20 Гбайт. ОЗУ работает *намного* быстрее жесткого диска, поэтому все приложения стремятся извлечь свои ресурсы, такие как изображения и звуки, с жесткого диска и разместить их в ОЗУ, чтобы иметь к ним более быстрый доступ.

[2] Один из недостатков управляемых языков — сложность точного управления механизмом освобождения памяти. Вместо этого память освобождается автоматически, в процессе *сборки мусора*. Иногда это может приводить к пиковому снижению частоты кадров на маломощных устройствах, таких как сотовые телефоны, но чаще остается незамеченным.

контейнера для значения. Например, изучая алгебру в школе, вы могли видеть такое выражение:

```
x = 5
```

Эта одна строка определяет переменную с именем `x` и присваивает ей значение `5`. Позднее, если я спрошу у вас значение `x+2`, вы наверняка дадите ответ `7`, потому что помните, что имени `x` соответствует значение `5`, при сложении которого с числом `2` получается `7`. Именно так действуют переменные в программах.

В большинстве интерпретируемых языков, таких как JavaScript, одна переменная может хранить данные любого типа. Переменная `x` может хранить число `5` в течение одной минуты, изображение — в течение другой и звуковой файл — в течение третьей. Такая способность одной и той же переменной хранить значения разных типов в программировании называется *слабой типизацией*.

C#, напротив, — *строго типизированный* язык. Это означает, что при создании переменной вы должны сообщить тип данных, которые она сможет хранить:

```
int x = 5;
```

Эта инструкция создает переменную с именем `x`, которая может хранить только значения типа `int` (то есть положительные или отрицательные числа без дробной части), и присваивает ей целое число `5`. Может показаться, что строгая типизация усложняет программирование, но в действительности она дает компилятору возможность применять разные оптимизации и позволяет среде разработки MonoDevelop проверять синтаксис в режиме реального времени (подобно тому, как Microsoft Word проверяет грамматику набираемого текста). Строгая типизация также помогает расширить поддержку автоматического дополнения кода в MonoDevelop, предсказывающую слова по мере их ввода и предлагающую возможные допустимые варианты завершения, основываясь на другом коде, написанном вами. Если в процессе ввода вы видите, что MonoDevelop предлагает правильный вариант завершения слова, можете просто нажать клавишу **Tab**, чтобы принять предложение. Когда вы начнете пользоваться этой функцией, она поможет вам сэкономить сотни нажатий клавиш в каждую минуту.

C# основан на функциях

Первые программы состояли из прямолинейных последовательностей команд. Они выполнялись линейно, от начала до конца, подобно инструкциям, которые вы могли бы дать другу, описывая, как добраться до вашего дома:

1. От школы поезжай на север, в Вермонт.
2. Затем примерно 7,5 мили на запад по дороге I-10.
3. На пересечении с I-405 поверни на юг и двигайся вперед 2 мили.
4. Сверни на бульвар Венис (Venice Blvd).

5. Поверни направо, на бульвар Сотел (Sawtelle Blvd).
6. Мой дом прямо на север от Венис на Сотел.

По мере развития программирования в языках разработки появились многократно повторяемые фрагменты в форме *циклов* (которые сами повторяют себя несколько раз) и *подпрограмм* (программа выполняет переход в подпрограмму, которая затем, после выполнения, возвращается в программу).

Создание *процедурных языков* (то есть языков, использующих функции)[1] позволило программистам создавать именованные фрагменты кода и заключать в них определенную функциональность (то есть группировать последовательности действий под одним именем функции). Например, если, описывая путь к вашему дому, как в предыдущем списке, вы попросите друга купить молока по дороге, он будет знать, что, увидев продуктовый магазин, должен остановиться, выйти из машины, зайти в магазин, найти молоко, заплатить за него, вернуться в машину и продолжить путь к вашему дому. Поскольку ваш друг уже знает, как покупается молоко, вам достаточно просто попросить его об этом, не давая подробных пошаговых инструкций. Например, так:

```
«Если увидишь магазин по пути, BuySomeMilk()»
```

Это предложение объединяет все инструкции по покупке молока в одну функцию с именем `BuySomeMilk()`. То же самое можно проделывать в любом процедурном языке. Когда компьютер, обрабатывающий код на C#, встретит имя функции, за которым следуют две круглые скобки, он *вызовет* эту функцию (то есть выполнит все действия, описанные в ней). Больше о функциях вы узнаете в главе 24 «Функции и параметры».

Другая фантастическая особенность функций: написав функцию `BuySomeMilk()` один раз, вам не придется делать это повторно. Даже при работе над другой программой часто есть возможность копировать и вставлять функции, такие как `BuySomeMilk()`, и повторно использовать их без необходимости переписывать с самого начала. Сценарий на C# с именем `Utils.cs`, который вы увидите в нескольких учебных примерах в этой книге, включает набор подобных функций многократного пользования.

C# — объектно-ориентированный язык

Через много лет после изобретения функций появилась идея *объектно-ориентированного программирования* (ООП). В соответствии с идеей ООП, функциональность и данные объединяются в конструкции, которые называются *объектами*, или, точнее, *классами*. Более подробно об этом рассказывается в главе 26 «Классы», тем не менее приведу здесь метафору.

---

[1] Существуют также *функциональные языки*, такие как Lisp, Scheme, Mathematica (язык Wolfram) и Haskell, но для этих функциональных языков слово «функциональный» означает нечто иное, чем возможность писать функции в C#.

Представьте группу разных животных. Каждое животное обладает конкретной информацией о себе. Примерами такой информации могут служить: вид, возраст, размеры, эмоциональное состояние, уровень голода, текущее местоположение и т. д. Каждое животное также может выполнять определенные действия: есть, двигаться, дышать и т. д. Данные о каждом животном сродни переменным в коде, а действия, выполняемые животными, аналогичны функциям.

До ООП представление животного в коде могло хранить информацию (то есть переменные), но не могло выполнять никаких действий. Действия выполнялись функциями, напрямую не связанными с животным. Программист мог написать функцию `Move()`, реализующую движение любых животных, но при этом ему нужно было написать в функции несколько строк кода для определения вида животного и выбора соответствующего типа движения. Например, собаки ходят, рыбы плавают, а птицы летают. Каждый раз, добавляя новое животное, программист должен был внести изменения в функцию `Move()`, чтобы привести ее в соответствие с новым типом передвижения, из-за чего `Move()` становилась все больше и сложнее.

Объектно-ориентированный подход изменил все это, введя идеи *классов* и *наследования*. *Класс* объединяет переменные и функции в один объект. В ООП вместо одной большой функции `Move()`, поддерживающей любых животных, за каждым животным закреплена более конкретная функция `Move()` намного меньшего размера. Это избавляет от необходимости расширять `Move()` при добавлении каждого нового типа животного, а также от необходимости проверять типы животных в `Move()`. Вместо этого каждый новый класс животных получает свою небольшую функцию `Move()`.

Объектно-ориентированный подход также включает понятие *наследования классов*. Наследование позволяет классам иметь более конкретные подклассы, а подклассам — наследовать или переопределять функции их *суперклассов*. Благодаря наследованию можно создать единый класс `Animal`, включающий объявление всех данных, общих для всех животных. Этот класс также мог бы иметь функцию `Move()`, но не конкретную. В подклассах класса `Animal`, таких как `Dog` или `Fish`, функция `Move()` могла бы переопределяться для придания конкретного способа передвижения, такого как ходьба или плавание. Это ключевой элемент программирования современных игр, и вы с успехом сможете использовать его, например, при создании общего базового класса врага `Enemy` и при создании более конкретных подклассов для каждого типа врага.

Чтение и понимание синтаксиса C#

Как любой другой язык, C# имеет конкретный синтаксис, которому вы должны следовать. Взгляните на следующие примеры предложений на русском языке:

- Собака лаяла на белку.
- На белку собака лаяла.
- Собака на белку лаяла.
- Лаяла собака на белку.

Все эти предложения состоят из тех же слов и знаков пунктуации, но расставленных в разном порядке и с изменением заглавных символов. Знакомые с русским языком сразу отметят, что только первое предложение составлено без использования инверсии. Другой, более абстрактный подход — рассмотреть предложение как набор членов предложений:

- [Подлежащее] [сказуемое] [дополнение].
- [Дополнение] [подлежащее] [сказуемое].
- [Подлежащее] [дополнение]. [сказуемое]
- [сказуемое] [Подлежащее] [дополнение].

Такая перестановка частей предложения изменяет синтаксис предложения, и последние три предложения неправильны, потому что содержат синтаксические ошибки. Как в любом другом языке, в C# есть конкретные синтаксические правила составления предложений. Рассмотрим подробнее следующую инструкцию:

```
int x = 5;
```

Как объяснялось выше, эта инструкция выполняет несколько действий:

- Объявляет переменную с именем `x` и с типом `int`.

  Всякий раз, когда инструкция начинается с типа переменной, второе слово в ней интерпретируется как имя новой переменной этого типа (глава 20 «Переменные и компоненты»). Эта инструкция называется *объявлением* переменной.

- Присваивает переменной `x` значение `5`.

  Символ `=` используется для *присваивания* значений переменным (или *инициализации*, если это первая инструкция присваивания для данной переменной). В этом случае имя переменной находится слева, а присваиваемое значение — справа.

- Завершается точкой с запятой (`;`).

  Каждая простая инструкция в C# должна завершаться точкой с запятой (`;`). Она напоминает точку в конце предложений на русском языке.

Почему инструкции в C# не завершаются точкой? Языки программирования компьютеров должны быть максимально ясными. Точка в конце инструкций в языке C# не используется потому, что она уже используется в числах как десятичная точка (например, точка в числе `3.14159`). Для большей ясности точка с запятой используется в C# только для завершения инструкций.

Теперь добавим вторую простую инструкцию:

```
int x = 5;
int y = x * ( 3 + x );
```

Вторая строка:

- Объявляет переменную с именем `y` и с типом `int`.
- Складывает `3 + x` (то есть 3 + 5, в результате чего получается 8).

  Так же как в алгебре, *порядок операций* изменяется круглыми скобками, то есть сумма `3+x` вычисляется первой, потому что заключена в скобки. Сумма равна 8, потому что в предыдущей инструкции переменной `x` было присвоено значение `5`. В приложении Б «Полезные идеи» есть раздел «Предшествование операторов и порядок операций», подробнее описывающий порядок выполнения операций в C#, но главное, что вы должны запомнить на будущее: если у вас есть какие-то сомнения о порядке выполнения операций, всегда используйте круглые скобки для устранения сомнений (и увеличения читаемости кода).
- Умножает `x * 8` (`x` — это 5, поэтому в результате получается 40).

  В отсутствие круглых скобок операции умножения и деления выполняются *перед* операциями сложения и вычитания. В отсутствие круглых скобок выражение `x * 3 + x` превратилось бы в `5 * 3 + 5`, затем в `15 + 5` и дало бы в результате `20`.
- Присваивает значение `40` переменной `y`.
- Завершается точкой с запятой (`;`).

Завершим эту главу исследованием пары инструкций на языке C#. На этот раз пронумеруем инструкции. Номера строк упрощают ссылку на определенную строку в коде, и я надеюсь, что они помогут вам читать и понимать код в этой книге при его вводе в свой компьютер. Важно понимать, что **вы не должны вводить номера строк** в MonoDevelop. MonoDevelop автоматически нумерует (и перенумерует) строки в процессе ввода:

```
1 string greeting = "Hello World!";
2 print( greeting );
```

Эти две инструкции работают со *строками* (последовательностями символов, такими как слова или предложения), а не с целыми числами. Первая инструкция (с номером `1`):

- Объявляет переменную с именем `greeting` и с типом `string`.

  `String` — еще один тип переменных, так же как `int`.
- Присваивает переменной `greeting` значение `"Hello World!"`.

  Двойные кавычки, окружающие `"Hello World!"`, сообщают компилятору C#, что символы между ними должны интерпретироваться как *строковый литерал* без любого побочного смысла. Если поместить в код строковый литерал `"x = 10"`, он **не** присвоит число `10` переменной `x`, потому что компилятор игнорирует строковые литералы между кавычками и не пытается интерпретировать как код на языке C#.

- Завершается точкой с запятой (`;`).

  Вторая инструкция (с номером `2`):

- Вызывает функцию `print()`.

  Как обсуждалось выше, функции — это именованные коллекции операций. Когда программа *вызывает* функцию, та выполняет содержащиеся в ней операции. Как можно предположить, `print()` содержит операции, которые выводят строку в панель Console (Консоль). Всякий раз, встречая в коде слово, за которым следуют круглые скобки, знайте, что это вызов или определение функции. Если записать имя функции с круглыми скобками за ним, программа вызовет функцию, заставив ее выполнить свои операции. Пример определения функции будет показан в следующей главе.

- Передает `greeting` в `print()`.

  Некоторые функции просто выполняют свою работу без всяких параметров, но многие требуют передачи им каких-либо значений. Любая переменная, заключенная в круглые скобки в вызове функции, будет передана в эту функцию как *аргумент*. В данном случае в функцию `print()` передается строка `greeting`, благодаря чему в панели Console (Консоль) появится последовательность символов `Hello World!`.

- Завершается точкой с запятой (`;`).

  Каждая простая инструкция должна завершаться точкой с запятой.

Итоги

Теперь, когда вы узнали чуть больше о C# и Unity, пришло время объединить их в вашу первую программу. Следующая глава проведет вас через процесс создания нового проекта Unity и нескольких сценариев на C#, добавления простого кода в эти сценарии и манипулирования трехмерными игровыми объектами `GameObject`.

19 Hello World: ваша первая программа

Добро пожаловать в мир программирования!

К концу этой главы вы создадите новый проект и напишете свой первый фрагмент кода. Мы начнем с классического проекта «Hello World», традиционной первой программы, с которой начинали изучать новый язык задолго до того, как я сам начал программировать, а затем мы перейдем к исследованию возможностей Unity.

Создание нового проекта

Теперь, после настройки окна Unity (в предыдущей главе), самое время приступить к программированию. Этот шаг мы начнем с создания нового проекта.

В приложении А «Стандартная процедура настройки проекта» приводятся подробные инструкции, как настраивать проекты Unity для глав этой книги. В начале описания каждого проекта вы будете видеть врезку, такую как ниже. Выполните рекомендации во врезке, чтобы настроить проект для этой главы.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, описанной в приложении А, создайте новый проект в Unity.

- **Имя проекта:** Hello World
- **Имя сцены:** (пока отсутствует)
- **Имена сценариев на C#:** (пока отсутствуют)

Обязательно прочитайте все описание процедуры в приложении А; в данном случае вам нужно лишь создать проект. Как создавать сцены и сценарии на C#, вы узнаете далее в этой главе.

Создавая проект в Unity, вы фактически создаете папку, где будут храниться все файлы, составляющие проект. Когда Unity завершит создание проекта, откроется

новая сцена, содержащая только объекты главной камеры Main Camera и направленного освещения Directional Light, которые появятся в панели Project (Проект). Прежде чем сделать что-то еще, сохраните сцену, выбрав в меню пункт File > Save Scene (Файл > Сохранить сцену). Unity автоматически выберет нужную папку для сохранения сцены, поэтому вам остается только ввести ее имя *_Scene_0* и щелкнуть на кнопке Save (Сохранить).[1] После сохранения сцены она появится в панели Project (Проект).

Щелкните правой кнопкой мыши на пустом месте в панели Project (Проект) и в контекстном меню выберите пункт Reveal in Finder (Показать в Finder, в macOS) или Show in Explorer (Показать в Проводнике, в Windows), как показано на рис. 19.1.

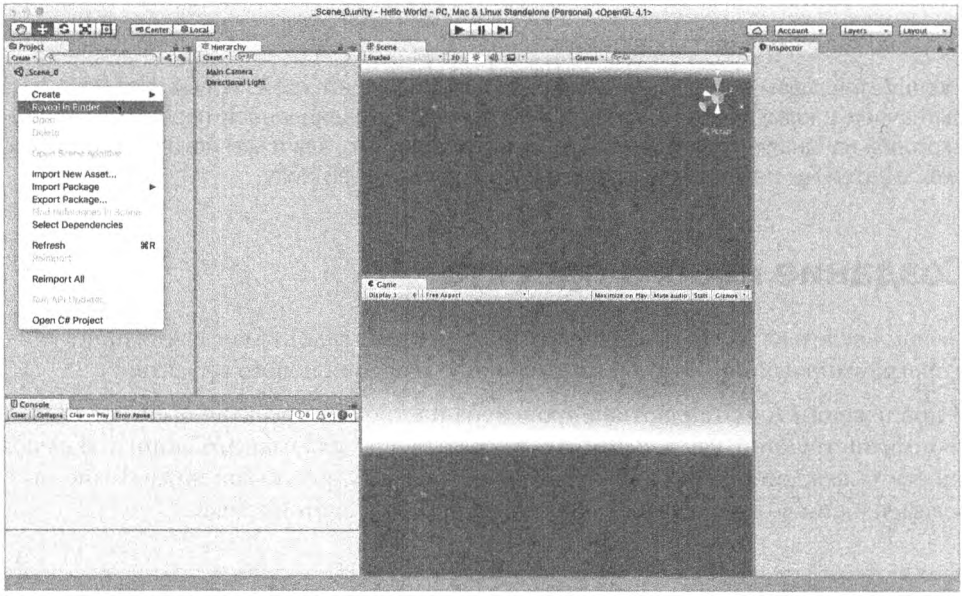


Рис. 19.1. Пустой холст нового проекта Unity (пункт Reveal in Finder (Показать в Finder) в контекстном меню панели Project (Проект))

Щелчок правой кнопкой мыши или сенсорной панели в macOS выполняется не так, как в Windows. Чтобы узнать, как его выполнить, прочитайте раздел «Щелчок правой кнопкой в macOS» в приложении Б «Полезные идеи».

Выбор пункта Reveal in Finder (Показать в Finder, в macOS) откроет окно Finder (или окно Explorer) и отобразит в нем содержимое папки проекта (рис. 19.2).

[1] Символ подчеркивания (_) в начале имени сцены *_Scene_0* обеспечит его вывод в верхней части панели Project (Проект) согласно правилам сортировки (в macOS).

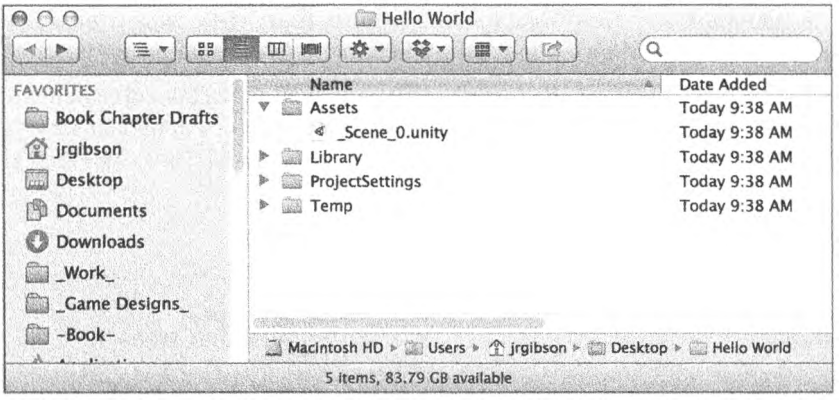


Рис. 19.2. Папка проекта Hello World в окне Finder в macOS

Как показано на рис. 19.2, папка *Assets* хранит все, что присутствует в панели Project (Проект). Теоретически папку *Assets* и панель Project (Проект) можно использовать взаимозаменяемо (например, если сбросить файл с изображением в папку *Assets*, он появится в панели Project (Проект), и наоборот), но я настоятельно советую работать только с панелью Project (Проект), не касаясь папки *Assets*. Прямые манипуляции с содержимым папки *Assets* иногда могут вызывать проблемы, тогда как операции с содержимым панели Project (Проект) обычно более безопасны. Кроме того, очень важно не трогать содержимое папок *Library*, *ProjectSettings* и *Temp*. Пренебрежение этим советом может стать причиной неожиданного поведения Unity и повреждения проекта.

А теперь вернемся в Unity.

НИКОГДА НЕ ИЗМЕНЯЙТЕ ИМЯ ПАПКИ ПРОЕКТА ПРИ РАБОТАЮЩЕЙ СРЕДЕ UNITY. Если изменить имя папки проекта или переместить ее в другое место при работающей среде Unity, это может вызвать ее крах. Unity активно управляет файлами проекта во время работы и изменение имени папки почти всегда приводит к краху. Если понадобится изменить имя папки проекта, закройте Unity, измените имя папки и запустите Unity снова.

Создание нового сценария на C#

Настал ваш час. Теперь вам предстоит написать свой первый фрагмент кода (в последующих главах вы узнаете больше о языке C#, а пока просто копируйте все, что увидите здесь):

1. Щелкните на кнопке Create (Создать) в панели Project (Проект) и в открывшемся меню выберите пункт Create > C# Script (Создать > Сценарий C#, как показано на рис. 19.3). В результате в панели Project (Проект) появится новый сценарий, а его имя будет автоматически выделено для изменения.

2. Дайте сценарию имя HelloWorld (обратите внимание, что между словами в имени не должно быть пробелов) и нажмите Return, чтобы подтвердить ввод имени.
3. Дважды щелкните на имени или на ярлыке сценария HelloWorld (в панели Project (Проект)), чтобы запустить MonoDevelop, наш редактор сценариев на C#. Сразу после запуска редактора ваш сценарий должен выглядеть в точности так, как показано на рис. 19.3, кроме строки 9.

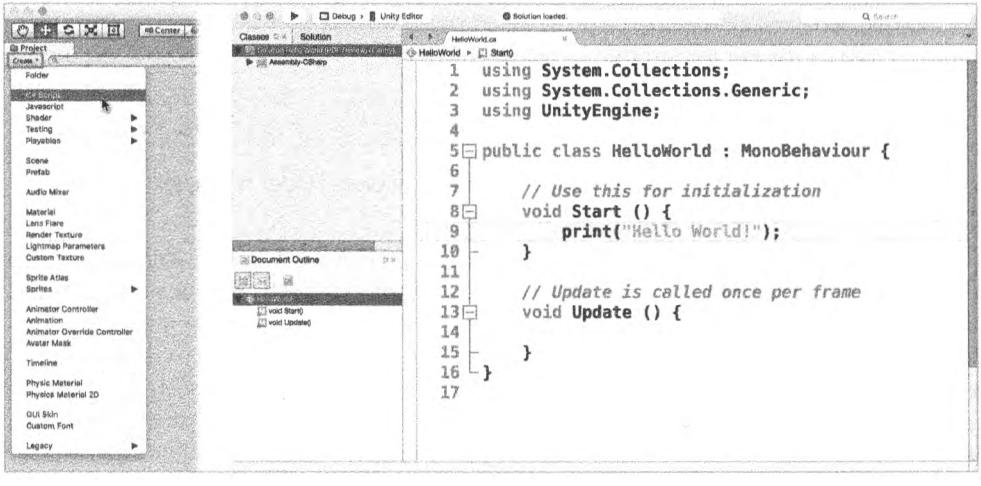


Рис. 19.3. Создание нового сценария на C# и его вид в окне MonoDevelop

4. В строке 9 сценария нажмите два раза клавишу Tab и введите код `print("Hello World");`, как показано в листинге, следующем далее. Обратите внимание на правильную расстановку заглавных символов и точку с запятой (`;`) в конце строки.

Ваш сценарий HelloWorld теперь должен выглядеть так, как показано в листинге ниже. На протяжении всей книги новые строки в листингах, а также строки, которые вы должны изменить, будут **выделяться жирным**, а код, присутствующий по умолчанию, — обычным шрифтом.

Каждая строка в листинге также имеет свой номер. Как можно видеть на рис. 19.3, MonoDevelop автоматически нумерует строки кода, поэтому вы не должны вводить их. Номера строк приводятся в книге, только чтобы вам проще было читать листинги.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class HelloWorld : MonoBehaviour {
6
7     // Используйте этот метод для инициализации
8     void Start () {
```

```
 9         print("Hello World!");
10     }
11
12     // Update вызывается в каждом кадре
13     void Update () {
14
15     }
16 }
```

Ваша версия MonoDevelop может автоматически добавлять дополнительные пробелы в некоторых местах. Например, она может добавить пробел между `print` и (в строке 9, в теле функции `Start()`. Это нормально, и вы не должны слишком волноваться об этом. В целом, даже при том, что правильная расстановка заглавных символов имеет большое значение, к расстановке пробелов предъявляются более мягкие требования. Последовательность из нескольких пробелов (или нескольких разрывов строк) интерпретируется компилятором C# как один пробел, поэтому вы свободно можете добавлять дополнительные пробелы и разрывы, если это поможет сделать код более читаемым (правда, из-за дополнительных разрывов номера строк в ваших листингах могут отличаться от номеров в листингах в книге).

Кроме того, не переживайте, если номера строк в ваших листингах будут отличаться от номеров строк в примерах. Если код один и тот же, номера строк ничего не значат.

5. Теперь сохраните сценарий, выбрав в главном меню MonoDevelop пункт **File > Save** (Файл > Сохранить), и вернитесь обратно в Unity.

Следующий шаг немного сложнее, но вы быстро освоите его, потому что его часто приходится повторять в Unity.

6. Наведите указатель мыши на имя сценария **HelloWorld** в панели **Project** (Проект), нажмите левую кнопку мыши и, удерживая ее, перетащите сценарий на элемент **Main Camera** в панели **Hierarchy** (Иерархия), а затем отпустите кнопку мыши, как показано на рис. 19.4. В процессе перетаскивания сценария слова **HelloWorld (Monoscript)** будут следовать за указателем мыши, а когда вы отпустите кнопку мыши над элементом **Main Camera**, слова **HelloWorld (Monoscript)** исчезнут.

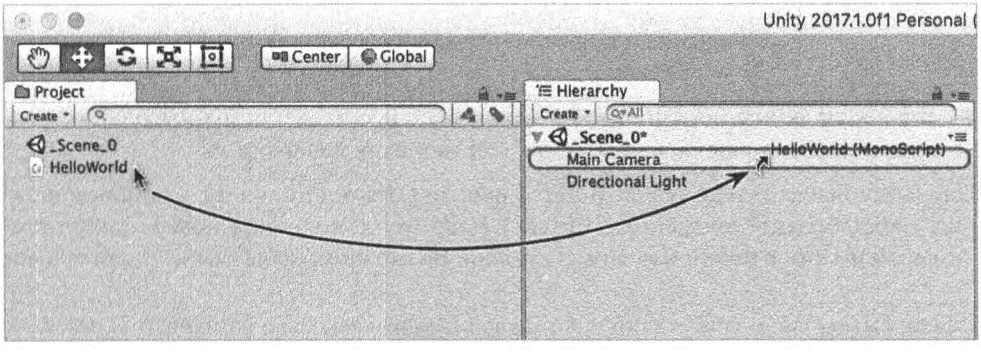


Рис. 19.4. Подключение сценария HelloWorld на C# к главной камере в панели Hierarchy (Иерархия)

Перетаскивание сценария HelloWorld на элемент Main Camera *подключит* его к главной камере как *компонент*. Все объекты в сцене, отображаемые в панели Hierarchy (Иерархия, например Main Camera), называют *игровыми объектами*, или `GameObject`, а игровые объекты состоят из *компонентов*. Если теперь щелкнуть на Main Camera в панели Hierarchy (Иерархия), вы должны увидеть HelloWorld (Script) в списке компонентов главной камеры в панели Inspector (Инспектор). Как показано на рис. 19.5, панель Inspector (Инспектор) содержит список из нескольких компонентов, составляющих главную камеру, в том числе Transform, Camera, GUILayer, Flare Layer, Audio Listener и HelloWorld (Script). В последующих главах вы узнаете больше об игровых объектах и компонентах.[1]

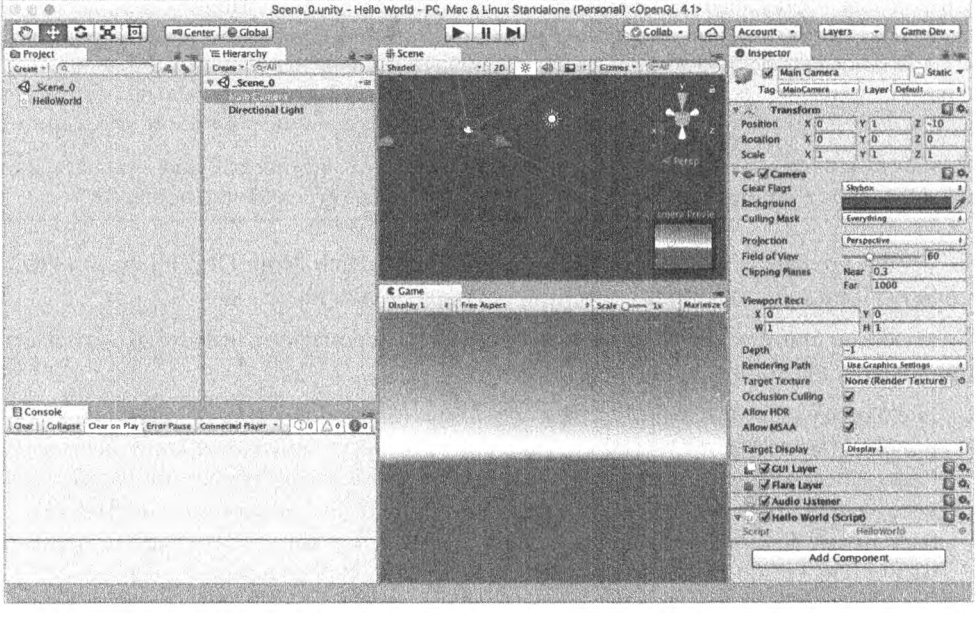


Рис. 19.5. Теперь сценарий HelloWorld появился в списке компонентов главной камеры в панели Inspector (Инспектор)

7. Теперь щелкните на кнопке Play (Играть) с изображением треугольника, направленного вправо, и наблюдайте, как творится волшебство!

Сценарий выведет текст *Hello World!* в панели Console (Консоль), как показано на рис. 19.6. Обратите внимание, что текст *Hello World!* появился также в узкой темно-серой полосе в нижнем левом углу окна. Возможно, это не самое удивительное

[1] Если вы случайно присоединили к главной камере несколько компонентов HelloWorld (Script), лишние всегда можно удалить, щелкнув на ярлыке с изображением шестеренки, справа от имени «HelloWorld (Script)», и выбрав в открывшемся меню пункт Remove Component (Удалить компонент).

событие в вашей жизни, но должны же вы с чего-то начать. Как сказал однажды один мудрый старик: «Вы сделали свой первый шаг в большой мир».

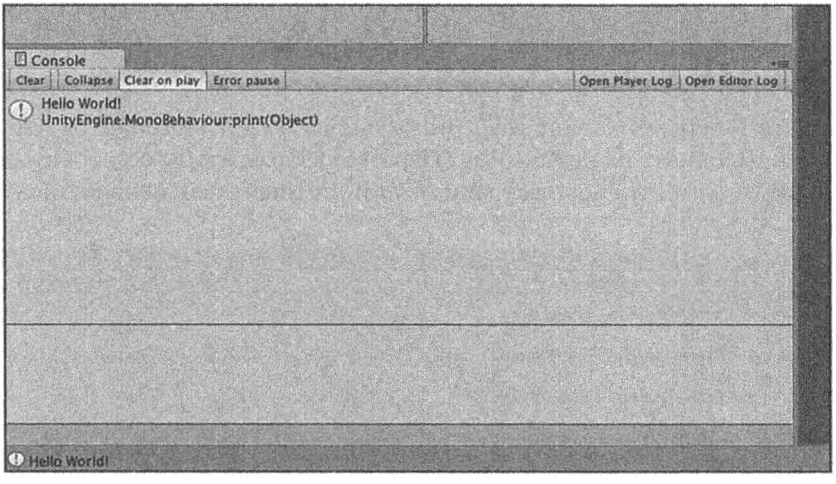


Рис. 19.6. Вывод текста Hello World! в панели Console (Консоль)

Start() и Update()

Теперь попробуйте перенести вызов функции `print()` из метода `Start()` в метод `Update()`.

1. Вернитесь в окно MonoDevelop и исправьте код, как показано в следующем листинге.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class HelloWorld : MonoBehaviour {
6
7     // Используйте этот метод для инициализации
8     void Start () {
9         // print("Hello World!"); // Теперь эта строка будет игнорироваться.
10    }
11
12    // Update вызывается в каждом кадре
13    void Update () {
14        print("Hello World!");
15    }
16 }
```

Добавив два символа слеша (`//`) в начало строки 9, мы превратили весь текст, следующий за слешами, в *комментарий*. Комментарии полностью игнорируются компьютером и используются, чтобы выключить какой-то код (как мы только

что сделали это в строке 9) или оставить сообщение для тех, кто будет читать код (как в строках 7 и 12). Добавив два символа слеша (`//`) в начало строки 9, мы, как говорят, *закомментировали* эту строку. Обязательно добавьте инструкцию `print("Hello World!");` в строку 14, чтобы сделать ее частью функции `Update()`.

2. Сохраните сценарий (заменив предыдущую версию) и попробуйте снова щелкнуть на кнопке **Play** (Играть).

Вы увидите теперь, что текст *Hello World!* снова и снова выводится в консоли (рис. 19.7). Щелкните на кнопке **Play** (Играть) еще раз, чтобы остановить выполнение, и вы увидите, что новые строки *Hello World!* перестали появляться.

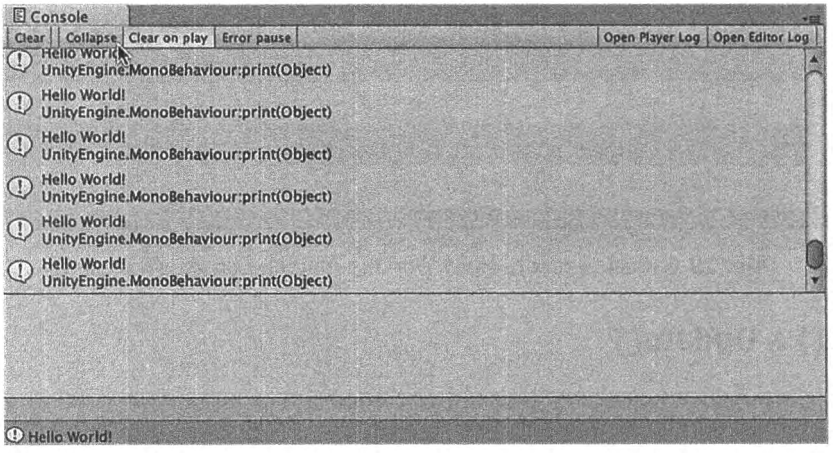


Рис. 19.7. Update() выводит слова Hello World! в каждом кадре

`Start()` и `Update()` — это две специальные функции в Unity-версии C#. `Start()` вызывается один раз в первом кадре существования объекта, а `Update()` вызывается в каждом кадре,[1] поэтому на рис. 19.6 наблюдается одно сообщение, а на рис. 19.7 — несколько. В Unity имеется целый список таких специальных функций, которые вызываются в определенные моменты времени. О многих из них я расскажу далее в этой книге.

На рис. 19.7 можно видеть, что одно и то же сообщение *Hello World!* выводится много раз. Если щелкнуть на кнопке **Collapse** (Свернуть) в панели **Console** (Консоль), под указателем мыши на рис. 19.7, все сообщения *Hello World!* свернутся в одно, рядом с которым будет отображаться число, соответствующее количеству одинаковых сообщений, отправленных в панель **Console** (Консоль). Это упростит наблюдение за появлением уникальных сообщений.

[1] Как рассказывалось выше (в частности, в главе 16 «Цифровое мышление»), каждый новый кадр начинается, когда Unity перерисовывает экран, что обычно происходит с частотой от 30 до 200 раз в секунду.

Пример поинтереснее

Теперь добавим больше Unity-стиля в нашу первую программу. В этом примере мы создадим много-много копий кубика. Каждая копия будет подпрыгивать независимо от других, учитывая при этом законы физики. Это упражнение демонстрирует скорость работы Unity и простоту создания контента.

Начнем с создания новой сцены:

1. Выберите в меню пункт File > New Scene (Файл > Новая сцена). Вы не заметите большой разницы, потому что в предыдущей сцене _Scene_0 у нас почти ничего не было, кроме сценария, присоединенного к камере, но если теперь щелкнуть на элементе Main Camera, вы увидите, что сценарий больше не присоединен к ней и заголовок окна Unity изменился: вместо _Scene_0.unity в нем отображается текст Untitled.
2. Как обычно, первое, что нужно сделать, — сохранить новую сцену. Выберите в меню пункт File > Save Scene (Файл > Сохранить сцену) и дайте сцене имя _Scene_1.
3. Выберите в меню пункт GameObject > 3D Object > Cube (Игровой объект > 3D объект > Куб), чтобы добавить игровой объект с именем Cube в панели Scene (Сцена) и Hierarchy (Иерархия). Если вы не видите куб в панели Scene (Сцена), попробуйте дважды щелкнуть на его имени в панели Hierarchy (Иерархия), в результате чего фокус в сцене переместится на объект Cube. Прочитайте врезку «Изменение вида сцены» далее в этой главе, где описывается, как управлять видом, отображаемым в панели Scene (Сцена).
4. Щелкните на имени Cube в панели Hierarchy (Иерархия), и вы увидите, как этот объект выделился в панели Scene (Сцена), а в панели Inspector (Инспектор) появился список его компонентов (рис. 19.8). Главное назначение панели Inspector (Инспектор) — дать возможность видеть и редактировать компоненты, составляющие любой игровой объект `GameObject`. В этом игровом объекте Cube имеются компоненты: Transform, Mesh Filter, Box Collider и Mesh Renderer:

   - **Transform:** компонент Transform задает местоположение, поворот и масштаб игрового объекта. Это единственный компонент, присутствующий во всех игровых объектах.

     Пока вы еще здесь, проверьте, что координаты X, Y и Z местоположения кубика в разделе Position имеют значение 0.

   - **Cube (Mesh Filter):** компонент Mesh Filter придает игровому объекту трехмерную форму, которая моделируется как сетка (mesh) из треугольников. Трехмерные модели в играх — это обычно поверхности с пустотой внутри. В отличие от настоящего яйца (с желтком и белком внутри), трехмерная модель яйца может быть лишь сеткой, имитирующей пустую скорлупу яйца. Компонент Mesh Filter присоединяет трехмерную модель к игровому объекту. В случае с кубом Mesh Filter использует простую трехмерную модель кубика,

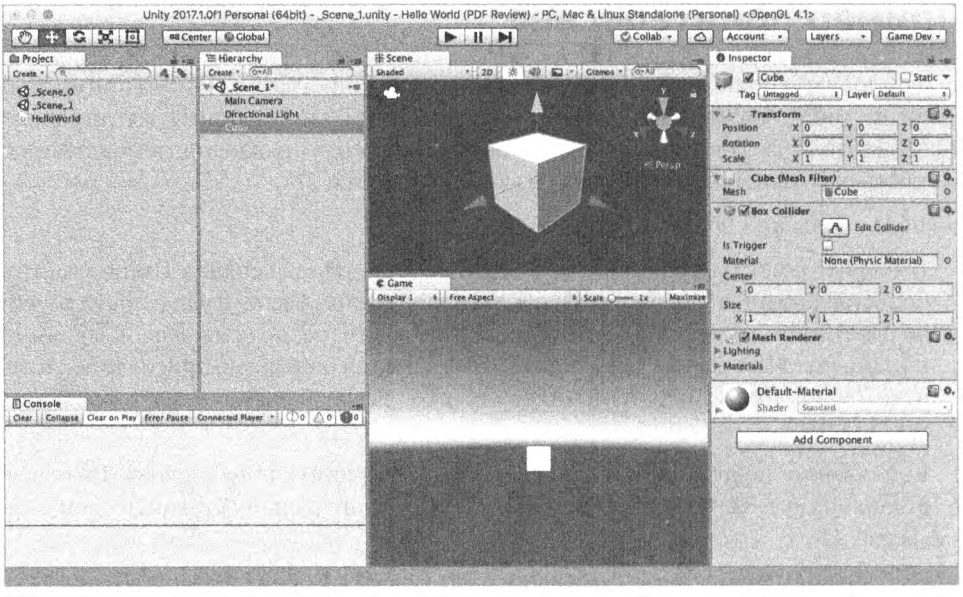


Рис. 19.8. Новый игровой объект Cube в панелях Scene (Сцена) и Hierarchy (Иерархия)

встроенную в Unity, но вообще вы можете импортировать в проект ресурсы со сложными трехмерными моделями и использовать их для создания сложных поверхностей в игре.

- **Box Collider:** компонент Collider позволяет игровому объекту `GameObject` взаимодействовать с другими объектами, моделируя действие законов физики. Физический движок PhysX, входящий в состав Unity, поддерживает коллайдеры нескольких разных форм, в том числе Sphere, Capsule, Box и Mesh (в порядке возрастания вычислительной сложности; то есть Mesh Collider требует от компьютера выполнить больше вычислений, чем Box Collider). Игровой объект с компонентом коллайдера (и без компонента твердого тела Rigidbody) действует как неподвижный объект в пространстве, с которым могут сталкиваться другие игровые объекты.
- **Mesh Renderer:** в отличие от компонента Mesh Filter, определяющего фактическую геометрию игрового объекта, компонент визуализации Mesh Renderer обеспечивает визуальное отображение этой геометрии. В Unity ничего не отображается на экране без компонента визуализации. Компоненты визуализации работают вместе с главной камерой для преобразования трехмерной геометрии Mesh Filter в пикселы на экране.

5. Теперь добавим в игровой объект еще один компонент: твердое тело Rigidbody. Пока объект Cube еще выбран в иерархии, выберите в меню пункт Component > Physics > Rigidbody (Компонент > Физика > Твердое тело), и вы увидите, как в панели Inspector (Инспектор) появится компонент Rigidbody:

- **Rigidbody:** компонент Rigidbody сообщает движку Unity, что для данного игрового объекта должно имитироваться действие законов физики. Сюда входит, например, действие сил притяжения, трения, столкновения и сопротивления. Компонент твердого тела Rigidbody позволяет игровому объекту с коллайдером перемещаться в пространстве. Без твердого тела, даже если перемещать игровой объект путем изменения его координат, компонент коллайдера будет работать ненадежно. Вы должны присоединять компоненты Rigidbody ко всем игровым объектам, чтобы обеспечить их правильное перемещение и реакцию на столкновение с другими коллайдерами.

6. Щелкните на кнопке Play (Играть); кубик упадет вниз, подчиняясь силе притяжения.

   Физическое моделирование в Unity осуществляется в метрической системе мер. Это означает, что:

   - 1 единица расстояния = 1 метр (например, координаты, описывающие местоположение).
   - 1 единица массы = 1 килограмм (например, масса твердого тела Rigidbody).
   - Ускорение свободного падения равно –9,8, то есть 9,8 м/с^2 в (отрицательном) направлении вниз.
   - Средний рост персонажа-человека составляет примерно 2 единицы (2 метра).

7. Щелкните на кнопке Play (Играть) еще раз, чтобы остановить воспроизведение.

В сцену уже включен компонент направленного освещения Directional Light. Именно поэтому кубик выглядит ярко освещенным. На данный момент это все, что вам нужно знать, более подробно об освещении будет рассказываться в следующих главах.

Создание шаблона

Пришло время превратить объект Cube в *шаблон* (prefab). Шаблон — это многократно используемый элемент в проекте, который можно использовать для создания любого количества экземпляров. Шаблон можно считать отливочной формой для игровых объектов, и любой игровой объект, полученный из шаблона, называется *экземпляром* шаблона. Чтобы создать шаблон, наведите указатель мыши на элемент Cube в панели Hierarchy (Иерархия), нажмите левую кнопку и, удерживая ее, перетащите элемент в панель Project (Проект), после чего отпустите кнопку мыши (рис. 19.9).

Вы заметите, как произойдут два изменения:

- В панели Project (Проект) появится новый шаблон с именем Cube. Шаблоны отличаются от других элементов ярлыком с изображением синего кубика. (Все шаблоны сопровождаются ярлыком с изображением кубика независимо от его фактической формы.)

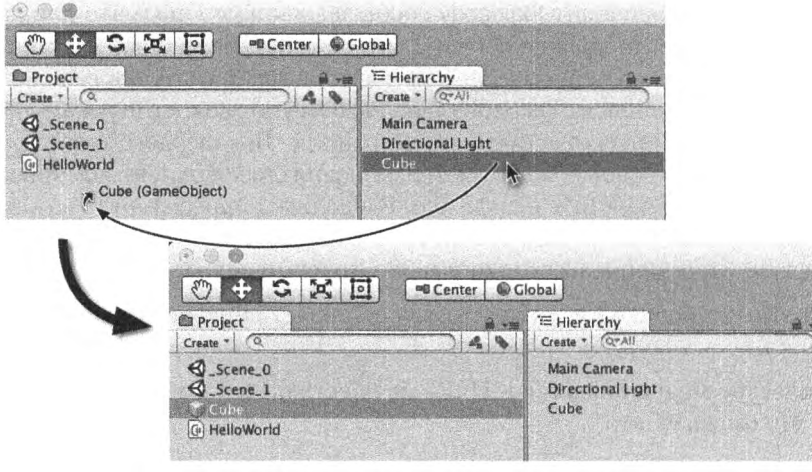


Рис. 19.9. Превращение кубика в шаблон

- Имя игрового объекта **Cube** в панели **Hierarchy** (Иерархия) окрасится в синий цвет. Если имя игрового объекта в панели **Hierarchy** (Иерархия) имеет синий цвет, это означает, что он является экземпляром шаблона (как копия, отлитая с помощью формы шаблона).

Для большей ясности переименуем шаблон кубика в панели **Project** (Проект) и дадим ему имя **Cube Prefab**.

1. Щелкните один раз на шаблоне **Cube**, чтобы выбрать его, затем щелкните второй раз, чтобы активировать режим переименования (можно также попробовать после первого щелчка нажать клавишу **Return** (или **F2** на PC)) и измените имя на **Cube Prefab**. Поскольку экземпляр шаблона в панели **Hierarchy** (Иерархия) получил имя по умолчанию **Cube**, его имя также изменится. Если бы вы переименовали экземпляр в панели **Hierarchy** (Иерархия), чтобы отличить его от шаблона, имя экземпляра не было бы затронуто.
2. Теперь, после создания шаблона, отпала надобность в присутствии его экземпляра в сцене. Щелкните на **Cube Prefab** в панели **Hierarchy** (Иерархия), не в панели **Project** (Проект)! Выберите в меню пункт **Edit > Delete** (Правка > Удалить); кубик исчезнет из сцены.

Теперь вам предстоит написать немного кода:

3. Выберите пункт меню **Assets > Create > C# Script** (Ресурсы > Создать > Сценарий C#) и сохраните вновь созданный сценарий с именем **CubeSpawner** (обратите внимание на две заглавные буквы и на отсутствие пробелов в имени).
4. Дважды щелкните на сценарии **CubeSpawner**, чтобы открыть MonoDevelop, добавьте код, выделенный жирным в следующем листинге, и сохраните сценарий:

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class CubeSpawner : MonoBehaviour {
6     public GameObject       cubePrefabVar;
7
8     // Используйте этот метод для инициализации
9     void Start () {
10        Instantiate( cubePrefabVar );
11    }
12
13    // Update вызывается в каждом кадре
14    void Update () {
15
16    }
17 }
```

Кроме добавления пробелов, как упоминалось в предыдущем примечании, некоторые версии MonoDevelop также удаляют лишние пробелы, когда вы добавляете точку с запятой в конец строки или когда нажимаете Return/Enter после этого. Это тоже нормально. В моих листингах вы часто будете видеть строки — как строка 6 в предыдущем листинге, где я добавил несколько табуляций перед именем переменной/поля (например, `cubePrefabVar`). Я поступаю так, потому что считаю, что выравнивание имен полей по вертикали упрощает чтение кода, но иногда в процессе ввода MonoDevelop будет удалять эти дополнительные пробелы или табуляции, добавленные для форматирования. Не волнуйтесь об этом: наличие или отсутствие таких дополнительных пробелов никак не влияет на код.

5. Так же как предыдущий сценарий, этот тоже надо присоединить к игровому объекту в сцене, чтобы его код выполнился. В Unity перетащите сценарий CubeSpawner на объект Main Camera, как показано на рис. 19.4.
6. Щелкните на элементе Main Camera в панели Hierarchy (Иерархия). Вы увидите, что к этому игровому объекту присоединился компонент Cube Spawner (Script) (рис. 19.10).

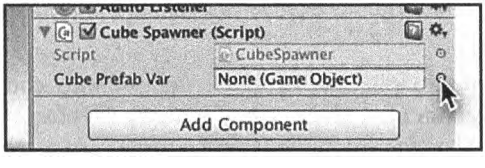


Рис. 19.10. Компонент сценария CubeSpawner в панели Inspector (Инспектор) со списком компонентов для главной камеры

В этом компоненте можно также заметить поле с именем Cube Prefab Var (которое в действительности должно иметь имя `cubePrefabVar`, причину такого отличия я объясню чуть позже). Оно является отражением инструкции `public GameObject`

cubePrefabVar; в строке 6. В общем случае если переменная в сценарии отмечена спецификатором public, она появится как поле в панели Inspector (Инспектор).

ИМЕНА ПЕРЕМЕННЫХ ВЫГЛЯДЯТ ИНАЧЕ В ПАНЕЛИ ИНСПЕКТОРА. Кто-то в команде разработчиков Unity решил, что имена переменных в панели Inspector (Инспектор) будут смотреться лучше, если в них добавить пробелы и изменить регистр символов. Я не знаю, почему эта традиция продолжилась в текущей версии, но это означает, что имена ваших переменных, такие как cubePrefabVar, будут неправильно отображаться в инспекторе, например: Cube Prefab Var. Будьте внимательны и всегда используйте правильные имена переменных в сценариях и игнорируйте лишние пробелы и неверный регистр символов, которые наблюдаются в инспекторе. На протяжении всей книги я буду ссылаться на такие имена переменных, какие записаны в коде, а не какие отображаются в инспекторе.

7. Как можно заметить в инспекторе, переменной cubePrefabVar в настоящее время не присвоено никакого значения. Щелкните на маленьком изображении мишени (под указателем мыши на рис 19.10) справа от поля ввода для переменной cubePrefabVar, чтобы открыть диалог Select GameObject (Выбор игрового объекта), где можно выбрать шаблон для присваивания этой переменной. Обязательно перейдите на вкладку Assets (Ресурсы). (Вкладка Assets отображает игровые объекты, перечисленные в панели Project (Проект), тогда как вкладка Scene (Сцена) отображает игровые объекты из панели Hierarchy (Иерархия).) Щелкните дважды на шаблоне Cube Prefab, чтобы выбрать его (рис. 19.11).

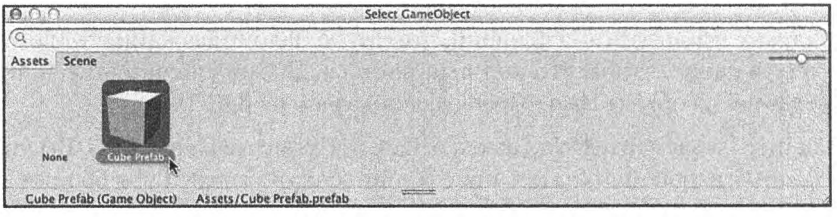


Рис. 19.11. Выбор шаблона Cube Prefab для присваивания переменной cubePrefabVar в сценарии CubeSpawner

8. Теперь в инспекторе можно заметить, что переменная cubePrefabVar получила значение Cube Prefab из панели Project (Проект). Чтобы лишний раз убедиться в этом, щелкните на значении Cube Prefab в инспекторе, и вы увидите, как Cube Prefab подсветится желтым цветом в панели Project (Проект).
9. Щелкните на кнопке Play (Играть). Вы увидите, как в иерархии появится единственный экземпляр Cube Prefab (Clone) игрового объекта. Как вы уже знаете из опыта создания сценария Hello World, функция Start() вызывается только один раз, и она создает единственный экземпляр (или клон, копию) шаблона Cube Prefab.

10. Теперь перейдите в окно MonoDevelop, закомментируйте вызов `Instantiate( cubePrefabVar )` в строке 10, в функции `Start()`, и добавьте инструкцию `Instantiate( cubePrefabVar );` в строку 15, в функцию `Update()`, как показано ниже.

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class CubeSpawner : MonoBehaviour {
 6     public GameObject       cubePrefabVar;
 7
 8     // Используйте этот метод для инициализации
 9     void Start () {
10         // Instantiate( cubePrefabVar );
11     }
12
13     // Update вызывается в каждом кадре
14     void Update () {
15         Instantiate( cubePrefabVar );
16     }
17 }
```

11. Сохраните сценарий `CubeSpawner`, вернитесь в Unity и щелкните на кнопке **Play** (Играть) снова. Это приведет к появлению множества кубиков, как показано на рис. 19.12[1].

Этот пример наглядно демонстрирует мощь Unity. Вы сумели быстро включиться в работу и создать нечто, вызывающее интерес. Теперь добавим в сцену еще несколько объектов, взаимодействующих с кубиками:

1. Щелкните на кнопке **Play** (Играть) еще раз, чтобы остановить воспроизведение.
2. В панели **Hierarchy** (Иерархия) щелкните на кнопке вызова меню **Create** (Создать) и выберите пункт **3D Object > Cube** (3D Объект > Куб). Дайте этому кубу имя **Ground**.
3. Выберите объект **Ground** в панели **Scene** (Сцена) или **Hierarchy** (Иерархия), нажмите клавишу **W**, **E** или **R**, чтобы переместить, повернуть или изменить масштаб игрового объекта. В результате вокруг объекта **Ground** появятся дополнительные графические элементы управления (стрелки, окружности и др., как показано на рис. 19.13).

В режиме перемещения (клавиша **W**) перетаскивание мышью одной из стрелок приводит к перемещению куба вдоль соответствующей оси (X, Y или Z). Цветные элементы, управляющие поворотом и масштабированием, аналогично выполняют

[1] Возможно, вам интересно, почему кубики разлетаются повсюду, а не падают сразу вниз. Хороший вопрос! Дело в том, что кубики создаются в одной точке пространства, и физическая система PhysX решает сместить их друг относительно друга (потому что коллайдеры в Unity не могут занимать одну и ту же область пространства), придавая им некоторую скорость, что заставляет кубики быстро разлетаться друг от друга.

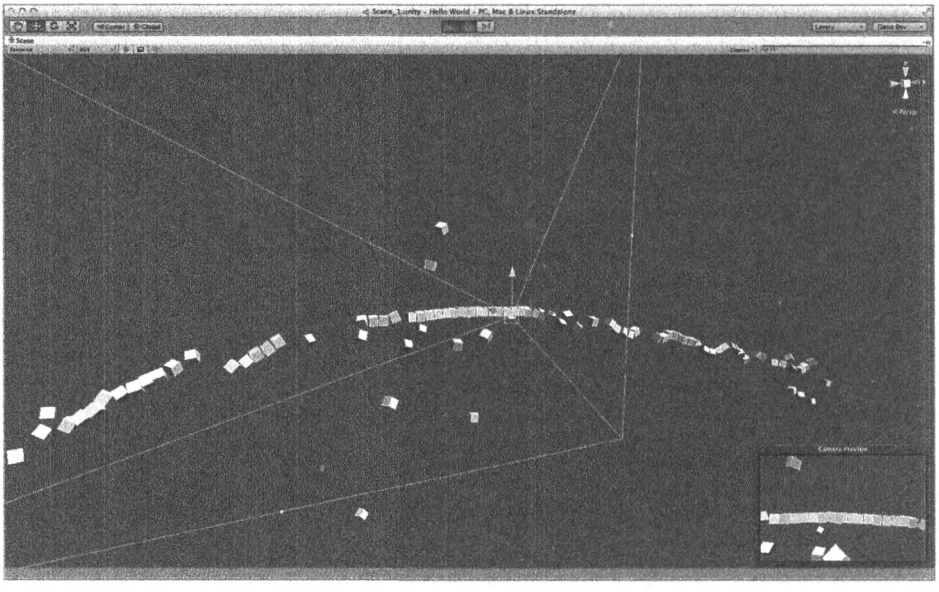


Рис. 19.12. Создание новых экземпляров шаблона CubePrefab в каждом вызове Update() быстро наполняет пространство кубиками!

назначенные им действия вдоль соответствующей оси. Подробнее об использовании инструментов, изображенных на рис. 19.13, вы узнаете во врезке «Изменение вида сцены».

4. Попробуйте переместить **Ground** в позицию с координатой Y, равной −4, и изменить его масштаб по осям X и Z до 10. Далее в книге я буду предлагать установить координаты, поворот и масштаб, используя такой формат.

```
Ground (Cube)     P:[ 0, -4, 0 ]     R:[ 0, 0, 0 ]     S:[ 10, 1, 10 ]
```

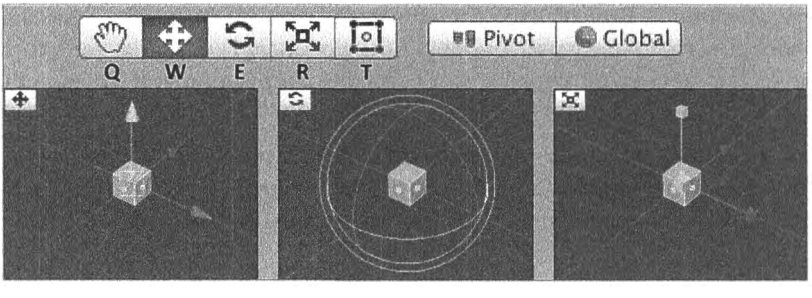


Рис. 19.13. Инструменты, управляющие перемещением, поворотом и масштабом. Q, W, E, R и T — это клавиши выбора каждого инструмента. Инструмент T используется для позиционирования двумерных игровых объектов и элементов графического интерфейса с пользователем

Здесь *Ground* — имя игрового объекта, а *(Cube)* — его тип. *P:[0, –4, 0]* означает, что координата X объекта должна иметь значение 0, координата Y — значение –4 и координата Z — значение 0. Аналогично *R:[0, 0, 0]* означает, что углы поворота относительно всех трех осей — X, Y и Z — должны быть равны 0. *S:[10, 1, 10]* означает, что масштаб вдоль оси X должен иметь значение 10, вдоль оси Y — значение 1 и вдоль оси Z — значение 10. Вы можете использовать инструменты и элементы управления, изображенные на рис. 19.13, или просто ввести предложенные значения в компонент Transform игрового объекта в панели Inspector (Инспектор).

Вы можете пока отложить книгу, поэкспериментировать со всем этим и добавить еще объекты. Экземпляры Cube Prefab будут отскакивать от *статических* объектов, которые вы поместите в сцену (рис. 19.14). Пока вы не добавите компонент Rigidbody в любую новую фигуру, она будет оставаться статической (то есть жесткой и неподвижной). Закончив, не забудьте сохранить сцену!

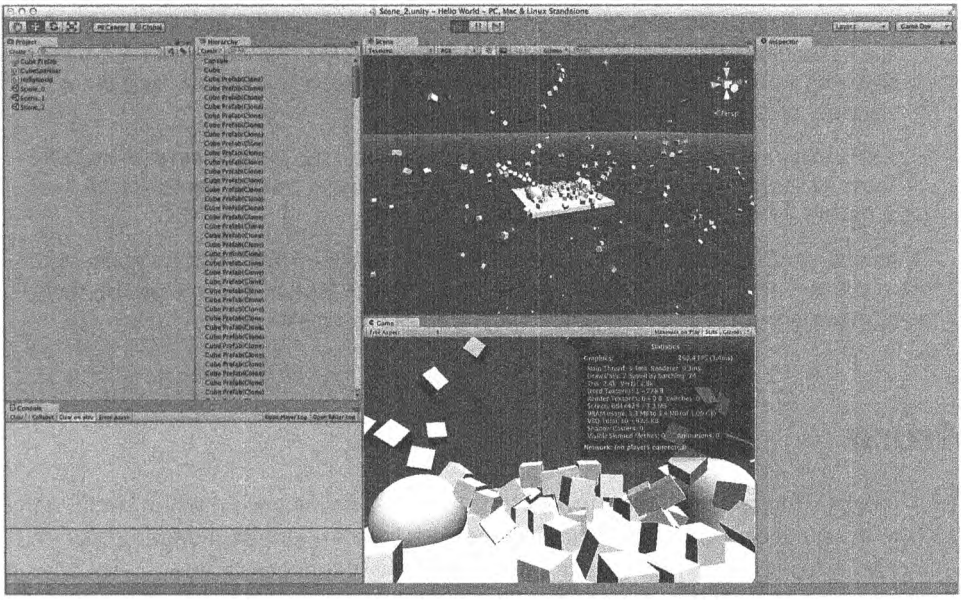


Рис. 19.14. Сцена с добавленными в нее статическими фигурами

ИЗМЕНЕНИЕ ВИДА СЦЕНЫ

Первый инструмент в панели на рис. 19.13, известный как *инструмент «рука»*, используется для управления видом, отображаемым в панели Scene (Сцена). Панель Scene (Сцена) имеет свою невидимую *камеру сцены*, отличную от главной камеры Main Camera в панели Hierarchy (Иерархия). Инструмент «рука» поддерживает выполнение нескольких разных операций. Выберите этот инструмент (щелкнув

на нем мышью или нажав клавишу Q на клавиатуре) и попробуйте выполнить следующие действия:

- Перетаскивание с левой кнопкой (то есть когда нажимается левая кнопка мыши и выполняется смещение указателя) в панели Scene (Сцена) вызывает изменение позиции камеры сцены без изменения местоположения других объектов в сцене. Технически камера сцены перемещается в плоскости, перпендикулярной направлению съемки камерой (то есть перпендикулярной вектору «вперед» камеры).
- Перетаскивание с правой кнопкой в панели Scene (Сцена) поворачивает камеру сцены вслед за указателем мыши. При перетаскивании с правой кнопкой позиция камеры не изменяется.
- Удержание клавиши Option (или Alt на PC) изменяет вид указателя мыши над панелью Scene (Сцена): изображение руки сменяется изображением глаза, а перетаскивание с левой кнопкой и удерживаемой клавишей Option заставляет вид сцены поворачиваться вокруг объектов в панели Scene (Сцена) — этот прием называют вращением камеры вокруг сцены. При перетаскивании с левой кнопкой и клавишей Option позиция камеры изменяется, но точка, куда смотрит камера, остается прежней.
- Скроллинг колесиком мыши заставляет камеру сцены увеличивать или уменьшать масштаб сцены. Управлять масштабом можно также перетаскиванием с правой кнопкой и клавишей Option.

Лучший способ понять, как работает инструмент «рука», — попробовать применить его в сцене с использованием разных методов, описанных в этой врезке. Немного поэкспериментировав, вы быстро овладеете этим инструментом.

Итоги

Примерно за 20 страниц вы из ничего получили работающий проект Unity, написав при этом совсем немного кода. Конечно, это был очень маленький проект, но я надеюсь, что он достаточно наглядно показал вам скорость работы Unity, а также скорость, с которой вы можете создавать что-нибудь работающее.

Следующая глава продолжит введение в C# и Unity, познакомив вас с переменными и расширив ваши познания о наиболее распространенных компонентах, которые можно добавлять в игровые объекты `GameObject`.

20 Переменные и компоненты

Эта глава познакомит вас со многими переменными и типами компонентов, используемыми повсюду в программировании на C# для Unity. К концу этой главы вы познакомитесь с некоторыми распространенными типами переменных в языке C# и с некоторыми важными типами, уникальными для Unity.

Эта глава также знакомит с игровыми объектами и компонентами. Любой объект в сцене в Unity является игровым объектом `GameObject`, а компоненты, входящие в состав игрового объекта, обеспечивают все его возможности, от позиционирования до моделирования физики, создания специальных эффектов, отображения трехмерной модели на экране, анимации и многого другого.

Введение в переменные

Как рассказывалось в главе 18 «Знакомство с нашим языком: C#», *переменная* — это лишь имя, которому можно присвоить определенное значение. Эта идея восходит к начальному курсу алгебры. В алгебре, например, можно записать такое определение:

```
x = 5
```

Оно *определяет переменную* `x` со *значением* 5. Иначе говоря, оно присваивает значение 5 имени `x`. Позднее, встретив определение:

```
y = x + 2
```

вы сможете вычислить значение переменной `y`, равное 7 (потому что x = 5, а 5 + + 2 = 7). `x` и `y` называются *переменными*, так как их значения можно в любой момент переопределить, и порядок следования этих переменных имеет значение. Взгляните на следующие определения. (Я добавил комментарии, начинающиеся с пары символов слеша [//], чтобы пояснить действие каждой инструкции.)

```
x = 10     // x получает значение 10
y = x - 4  // y получает значение 6, потому что 10-4 = 6
x = 12     // x получает значение 12, но y сохраняет значение 6
z = x + 3  // z получает значение 15, потому что 12+3 = 15
```

После этой последовательности определений переменные x, y и z получают значения 12, 6 и 15 соответственно. Как видите, изменение значения x не влияет на значение y, потому что переменная y получила значение 6 до того, как переменной x было присвоено новое значение 12, а значение y не имеет обратной силы.

Строго типизированные переменные в C#

Переменные в C# строго типизированы, и им нельзя присваивать произвольные значения, то есть они могут иметь значения только определенных типов. Это необходимо, потому что компьютер должен знать, сколько памяти выделить для каждой переменной. Большое изображение может занимать много мегабайтов и даже гигабайтов пространства, тогда как для хранения логического значения (которое может быть 1 или 0) в действительности достаточно одного бита. Даже один мегабайт содержит 8 388 608 бит!

Объявление переменных и присваивание им значений в C#

В C# нужно объявить переменную и присвоить ей значение, чтобы она имела практическую ценность.

Объявление создает переменную и задает ее имя и тип, но не присваивает ей значения (хотя для переменных некоторых простых типов предусмотрены значения по умолчанию).

```
bool bravo;    // Объявление переменной с именем bravo и типом bool (логический)
int india;     // Объявление переменной с именем india и типом int (целочисленный)
float foxtrot; // Объявление переменной с именем foxtrot и типом float
                  (вещественный)
char charlie;  // Объявление переменной с именем charlie и типом char (символьный)
```

Присваивание придает переменной значение. Вот несколько примеров использования объявленных переменных:

```
bravo = true;
india = 8;
foxtrot = 3.14f; // Символ f указывает, что литерал числа имеет тип float,
                    как описывается ниже
charlie = 'c';
```

Любое конкретное значение в коде (например, `true`, `8` или `'c'`) называется *литералом*. В предыдущем листинге `true` — это литерал типа `bool`, `8` — литерал типа `int` (целое число), `3.14f` — литерал типа `float` и `'c'` — литерал типа `char`. По умолчанию MonoDevelop отображает литералы ярко-оранжевым цветом (хотя на некоторых компьютерах по непонятным причинам значение `true` отображается сине-зеленым цветом), и для каждого типа действуют определенные правила пред-

ставления литералов. Больше информации о каждом типе переменных вы узнаете в следующих разделах.

Объявление перед присваиванием

Прежде чем присвоить значение переменной, ее нужно объявить, хотя часто эти две операции выполняются в одной строке:

```
string sierra = "Mountain";
```

Инициализация переменных в C# перед обращением к ним

Первое присваивание значения переменной называют ее *инициализацией*. Некоторые простые типы переменных (такие, как `bool`, `int`, `float` и другие в примерах выше) получают значение по умолчанию в момент объявления (соответственно `false`, `0` и `0f`). Переменные сложных типов (например, `GameObject`, `List`, и др.) по умолчанию получают значение `null`, сообщающее о неинициализированном состоянии, и не могут полноценно использоваться до их инициализации.

Вообще, даже если простая переменная имеет значение по умолчанию, Unity будет предупреждать и выводить сообщения компилятора об ошибках[1] при попытке обратиться (то есть прочитать) к переменной, объявленной, но не инициализированной.

Важные типы переменных в C#

В C# доступно несколько разных типов переменных. Ниже перечисляются наиболее важные из них, с которыми вы будете встречаться чаще всего. Имена всех этих основных типов переменных начинаются со строчной буквы, тогда как большинство имен типов данных в Unity начинаются с прописной буквы. Для каждого типа я привожу некоторую основную информацию и даю пример объявления и определения переменной.

bool: 1-битное значение «ложь» или «истина»

Имя `bool` — сокращенная форма от Boolean (логический). В своей основе все переменные состоят из битов, которые могут принимать ложное (false) или истинное (true) значение. Тип `bool` имеет размер в 1 бит, то есть переменные этого типа за-

[1] В главе 18 «Знакомство с нашим языком: C#» вы узнали, что C# — компилируемый язык. Ошибки компилятора — это ошибки, обнаруживаемые в процессе компиляции, когда Unity пытается интерпретировать код на C#, написанный вами. Подробнее об ошибках и их типах рассказывается в главе 25 «Отладка».

нимают меньше всего места в памяти[1]. Логические переменные необычайно полезны в логических операциях, например в инструкциях `if` и других условных операторах, которые подробнее описываются в двух следующих главах. В языке C# литералы типа `bool` представлены двумя словами: `true` и `false`:

```
bool verified = true;
```

int: 32-битное целое

Переменная типа `int` (сокращенно от integer — целое число) может хранить единственное целое число (целые числа — это числа, не имеющие дробной части, такие как `5`, `2` или `-90`). Целочисленная математика очень точная и *очень* быстрая. Переменная типа `int` в Unity может хранить числа в диапазоне от −2 147 483 648 до 2 147 483 647 *включительно* (то есть любое из этих двух граничных значений, а не только числа между ними) и использует 1 бит для представления знака и 31 бит для представления его значения:

```
int nonFractionalNumber = 12345;
```

float: 32-битное десятичное число

Вещественное число (число с плавающей запятой[2]), или тип `float`, — это наиболее распространенная форма представления десятичных чисел в Unity. Называется «с плавающей запятой», потому что хранится с использованием системы, напоминающей *научную форму записи*. Научная форма записи представляет числа в формате $a \times 10^b$ (например, число 300 можно записать, как 3×10^2, а число 12 345 — как $1{,}2345 \times 10^4$). Числа с плавающей запятой хранятся в похожем формате, как $a \times 2^b$. Когда число типа `float` сохраняется в памяти компьютера, 1 бит отводится для представления знака числа, 23 бита — для мантиссы (часть a числа) и 8 бит — для экспоненты (часть b). Такой способ хранения не позволяет точно представлять очень большие числа и числа в диапазоне между 1 и −1, которые сложно представить как степень двойки. Например, число 1/3 нельзя точно представить в виде числа с плавающей запятой[3].

В большинстве случаев неточная природа чисел с плавающей запятой не имеет большого значения для игр, но она может стать причиной множества мелких ошибок, например, в определении моментов столкновения; поэтому использование в игре объектов с размерами больше одной единицы и меньше нескольких тысяч единиц

---

[1] Из-за особенностей работы с памятью в современных компьютерах и в C# переменная типа `bool` в действительности занимает от 32 до 64 бит памяти, но вообще для хранения логического значения вполне достаточно единственного бита.

[2] https://ru.wikipedia.org/wiki/Число_с_плавающей_запятой

[3] Недостаточная точность чисел с плавающей запятой также является причиной, почему координаты и углы вращения в компоненте Transform иногда выглядят как очень сложные числа, близкие к нулю, хотя должны быть равны нулю.

обеспечит чуть более высокую точность определения столкновений. Литералы типа `float` должны быть целыми или десятичными числами и завершаться символом `f`. Это связано с тем, что любой литерал десятичного числа без завершающего символа `f` интерпретируется компилятором C# как значение типа `double` (представляет вещественные числа с двойной точностью), в противоположность типу `float` с одинарной точностью, который используется в Unity. Во всех встроенных функциях Unity вместо `double` используется тип `float`, что обеспечивает более высокую скорость вычислений, хотя и за счет меньшей точности:

```
float notPreciselyOneThird = 1.0f/3.0f;
```

Один из способов справиться с невысокой точностью типа `float` — использовать функцию сравнения `Mathf.Approximately()`, описанную в разделе «Mathf: библиотека математических функций» далее в этой главе. Она возвращает `true`, если два значения с плавающей запятой достаточно близки друг к другу.

Если вы встретитесь со следующей ошибкой при компиляции своего кода

```
error CS0664: Literal of type double cannot be implicitly converted to type
'float'. Add suffix 'f' to create a literal of this type
```

```
(error CS0664: Литерал с типом double нельзя неявно преобразовать в тип 'float'.
Добавьте суффикс 'f', чтобы создать литерал этого типа)
```

значит, где-то вы забыли добавить `f` после литерала числа с плавающей запятой.

char: 16-битный символ

Тип `char` представляет единственный символ, используя 16 бит информации. Символы в диалекте C# для Unity хранятся в виде Юникода (Unicode)[1], что позволяет представлять более 110 000 разных символов из более чем 100 разных алфавитов и языков (включая, например, все символы упрощенного китайского письма). Литералы символов в коде заключаются в одиночные кавычки (апострофы):

```
char theLetterA = 'A';
```

string: последовательности 16-битных символов

Тип `string` используется для представления всего, что только можно, от одиночного символа до текста целой книги. Теоретическая длина строки в C# превышает 2 миллиарда символов, но большинство компьютеров столкнется с проблемой исчерпания памяти задолго до того, как будет достигнуто это ограничение. Чтобы вы могли получить хоть какое-то представление об этой длине, отмечу, что полная версия пьесы Шекспира «Гамлет» содержит чуть больше 175 000 символов[2],

[1] https://ru.wikipedia.org/wiki/Юникод

[2] http://shakespeare.mit.edu/hamlet/full.html

включая режиссерские ремарки, разрыв строк и т. п. Это значит, что пьеса «Гамлет» уместилась бы в одну строку типа `string` более 12 000 раз. Литералы строк заключаются в двойные кавычки:

```
string theFirstLineOfHamlet = "Who's there?";
```

Квадратные скобки и строки

Для доступа к отдельным символам строк можно использовать *квадратные скобки*:

```
char theCharW = theFirstLineOfHamlet[0]; // W - это 0-й символ строки
char theChart = theFirstLineOfHamlet[6]; // t - это 6-й символ строки
```

Добавление числа в квадратных скобках после имени переменной позволяет извлечь символ в этой позиции в строке (не влияя на саму строку). Нумерация позиций символов в строке начинается с нуля; соответственно, в 0-й позиции в первой строке пьесы «Гамлет» находится символ `W`, а в 6-й — символ `t`. Подробнее об использовании квадратных скобок для доступа к элементам коллекций рассказывается в главе 23 «Коллекции в C#».

Если вы увидите любую из следующих ошибок времени компиляции

```
error CS0029: Cannot implicitly convert type 'string' to 'char'
error CS0029: Cannot implicitly convert type 'char' to 'string'
error CS1012: Too many characters in character literal
error CS1525: Unexpected symbol '<internal>'
```

```
(error CS0029: Невозможно неявно преобразовать тип 'string' в 'char'
error CS0029: Невозможно неявно преобразовать тип 'char' в 'string'
error CS1012: Слишком много символов в символьном литерале
error CS1525: Неожиданный символ '<internal>')
```

это обычно означает, что где-то по ошибке использованы двойные кавычки (`" "`) для символьного литерала или одиночные кавычки (`' '`) для строкового литерала. Строковые литералы всегда должны заключаться в двойные кавычки, а символьные — в одиночные.

class: определение нового типа переменных

Ключевое слово `class` определяет новый тип переменных, который можно считать комбинацией переменных и функций. Все типы переменных и компоненты в Unity, перечисленные в разделе «Важные типы переменных в Unity» далее в этой главе, являются примерами классов. Более подробно о классах рассказывается в главе 26 «Классы».

Область видимости переменных

Еще одним важным понятием, кроме типа, является *область видимости* переменной. Под областью видимости подразумевается область кода, в которой переменная

существует и доступна. Переменная, объявленная в одной части кода, может быть недоступна в другой. Я буду пояснять эту сложную проблему на протяжении всей книги. Если вы хотите освоить ее постепенно, просто продолжайте читать книгу по порядку. Но если вы хотите получить больше информации об области видимости переменных прямо сейчас, прочитайте раздел «Область видимости переменных» в приложении Б «Полезные идеи».

Соглашения об именах

Код в книге следует целому ряду правил, управляющих выбором имен для переменных, функций, классов и т. д. Ни одно из этих правил не является обязательным, но следуя им, вы сделаете свой код более удобочитаемым не только для тех, кто будет пытаться расшифровать его, но и для вас самих, если спустя месяцы вам придется вернуться к нему и понять, что вы написали. Каждый программист использует в своей практике немного отличающиеся правила — даже мои личные правила менялись с течением времени, но правила, которые я хочу представить здесь, оказались весьма эффективными для меня и моих студентов, и они совместимы с большинством кода на C#, который мне довелось встречать в Unity.

ВЕРБЛЮЖИЙ РЕГИСТР

ВерблюжийРегистр — распространенный в программировании способ записи имен переменных. Он позволяет программисту и любому, кто будет читать его код, легко разбирать длинные имена переменных. Вот несколько примеров:

- `aVeryLongNameThatIsEasierToReadBecauseOfCamelCase`
- `variableNamesStartWithALowerCaseLetter`
- `ClassNamesStartWithACapitalLetter`

Главная особенность верблюжьегоРегистра — возможность объединить несколько слов в одно, отметив заглавной буквой начало каждого исходного слова. Она называется верблюжьимРегистром, потому что немного напоминает горбы на спине верблюда.

- Используйте *верблюжийРегистр* для любых имен (см. врезку «Верблюжий Регистр»).
- Имена переменных должны начинаться с символа нижнего регистра (например, `someVariableName`).
- Имена функций должны начинаться с символа верхнего регистра (например, `Start()`, `Update()`).
- Имена классов должны начинаться с символа верхнего регистра (например, `GameObject`, `ScopeExample`).

- Имена приватных переменных часто начинают с подчеркивания (например, `_hiddenVariable`).
- Имена статических переменных часто состоят только из символов верхнего регистра и записываются с применением «змеиной_нотации» (например, `NUM_INSTANCES`). В змеиной_нотации слова, составляющие имя, объединяются символом подчеркивания.

Для справки: я повторю и дополню эту информацию в разделе «Соглашения об именах» в приложении Б.

Важные типы переменных в Unity

Unity определяет большое количество типов переменных, которые будут встречаться вам почти в каждом проекте. Все эти типы в действительности являются классами, и их имена следуют соглашениям, принятым в Unity, когда все имена классов начинаются с заглавной буквы.[1] Для каждого типа переменных, реализованного в Unity, вы увидите информацию, описывающую порядок создания новых экземпляров этого класса (см. врезку, рассказывающую об экземплярах класса), сопровождаемую списками важных переменных и функций этого типа данных. В большинстве классов, перечисленных в этом разделе, переменные и функции разбиты на две группы:

- **Переменные и функции экземпляра:** эти переменные и функции непосредственно связаны с единственным экземпляром типа. Заглянув в описание `Vector3`, следующее далее, вы увидите, что `x`, `y`, `z` и `magnitude` — это переменные экземпляра типа `Vector3` и доступ к каждому из них осуществляется с использованием имени переменной, точки и имени переменной экземпляра (например, `position.x`). Экземпляры `Vector3` могут иметь разные значения этих переменных. Аналогично, функция `Normalize()` воздействует на единственный экземпляр типа `Vector3` и записывает значение 1 в переменную `magnitude` этого экземпляра. Переменные экземпляра часто называют *полями*, а функции экземпляра — *методами*.
- **Статические переменные и функции класса:** статические переменные связаны с определением самого класса, а не с его отдельными экземплярами. Они часто используются для хранения информации, общей для всех экземпляров класса (например, `Color.red` всегда хранит значение красного (red) цвета), или обрабатывают несколько экземпляров класса, не воздействуя на них (например, `Vector3.Cross(v3a, v3b)` вычисляет векторное произведение двух векторов типа `Vector3` и возвращает новое значение типа `Vector3`, не изменяя `v3a` и/или `v3b`).

[1] Вернее, некоторые из этих типов переменных в Unity являются классами, а другие — структурами. Структуры во многом напоминают классы, но вам не придется писать их здесь, поэтому я решил назвать все типы классами.

За более полной информацией обо всех типах в Unity обращайтесь к документации, ссылки на которую вы найдете в сносках.

ЭКЗЕМПЛЯРЫ КЛАССА И СТАТИЧЕСКИЕ ФУНКЦИИ

Подобно шаблонам (prefabs), с которыми вы познакомились в главе 19 «Hello World: ваша первая программа», классы тоже могут иметь *экземпляры*. Экземпляр любого класса — это объект данных, имеющий тип, определяемый классом.

Например, можно определить класс `Human`, и тогда все, кого вы знаете, могли бы быть его экземплярами. Некоторые функции определены для всех людей (например, `Eat()`, `Sleep()`, `Breathe()`).

Так же как все люди, окружающие вас, имеют свои особенности, все экземпляры класса отличаются друг от друга. Даже если два экземпляра имеют совершенно одинаковые значения в своих переменных, они хранятся в разных областях памяти компьютера и считаются двумя разными объектами. (Продолжая аналогию с человеком, представьте двух одинаковых близнецов.) Экземпляры класса передаются *по ссылке*, а не по значению. То есть если вы будете сравнивать два экземпляра класса, чтобы узнать, не одинаковые ли они, сравниваться будут их *адреса в памяти*, а не значения переменных (так же, как два одинаковых близнеца — это два разных человека, несмотря на то что имеют одну и ту же ДНК).

Также есть возможность сослаться на один и тот же экземпляр с помощью разных переменных. Подобно тому как я могу называть свою дочь «дочкой», а мои родители — «внучкой», ссылку на экземпляр класса можно присвоить переменным с разными именами, но это будет один и тот же объект данных, как показано в следующем листинге:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Определение класса Human
public class Human {
    public string name;
    public Human partner;
}

public class Family : MonoBehaviour {
    // объявление общедоступных переменных
    public Human husband;
    public Human wife;

    void Start() {
        // Начальное состояние
        husband = new Human();
        husband.name = "Jeremy Gibson";
        wife = new Human();
        wife.name = "Melanie Schuessler";

        // Мы, я и моя жена, женаты
```

```
24          husband.partner = wife;
25          wife.partner = husband;
26
27          // Мы изменили свои имена
28          husband.name = "Jeremy Gibson Bond";
29          wife.name = "Melanie Schuessler Bond";
30
31          // Так как wife.partner ссылается на тот же экземпляр, что
32          // и husband, переменная name в wife.partner также изменила
            // значение
33          print(wife.partner.name);
34          // выведет "Jeremy Gibson Bond"
35      }
36  }
```

В классе Human также можно создать *статические функции*, использующие один или более экземпляров класса. Статическая функция Marry() позволяет связать брачными узами двух людей, как показано в следующем листинге.

```
33          print(wife.partner.name);
34          // выведет "Jeremy Gibson Bond"
35      }
36      // Этот код нужно вставить в предыдущий листинг между строками 35 и 36
37      // В результате строка 36 станет строкой 42
38      static public void Marry(Human h0, Human h1) {
39          h0.partner = h1;
40          h1.partner = h0;
41      }
42  }
```

После добавления этой функции можно заменить код в строках 23 и 24 в первом листинге единственной строкой Human.Marry(wife, husband);. Так как Marry() — это статическая функция, ее можно использовать почти повсюду. Далее в книге вы узнаете больше о статических функциях и переменных.

Vector3: коллекция из трех значений типа float

Vector3[1] — очень распространенный тип данных для вычислений в трехмерном пространстве. Чаще всего в Unity он используется для хранения трехмерных координат объектов. Пройдите по ссылке в сноске, чтобы узнать больше о типе Vector3.

```
Vector3 position = new Vector3( 0.0f, 3.0f, 4.0f ); // Устанавливает значения x, y и z
```

Переменные и функции экземпляра Vector3

Каждый экземпляр класса Vector3 содержит также несколько удобных встроенных значений и функций:

[1] http://docs.unity3d.com/Documentation/ScriptReference/Vector3.html

```
print( position.x ); // 0.0, Значение x экземпляра Vector3
print( position.y ); // 3.0, Значение y экземпляра Vector3
print( position.z ); // 4.0, Значение z экземпляра Vector3
print( position.magnitude ); // 5.0, Расстояние экземпляра Vector3 от 0,0,0
                             // другое название для magnitude - "length" (длина).
position.Normalize(); // Устанавливает величину magnitude экземпляра position
                      // равной 1 в том смысле, что переменные x, y и z экземпляра
                      // position теперь имеют значения [0.0, 0.6, 0.8]
```

Статические переменные и функции экземпляра Vector3

Кроме того, сам класс `Vector3` имеет несколько статических переменных и функций:

```
print( Vector3.zero );      // (0,0,0), Краткая форма для: new Vector3( 0, 0, 0 )
print( Vector3.one );       // (1,1,1), Краткая форма для: new Vector3( 1, 1, 1 )
print( Vector3.right );     // (1,0,0), Краткая форма для: new Vector3( 1, 0, 0 )
print( Vector3.up );        // (0,1,0), Краткая форма для: new Vector3( 0, 1, 0 )
print( Vector3.forward );   // (0,0,1), Краткая форма для: new Vector3( 0, 0, 1 )
Vector3.Cross( v3a, v3b ); // Вычисляет векторное произведение двух экземпляров
                           Vector3
Vector3.Dot( v3a, v3b );   // Вычисляет скалярное произведение двух экземпляров
                           Vector3
```

Это лишь часть полей и методов класса `Vector3`. Познакомиться с другими можно в документации к Unity по ссылке в сноске.

Color: цвет с информацией о прозрачности

Переменная типа `Color`[1] может хранить информацию о цвете и прозрачности (альфа-значение). Цвета в компьютерах получаются смешиванием трех основных цветов: красного, зеленого и синего. Этот перечень отличается от основных цветов, используемых при смешивании красок, с которыми вы могли познакомиться еще в детстве (красный, желтый и синий), потому что на экране компьютера цвета *складываются*, а не *вычитаются*. В системе с вычитанием цветов, как при использовании красок, смешивание все большего и большего количества цветов смещает результат в сторону черного (или, в действительности, к темному грязно-коричневому). В системе со сложением цветов (как на экране компьютера, в театральном освещении или в HTML-цветах в интернете), напротив, смешивание все большего и большего количества цветов смещает результат ко все более ярким оттенкам и в конечном итоге приводит к белому. Красный, зеленый и синий компоненты цвета в C# хранятся как значения типа `float` в диапазоне от `0.0f` до `1.0f`, где `0.0f` обозначает отсутствие данного цвета, а `1.0f` — максимальную его интенсивность. Четвертое значение типа `float` с именем `alpha` определяет прозрачность цвета типа `Color`. Цвет со значением `alpha`, равным `0.0f`, полностью прозрачен, а цвет со значением `alpha`, равным `1.0f`, полностью непрозрачен:

[1] http://docs.unity3d.com/Documentation/ScriptReference/Color.html

```
// Цвета определяются float-значениями красного, зеленого и синего компонентов,
// а также альфа-канала
Color darkGreen = new Color( 0f, 0.25f, 0f ); // Если альфа-значение не указано,
                                              // оно по умолчанию получает значение
                                              // 1 (полностью непрозрачный цвет)
Color darkRedTranslucent = new Color( 0.25f, 0f, 0f, 0.5f );
```

Как видите, есть два разных способа определения цвета: один с тремя параметрами (красный, зеленый и синий) и один — с четырьмя (красный, зеленый, синий и альфа)[1].

Переменные и функции экземпляра Color

Все каналы цвета доступны через переменные экземпляра:

```
print( Color.yellow.r ); // 1, Красный компонент желтого цвета
print( Color.yellow.g ); // 0.92f, Зеленый компонент желтого цвета
print( Color.yellow.b ); // 0.016f, Синий компонент желтого цвета
print( Color.yellow.a ); // 1, Альфа-компонент желтого цвета
```

Статические переменные и функции класса Color

Некоторые цвета предопределены в Unity в виде статических переменных класса:

```
// Основные цвета: красный, зеленый и синий
Color.red = new Color(1, 0, 0, 1);   // Чистый красный
Color.green = new Color(0, 1, 0, 1); // Чистый зеленый
Color.blue = new Color(0, 0, 1, 1);  // Чистый синий

// Вторичные цвета: циан, фуксин и желтый
Color.cyan = new Color(0, 1, 1, 1);            // Циан, яркий сине-зеленый
Color.magenta = new Color(1, 0, 1, 1);         // Фуксин, розово-фиолетовый
Color.yellow = new Color(1, 0.92f, 0.016f, 1); // Приятный для глаз желтый
// Нетрудно догадаться, что стандартный желтый цвет можно получить как new
// Color(1,1,0,1), но, по мнению разработчиков Unity, этот желтый смотрится лучше.

// Черный, белый и прозрачный
Color.black = new Color(0, 0, 0, 1);        // Чистый черный
Color.white = new Color(1, 1, 1, 1);        // Чистый белый
Color.gray = new Color(0.5f, 0.5f, 0.5f, 1) // Серый
Color.grey = new Color(0.5f, 0.5f, 0.5f, 1) // Британское написание цвета gray
(серый)
Color.clear = new Color(0, 0, 0, 0);        // Полностью прозрачный
```

Quaternion: информация о повороте

Объяснение внутреннего устройства класса `Quaternion`[2] и особенностей его работы далеко выходит за рамки этой книги, но вам часто придется пользоваться им для изменения углов поворота объектов посредством значения `GameObject.transform`.

---

[1] Возможность функции `new Color()` принимать три или четыре аргумента называется *перегрузкой функций*. Подробнее об этом рассказывается в главе 24 «Функции и параметры».

[2] http://docs.unity3d.com/Documentation/ScriptReference/Quaternion.html

rotation типа Quaternion, которое является частью любого игрового объекта. Экземпляры Quaternion определяют углы поворота так, чтобы избежать складывания шарнирных соединений — проблемы, характерной при использовании стандартного способа поворота относительно осей X, Y, Z (или эйлерова поворота), когда одна ось может совмещаться с другой и ограничивать возможность поворота. В большинстве случаев вы будете определять Quaternion, передавая углы эйлерова поворота и давая Unity возможность самостоятельно преобразовать их в эквивалентный Quaternion:

```
Quaternion lookUp45Deg = Quaternion.Euler( -45f, 0f, 0f );
```

В случаях, подобных этому, функции Quaternion.Euler() передаются три числа типа float — углы поворота в градусах относительно осей X, Y и Z (которые окрашиваются в Unity в красный, зеленый и синий цвета соответственно). Игровые объекты, включая Main Camera в сцене, первоначально ориентированы так, что смотрят в положительном направлении вдоль оси Z. Операция выше повернет камеру на −45° относительно красной оси X, которая в результате будет смотреть на игровой мир в положительном направлении вдоль оси Z под углом 45°. Если это последнее предложение показалось вам непонятным, не волнуйтесь. Позднее вы сможете перейти в Unity и поэкспериментировать с углами поворота относительно осей X, Y и Z в компоненте Transform игрового объекта в инспекторе, чтобы понять, как они влияют на его ориентацию.

Переменные и функции экземпляра Quaternion

Обратившись к переменной eulerAngles экземпляра Quaternion, можно узнать углы эйлерова поворота объекта относительно осей в виде экземпляра Vector3:

```
print( lookUp45Deg.eulerAngles ); // ( -45, 0, 0 ), Эйлеров поворот
```

Mathf: библиотека математических функций

В действительности Mathf[1] является не типом переменных, а фантастически полезной библиотекой математических функций. Все переменные и функции типа Mathf являются статическими; создать экземпляр типа Mathf невозможно. Библиотека Mathf включает слишком много полезных функций, чтобы можно было перечислить их здесь, тем не менее вот некоторые из них:

```
Mathf.Sin(x);          // Вычисляет синус угла x
Mathf.Cos(x);          // Также имеются функции .Tan(), .Asin(), .Acos() и .Atan()
Mathf.Atan2( y, x );   // Возвращает угол поворота относительно оси Z,
                       //  на который нужно повернуть объект, смотрящий
                       //  вдоль оси X, лицом к точке x, y.[2]
print(Mathf.PI);       // 3.141593; отношение длины окружности к диаметру
```

[1] http://docs.unity3d.com/Documentation/ScriptReference/Mathf.html

[2] http://docs.unity3d.com/Documentation/ScriptReference/Mathf.Atan2.html

```
Mathf.Min( 2, 3, 1 ); // 1, наименьшее из трех чисел (float или int)
Mathf.Max( 2, 3, 1 ); // 3, наибольшее из трех чисел (float или int)
Mathf.Round( 1.75f ); // 2, округление вверх или вниз до ближайшего целого
Mathf.Ceil( 1.75f );  // 2, округление вверх до следующего большего целого
Mathf.Floor( 1.75f ); // 1, округление вниз до следующего меньшего целого
Mathf.Abs( -25 );     // 25, абсолютное значение числа -25

Mathf.Approximately( a, b ); // Сравнение двух значение float с заданной точностью
```

`Mathf.Approximately()` — отличный инструмент, позволяющий справиться с невысокой точностью типа `float`, потому что (в отличие от оператора ==) возвращает `true`, если два значения `float` настолько близки друг к другу, что неточность, свойственная типу `float`, может заставить их выглядеть неравными. Этот метод не используется в книге, потому что ни в одном примере не выполняется сравнение двух значений `float` с использованием ==, но если вам в вашей практике понадобится сравнить два числа `float`, используйте `Mathf.Approximately()` вместо ==.

Screen: информация о дисплее

`Screen`[1] — еще одна библиотека, подобная `Mathf`, позволяющая получить информацию об экране, на котором отображается игра. Библиотека `Screen` работает на любых устройствах и дает возможность получить точную информацию о дисплее в PC, macOS, а также на устройствах iOS, Android и WebGL:

```
print( Screen.width );  // Выведет ширину экрана в пикселах
print( Screen.height ); // Выведет dscjne экрана в пикселах
```

SystemInfo: информация об устройстве

`SystemInfo`[2] предоставляет информацию об устройстве, на котором выполняется игра, в том числе информацию об операционной системе, количестве процессоров, графическом оборудовании и многом другом. Я советую пройти по ссылке в сноске, чтобы узнать больше.

```
print( SystemInfo.operatingSystem ); // Например: Mac OS X 10.8.5
```

GameObject: тип любого объекта в сцене

`GameObject`[3] — базовый класс всех сущностей, присутствующих в сценах Unity. Все объекты, которые вы видите на экране в игре Unity, являются подклассами класса `GameObject`. Игровые объекты `GameObject` могут содержать разные *компоненты*, включая перечисленные в следующем разделе «Игровые объекты и компоненты

[1] http://docs.unity3d.com/Documentation/ScriptReference/Screen.html
[2] http://docs.unity3d.com/Documentation/ScriptReference/SystemInfo.html
[3] http://docs.unity3d.com/Documentation/ScriptReference/GameObject.html

в Unity». Однако все игровые объекты имеют несколько важных переменных, кроме тех, что будут перечислены там:

```
GameObject gObj = new GameObject("MyGO"); // Создание нового игрового объекта
с именем MyGO
print( gObj.name ); // MyGO, имя игрового объекта gObj
Transform trans = gObj.GetComponent<Transform>(); // Определение ссылки trans
                                                  //  на компонент Transform
                                                  //  объекта gObj
Transform trans2 = gObj.transform; // Сокращенный способ доступа к тому же
                                   // компоненту Transform
gObj.SetActive(false); // Деактивирует объект gObj, в результате он становится
                       // невидимым и его код прекращает выполняться.
```

Метод[1] `gObj.GetComponent<Transform>()`, показанный здесь, играет особую роль, позволяя получить доступ к любому компоненту, присоединенному к игровому объекту `GameObject`. Иногда вам будут встречаться методы с угловыми скобками `<>`, такие как `GetComponent<>()`. Такие методы называют *обобщенными*, потому что они способны работать с разными типами данных. В случае с `GetComponent<Transform>()` используется тип данных `Transform`, который сообщает методу `GetComponent<>()`, что тот должен отыскать компонент Transform в игровом объекте `GameObject` и вернуть его вам. Аналогично можно получать любые другие компоненты, подключенные к `GameObject`, указывая тип компонента в угловых скобках вместо `Transform`. Например:

```
Renderer rend = gObj.GetComponent<Renderer>(); // Вернет компонент Renderer
Collider coll = gObj.GetComponent<Collider>(); // Вернет компонент Collider
HelloWorld hwInstance = gObj.GetComponent<HelloWorld>();
```

Как показано в третьей строке предыдущего листинга, с помощью `GetComponent<>()` можно получить экземпляр любого класса C#, присоединенный к игровому объекту `GameObject`. Если предположить, что к объекту `gObj` присоединен сценарий HelloWorld на C, тогда `gObj.Getcomponent <HelloWorld>()` вернет его. Этот прием несколько раз используется в примерах в этой книге.

Игровые объекты и компоненты в Unity

Как отмечалось в предыдущем разделе, все объекты, видимые на экране Unity, являются игровыми объектами типа `GameObject`, и все игровые объекты содержат один или более компонентов (компонент Transform присоединен *всегда*). Если выбрать игровой объект в панели Hierarchy (Иерархия) или в панели Scene (Сцена), его компоненты появятся в панели Inspector (Инспектор), как показано на рис. 20.1.

[1] Функция и метод в своей основе суть одно и то же. Единственное отличие — *функциями* называют автономные функции, а *методами* — функции, являющиеся частью класса.

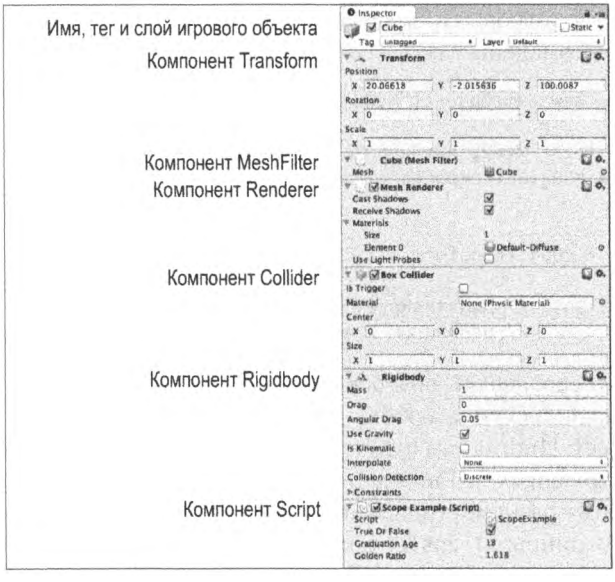


Рис. 20.1. В панели Inspector (Инспектор) отображаются разные важные компоненты

Transform: местоположение, поворот и масштаб

Transform[1] — обязательный компонент, присутствующий во всех игровых объектах. Компонент Transform управляет важной информацией об игровом объекте, такой как *местоположение* (координаты игрового объекта), *поворот* (ориентация игрового объекта) и *масштаб* (размер игрового объекта). Даже при том, что эта информация не отображается в панели Inspector (Инспектор), тем не менее компонент Transform отвечает также за отношения родитель — потомок в панели Hierarchy (Иерархия). Когда один объект является потомком другого, он перемещается вместе с родителем, если присоединен к нему.

MeshFilter: видимая модель

Компонент MeshFilter[2] присоединяет к игровому объекту трехмерную сетку, присутствующую в панели Project (Проект). Чтобы увидеть модель на экране, к игровому объекту должны быть присоединены оба компонента: MeshFilter, хранящий информацию о фактической трехмерной сетке, и MeshRenderer, объединяющий эту сетку с шейдером или материалом и отображающий изображение на экране. MeshFilter создает поверхность игрового объекта, а MeshRenderer определяет форму, цвет и текстуру этой поверхности.

[1] http://docs.unity3d.com/Documentation/Components/class-Transform.html
[2] http://docs.unity3d.com/Documentation/Components/class-MeshFilter.html

Renderer: позволяет увидеть игровой объект

Компонент Renderer[1] — в большинстве случаев MeshRenderer — позволяет увидеть игровой объект в панелях Scene (Сцена) и Game (Игра). MeshRenderer требует наличия MeshFilter с трехмерной сеткой, а также хотя бы одного компонента Material, если вы хотите, чтобы ваш объект выглядел привлекательнее кирпича унылого розово-фиолетового цвета (компоненты Material применяют текстуры к объектам, и в их отсутствие Unity по умолчанию применяет сплошной розово-фиолетовый материал, чтобы привлечь внимание к проблеме). Компоненты Renderer объединяют вместе MeshFilter, компонент(ы) Material и освещение, чтобы показать игровой объект `GameObject` на экране.

Collider: физическое присутствие игрового объекта

Компонент Collider[2] обеспечивает физическое присутствие объекта `GameObject` в игровом мире и определяет момент столкновения с другими объектами. В Unity имеется четыре вида компонентов Collider, перечисленные ниже в порядке уменьшения скорости работы. Самый быстрый коллайдер — Sphere Collider — выполняет расчеты по определению столкновения с другим объектом чрезвычайно быстро, тогда как Mesh Collider работает намного медленнее:

- Sphere Collider:[3] шар, или сфера. Столкновения с этой формой вычисляются быстрее всего.
- Capsule Collider:[4] цилиндр со сферами на обоих концах. Второй тип по быстродействию.
- Box Collider:[5] сплошной прямоугольник. Удобно использовать для моделирования коробок, ящиков и других предметов прямоугольной формы.
- Mesh Collider:[6] коллайдер, имеющий форму трехмерной сетки. Это удобный и очень точный коллайдер, но работает намного, намного медленнее трех предыдущих. Кроме того, Mesh Collider сможет обнаруживать столкновения с другими коллайдерами Mesh Collider, только если установить его признак Convex в значение `true`.

Физика и столкновения обсчитываются в Unity движком PhysX компании NVIDIA. Этот движок определяет столкновения очень быстро и с высокой точностью, тем не менее все физические движки имеют ограничения, и даже PhysX иногда ошибается при работе с быстро движущимися объектами или тонкими стенами.

В последующих главах коллайдеры рассматриваются более подробно. Дополнительную информацию о них вы найдете в документации к Unity.

---

[1] http://docs.unity3d.com/Documentation/Components/class-MeshRenderer.html
[2] http://docs.unity3d.com/Documentation/Components/comp-DynamicsGroup.html
[3] http://docs.unity3d.com/Documentation/Components/class-SphereCollider.html
[4] http://docs.unity3d.com/Documentation/Components/class-CapsuleCollider.html
[5] http://docs.unity3d.com/Documentation/Components/class-BoxCollider.html
[6] http://docs.unity3d.com/Documentation/Components/class-MeshCollider.html

Rigidbody: моделирование поведения физических тел

Компонент Rigidbody[1] моделирует поведение игрового объекта с учетом законов физики. Он пересчитывает ускорение и скорость движения в каждом вызове `FixedUpdate`[2] (обычно 50 раз в секунду), обновляя местоположение и поворот компонента Transform с течением времени. Он также использует компонент Collider для определения столкновений с другими игровыми объектами. Кроме того, компонент Rigidbody может моделировать действие сил тяготения, сопротивления и др., таких как сила ветра или взрыва. Установите переменную `isKinematic` этого компонента в значение `true`, если хотите сами позиционировать игровой объект, без моделирования физики, поддерживаемой компонентом Rigidbody.

Чтобы координаты компонента Collider изменялись синхронно с координатами его игрового объекта, этот игровой объект должен иметь компонент Rigidbody. Иначе — по крайней мере, когда поведением объекта управляет движок PhysX, — коллайдер не будет перемещаться. Проще говоря, если игровой объект не имеет компонента Rigidbody, он может перемещаться по экрану, но для PhysX местоположение компонента Collider будет оставаться неизменным, соответственно, будет считаться, что физически игровой объект находится в начальной точке.

Script: сценарии на C#, которые вы пишете

Все сценарии на C# также являются компонентами игрового объекта. Одно из преимуществ принадлежности сценариев к компонентам — возможность присоединить к игровому объекту несколько сценариев, и она будет использована в некоторых учебных примерах в части III. Далее в этой книге вы узнаете больше о компонентах Script и о том, как получить доступ к ним.

ИМЕНА ПЕРЕМЕННЫХ ИЗМЕНЯЮТСЯ В ИНСПЕКТОРЕ. На рис. 20.1 видно, что для сценария отображается имя Scope Example (Script), но это нарушает правила именования классов, потому что имена классов не могут содержать пробелов.

Фактически в моем коде имя сценария записано как одно слово в верблюжьемРегистре: `ScopeExample`. Точная причина мне неизвестна, но в инспекторе имена классов и переменных отображаются не так, как они записаны в сценариях на C#, согласно следующим правилам:

- Имя класса `ScopeExample` превращается в Scope Example (Script).
- Имя переменной `trueOrFalse` превращается в True Or False.

[1] http://docs.unity3d.com/Documentation/Components/class-Rigidbody.html

[2] В Unity метод `Update` вызывается перед отображением каждого кадра (обычно от 30 до 300 раз в секунду, в зависимости от быстродействия компьютера), тогда как метод `FixedUpdate` вызывается через равные интервалы времени (по умолчанию 50 раз в секунду, независимо от платформы). Физические движки работают лучше, когда пересчет происходит через равные интервалы времени, поэтому эти два метода были разделены.

- Имя переменной `graduationAge` превращается в Graduation Age.
- Имя переменной `goldenRatio` превращается в Golden Ratio.

Это важное отличие, которое не раз вызывало путаницу у моих студентов в прошлом. Но несмотря на то что в инспекторе имена выглядят иначе, в коде они остаются прежними. На протяжении всей книги я буду ссылаться на переменные по их настоящим именам, как они записаны в коде, независимо от того, как они отображаются в инспекторе.

Итоги

Это была длинная глава с большим объемом информации, и, может быть, вам понадобится прочитать ее еще раз или вернуться к ней позже, когда у вас появится некоторый опыт программирования. Но вся эта информация пригодится вам, если вы продолжите читать книгу и начнете писать свой код. Когда вы разберетесь во взаимоотношениях между игровыми объектами и компонентами, а также освоите установку и изменение переменных в инспекторе Unity, то заметите, что писать код получается намного быстрее и с меньшими затруднениями.

21 Логические операции и условия

Многие слышали, что на самом низком уровне данные в компьютере состоят исключительно из нулей и единиц — битов, которые могут иметь одно из двух значений: истина (true) или ложь (false). Но только программисты по-настоящему понимают, как много кода приходится писать, чтобы свести задачу к значениям «истина» или «ложь» и затем решить ее.

В этой главе вы познакомитесь с логическими операциями, такими как И, ИЛИ и НЕ; с операторами сравнения, такими как >, <, == и !=; и освоите условные инструкции, такие как `if` и `switch`. Все это составляет основу программирования.

Булева математика

Как вы узнали в предыдущей главе, переменные типа `bool` могут хранить одно из двух значений: `true` (истина) или `false` (ложь). Булева математика получила свое название в честь Джорджа Буля (George Boole), английского математика и логика, работавшего со значениями «истина» и «ложь» и логическими операциями (известными также как «булевы операции»). Хотя в то время не было компьютеров, вся компьютерная логика основана на его исследованиях.

В программировании на C# логические переменные широко используются для хранения простой информации о состоянии игры (например, `bool gameOver = false;`) и управления потоком выполнения программы с помощью операторов `if` и `switch`, о которых рассказывается далее в этой главе.

Логические операции

Логические операции позволяют программистам изменять или объединять логические переменные значимыми способами.

! (оператор НЕ)

Оператор ! (читается как «не») меняет логическое значение на противоположное. Ложь становится истиной, а истина — ложью:

```
print( !true );    // Выведет: false
print( !false );   // Выведет: true
print( !(!true) ); // Выведет: true (двойное отрицание значения true дает true)
```

Иногда оператор ! называют *оператором логического отрицания*, чтобы подчеркнуть отличие от оператора ~ (поразрядный оператор НЕ), о котором рассказывается в разделе «Поразрядные логические операторы и маски уровней» приложения Б «Полезные идеи».

&& (оператор И)

Оператор && возвращает `true`, только если оба операнда имеют значение `true`:[1]

```
print( false && false ); // false
print( false && true );  // false
print( true  && false ); // false
print( true  && true );  // true
```

|| (оператор ИЛИ)

Оператор || возвращает `true`, если хотя бы один из операндов имеет значение `true`:

```
print( false || false ); // false
print( false || true );  // true
print( true  || false ); // true
print( true  || true );  // true
```

Логические операторы с короткой и полной схемой вычисления

Стандартные операторы И и ИЛИ (&& и ||) выполняют вычисления *по короткой схеме*, то есть они возвращают результат, как только он становится очевиден, не выполняя оставшийся код. Например, инструкция `false && SomeFunction()` никогда не вызовет функцию `SomeFunction()`, потому что после вычисления первого операнда `false` результат становится очевиден и && вернет его до того, как дойдет очередь до вызова `SomeFunction()`. Существуют также логические операторы И и ИЛИ, выполняющие вычисления *по полной схеме* (& и |) — они вычисляют оба операнда независимо от значения первого. Следующий листинг поясняет, как действуют эти операторы.

В следующем листинге буква, идущая за двумя слешами (например, `// a`), справа от строки кода, указывает, что ниже, внутри листинга, даются дополнительные пояснения к этой строке. На протяжении всей книги пояснения обычно будут даваться в конце листингов, но в этом примере они включены в листинг, чтобы вам было проще.

[1] В некоторых листингах я добавляю дополнительные пробелы, чтобы сделать их более удобочитаемыми. Не забывайте, что любое количество пробелов, следующих подряд, интерпретируется компилятором C# как один пробел.

```
 1 // Эта функция выводит "--true" и возвращает значение true.
 2 bool printAndReturnTrue() {
 3     print( "--true" );
 4     return( true );
 5 }
 6
 7 // Эта функция выводит "--false" и возвращает значение false.
 8 bool printAndReturnFalse() {
 9     print( "--false" );
10     return( false );
11 }
12
13 void ShortingOperatorTest() {
14     // В строках 15, 17, 19 и 21 используются операторы && и || с короткой
       // схемой вычислений
15     bool andTF = ( printAndReturnTrue() && printAndReturnFalse() ); // a
16     print( "andTF: "+andTF ); // Выведет: "--true --false andTF: false"
```

a. Эта строка выведет `--true` и `--false`, а затем присвоит переменной `andTF` значение `false`. Первый аргумент оператора `&&`, действующего по короткой схеме, вычисляется как `true`, что заставляет оператор вычислить второй аргумент, в результате чего получается `false`.

```
17     bool andFT = ( printAndReturnFalse() && printAndReturnTrue() ); // b
18     print( "andFT: "+andFT ); // Выведет: " --false andFT: false"
```

b. Эта строка выведет только `--false`, а затем присвоит переменной `andFT` значение `false`. Первый аргумент оператора `&&`, действующего по короткой схеме, вычисляется как `false`, что заставляет оператор вернуть `false`, не вычисляя второй аргумент. В этой строке вызов `printAndReturnTrue()` не выполняется.

```
19     bool orTF = ( printAndReturnTrue() || printAndReturnFalse() ); // c
20     print( "orTF: "+orTF ); // Выведет: "--true orTF: true"
```

c. Эта строка выведет только `--true`, а затем присвоит переменной `orTF` значение `true`. Первый аргумент оператора `||`, действующего по короткой схеме, вычисляется как `true`, что заставляет оператор вернуть `true`, не вычисляя второй аргумент.

```
21     bool orFT = ( printAndReturnFalse() || printAndReturnTrue() ); // d
22     print( "orFT: "+orFT ); // Выведет: "--false --true orTF: true"
```

d. Эта строка выведет `--false` и `--true`, а затем присвоит переменной `orFT` значение `true`. Первый аргумент оператора `||`, действующего по короткой схеме, вычисляется как `false`, что заставляет оператор вычислить второй аргумент, чтобы определить возвращаемый результат.

```
23     // В строках 24 и 26 используются операторы & и | с полной схемой вычислений
24     bool nsAndFT = ( printAndReturnFalse() & printAndReturnTrue() ); // e
25     print( "nsAndFT: "+nsAndFT ); // Выведет: "--false --true nsAndFT: false"
```

e. Оператор &, действующий по полной схеме, вычисляет оба аргумента, независимо от значения первого. Эта строка выведет `--false` и `--true`, а затем присвоит переменной `nsAndFT` значение `false`.

```
26      bool nsOrTF = (printAndReturnTrue() | printAndReturnFalse() ); // f
27      print( "nsOrTF: "+nsOrTF ); // Выведет: "--true --false nsOrTF: false"
28  }
```

f. Оператор |, действующий по полной схеме, вычисляет оба аргумента, независимо от значения первого. Эта строка выведет `--true` и `--false`, а затем присвоит переменной `nsOrTF` значение `true`.

Знание обоих видов логических операторов, действующих по короткой и по полной схеме, пригодится вам при разработке своего кода. Чаще используются операторы с короткой схемой вычислений (&& и ||), но когда важно гарантировать вычисление всех аргументов, применяются операторы & и |.

Я советую ввести этот код в Unity и запустить его под управлением отладчика в пошаговом режиме, чтобы до конца понять, как он действует. Чтобы познакомиться с фантастическим отладчиком, входящим в состав MonoDevelop и Unity, прочитайте главу 25 «Отладка».

Поразрядные логические операторы

Иногда | и & называют *поразрядным ИЛИ* и *поразрядным И*, потому что они могут также использоваться для выполнения поразрядных операций с целыми числами. Их можно использовать для выполнения некоторых экзотических операций, связанных с обнаружением столкновений, о чем более подробно рассказывается в разделе «Поразрядные логические операторы и маски уровней» приложения Б «Полезные идеи».

Объединение логических операций

Часто бывает полезно объединить несколько логических операций в одну строку:

```
bool tf = true || false && false;
```

Но будьте внимательны, потому что порядок выполнения операций распространяется и на логические операторы. В C# логические операции выполняются в следующем порядке:

!	NOT
&	И с полной схемой вычислений / поразрядное И
|	ИЛИ с полной схемой вычислений / поразрядное ИЛИ
&&	И
||	ИЛИ

То есть предыдущая строка будет интерпретироваться компилятором как

```
bool tf = true || ( false && false );
```

Оператор && всегда выполняется раньше оператора ||. Не зная этих правил, вы могли бы полагать, что операторы выполняются слева направо, и ожидать результат false (например, (true || false) && false даст в результате false), но фактически без круглых скобок строка дает значение true!

Независимо от знания порядка выполнения операций, старайтесь использовать круглые скобки как можно чаще, чтобы сделать свой код более очевидным. Ясность кода особенно важна, если вы планируете работать с кем-то вместе (или даже для вас самого, если вы решите вернуться к своему коду спустя несколько недель или месяцев). Лично я следую простому правилу: если есть вероятность понять код неправильно, я использую круглые скобки и добавляю комментарий, поясняющий мои намерения и то, как этот код будет интерпретироваться компьютером.

Логическая эквивалентность булевых операций

Глубокое освещение булевой логики выходит далеко за рамки этой книги, тем не менее замечу, что, объединяя логические операции, можно получить очень интересные результаты. В примерах логических правил, перечисленных ниже, a и b — это переменные типа bool. Эти правила верны независимо от значений переменных a и b и независимо от того, по какой схеме — короткой или полной — действуют используемые операторы:

- Ассоциативность: (a & b) & c эквивалентно a & (b & c)
- Коммутативность: (a & b) эквивалентно (b & a)
- Дистрибутивность И перед ИЛИ: a & (b | c) эквивалентно (a & b) | (a & c)
- Дистрибутивность ИЛИ перед И: a | (b & c) эквивалентно (a | b) & (a | c)
- (a & b) эквивалентно !(!a | !b)
- (a | b) эквивалентно !(!a & !b)

Если вам будет интересно познакомиться с другими эквивалентными парами выражений и узнать, как их можно использовать, поищите в интернете ресурсы, посвященные булевой математике.

Операторы сравнения

Логические значения можно создавать не только с помощью логических операторов, но и с помощью операторов сравнения, применяя их к значениям других типов.

== (равно)

Оператор равенства проверяет равенство значений двух переменных или эквивалентность литералов. Результатом является логическое значение `true` или `false`.

НЕ ПУТАЙТЕ ОПЕРАТОРЫ = И == Начинающие программисты часто путают операторы присваивания (=) и равенства (==). Оператор присваивания (=) используется, чтобы записать некоторое значение в переменную, а оператор равенства (==) сравнивает два значения. Взгляните на следующий листинг:

```
1 bool f = false;
2 bool t = true;
3 print( f == t ); // выведет: false
4 print( f = t );  // выведет: true
```

В строке 3 переменная `f` сравнивается с переменной `t`, и поскольку они не равны, возвращается и выводится `false`. Однако в строке 4 переменной `f` присваивается значение переменной `t`, в результате `f` получает значение `true` и на экран выводится `true`.

Нередко также путаница возникает в разговорах при упоминании этих двух операторов. Чтобы избежать ее, об инструкции `i=5` я обычно говорю: «присвоить 5 переменной i», а об инструкции `i==5`: «i равно 5».

```
 1 int i0 = 10;
 2 int i1 = 10;
 3 int i2 = 20;
 4 float f0 = 1.23f;
 5 float f1 = 3.14f;
 6 float f2 = Mathf.PI;
 7
 8 print( i0 == i1 );    // Выведет: True
 9 print( i1 == i2 );    // Выведет: False
10 print( i2 == 20 );    // Выведет: True
11 print( f0 == f1 );    // Выведет: False
12 print( f0 == 1.23f ); // Выведет: True
13 print( f1 == f2 );    // Выведет: False // a
```

a. Сравнение в строке 13 вернет `false`, потому что `Math.PI` возвращает более точное значение, чем `3.14f`, а оператор == требует точного совпадения значений.

Более подробно о сравнении значений переменных разных типов рассказывается во врезке «Проверка равенства по значению и по ссылке».

ПРОВЕРКА РАВЕНСТВА ПО ЗНАЧЕНИЮ И ПО ССЫЛКЕ

В диалекте языка C# для Unity большинство простых типов данных сравнивается *по значению*. То есть две переменные считаются равными, если они хранят одинаковые значения. Это верно для всех следующих типов данных:

- bool
- int
- float
- char
- string
- Vector3
- Color
- Quaternion

Но для более сложных типов переменных, таких как GameObject, Material, Renderer и т. д., равенство проверяется *по ссылке*. При сравнении по ссылке проверяется не сходство значений, на которые ссылаются эти две переменные, а эквивалентность ссылок в них. То есть проверяется, ссылаются (или указывают) ли две переменные на один и тот же объект в памяти компьютера. В следующем примере, где выполняется сравнение по ссылке, boxPrefab — это существующая переменная, ссылающаяся на шаблон игрового объекта.

```
1 GameObject go0 = Instantiate<GameObject>( boxPrefab );
2 GameObject go1 = Instantiate<GameObject>( boxPrefab );
3 GameObject go2 = go0;
4 print( go0 == go1 ); // Выведет: false
5 print( go0 == go2 ); // Выведет: true
```

Даже при том, что экземпляры boxPrefabs, присвоенные переменным go0 и go1, имеют одинаковые значения (одни и те же координаты по умолчанию, поворот и т. д.), оператор == посчитал их разными, потому что фактически это два разных объекта, хранящихся в разных областях памяти. Оператор == посчитал равными переменные go0 и go2, потому что обе они ссылаются на один и тот же объект. Давайте продолжим предыдущий листинг:

```
6 go0.transform.position = new Vector3( 10, 20, 30);
7 print( go0.transform.position); // Выведет: (10.0, 20.0, 30.0)
8 print( go1.transform.position); // Выведет: ( 0.0, 0.0, 0.0)
9 print( go2.transform.position); // Выведет: (10.0, 20.0, 30.0)
```

Здесь мы изменили координаты в go0. Так как go1 ссылается на другой экземпляр игрового объекта, его координаты остались прежними. Однако так как go2 и go0 ссылаются на один и тот же экземпляр игрового объекта, изменения в координатах отразились также на go2.transform.position.

Теперь изменим координаты в go1, приведя их в соответствие с координатами в go0 (тот же игровой объект, на который ссылается go2).

```
10 go1.transform.position = new Vector3( 10, 20, 30);
11 print( go0.transform == go1.transform);                    // Выведет: false
12 print( go0.transform.position == go1.transform.position); // Выведет: true
```

Преобразования в go0 и go1 не эквивалентны, но их местоположения равны, потому что позиции типа Vector3 сравниваются по значению.

!= (не равно)

Оператор неравенства возвращает `true`, если два значения не равны, и `false`, если равны. Он является полной противоположностью оператору `==`. При сравнении объектов по ссылке оператор `!=` возвращает `true`, когда два объекта, на которые указывают ссылки, находятся в разных областях памяти. (При описании других операторов сравнения для ясности и экономии места вместо переменных будут использоваться литералы.)

```
print( 10 != 10 );          // Выведет: False
print( 10 != 20 );          // Выведет: True
print( 1.23f != 3.14f );    // Выведет: True
print( 1.23f != 1.23f );    // Выведет: False
print( 3.14f != Mathf.PI ); // Выведет: True
```

> (больше) и < (меньше)

Оператор `>` возвращает `true`, если значение слева больше значения справа:

```
print( 10 > 10 );       // Выведет: False
print( 20 > 10 );       // Выведет: True
print( 1.23f > 3.14f ); // Выведет: False
print( 1.23f > 1.23f ); // Выведет: False
print( 3.14f > 1.23f ); // Выведет: True
```

Оператор `<` возвращает `true`, если значение слева меньше значения справа:

```
print( 10 < 10 );       // Выведет: False
print( 20 < 10 );       // Выведет: False
print( 1.23f < 3.14f ); // Выведет: True
print( 1.23f < 1.23f ); // Выведет: False
print( 3.14f < 1.23f ); // Выведет: False
```

Символы `<` и `>` иногда называют также *угловыми скобками*, особенно когда они используются в роли тегов в таких языках, как HTML и XML, или в обобщенных функциях в C#. Но когда они используются в роли операторов сравнения, их всегда называют *больше* и *меньше*. Сравнение объектов по ссылке с помощью `>`, `<`, `>=` и `<=` невозможно.

>= (больше или равно) и <= (меньше или равно)

Оператор `>=` возвращает `true`, если значение слева больше или равно значению справа:

```
print( 10 >= 10 );       // Выведет: True
print( 10 >= 20 );       // Выведет: False
print( 1.23f >= 3.14f ); // Выведет: False
print( 1.23f >= 1.23f ); // Выведет: True
print( 3.14f >= 1.23f ); // Выведет: True
```

Оператор `<=` возвращает `true`, если значение слева меньше или равно значению справа:

```
print( 10 <= 10 );       // Выведет: True
print( 10 <= 20 );       // Выведет: True
print( 1.23f <= 3.14f ); // Выведет: True
print( 1.23f <= 1.23f ); // Выведет: True
print( 3.14f <= 1.23f ); // Выведет: False
```

Условные инструкции

Логические инструкции можно объединять с логическими значениями и операторами сравнения для управления потоком выполнения программ. Это значит, что значение `true` может направить выполнение кода по одному пути, а значение `false` — по-другому. Наиболее распространенными формами условных инструкций являются `if` и `switch`.

Инструкция if

Инструкция `if` выполняет код в фигурных скобках `{}`, только если значение в круглых скобках `()` вычисляется как `true`:

```
if (true) {
    print( "The code in the first if statement executed." );
}
if (false) {
    print( "The code in the second if statement executed." );
}

// Этот код выведет:
//     The code in the first if statement executed.
```

Здесь выполнится код в фигурных скобках `{}`, относящихся к первой инструкции `if`, а код в фигурных скобках, относящихся ко второй инструкции `if`, — нет.

Инструкции с фигурными скобками не требуют добавлять точку с запятой после закрывающей фигурной скобки. Все другие инструкции обязательно должны оканчиваться точкой с запятой:

```
float approxPi = 3.14159f; // Стандартная инструкция с точкой с запятой
```

Составные инструкции (то есть заключенные в фигурные скобки) не требуют наличия точки с запятой после закрывающей фигурной скобки:

```
if (true) {
    print( "Hello" ); // Здесь нужна точка с запятой.
    print( "World" ); // Здесь нужна точка с запятой.
}                     // После закрывающей фигурной скобки точка с запятой
                      // не требуется!
```

То же верно для любых других составных инструкций, заключенных в фигурные скобки.[1]

[1] Насколько я помню, это правило не действует только в одном случае: в особой форме инициализации массива.

Объединение инструкций if с операциями сравнения и логическими операциями

Для определения реакции игры на разные ситуации логические операторы можно объединять с инструкциями `if`:

```
bool night = true;
bool fullMoon = false;

if (night) {
    print( "It's night." );
}
if (!fullMoon) {
    print( "The moon is not full." );
}
if (night && fullMoon) {
    print( "Beware werewolves!!!" );
}
if (night && !fullMoon) {
    print( "No werewolves tonight. (Whew!)" );
}

// Этот код выведет:
//     It's night.
//     The moon is not full.
//     No werewolves tonight. (Whew!)
```

И конечно, инструкции `if` можно объединять с операторами сравнения:

```
if (10 == 10 ) {
    print( "10 is equal to 10." );
}
if ( 10 > 20 ) {
    print( "10 is greater than 20." );
}
if ( 1.23f <= 3.14f ) {
    print( "1.23 is less than or equal to 3.14." );
}
if ( 1.23f >= 1.23f ) {
    print( "1.23 is greater than or equal to 1.23." );
}
if ( 3.14f != Mathf.PI ) {
    print( "3.14 is not equal to "+Mathf.PI+"." );
    // оператор + можно использовать для конкатенации строк с данными других типов.
    // В этом случае данные других типов преобразуются в строки.
}

// Этот код выведет:
//     10 is equal to 10.
//     1.23 is less than or equal to 3.14.
//     1.23 is greater than or equal to 1.23.
//     3.14 is not equal to 3.141593.
```

ИЗБЕГАЙТЕ ИСПОЛЬЗОВАНИЯ = В ИНСТРУКЦИИ if. Как уже упоминалось в предыдущем предупреждении, == — это оператор сравнения, определяющий эквивалентность двух значений, а = — это оператор присваивания, записывающий новое значение

в переменную. Если в инструкции `if` по ошибке использовать =, результатом фактически будет присваивание, а не сравнение.

Иногда Unity обнаруживает такие ошибки, сообщая о невозможности неявно преобразовать значение в логический тип. Например, такая ошибка будет обнаружена в следующем коде:

```
float f0 = 10f;
if ( f0 = 10 ) {
    print( "f0 is equal to 10.");
}
```

Иногда Unity просто выводит вежливое предупреждение, отмечая, что в инструкции `if` встречен оператор =, и, возможно, вы имели в виду ==. Но иногда Unity не выводит никаких предупреждений, поэтому будьте внимательны и отслеживайте это самостоятельно.

Инструкция if...else

Часто бывает нужно сделать что-то одно, если значение условного выражения равно `true`, и что-то другое в противном случае. В таких случаях можно добавить в инструкцию `if` предложение `else`:

```
bool night = false;

if (night) {
    print( "It's night." );
} else {
    print( "It's daytime. What are you worried about?" );
}

// Этот код выведет:
//     It's daytime. What are you worried about?
```

В данном случае, так как переменная `night` имеет значение `false`, код выполнит предложение `else`.

Цепочки if...else if...else

Из предложений `else` можно также составлять цепочки:

```
bool night = true;
bool fullMoon = true;

if (!night) {               // Условие 1 (вычисляется как false)
    print( "It's daytime. What are you worried about?" );
} else if (fullMoon) {      // Условие 2 (вычисляется как true)
    print( "Beware werewolves!!!" );
} else {                    // Условие 3 (не вычисляется)
    print( "It's night, but the moon is not full." );
}

// Этот код выведет:
//     Beware werewolves!!!
```

Если какое-то из условий в цепочке `if...else if...else` вычисляется как `true`, все остальные условия не проверяются (остальная часть цепочки просто пропускается). В предыдущем листинге Условие 1 вычисляется как `false`, поэтому проверяется

Условие 2. Так как Условие 2 вычисляется как `true`, компьютер перепрыгивает через Условие 3 и не проверяет его.

Вложенные инструкции if

Для реализации более сложного поведения допускается вкладывать инструкции `if` друг в друга:

```
bool night = true;
bool fullMoon = false;

if (!night) {
    print( "It's daytime. What are you worried about?" );
} else {
    if (fullMoon) {
        print( "Beware werewolves!!!" );
    } else {
        print( "It's night, but the moon is not full." );
    }
}

// Этот код выведет:
//     It's night, but the moon is not full.
```

Инструкция switch

Инструкция `switch` может заменить несколько инструкций `if...else`, но у нее есть некоторые ограничения:

- инструкция `switch` проверяет только равенство;
- инструкция `switch` выполняет сравнение только с одной переменной;
- инструкция `switch` сравнивает переменную только с литералами (но не с другими переменными).

Например:

```
int num = 3;

switch (num) { // Переменная в круглых скобках (num) используется для сравнения
    case (0):  // Каждый вариант case – это литерал числа, сравниваемый
               // с переменной num
        print( "The number is zero." );
        break; // Каждый вариант case должен завершаться инструкцией break.
    case (1):
        print( "The number is one." );
        break;
    case (2):
        print( "The number is two." );
        break;
    default:   // Если ни для одного из вариантов сравнение не вернуло true,
               // выполняется default
        print( "The number is more than a couple." );
        break;
} // Инструкция switch завершается закрывающей фигурной скобкой.
```

```
// Этот код выведет:
//     The number is more than a couple.
```

Если один из вариантов `case` содержит литерал со значением, равным значению проверяемой переменной, компьютер перейдет к этому варианту `case` и будет выполнять его код, пока не встретит инструкцию `break`. Встретив `break`, он выйдет из инструкции `switch`, пропустив код в других вариантах `case`.

Также можно организовать «проваливание» из одного варианта `case` в следующий за ним, но только если между предложениями `case` нет никакого другого кода (например, варианты 3, 4 и 5 в следующем листинге):

```
int num = 4;

switch (num) {
    case (0):
        print( "The number is zero." );
        break;
    case (1):
        print( "The number is one." );
        break;
    case (2):
        print( "The number is a couple." );
        break;
    case (3):
    case (4):
    case (5):
        print( "The number is a few." );
        break;
    default:
        print( "The number is more than a few." );
        break;
}

// Этот код выведет:
//     The number is a few.
```

В этом примере, если переменная `num` хранит число 3, 4 или 5, будет выведена строка `The number is a few`.

Учитывая возможность объединения условий и инструкций `if`, возникает вопрос: какой смысл использовать инструкцию `switch`, если она имеет так много ограничений? На самом деле, она часто применяется для обработки разных возможных состояний игровых объектов. Например, если вы пишете игру, в которой игрок может превращаться в человека, птицу, рыбу или росомаху, вы можете использовать такой фрагмент кода:

```
string species = "fish";
bool   onLand = false;

// Разные животные перемещаются по-разному
public function Move() {
    switch (species) {
        case ("person"):
            Run(); // Вызов функции с именем Run()
```

```
            break;
        case ("bird"):
            Fly();
            break;
        case ("fish"):
            if (!onLand) {
                Swim();
            } else {
                FlopAroundPainfully();
            }
            break;
        case ("wolverine"):
            Scurry();
            break;
        default:
            print( "Unknown species type: "+species );
            break;
    }
}
```

В этом примере, если игрок принял обличье рыбы (и находится в воде), будет вызвана функция `Swim()`. Важно отметить, что вариант `default` здесь используется для выявления обличий, которые не поддерживаются инструкцией `switch`, и выводит информацию о них. Например, если переменной `species` присвоить строку `"lion"`, этот код выведет:

```
Unknown species type: lion
```

В предыдущем листинге также присутствуют имена функций, которые еще не определены (например, `Run()`, `Fly()`, `Swim()`). В следующей главе мы как раз рассмотрим тему создания своих функций.[1]

Итоги

Несмотря на то что логические операции могут показаться излишне сухими, они составляют большую часть программирования. Компьютерные программы содержат сотни и даже тысячи точек ветвления, где компьютер выбирает, по какому пути пойти, и все эти решения сводятся к логическим операциям и сравнениям. Продолжая читать книгу, вы можете время от времени возвращаться к этому разделу, если почувствуете, что не понимаете, как выполняется сравнение в коде, находящемся перед вами.

[1] По некоторым причинам создание отдельных функций для разных способов перемещения, как в этом примере, считается не самым лучшим стилем программирования. Однако объяснение этих причин выходит далеко за рамки этой книги. Когда вы закончите читать эту книгу, посетите веб-сайт Роберта Найстрома (Robert Nystrom) и его книги «Game Programming Patterns» (http://gameprogrammingpatterns.com), где можно найти информацию о хороших стратегиях программирования.

22 Циклы

Компьютерные программы обычно многократно выполняют одни и те же действия. В стандартном игровом цикле игра рисует кадр на экране, проверяет ввод игрока, рассматривает этот ввод и рисует следующий кадр, повторяя эти действия не менее 30 раз в секунду.

Цикл в коде на C# заставляет компьютер повторять одни и те же действия несколько раз. Это может быть все что угодно — от обхода всех врагов в сцене и передачи управления их искусственному интеллекту до обхода всех физических объектов в сцене и проверки столкновений с ними. К концу этой главы вы будете знать о циклах все, что необходимо, а в следующей главе вы узнаете, как использовать их для работы с разными коллекциями C#.

Виды циклов

В C# есть всего четыре вида циклов: `while`, `do...while`, `for` и `foreach`. Из них чаще всего вы будете использовать `for` и `foreach`, потому что в целом они более безопасны и лучше приспособлены к решению задач, с которыми вы будете сталкиваться при создании игр:

- Цикл `while`: самый простой вид цикла. Перед каждой итерацией проверяет условие, чтобы убедиться, что можно продолжить выполнять цикл.
- Цикл `do...while`: похож на цикл `while`, но выполняет проверку *после* каждой итерации.
- Цикл `for`: инструкция цикла включает выражение инициализации, условие завершения цикла и выражение наращивания переменной цикла. Это наиболее часто используемый вид цикла.
- Цикл `foreach`: инструкция цикла автоматически выполняет итерации по всем элементам перечисляемого объекта или коллекции. Эта глава включает первую часть обсуждения инструкции `foreach`, а другая часть обсуждения, более обширная, приводится в следующей главе как часть дискуссии о коллекциях в C#, таких как списки и массивы.

Подготовка проекта

В приложении А «Стандартная процедура настройки проекта» приводятся подробные инструкции, как настраивать проекты Unity для глав этой книги. В начале описания каждого проекта вы будете видеть врезку, подобную данной ниже. Выполните рекомендации во врезке, чтобы настроить проект для этой главы.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. За информацией о стандартной процедуре обращайтесь к приложению А.

Имя проекта: Loop Examples

Имя сцены: _Scene_Loops

Имена сценариев на C#: Loops

Подключите сценарий Loops к главной камере Main Camera в сцене.

Цикл while

Цикл `while` — самый простой вид цикла. Однако поэтому он и недостаточно безопасный в сравнении с более современными видами циклов. В своей практике я почти никогда не использую циклы `while` из-за опасности попасть в *бесконечный цикл*.

Опасность бесконечных циклов

Цикл становится бесконечным, когда программа, начав выполнять его, не может остановиться. Давайте напишем такой цикл и посмотрим, что из этого получится. Откройте сценарий **Loops** в MonoDevelop (дважды щелкнув на нем в панели **Project** (Проект)), добавьте в него строки, выделенные жирным в листинге ниже (строки 7–9), и удалите все лишние строки, добавляемые в сценарий по умолчанию.

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class Loops : MonoBehaviour {
 6     void Start () {
 7         while (true) {
 8             print( "Loop" );
 9         }
10     }
11 }
```

Сохраните сценарий, выбрав в MonoDevelop пункт меню **File > Save** (Файл > Сохранить). Переключитесь обратно в Unity и щелкните на кнопке **Play** (Играть)

с треугольником в верхней части окна Unity. Видите, что ничего не происходит... ничего не происходит и не произойдет *никогда*? Фактически вам придется *принудительно завершить* Unity (как это сделать, описывается во врезке). Вы только что столкнулись с *бесконечным циклом*, и, как видите, бесконечный цикл полностью подвесил Unity. Вам повезло, что в настоящее время вы пользуетесь многопоточной компьютерной операционной системой, потому что прежде, во времена однопоточных систем, бесконечный цикл подвесил бы не только одно приложение, но и весь компьютер целиком, и тогда остался бы только один выход — перезапуск.

Но что сделало цикл бесконечным? Чтобы понять это, взгляните на цикл `while`.

```
7         while (true) {
8             print( "Loop" );
9         }
```

Код, заключенный в фигурные скобки, выполняется многократно, пока *условное выражение* в круглых скобках остается истинным. В строке 7 выражение всегда истинно, поэтому строка `print( "Loop" );` будет выполняться снова и снова, до бесконечности.

Но возникает вопрос: если строка выполняется бесконечно, почему текст `"Loop"` не появляется в панели Console (Консоль)? Несмотря на то что функция `print()` была вызвана много раз (может быть, сотни тысяч или даже миллионы, прежде чем вы решили принудительно завершить Unity), вы не увидите вывода в панели Console (Консоль), потому что Unity попал в бесконечный цикл и не в состоянии перерисовать свое окно (что необходимо, чтобы вы смогли увидеть изменения в панели Console (Консоль)).

КАК ПРИНУДИТЕЛЬНО ЗАВЕРШИТЬ ПРИЛОЖЕНИЕ

В macOS

Чтобы принудительно завершить приложение, выполните следующие действия:

- Нажмите комбинацию Command-Option-Esc на клавиатуре. Откроется окно Force Quit (Принудительное завершение).
- Найдите зависшее приложение, которое нужно завершить. Часто его имя в списке сопровождается текстом (Not responding) (Не отвечает).
- Щелкните на имени приложения в списке и затем на кнопке Force Quit (Завершить). Возможно, вам придется подождать несколько секунд, пока приложение завершится.

В Windows

Чтобы принудительно завершить приложение, выполните следующие действия:

- Нажмите комбинацию Shift+Ctrl+Esc на клавиатуре. Откроется окно Windows Task Manager (Диспетчер задач Windows).
- Найдите зависшее приложение.

- Щелкните на имени приложения и затем на кнопке End Task (Снять задачу). Возможно, вам придется подождать несколько секунд, пока приложение завершится.

Принудительно завершив работающую среду Unity, вы потеряете все изменения, которые добавили с момента последнего сохранения. Поскольку вы и без того вынуждены постоянно сохранять сценарии на C#, к ним это предупреждение относится меньше всего, но вам, возможно, придется восстанавливать несохраненные изменения в сцене. Например, в _Scene_Loops. Если не сохранить ее после подключения сценария Loops к главной камере Main Camera, вам придется вновь подключать сценарий после перезапуска Unity.

Более полезный цикл while

Откройте сценарий Loops в MonoDevelop и измените его, как показано ниже:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Loops : MonoBehaviour {
    void Start () {
        int i=0;
        while ( i<3 ) {
            print( "Loop: "+i );
            i++; // См. врезку "Операторы инкремента и декремента"
        }
    }
}
```

Сохраните код, переключитесь обратно в Unity и щелкните на кнопке Play (Играть). На этот раз Unity не застрянет в бесконечном цикле, потому что условное выражение `( i<3 )` в инструкции `while` рано или поздно станет ложным. Вот как выглядит вывод этой программы в консоли (за исключением всех дополнительных строк, которые выводит Unity):

```
Loop: 0
Loop: 1
Loop: 2
```

Эти строки выводятся вызовом `print( i )` в каждой итерации цикла `while`. Важно отметить, что условное выражение проверяется *перед* каждой итерацией.

В большинстве примеров в этой главе в роли *переменной цикла* будет использоваться переменная с именем `i`. Имена `i`, `j` и `k` часто используются программистами для обозначения переменных цикла (то есть переменных, наращиваемых в каждой итерации), и, как результат, эти имена редко используются в других ситуациях. Поскольку эти переменные очень часто создаются и уничтожаются в разных циклах, старайтесь вообще не использовать имена переменных `i`, `j` и `k` для чего-то еще.

ОПЕРАТОРЫ ИНКРЕМЕНТА И ДЕКРЕМЕНТА

В строке 10 примера «более полезного» цикла `while` впервые в этой книге демонстрируется *оператор инкремента* (`++`), который увеличивает значение соответствующей переменной на 1. То есть если допустить, что `i=5`, тогда инструкция `i++;` увеличит значение переменной `i` до 6.

Существует также *оператор декремента* (`--`), уменьшающий значение соответствующей переменной на 1.

Операторы инкремента и декремента можно помещать перед или после имени переменной, но в этих случаях инструкции будут работать по-разному (например, `++i` и `i++` действуют немного по-разному). Разница в том, какое значение возвращается, начальное (`i++`) или уже увеличенное (`++i`). Вот пример, который поможет понять суть.

```
 6     void Start() {
 7         int i = 1;
 8         print( i );   // Выведет: 1
 9         print( i++ ); // Выведет: 1
10         print( i );   // Выведет: 2
11         print( ++i ); // Выведет: 3
12     }
```

Как видите, строка 8 вывела текущее значение переменной `i`, равное 1. Затем, в строке 9, оператор постинкремента `i++` сначала вернул текущее значение `i`, которое было выведено в консоль (1), а затем увеличил значение `i` до 2.

Строка 10 вывела текущее значение `i`, равное 2. Затем, в строке 10, оператор преинкремента `++i` сначала увеличил значение `i` с 2 до 3 и затем вернул его функции `print`, которая вывела 3.

Цикл do...while

Цикл `do...while` действует точно так же, как цикл `while`, только условное выражение проверяется *после* каждой итерации. Это гарантирует, что тело цикла выполнится хотя бы один раз. Измените код, как показано ниже:

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class Loops : MonoBehaviour {
 6     void Start () {
 7         int i=10;
 8         do {
 9             print( "Loop: "+i );
10             i++;
11         } while ( i<3 );
12     }
13 }
```

Проверьте еще раз, что не забыли заменить строку 7 в функции `Start()` на `int i=10;`. Даже при том, что условие в `while` никогда не выполнится (10 никогда не будет меньше 3), цикл все равно произведет одну итерацию перед проверкой условного выражения в строке 11. Если инициализировать переменную `i` числом `0`, вывод этого примера не будет отличаться от предыдущего, поэтому я использовал инструкцию `i=10` в строке 7, чтобы показать, что цикл `do...while` всегда выполняет хотя бы одну итерацию, независимо от значения `i`. Условное выражение в `do...while` всегда должно завершаться точкой с запятой.

Сохраните сценарий и запустите его в Unity, чтобы увидеть результат.

Цикл for

В обоих примерах, `while` и `do...while`, требовалось объявлять и определять переменную `i`, увеличивать ее и проверять в условном выражении, и каждое из этих действий оформлялось отдельной инструкцией. Цикл `for` позволяет описать все эти действия в одной строке. Измените код в сценарии **Loops**, как показано ниже, сохраните его и запустите в Unity.

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class Loops : MonoBehaviour {
 6     void Start() {
 7         for ( int i=0; i<3; i++ ) {
 8             print( "Loop: "+i );
 9         }
10     }
11 }
```

Цикл `for` в этом примере выводит в панель **Console** (Консоль) все то же самое, что и предыдущий «более полезный» цикл `while`, но при этом вам потребовалось писать меньше строк кода. Структура цикла `for` требует указать допустимые *выражение инициализации*, *условное выражение* и *выражение итерации*. Ниже жирным выделены все эти выражения из предыдущего примера:

Выражение инициализации:	`for ( int i=0; i<3; i++ ) {`
Условное выражение:	`for ( int i=0; i<3; i++ ) {`
Выражение итерации:	`for ( int i=0; i<3; i++ ) {`

Выражение инициализации (`int i=0;`) выполняется перед началом цикла `for`. Оно объявляет и определяет переменную с локальной областью видимости в теле цикла. Это означает, что переменная `int i` перестанет существовать после выхода из цикла. За дополнительной информацией об областях видимости переменных обращайтесь к разделу «Области видимости переменных» приложения Б «Полезные идеи».

Условное выражение (`i<3`) проверяется перед первой итерацией цикла `for` (в точности как проверяется условное выражение перед первой итерацией цикла `while`). Если условие выполняется, выполняется тело цикла `for` в фигурных скобках.

После выполнения тела цикла `for` в фигурных скобках выполняется *выражение итерации* (`i++`), то есть после инструкции `print( "Loop: "+i );` выполняется выражение `i++`. Затем снова проверяется условное выражение, и если оно все еще истинно, вновь выполняется тело цикла в фигурных скобках и выражение итерации. Так продолжается, пока условное выражение не станет ложным, после чего цикл `for` завершится.

Поскольку цикл `for` требует включать все эти выражения в одну строку, избежать создания бесконечного цикла намного проще с циклами `for`.

НЕ ЗАБЫВАЙТЕ ДОБАВЛЯТЬ ТОЧКИ С ЗАПЯТОЙ ПОСЛЕ КАЖДОГО ВЫРАЖЕНИЯ В ИНСТРУКЦИИ `for`. Важно не забывать разделять выражения инициализации, условия и итерации точками с запятой. Каждое из них является независимым выражением и, как все независимые выражения в C#, должно завершаться точкой с запятой. Так же как большинство строк в C# должны завершаться точкой с запятой, независимые выражения в цикле `for` тоже должны завершаться точкой с запятой.

Выражение инкремента необязательно должно быть ++

В роли выражения итерации часто используют инструкцию инкремента, такую как `i++`, но это не является обязательным требованием. В выражении итерации допускается использовать любые операции.

Декремент

Часто в качестве выражения итерации используют счет в обратную сторону вместо счета вперед. Реализовать это можно с помощью оператора декремента.

```
 6     void Start() {
 7         for ( int i=5; i>2; i-- ) {
 8             print( "Loop: "+i );
 9         }
10     }
```

Этот пример выведет в панель Console (Консоль) следующие строки:

```
Loop: 5
Loop: 4
Loop: 3
```

Цикл foreach

Цикл `foreach` действует как автоматизированный цикл `for` и может использоваться для итераций по элементам перечисляемых объектов. В C# к таким объектам отно-

сится большинство коллекций, включая строки (как коллекции символов), списки и массивы, о которых рассказывается в следующей главе. Попробуйте выполнить следующий пример в Unity.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Loops : MonoBehaviour {
    void Start() {
        string str = "Hello";
        foreach( char chr in str ) {
            print( chr );
        }
    }
}
```

В каждой итерации этот цикл будет выводить очередной символ из строки `str`, как показано ниже:

```
H
e
l
l
o
```

Цикл `foreach` гарантирует обход всех элементов перечислимого объекта. В данном случае он выполняет обход всех символов в строке `"Hello"`. Я подробнее остановлюсь на цикле `foreach` в следующей главе, где мы обсудим списки и массивы[1].

Инструкции перехода в циклах

Инструкцией перехода называется любая инструкция, которая заставляет поток выполнения перепрыгивать в другую точку в коде. Примером может служить уже знакомая нам инструкция `break`, помещаемая в конец каждой ветки `case` в инструкции `switch`.

Инструкция break

Инструкция `break` может также использоваться в любых циклах для преждевременного выхода из них. Для примера измените функцию `Start()`, как показано ниже:

[1] Это не должно вас беспокоить в данный момент, но вы должны знать, что циклы `foreach` менее эффективны, чем другие виды циклов, в том смысле, что они действуют немного медленнее и генерируют больше мусора в памяти, который автоматически собирается и удаляется компьютером. При создании игр для слабых устройств, таких как мобильные телефоны, где ощущается нехватка быстродействия и памяти, вам, возможно, придется отказаться от циклов `foreach`. Кроме того, цикл `foreach` не гарантирует перебора элементов коллекций (то есть массивов, списков и т. п.) в ожидаемом порядке, хотя обычно порядок соблюдается.

```
 6     void Start() {
 7         for ( int i=0; i<10; i++ ) {
 8             print( i );
 9             if ( i==3 ) {
10                 break;
11             }
12         }
13     }
```

Обратите внимание: в этом листинге я опустил строки 1–5, а также последнюю строку с закрывающей фигурной скобкой } (строка 12 в предыдущих листингах), потому что они не изменились. Эти строки должны присутствовать в сценарии — вам нужно лишь заменить строки 7–10 с циклом `foreach` в предыдущем листинге строками 7–12 из этого листинга.

Запустив этот код в Unity, вы получите вывод:

```
0
1
2
3
```

Инструкция `break` производит преждевременный выход из цикла `for`. Ее также можно использовать для прерывания циклов `while`, `do...while` и `foreach`.

Примеры кода:	**Вывод в консоль:**

`for ( int i=0; i<10; i++ ) {`
`  print( i );`
`  if ( i==3 ) {`
`  break;`
`  }`
`}`
	`0`
	`1`
	`2`
	`3`

`int i = 0;`
`while (true) {`
`  print( i );`
`  if ( i > 2 ) break; // a`
`  i++;`
`}`
	`0`
	`1`
	`2`
	`3`

`int i = 3;`
`do {`
`  print( i );`
`  i--;`
`  if ( i==1 ) break; // b`
`} while ( i > 0 );`
	`3`
	`2`

`foreach (char c in "Hello") {`
`  if (c == 'l') {`
`  break;`
`  }`
`  print( c );`
`}`
	`H`
	`e`

Следующие абзацы, начинающиеся с латинских символов, относятся к строкам в предыдущем коде и отмечены комментариями `// a` и `// b` (для большей заметности они выделены в листингах жирным).

1. Эта строка демонстрирует однострочную версию инструкции `if`. Если инструкция записывается в одну строку, как здесь, фигурные скобки можно опустить.
2. Этот код выведет только 3 и 2, потому что во второй итерации операция декремента `i--` уменьшит значение `i` до `1`. После этого условное выражение в инструкции `if` станет истинным и выполнится инструкция `break`, прервав цикл `do...while`.

Приостановитесь ненадолго и исследуйте предыдущие примеры кода, чтобы до конца понять, почему каждый из них генерирует именно такой вывод, как показано в колонке справа. Если что-то покажется вам непонятным, введите код и исследуйте его работу в отладчике. (См. главу 25 «Отладка».)

Инструкция continue

Инструкция `continue` заставляет программу пропустить код, оставшийся в теле цикла, и начать следующую итерацию.

Код:

```
for (int i=0; i<=360; i++) {
    if (i % 90 != 0) {
        continue;
    }
    print( i );
}
```

Вывод:

```
0
90
180
270
360
```

В предыдущем коде, всякий раз, когда выполняется условие `i%90 != 0` (то есть когда остаток от деления `i/90` не равен нулю), инструкция `continue` заставляет цикл `for` начать новую итерацию, пропустив строку `print( i );`. Инструкцию `continue` можно также использовать в циклах `while`, `do...while` и `foreach`.

% — ОПЕРАТОР ДЕЛЕНИЯ ПО МОДУЛЮ

Строка `if (i % 90 != 0) {` в примере использования инструкции `continue` впервые в этой книге демонстрирует применение оператора *деления по модулю* (`%`). Он возвращает остаток от деления двух чисел. Например, `12 % 10` вернет значение `2`, потому что остаток от деления `12/10` равен 2.

Оператор деления по модулю можно также применять к вещественным числам, например: `12.4f % 1f` вернет `0.4f`, остаток от деления 12.4 на 1. Но имейте в виду, что оператор деления по модулю восприимчив к врожденной неточности стандартного типа `float`, то есть `12.4f % 1f` может вернуть результат `0.3999996f` или нечто похожее.

Итоги

Понимание работы циклов — важное условие на пути становления хорошего программиста. Однако ничего страшного, если прямо сейчас что-то останется для вас непонятным. Когда вы начнете использовать циклы, занявшись разработкой действующих прототипов игр, они станут для вас более понятными. Просто вводите каждый пример кода в Unity и запускайте его, чтобы помочь себе приблизиться к пониманию.

Также запомните, что в своей практике я часто использую циклы `for` и `foreach` и крайне редко циклы `while` и `do...while` из-за опасности создать бесконечный цикл. В следующей главе вы познакомитесь с массивами, списками и другими видами упорядоченных коллекций однотипных элементов и увидите, как можно использовать циклы для итераций по их элементам.

23 Коллекции в C#

Коллекции в C# позволяют воздействовать на несколько однотипных объектов как на группу. Например, можно сохранить все игровые объекты, представляющие врагов, в списке и выполнять итерации по этому списку в каждом кадре, чтобы обновить их местоположения и состояния.

Эта глава охватывает три основных вида таких коллекций: списки, массивы и словари. К концу этой главы вы будете понимать, как работают эти коллекции и какие из них лучше использовать в той или иной ситуации.

Коллекции в C#

Коллекция — это группа элементов, на которую можно сослаться посредством единственной переменной. Примерами коллекций в обычной жизни могут служить: группа людей, прайд львов, сборище грачей или стая ворон. Подобно этим группам животных, коллекции в языке C#, которые вы будете использовать, могут содержать данные только одного типа (например, вы не сможете включить тигра в прайд львов), хотя некоторые редко используемые коллекции способны хранить разнотипные данные. Массивы реализованы в C# на низком уровне, а другие коллекции, представленные в этой главе, опираются на код из библиотеки `System.Collections.Generic`, как вы узнаете далее.

Наиболее распространенные коллекции

Далее следует краткий обзор некоторых наиболее распространенных коллекций. Если далее в главе коллекция описывается более подробно, рядом с ее именем в следующем списке будет стоять звездочка (*).

- **Массив***: массив — это индексируемый, упорядоченный список объектов. Определяя массив, вы должны задать его размер, который нельзя изменить позднее, что отличает массив от более гибкого типа `List`. В языке C# имеется также класс `Array`, но он отличается от простых массивов данных, которые

описываются далее в главе. Подобно большинству основных видов коллекций, массивы имеют несколько специальных функций класса. Но кроме того, массивы поддерживают доступ к отдельным элементам с помощью квадратных скобок и числовых индексов, благодаря чему есть возможность добавлять и читать объекты из массива с использованием имени переменной, представляющей этот массив, и `[]`, например:

```
stringArray[0] = "Hello";
stringArray[1] = "World";
print( stringArray[0]+" "+stringArray[1] ); // Выведет: Hello World
```

- **Список `List`***: тип `List` (или список) напоминает массив, но, в отличие от последнего, его размер не фиксирован и лишь немного уступает ему в производительности. В этой книге я буду использовать название типа `List` в языке C#, чтобы более четко отделить его от обычного слова «список». Тип `List` является одной из наиболее часто используемых коллекций в этой книге. Коллекции типа `List` поддерживают индексирование с квадратными скобками подобно массивам. Кроме того, тип `List` включает следующие методы:
  - `new List<T>()` — объявляет новую коллекцию `List` с элементами типа `T`[1].
  - `Add(X)` — добавляет объект `X` типа `T` в конец `List`.
  - `Clear()` — удаляет все объекты из `List`.
  - `Contains(X)` — возвращает `true`, если объект `X` (типа `T`) имеется в `List`.
  - `Count` — свойство[2], возвращающее количество объектов в `List`.
  - `IndexOf(X)` — возвращает числовой индекс объекта `X` в `List`. Если объект `X` отсутствует в коллекции, возвращается –1.
  - `Remove(X)` — удаляет объект `X` из List.
  - `RemoveAt(#)` — удаляет из `List` объект с индексом `#`.
- **Словарь*:** словари позволяют хранить пары *ключ-значение* и связывать объекты с определенными ключами. Примером словаря из обычной жизни может служить библиотека, где *ключ* в десятичной системе Дьюи позволяет читателям получить доступ к *значению*, то есть к конкретной книге. В отличие от всех других коллекций, рассматриваемых в этой главе, словари объявляются с двумя

---

[1] Тип `List` — это *обобщенная коллекция*. В C# слово *обобщенный* означает возможность использования с разными типами. Элемент `<T>` указывает, что при создании коллекции типа `List` вы должны определить тип ее элементов (например, `new List<GameObject>()` или `new List<Vector3>()`). Указание на обобщенный тип `<T>` вы также увидите в некоторых методах, таких как `GetComponent<T>()`, поддерживаемых игровыми объектами `GameObject`, где между угловыми скобками нужно указать тип компонента, который вы хотите получить (например, `gameObject.GetComponent<Rigidbody>()`).

[2] Будучи свойством, `Count` выглядит как поле, но в действительности это функция (см. главу 26 «Классы»).

типами (с типом ключа и типом значения)[1]. Добавлять и читать значения из словаря можно с помощью квадратных скобок (например, `dict["key"]`). Словари поддерживают следующие методы:

- `new Dictionary<Tkey, Tvalue>()` — объявляет новый словарь `Dictionary` с указанными типами ключей и значений.
- `Add(TKey, Tvalue)` — добавляет в словарь объект `Tvalue` с ключом `Tkey`.
- `Clear()` — удаляет все объекты из словаря.
- `ContainsKey(TKey)` — возвращает `true`, если ключ `TKey` имеется в словаре.
- `ContainsValue(TValue)` — возвращает `true`, если значение `TValue` имеется в словаре.
- `Count` — свойство, возвращающее количество пар *ключ-значение* в словаре.
- `Remove(TKey)` — удаляет из словаря значение с ключом `TKey`.

- **Очередь:** упорядоченная коллекция, действующая по принципу «первым пришел, первым ушел» (first-in, first-out, FIFO). Очередь `Queue` действительно напоминает очередь, в которой вы можете стоять, желая попасть в парк развлечений. Объекты добавляются в конец очереди вызовом `Enqueue()` и удаляются из начала очереди вызовом `Dequeue()`. Очереди поддерживают следующие методы:
  - `Clear()` — удаляет все объекты из очереди.
  - `Contains(X)` — возвращает `true`, если объект `X` имеется в очереди.
  - `Count` — свойство, возвращающее количество объектов в очереди.
  - `Dequeue()` — удаляет объект из начала очереди и возвращает его.
  - `Enqueue(X)` — добавляет объект `X` в конец очереди.
  - `Peek()` — возвращает объект из начала очереди, не удаляя его.
- **Стек:** упорядоченная коллекция, действующая по принципу «первым пришел, последним ушел» (first-in, last-out, FILO). Стек напоминает колоду карт. Объекты добавляются на вершину стека вызовом `Push()` и удаляются с вершины вызовом `Pop()`. Стеки поддерживают следующие методы:
  - `Clear()` — удаляет все объекты из стека.
  - `Contains(X)` — возвращает `true`, если объект `X` имеется в стеке.
  - `Count` — свойство, возвращающее количество объектов в стеке.
  - `Peek()` — возвращает объект с вершины стека, не удаляя его.
  - `Pop()` — удаляет и возвращает объект с вершины стека.
  - `Push(X)` — добавляет объект `X` на вершину стека.

[1] Функция `new Dictionary<Tkey, Tvalue>()` включает два обобщенных типа, позволяя создать словарь `Dictionary` с ключами типа `Tkey` и значениями типа `Tvalue`. Подробнее об этом рассказывается далее в главе.

Поскольку списки типа `List` являются самым часто используемым в книге видом коллекций, я начну с них, а затем перейду к массивам и словарям.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. Чтобы освежить в памяти стандартную процедуру настройки проекта, обращайтесь к приложению А «Стандартная процедура настройки проекта».

- **Имя проекта:** Collections Project
- **Имя сцены:** _Scene_Collections
- **Имена сценариев на C#:** ArrayEx, DictionaryEx, ListEx

Подключите все три сценария на C# к главной камере Main Camera в сцене _Scene_Collections.

Обобщенные коллекции

В начало каждого сценария на C# Unity автоматически добавляет три строки, начинающиеся со слова `using`[1]:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
```

Каждая из этих строк `using` загружает библиотеку кода и дает сценарию возможность использовать эти библиотеки. Первая строка — самая важная для проектов Unity, так как она открывает сценарию доступ ко всем стандартным объектам Unity, включая `MonoBehaviour`, `GameObject`, `Rigidbody`, `Transform` и т. д.

Вторая строка позволяет сценарию использовать некоторые *нетипизированные* коллекции, такие как `ArrayList` (которая встретится нам в учебных примерах). Нетипизированные коллекции могут одновременно хранить данные разных типов (например, строку в одном элементе и изображение или мелодию — в другом). Такая гибкость может привести к небрежности в программировании и усложнить отладку, поэтому я настоятельно советую избегать использования нетипизированных коллекций.

Третья строка особенно важна для этой главы, потому что открывает доступ к нескольким *обобщенным коллекциям*, включая `List` и `Dictionary`. Обобщенные коллекции являются *строго типизированными* в том смысле, что могут хранить эле-

[1] До версии Unity 5.5 строка `using System.Collections.Generic;` не добавлялась по умолчанию и разработчикам приходилось делать это вручную. Кроме того, в версии Unity 5.5 и выше эти строки следуют в другом порядке, но в данном случае порядок не имеет значения. Я расположил их так, потому что именно в таком порядке буду их описывать.

менты только одного типа, указанного в угловых скобках[1]. Вот несколько примеров объявления обобщенных коллекций (то есть создания конкретных экземпляров обобщенных коллекций):

- `public List<string> sList;` — объявляет список `List` строк.
- `public List<GameObject> goList;` — объявляет список `List` игровых объектов `GameObject`.
- `public Dictionary<char,string> acronymDict;` — объявляет словарь `Dictionary` строковых значений с символьными ключами (например, вы сможете использовать символ `'o'` для доступа к строке `"Okami"`).

В `System.Collections.Generic` также определяется несколько других обобщенных типов данных, которые не будут рассматриваться в этой главе. К ним относятся обобщенные версии очереди `Queue` и стека `Stack`, упоминавшиеся выше. В отличие от массивов, которые имеют фиксированный размер, все обобщенные коллекции способны изменять свои размеры динамически.

List

Щелкните дважды на сценарии ListEx в панели Project (Проект), чтобы открыть его в MonoDevelop, и добавьте следующий код, выделенный жирным. Комментарии в стиле `// a` в строках справа ссылаются на пояснения, следующие за листингом. Строки, которые вы должны добавить, выделены жирным.

```
 1 using System.Collections;                                    // a
 2 using System.Collections.Generic;                            // b
 3 using UnityEngine;                                           // c
 4
 5 public class ListEx : MonoBehaviour {
 6     public List<string>        sList;                        // d
 7
 8     void Start () {
 9         sList = new List<string>();                          // e
10         sList.Add( "Experience" );                           // f
11         sList.Add( "is" );
12         sList.Add( "what" );
13         sList.Add( "you" );
14         sList.Add( "get" );
15         sList.Add( "when" );
16         sList.Add( "you" );
17         sList.Add( "don't" );
```

[1] Для «обобщенных» (или универсальных) коллекций может показаться странным, что они способны хранить данные только одного типа. Слово *обобщенный* в данном случае означает возможность (посредством `<T>`) определить обобщенный тип данных, такой как `List`, который можно конкретизировать любым типом данных. Создание обобщенных классов, таких как `List`, выходит далеко за рамки этой книги, но вы можете поискать в интернете по фразе «обобщенные классы в C#», чтобы узнать больше.

```
18          sList.Add( "get" );
19          sList.Add( "what" );
20          sList.Add( "you" );
21          sList.Add( "wanted." );
22          // Цитата моего профессора, доктора Рэнди Пауша (Randy Pausch, 1960-2008)
23
24          print( "sList Count = "+sList.Count );                          // g
25          print( "The 0th element is: "+sList[0] );                       // h
26          print( "The 1st element is: "+sList[1] );
27          print( "The 3rd element is: "+sList[3] );
28          print( "The 8th element is: "+sList[8] );
29
30          string str = "";
31          foreach (string sTemp in sList) {                               // i
32              str += sTemp+" ";
33          }
34          print( str );
35      }
36  }
```

a. Подключение библиотеки `System.Collections` в начале всех сценариев на C# открывает доступ к типу `ArrayList` (в числе других). `ArrayList` — это еще один тип коллекций в C#, напоминающий `List`, но, в отличие от `List`, он не требует, чтобы все элементы имели один и тот же тип данных. Это дает больше гибкости, но, как мне кажется, от него больше вреда, чем пользы, по сравнению со списками `List` (в том числе и значительно страдает производительность).

b. Тип коллекций `List` является частью библиотеки `System.Collections.Generic`, поэтому ее обязательно нужно импортировать, чтобы получить возможность использовать тип `List`. В версии Unity 5.5 и выше эта строка добавляется автоматически, но в более ранних версиях ее приходится добавлять вручную. Как отмечалось выше, эта библиотека включает множество других обобщенных типов коллекций, кроме `List`. Чтобы узнать больше о них, попробуйте поискать в интернете по фразе «C# System.Collections.Generic».

c. Библиотека `UnityEngine` открывает доступ ко всем классам и типам, характерным для Unity (например, `GameObject`, `Renderer`, `Mesh`). Ее подключение в любом сценарии `MonoBehaviour` обязательно.

d. Объявляет коллекцию `List<string> sList`. Все объявления коллекций обобщенных типов должны сопровождаться угловыми скобками `<>`, заключающими конкретный тип. В данном случае объявляется коллекция — список строк. Сильная сторона обобщенных типов заключается в возможности создавать коллекции с элементами любых типов. Так же легко можно создать `List<int>`, `List<GameObject>`, `List<Transform>`, `List<Vector3>` и т. д. Объявляя коллекцию `List`, вы обязательно должны указать тип ее элементов.

e. В строке 6 объявляется имя переменной `sList`, которая может хранить список `List` строк, но сама переменная `sList` получает значение по умолчанию `null` (то есть не имеет значения), поэтому в строке 9 выполняется ее инициализация.

Любая попытка добавить элемент в `sList` до инициализации вызовет ошибку. В выражении инициализации `new` должен повторно указываться тип `List` с типом элементов. Инициализированный список `List` не содержит элементов, и его свойство `Count` вернет ноль.

f. Функция `Add()` класса `List` добавляет элемент в список. В данном случае в 0-й (нулевой) элемент списка вставляется строковый литерал `"Experience"`. Подробнее об индексации с нуля рассказывается во врезке «Индексация списков и массивов начинается с нуля».

g. Свойство `Count` класса `List` возвращает значение типа `int`, представляющее количество элементов в списке. Эта строка выведет:

```
sList Count = 12
```

h. Строки 25–28 демонстрируют использование квадратных скобок для доступа к элементам списка (например, `sList[0]`). Чтобы получить элемент списка, после имени переменной нужно добавить квадратные скобки `[]` и указать в них целое число, соответствующее позиции требуемого элемента в списке или в массиве. Целое число в квадратных скобках называют индексом. Эти строки выведут:

```
The 0th element is: Experience
The 1st element is: is
The 3rd element is: you
The 8th element is: get
```

i. Для обхода элементов списка `List` и других коллекций часто используется цикл `foreach` (представлен в предыдущей главе). Так же как строка является коллекцией символов, `List<string> sList` является коллекцией строк. Переменная `string sTemp` доступна только внутри инструкции `foreach` и исчезает после выхода из цикла. Поскольку списки `List` являются строго типизированными (то есть компилятор C# знает, что переменная `sList` имеет тип `List<string>`), элементы списка `sList` можно присваивать переменной `string sTemp` без любых дополнительных преобразований. Это одно из главных преимуществ типа `List` перед другим типом нетипизированных коллекций `ArrayList`. Эта строка выведет:

```
Experience is what you get when you don't get what you wanted.[1]
```

Как обычно, не забудьте сохранить сценарий в MonoDevelop, закончив правку. Затем вернитесь в Unity и выберите **Main Camera** в панели **Hierarchy** (Иерархия). Вы увидите, что в панели **Inspector** (Инспектор), в компоненте **ListEx (Script)**, появилась переменная **List<string> sList**. Если теперь запустить сцену в Unity, можно распахнуть раздел **sList**, щелкнув на значке с изображением треугольника слева, и увидеть значения в списке. Массивы и списки `List` отображаются в инспекторе, но словари — нет.

[1] Опыт — это то, что вы получаете, когда получаете не то, что хотите. — *Примеч. пер.*

ИНДЕКСАЦИЯ СПИСКОВ И МАССИВОВ НАЧИНАЕТСЯ С НУЛЯ

Индексация массивов и списков `List` *начинается с нуля*, то есть «первый» элемент коллекции в действительности имеет индекс [0]. На протяжении всей книги я буду называть этот элемент нулевым или 0-м.

Рассмотрим для примера *псевдокод* с коллекцией `coll`. Псевдокод — это код, не имеющий отношения ни к какому конкретному языку программирования и используемый для иллюстрации понятий.

```
coll = [ "A", "B", "C", "D", "E" ]
```

Счетчик элементов, или *длина*, коллекции `coll` равен 5, а допустимыми индексами элементов являются числа от `0` до `coll.Count-1` (то есть `0`, `1`, `2`, `3` и `4`).

```
print( coll.Count ); // 5

print( coll[0] );    // A
print( coll[1] );    // B
print( coll[2] );    // C
print( coll[3] );    // D
print( coll[4] );    // E

print( coll[5] );    // Исключение выхода за диапазон индексов!!!
```

Попытка обратиться к элементу с индексом не из этого диапазона приведет к следующей ошибке времени выполнения:

```
IndexOutOfRangeException: Array index is out of range.
```

Помните об этом, работая с любыми коллекциями в языке C#.

ВАЖНЫЕ СВОЙСТВА И МЕТОДЫ СПИСКОВ LIST

Списки `List` поддерживают множество свойств и методов, и ниже перечислены наиболее часто используемые из них. Все следующие примеры ссылаются на определение `List<string> sL` ниже и не связаны между собой. То есть в каждом примере предполагается, что первоначально список `List sL` находится в состоянии, определенном в трех следующих строках, и не подвергался изменениям в предыдущих примерах.

```
List<string> sL = new List<string>();
sL.Add( "A" ); sL.Add( "B" ); sL.Add( "C" ); sL.Add( "D" );
// В результате получается список List: [ "A", "B", "C", "D" ]
```

Свойства

Свойства возвращают информацию о списке `List`.

- `sL[2]` (доступ к элементу по индексу): вернет элемент списка `List` с указанным индексом (2). Так как во втором элементе списка хранится строка `"C"`, `sL[2]` вернет `"C"`.

- `sL.Count`: вернет текущее количество элементов в списке `List`. Так как длина списка `List` может изменяться с течением времени, свойство `Count` играет очень важную роль. Последний допустимый индекс в списке `List` всегда равен `Count-1`. Значение `sL.Count` равно 4, поэтому последний допустимый индекс равен 3.

Методы

Методы — это функции, позволяющие изменять список.

- `sL.Add("Hello")`: добавит параметр `"Hello"` в конец списка `sL`. После этого вызова `sL` будет хранить список: `[ "A", "B", "C", "D", "Hello" ]`.
- `sL.Clear()`: удалит все элементы из `sL`, в результате чего останется пустой список `sL`: `[ ]`.
- `sL.IndexOf("A")`: найдет первый элемент в списке `sL`, соответствующий параметру `"A"`, и вернет его индекс. Так как строка `"A"` хранится в 0-м элементе списка `sL`, этот вызов вернет `0`.
- Если элемент с указанным значением отсутствует в списке `List`, метод вернет значение `-1`. Это безопасный и быстрый способ проверки наличия элемента в списке `List`.
- `sL.Insert(2, "B.5")`: вставит второй параметр (`"B.5"`) в список `sL` как элемент с индексом в первом параметре (`2`). Все последующие элементы списка `List` сдвинутся к концу. После этого вызова `sL` будет хранить список: `[ "A", "B", "B.5", "C", "D" ]`. В первом параметре допускается передавать значения из диапазона от `0` до `sL.Count`. Значения не из этого диапазона вызовут ошибку времени выполнения.
- `sL.Remove("C")`: удалит указанный элемент из списка `List`. Если по стечению обстоятельств в списке окажутся две строки `"C"`, удалена будет только первая из них. После этого вызова `sL` будет хранить список: `[ "A", "B", "D" ]`.
- `sL.RemoveAt(0)`: удалит элемент с указанным индексом. Так как 0-й элемент хранит строку `"A"`, после этого вызова `sL` будет хранить список: `[ "B", "C", "D" ]`.

Преобразование списка в массив

Список `List` можно преобразовать в простой массив (описывается далее в этой главе). Это может пригодиться при использовании некоторых функций Unity, принимающих простой массив объектов вместо списка `List`.

- `sL.ToArray()`: сгенерирует массив, содержащий все элементы из списка `sL`. Тип нового массива будет соответствовать типу списка. В данном случае вызов метода вернет новый массив строк `[ "A", "B", "C", "D" ]`.

Прежде чем перейти к знакомству со словарями, остановите сцену в Unity и снимите флажок рядом с именем компонента **ListEx (Script)** в панели **Inspector** (Инспектор), чтобы сделать сценарий **ListEx** неактивным (как показано на рис. 23.1).

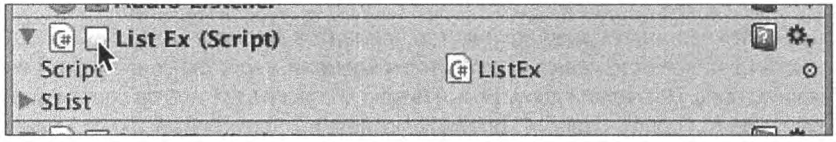


Рис. 23.1. Щелкните на флажке, чтобы сделать компонент ListEx (Script) неактивным

Dictionary

Словари `Dictionary` не отображаются в инспекторе, но, несмотря на это, могут оказаться фантастическим средством хранения информации. Одно из главных преимуществ словаря `Dictionary` — *постоянное время доступа* к элементам. То есть скорость доступа к элементам не зависит от их количества в словаре. Возьмите для сравнения список `List` или массив, где приходится перебирать все элементы один за другим: с увеличением размера списка `List` или массива время поиска конкретного элемента также увеличивается, особенно если искомый элемент оказывается последним в коллекции.

Словари хранят пары *ключ-значение*. После создания словаря к значениям можно обращаться по их ключам. Чтобы увидеть словарь в действии, откройте сценарий DictionaryEx и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;                                    // a
using UnityEngine;

public class DictionaryEx : MonoBehaviour {
    public Dictionary<string,string>    statesDict;                  // b

    void Start () {
        statesDict = new Dictionary<string, string>();               // c

        statesDict.Add( "MD", "Maryland" );                          // d
        statesDict.Add( "TX", "Texas" );
        statesDict.Add( "PA", "Pennsylvania" );
        statesDict.Add( "CA", "California" );
        statesDict.Add( "MI", "Michigan" );

        print("There are "+statesDict.Count+" elements in statesDict."); // e

        foreach (KeyValuePair<string,string> kvp in statesDict) {    // f
            print( kvp.Key + ": " + kvp.Value );
        }

        print( "MI is " + statesDict["MI"] );                        // g

        statesDict["BC"] = "British Columbia";                       // h

        foreach (string k in statesDict.Keys) {                      // i
            print( k + " is " + statesDict[k] );
```

```
29          }
30      }
31
32 }
```

a. Чтобы получить возможность использовать словари, нужно подключить библиотеку `System.Collections.Generic`.

b. Словарь `Dictionary` объявляется с типами ключей и значений, как здесь. В данном словаре ключи и значения имеют тип `string`, но вообще можно использовать любые другие типы.

c. Так же как список `List`, словарь `Dictionary` нельзя использовать до инициализации, как здесь.

d. Добавляя элементы в словарь `Dictionary`, вы должны передавать ключ и значение каждого элемента. Эти пять инструкций `Add` добавляют в словарь почтовые индексы и названия штатов, где я жил.

e. Подобно другим коллекциям в C#, словари поддерживают свойство `Count`, возвращающее количество хранящихся в них элементов. Эта строка выведет:

```
There are 5 elements in the Dictionary.
```

f. Для обхода элементов словаря можно использовать цикл `foreach`, но переменная цикла в этом случае должна иметь тип `KeyValuePair<,>`. Два типа в угловых скобках `KeyValuePair<,>` должны совпадать с типами, указанными в объявлении переменной словаря (в данном примере `<string,string>`). Этот цикл выведет:

```
MD: Maryland
TX: Texas
PA: Pennsylvania
CA: California
MI: Michigan
```

g. Если ключ известен, его можно использовать для доступа к значению элемента в словаре посредством квадратных скобок. Эта строка выведет:

```
MI: Michigan
```

h. Другой способ добавления значений в словарь — использовать квадратные скобки, как показано здесь. Да, мне довелось одно время жить в Британской Колумбии.

i. Для обхода элементов словаря в цикле `foreach` можно также использовать метод `Keys`. Эта строка выведет:

```
MD is Maryland
TX is Texas
PA is Pennsylvania
CA is California
MI is Michigan
BC is British Columbia
```

Сохраните сценарий DictionaryEx, вернитесь в Unity и щелкните на кнопке Play (Играть). Вы должны увидеть вывод, описанный выше. Напомню, что словари не отображаются в инспекторе Unity, поэтому, даже при том, что `statesDict` является общедоступной переменной, вы не увидите ее в панели Inspector (Инспектор).

ВАЖНЫЕ СВОЙСТВА И МЕТОДЫ СПИСКОВ DICTIONARY

Словари `Dictionary` поддерживают множество свойств и методов, но в этой врезке перечислены только наиболее часто используемые из них. Все следующие примеры ссылаются на определение `Dictionary<int, string> dIS` ниже и не связаны между собой. То есть в каждом примере предполагается, что первоначально словарь `Dictionary dIS` находится в состоянии, как определено в следующих строках, и не подвергался изменениям в предыдущих примерах.

```
Dictionary<int,string> dIS;
dIS = new Dictionary<int, string>();
dIS[0] = "Zero";
dIS[1] = "One";
dIS[10] = "Ten";
dIS[1234567890] = "A lot!";
```

Другой способ объявить и инициализировать тот же словарь:

```
dIS = new Dictionary<int, string>() {
    { 0, "Zero" },
    { 1, "One" },
    { 10, "Ten" },
    { 1234567890, "A lot!" }
};
```

Этот способ одновременного объявления и определения словаря `Dictionary` — один из редких случаев, когда требуется добавлять точку с запятой после закрывающей фигурной скобки.

Свойства

- `dIS[10]` (доступ к элементу по индексу): вернет значение из словаря `Dictionary` с указанным индексом (`10`). Так как ключу `10` соответствует значение `"Ten"`, `dIS[10]` вернет `"Ten"`. Попытка указать ключ, отсутствующий в словаре, приведет к ошибке времени выполнения `KeyNotFoundException`, которая аварийно завершит работу сценария.
- `dIS.Count`: вернет текущее количество пар ключ-значение в словаре `Dictionary`. Так как длина словаря `Dictionary` может изменяться с течением времени, свойство `Count` играет очень важную роль.

Методы

- `dIS.Add(12,"Dozen")`: добавит в словарь `Dictionary` значение `"Dozen"` с ключом `12`.

- `dIS[13] = "Baker's Dozen"`: добавит в словарь ключ 13 со значением `"Baker's Dozen"`. Если в квадратных скобках указать существующий ключ, существующее значение будет заменено новым. Например `dIS[0] = "None"` заменит значение ключа `0` на `"None"`.
- `dIS.Clear()`: удалит из `dIS` все пары ключ-значение, в результате чего останется пустой словарь.
- `dIS.ContainsKey(1)`: вернет `true`, если ключ 1 присутствует в словаре. Этот вызов выполняется очень быстро, потому что тип `Dictionary` поддерживает очень быстрый поиск по ключу. Ключи в словарях уникальны, то есть каждому ключу в словаре может соответствовать только одно значение.
- `dIS.ContainsValue("A lot!")`: вернет `true`, если значение `"A lot!"` присутствует в словаре. Это довольно медленный вызов, потому что словари оптимизируют поиск ключей, но не значений. Требование уникальности не предъявляются к значениям, то есть нескольким разным ключам могут соответствовать одинаковые значения.
- `dIS.Remove(10)`: удалит из словаря пару ключ-значение с ключом `10`.

Иногда желательно видеть некоторое подобие словаря `Dictionary` в инспекторе. В таких случаях я часто создаю простой список `List`, содержащий ключи и значения. Пример такого списка можно увидеть в главе 31 «Прототип 3.5: Space SHMUP Plus».

Прежде чем перейти к знакомству с массивами, остановите сцену в Unity и снимите флажок рядом с именем компонента DictionaryEx (Script) в панели Inspector (Инспектор), чтобы сделать сценарий `DictionaryEx` неактивным.

Массивы

Массив — простейший и самый быстрый тип коллекций. Для работы с массивами не требуется импортировать дополнительные библиотеки (инструкцией `using`), потому что они встроены в ядро языка C#. Кроме того, массивы могут быть многомерными и ступенчатыми, что иногда очень удобно.

Массивы имеют фиксированную длину, которая должна быть определена при инициализации. Щелкните дважды на сценарии ArrayEx в панели Project (Проект), чтобы открыть его в MonoDevelop, и введите следующий код:

```
1 using System.Collections;                                          // a
2 using UnityEngine;
3
4 public class ArrayEx : MonoBehaviour {
5     public string[] sArray;                                        // b
6
7     void Start () {
8         sArray = new string[10];                                   // c
```

```
 9
10          sArray[0] = "These";                                          // d
11          sArray[1] = "are";
12          sArray[2] = "some";
13          sArray[3] = "words";
14
15          print( "The length of sArray is: "+sArray.Length );           // e
16
17          string str = "";
18          foreach (string sTemp in sArray) {                            // f
19              str += "|"+sTemp;
20          }
21          print( str );
22      }
23  }
```

a. В отличие от списков и словарей, массивы не требуют импортировать библиотеку `System.Collections.Generic`.

b. Также, в отличие от списков и словарей, массивы в C# не являются отдельным типом данных — это коллекция, сформированная из другого доступного типа данных добавлением квадратных скобок после имени типа. Типом переменной `sArray` является не `string`, а `string[]` — коллекция нескольких строк. Обратите внимание: несмотря на то что переменная `sArray` объявляется как массив, длина массива не указывается в объявлении.

c. Здесь `sArray` инициализируется как массив `string[]` с длиной 10. При инициализации массива все его элементы заполняются значением по умолчанию для выбранного типа данных. Для `int[]` или `float[]` значением по умолчанию является `0`. Для `string[]` и других сложных типов данных, таких как `GameObject[]`, все элементы массива заполняются значением `null` (которое указывает на отсутствие значения).

d. Вместо метода `Add()`, как при работе со списками, для присваивания значений элементам стандартных массивов и чтения этих элементов используются квадратные скобки с индексами.

e. Вместо свойства `Count`, как обобщенные коллекции в C#, массивы поддерживают свойство `Length`. Важно отметить, что свойство `Length` (как можно видеть в выводе сценария) возвращает полную длину массива, включая инициализированные элементы (например, с `sArray[0]` по `sArray[3]` в сценарии) и пустые (то есть все еще хранящие неопределенное значение по умолчанию, как элементы с `sArray[4]` по `sArray[9]`). Эта строка выведет:

```
The length of sArray is 10.
```

f. Цикл `foreach` работает с массивами в точности как с другими коллекциями. Единственное отличие — `foreach` перебирает все элементы массива, в том числе пустые и со значением `null`. Так же как в предыдущем примере со списком `List<string>`, `sTemp` здесь — это временная строковая переменная, которой

в цикле присваивается значение каждого элемента массива sArray. Этот цикл выведет:

```
|These|are|some|words||||||
```

Запустив сценарий, выберите главную камеру Main Camera в панели Hierarchy (Иерархия). Это позволит распахнуть элемент sArray в компоненте ArrayEx (Script) в панели Inspector (Инспектор) и увидеть содержимое массива.

Сценарий выведет следующие строки:

```
The length of sArray is: 10
|These|are|some|words||||||
```

Пустые элементы в середине массива

В отличие от списков List[1], массивы могут иметь пустые элементы в середине. Это свойство можно использовать в игре для создания чего-то похожего на шкалу очков, на которой каждый игрок отображается своей меткой и между метками может быть пустое пространство.

Измените строки 12 и 13 в предыдущем сценарии, как показано ниже:

```
10          sArray[0] = "These";
11          sArray[1] = "are";
12          sArray[3] = "some";
13          sArray[6] = "words";
```

Этот код должен вывести: |These|are||some|||words|||

Как следует из этого вывода, теперь массив sArray имеет пустые элементы с индексами 2, 4, 5, 7, 8 и 9. Если индекс (как 0, 1, 3 и 6 здесь) находится в допустимых пределах для данного массива, вы можете использовать квадратные скобки для изменения значений любых его элементов, а цикл foreach благополучно их обработает.

Попытка присвоить значение по несуществующему индексу, то есть по индексу вне диапазона, заданного при инициализации массива (например, sArray[10] = "oops!"; или sArray[99] = "error!";), завершится ошибкой времени выполнения: IndexOutOfRangeException: Array index is out of range.

Попытка прочитать элемент с использованием несуществующего индекса завершится той же ошибкой. Например, print(sArray[20]); завершится ошибкой IndexOutOfRangeException.

Верните код в исходное состояние:

```
10          sArray[0] = "These";
11          sArray[1] = "are";
12          sArray[2] = "some";
13          sArray[3] = "words";
```

[1] Технически можно добавить в список значение null, чтобы создать пустой элемент в середине, но я так никогда не делал.

Пустые элементы массива и foreach

Снова запустите проект и посмотрите на вывод:

```
|These|are|some|words||||||
```

Инструкция `str += "|"+sTemp;` в строке 19 добавляет вертикальную черту (`|`) в конец `str` перед каждым элементом массива. Даже при том, что элементы с `sArray[4]` по `sArray[9]` хранят значение по умолчанию `null`, они все еще перечисляются циклом `foreach`. Это хорошая возможность использовать инструкцию перехода `break`, чтобы выйти из цикла `foreach` раньше. Измените код, как показано ниже:

```
18          foreach (string sTemp in sArray) {
19              str += "|"+sTemp;
20              if (sTemp == null) break;
21          }
```

Новый код выведет: `|These|are|some|words|`

Достигнув элемента `sArray[4]`, сценарий добавит `"|"+null` в конец строки `str`, затем проверит значение `sArray[4]`, обнаружит `null` и прервет цикл `foreach`, предотвратив обход элементов с `sArray[5]` по `sArray[9]`.

ВАЖНЫЕ СВОЙСТВА И МЕТОДЫ МАССИВОВ

Ниже в этой врезке перечислены наиболее часто используемые свойства и методы массивов. Все дальнейшие примеры ссылаются на следующее определение массива и не связаны между собой.

```
string[] sA = new string[] { "A", "B", "C", "D" };
// В результате получится массив: [ "A", "B", "C", "D" ]
```

Такой способ определения массива позволяет объявить, инициализировать и заполнить массив в одной строке (в противоположность инструкции инициализации массива, как показано в строке 8 в предыдущем листинге). Обратите внимание, что при таком способе инициализации массива значение свойства `Length` определяется количеством элементов между фигурными скобками и его не требуется указывать явно; в действительности, если в определении массива используются фигурные скобки, вы не сможете указать размер массива в квадратных скобках, не совпадающий с числом элементов в фигурных скобках.

Свойства

- `sA[2]` (доступ к элементу по индексу): вернет элемент массива с указанным индексом (2). Так как во втором элементе массива хранится строка `"C"`, `sA[2]` вернет `"C"`.
- Если индекс находится вне допустимого диапазона для данного массива (для `sA` он охватывает индексы от 0 до 3), его использование приведет к ошибке времени выполнения `IndexOutOfRangeException`.

- `sA[1] = "Bravo"` (присваивание значения элементу по индексу): присвоит значение справа от оператора присваивания = элементу массива с указанным индексом, заменив предыдущее значение. В результате этой операции `sA` будет хранить массив: `[ "A", "Bravo", "C", "D" ]`.
- Если индекс находится вне допустимого диапазона для данного массива, его использование приведет к ошибке времени выполнения `IndexOutOfRangeException`.
- `sA.Length`: вернет общий размер массива. Учитываются все элементы, независимо от того, было им присвоено какое-то значение или они все еще хранят значение по умолчанию. В данном случае вернет 4.

Статические методы

Статические методы массивов являются частью класса `System.Array` (то есть определены в библиотеке `System.Collections`) и добавляют в массивы некоторые возможности, свойственные спискам `List`.

- `System.Array.IndexOf( sA, "C" )`: найдет первый элемент в массиве `sA` со строкой `"C"` и вернет его индекс. Так как строка `"C"` хранится во втором элементе массива `sA`, этот вызов вернет 2.
- Если искомое значение отсутствует в массиве, возвращается `-1`. Эта особенность часто используется, чтобы выяснить присутствие в массиве конкретного элемента.
- `System.Array.Resize( ref sA, 6 )`: изменит размер массива. В первом параметре этот метод принимает ссылку на экземпляр массива (именно поэтому требуется использовать ключевое слово `ref`), а во втором — новый размер. В результате этой операции `sA` будет хранить массив: `[ "A", "B", "C", "D", null, null ]`.
- Если во втором параметре передать новый размер меньше текущего значения `Length`, лишние элементы в конце массива будут отброшены. Вызов `System.Array.Resize( ref sA, 2 )` превратит массив `sA` в: `[ "A", "B" ]`. `System.Array.Resize()` не работает с многомерными массивами, которые описываются далее в этой главе.

Преобразование массива в список List

Как рассказывалось в разделе с описанием типа `List`, списки можно преобразовывать в простые массивы. Также возможно обратное преобразование массивов в списки `List`.

- `List<string> sL = new List<string>( sA )`: эта строка создаст список `List` `sL`, содержащий копии элементов массива `sA`.

Для объявления, инициализации и заполнения списка `List` также можно использовать выражение инициализации массива, хотя это не очень удобно:

- `List<string> sL = new List<string>( new string[] { "A", "B", "C" } );`

Эта строка объявляет, определяет и заполняет новый анонимный массив `string[]`, который тут же передается в вызов функции `new List<string>()`.

Прежде чем перейти к следующему примеру, деактивируйте сценарий ArrayEx, сняв флажок рядом с его именем в панели Inspector (Инспектор).

Многомерные массивы

Создание многомерных массивов с двумя или более индексами часто очень удобно. При работе с подобными массивами в квадратных скобках можно вместо одного использовать два или более индекса. Таким способом можно создать двумерную таблицу с элементами-ячейками.

Создайте новый сценарий на C# с именем Array2dEx и подключите его к главной камере Main Camera. Откройте Array2dEx в MonoDevelop и введите следующий код:

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class Array2dEx : MonoBehaviour {
 6
 7     public string[,]                sArray2d;
 8
 9     void Start () {
10         sArray2d = new string[4,4];
11
12         sArray2d[0,0] = "A";
13         sArray2d[0,3] = "B";
14         sArray2d[1,2] = "C";
15         sArray2d[3,1] = "D";
16
17         print( "The Length of sArray2d is: "+sArray2d.Length );
18     }
19 }
```

Этот сценарий выведет: `The Length of sArray2d is: 16`

Как видите, свойство `Length` возвращает единственное целое число даже для многомерных массивов. Оно просто отражает общее количество элементов в массиве, поэтому программист должен сам следить за каждым измерением массива.

Теперь реализуем форматированный вывод значений из массива `sArray2d`. Закончив, мы должны получить такой вывод:

```
|A| | |B|
| | |C| |
| | | | |
| |D| | |
```

Как видите, значение `A` здесь находится в 0-й строке и в 0-м столбце (`[0,0]`), значение `B` — в 0-й строке и в 3-м столбце (`[0,3]`), и т. д. Для реализации такого форматирования добавьте в сценарий строки, выделенные жирным в следующем листинге:

```
17          print( "The Length of sArray2d is: "+sArray2d.Length );
18          string str = "";
19          for ( int i=0; i<4; i++ ) {                                    // a
20              for ( int j=0; j<4; j++ ) {
21                  if (sArray2d[i,j] != null) {                           // b
22                      str += "|"+sArray2d[i,j];
23                  } else {
24                      str += "|_";
25                  }
26              }
27              str += "|"+"\n";                                           // c
28          }
29          print( str );
30      }
31  }
```

a. Строки 19 и 20 демонстрируют использование двух вложенных циклов `for` для обхода элементов двумерного массива. Вот как они действуют:

   - Итерации начинаются с `i=0` (строка 19).
   - Выполняется обход всех значений `j`, от 0 до 3 (строки с 20-й по 26-ю).

     Переменная `str` получает значение `"|A| | |B|\n"` (строка 27).
   - Переменная `i` увеличивается до `1` (строка 19).
   - Выполняется обход всех значений `j`, от 0 до 3 (строки с 20-й по 26-ю).

     Переменная `str` получает значение `"|A| | |B|\n| | |C| |\n"` (строка 27).
   - Переменная `i` увеличивается до `2` (строка 19).
   - Выполняется обход всех значений `j`, от 0 до 3 (строки с 20-й по 26-ю).

     Переменная `str` получает значение `"|A| | |B|\n| | |C| |\n| | |\n"` (строка 27).
   - Переменная `i` увеличивается до `3` (строка 19).
   - Выполняется обход всех значений `j`, от 0 до 3 (строки с 20-й по 26-ю).

     Переменная `str` получает значение `"|A| | |B|\n| | |C| |\n| | |\n|D| | |\n"` (строка 27).

   Это гарантирует обход элементов многомерного массива по порядку. В данном примере таблицы код последовательно обходит все элементы строки (увеличивая `j` с 0 до 3) и затем перемещается на следующую строку, увеличивая `i`.

b. Строки 21–25 сравнивают строку в элементе `sArray[i,j]` со значением `null`. Если они отличаются, в конец переменной `str` добавляется символ вертикальной черты и значение `sArray2d[i,j]`. Если элемент содержит `null`, в конец `str` добавляется символ вертикальной черты и один пробел. Символ вертикальной черты на клавиатуре находится над клавишей Return (или Enter). Обычно его можно ввести, нажав комбинацию Shift+\ (обратный слеш).

c. Эта строка выполняется после перебора всех значений j во вложенном цикле for, но перед началом следующей итерации по i во внешнем цикле. Она добавляет в str завершающий символ вертикальной черты и символ перевода строки, что придает выводу законченный вид[1].

Обновленный сценарий выведет:

```
The Length of sArray2d is: 16
|A| | |B|
| | |C| |
| | | | |
| |D| | |
```

В панели Console (Консоль) вы увидите только первые две строки таблицы с содержимым массива sArray2d. Но если щелкнуть на них, в нижней половине консоли откроется весь вывод целиком (рис. 23.2).

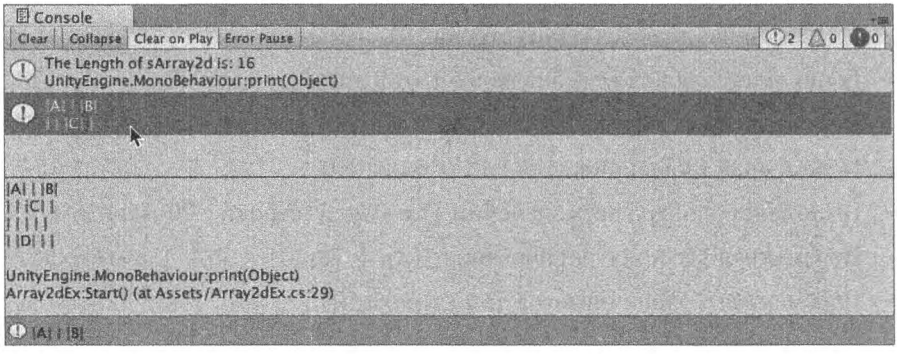


Рис. 23.2. Щелчок на сообщении в консоли приводит к отображению полного текста ниже. Обратите внимание, что самое свежее сообщение также отображается в левом нижнем углу консоли

Как показано на рисунке, вид отформатированного текста не точно соответствует задуманному в панели Console (Консоль), потому что она использует *не моноширинный шрифт* (то есть шрифт, в котором символы *i* и *m* имеют разную ширину; в моноширинном шрифте для символов i и m отводится пространство одинаковой ширины). Вы можете щелкнуть на любой строке в панели Console (Консоль), выбрать пункт меню Edit > Copy (Правка > Копировать), чтобы скопировать данные, и затем вставить скопированный текст в окно другой программы. Я часто поступаю так и выполняю вставку в окне текстового редактора. (Я предпочитаю *BBEdit*[2]

[1] Строку "\n" компилятор C# воспринимает как один символ, отмечающий начало новой строки. Он обеспечивает перенос строк при выводе на экран.

[2] На сайте компании BareBones Software (http://www.barebones.com) имеется бесплатная демонстрационная версия *BBEdit*.

в macOS или *EditPad Pro*[1] в Windows, оба редактора обладают довольно широкими возможностями.)

Также вы должны знать, что панель Inspector (Инспектор) в Unity не отображает многомерные массивы. Так же как со словарями, если инспектор не знает, как правильно отобразить переменную, он просто игнорирует ее, поэтому имя многомерного массива, даже общедоступного, не появится в панели Inspector (Инспектор).

Остановите сцену в Unity, еще раз щелкнув на кнопке Play (Играть), чтобы исчезла синяя подсветка, и затем деактивируйте компонент Array2dEx (Script) в инспекторе.

Ступенчатые массивы

Ступенчатый массив — это массив массивов. Он напоминает многомерный массив, но может иметь вложенные массивы разной длины. Мы создадим ступенчатый массив, хранящий следующие данные:

```
| A | B | C | D |
| E | F | G |
| H | I |
| J |   |   | K |
```

Как видите, 0-я и 3-я строки содержат по четыре элемента, а строки 1 и 2 содержат три и два элемента соответственно. Обратите также внимание, что допускается наличие пустых элементов, как в показано 3-й строке. На самом деле C# допускает даже случай, когда вся строка может быть пустой (хотя в коде, представленном в следующем листинге, это вызвало бы ошибку в строке 33).

Создайте новый сценарий на C# с именем JaggedArrayEx и подключите его к главной камере Main Camera. Откройте JaggedArrayEx в MonoDevelop и введите следующий код:

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class JaggedArrayEx : MonoBehaviour {
 6     public string[][]        jArray;                                    // a
 7
 8     void Start () {
 9         jArray = new string[4][];                                       // b
10
11         jArray[0] = new string[4];                                      // c
12         jArray[0][0] = "A";
13         jArray[0][1] = "B";
```

[1] На сайте компании Just Great Software (http://editpadpro.com) имеется бесплатная демонстрационная версия *EditPad Pro*.

```
14          jArray[0][2] = "C";
15          jArray[0][3] = "D";
16
17          // Примеры инициализации массива в одной строке                  // d
18          jArray[1] = new string[] { "E", "F", "G" };
19          jArray[2] = new string[] { "H", "I" };
20
21          jArray[3] = new string[4];                                       // e
22          jArray[3][0] = "J";
23          jArray[3][3] = "K";
24
25          print( "The Length of jArray is: "+jArray.Length );             // f
26          // Выведет: The Length of jArray is: 4
27
28          print( "The Length of jArray[1] is: "+jArray[1].Length );       // g
29          // Выведет: The Length of jArray[1] is: 3
30
31          string str = "";
32          foreach (string[] sArray in jArray) {                            // h
33              foreach( string sTemp in sArray ) {
34                  if (sTemp != null) {
35                      str += " | "+sTemp;                                  // i
36                  } else {
37                      str += " | ";                                        // j
38                  }
39              }
40              str += " | \n";
41          }
42
43          print( str );
44      }
45  }
```

a. Строка 6 объявляет `jArray` как ступенчатый массив (то есть массив массивов). Если `string[]` — это коллекция строк, то `string[][]` — это коллекция массивов строк (массивов типа `string[]`).

b. Строка 9 определяет `jArray` как ступенчатый массив с длиной 4. Обратите внимание, что вторая пара квадратных скобок остается пустой, подсказывая, что вложенные массивы могут быть любой длины, хотя после определения их длина фиксируется и изменить ее становится непросто.

c. Строка 11 определяет 0-й элемент `jArray` как массив строк с длиной 4. Строки 12–15 вставляют элементы в этот вложенный массив, используя первую пару квадратных скобок (`[0]`) для доступа к 0-му вложенному массиву в `jArray`, а вторую пару — для вставки строк во все четыре элемента вложенного массива.

d. В строках 18 и 19 используется однострочная форма определения массива. Так как элементы массива перечисляются в фигурных скобках, длину массива можно не указывать явно (поэтому в `new string[]` пустые квадратные скобки).

e. В строках 21–23 определяется 3-й элемент `jArray` как `string[]` с длиной 4, и затем заполняются 0-й и 3-й элементы этого массива `string[]`, а элементы 1 и 2 остаются пустыми.

f. Строка 25 выводит `"The Length of jArray is: 4"`. Так как `jArray` — это массив массивов (а не многомерный массив), `jArray.Length` подсчитывает количество элементов, доступных посредством первой пары квадратных скобок (то есть четыре вложенных массива).

g. Строка 28 выводит `"The Length of jArray[1] is: 3"`. Так как `jArray` — это массив массивов, определить длину вложенного массива не составляет труда.

h. При работе со ступенчатыми массивами цикл `foreach` различает основной и вложенные массивы. При обходе `jArray` цикл `foreach` выполняет итерации по вложенным массивам `string[]` (массивам строк), элементам `jArray`, а при обходе любого из вложенных массивов `string[]` цикл `foreach` выполняет итерации по строкам внутри него. Обратите внимание, что `sArray` имеет тип `string[]` (массив строк), а `sTemp` имеет тип `string`.

   Как я уже упоминал, строка 33 могла бы возбудить ошибку `NullReferenceException`, если бы один из элементов `jArray` имел значение `null`. В этом случае переменная `sArray` получила бы значение `null`, и попытка применить к ней инструкцию `foreach` в строке 33 привела бы к ошибке `NullReferenceException` (попытка сослаться на элемент по пустой ссылке `null`). Инструкция `foreach` могла бы попытаться обратиться к данным в `sArray`, как `sArray.Length` и `sArray[0]`. Так как значение `null` не имеет элементов, обращение к `null.Length` вызывает эту ошибку.

i. Строковый литерал в строке 35 должен вводиться с клавиатуры как пробел, вертикальная черта, пробел.

j. Строковый литерал в строке 37 должен вводиться с клавиатуры как пробел, вертикальная черта, пробел, пробел.

Этот сценарий выведет следующие строки в панели Console (Консоль):

```
The Length of jArray is: 4
The Length of jArray[1] is: 3
| A | B | C | D |
| E | F | G |
| H | I |
| J |   |   | K |
```

Обход ступенчатых массивов с помощью for вместо foreach

Для обхода элементов ступенчатых массивов также можно использовать циклы `for`, опираясь на свойство `Length` основного и вложенных массивов. Например, строки 32–41 в предыдущем листинге можно заменить следующим кодом:

```
        string str = "";
        for (int i=0; i<jArray.Length; i++) {
            for (int j=0; j<jArray[i].Length; j++) {
                if (jArray[i][j] != null) {
                    str += " | "+jArray[i][j];
                } else {
                    str += " | ";
                }
            }
            str += " | \n";
        }
```

Этот код выведет то же самое, что и предыдущий вариант с циклами `foreach`. Выбор между `for` и `foreach` во многом зависит от конкретной ситуации.

Ступенчатые списки

Наконец, завершая обсуждение ступенчатых коллекций, замечу, что также возможно создать ступенчатый список `List`. Ступенчатый двумерный список строк можно объявить так: `List<List<string>> jaggedStringList`. По аналогии со ступенчатыми массивами, вложенные списки получают значение по умолчанию `null`, поэтому их нужно инициализировать при добавлении, как показано в следующем листинге. Как и любые списки, ступенчатые списки не могут иметь пустых элементов. Создайте новый сценарий на C# с именем `JaggedListTest`, подключите его к главной камере Main Camera и введите в него следующий код:

```
using System.Collections;                                                    // a
using System.Collections.Generic;
using UnityEngine;

public class JaggedListTest : MonoBehaviour {
    public List<List<string>> jaggedList;

    // Используйте этот метод для инициализации
    void Start () {
        jaggedList = new List<List<string>>();

        // Добавить два списка List<string> в jaggedList
        jaggedList.Add( new List<string>() );
        jaggedList.Add( new List<string>() );

        // Добавить две строки в jaggedList[0]
        jaggedList[0].Add ("Hello");
        jaggedList[0].Add ("World");

        // Добавить третий список List<string> в jaggedList, включая данные
        jaggedList.Add ( new List<string>( new string[] {"complex",
        ➥"initialization"}) );                                              // b

        string str = "";
        foreach (List<string> sL in jaggedList) {
```

```
25              foreach (string sTemp in sL) {
26                  if (sTemp != null) {
27                      str += " | "+sTemp;
28                  } else {
29                      str += " | ";
30                  }
31              }
32              str += " | \n";
33          }
34          print( str );
35      }
36  }
```

a. Инструкция `using System.Collections;` автоматически добавляется во все сценарии на C# в Unity, но вообще она не нужна для работы со списками `List` (достаточно импортировать библиотеку `System.Collections.Generic`).

b. Здесь впервые в книге используется символ продолжения кода ➥. Он будет использоваться везде, где строки кода не умещаются по ширине книжной страницы.

Вы не должны вводить символ ➥; он используется, только чтобы показать, что код должен вводиться в одну строку. Без пробелов в начале строка 21 должна выглядеть так:

```
jaggedList.Add ( new List<string>( new string[] {"complex","initialization"} ) );
```

Этот сценарий выведет в панель Console (Консоль) следующие строки:

```
| Hello | World |
|
| complex | initialization |
```

Когда использовать массивы или списки

Массивы и списки очень похожи, поэтому многие часто теряются, не зная, какую коллекцию лучше выбрать в той или иной ситуации. Вот основные отличия между массивами и списками:

- Списки легко изменяют свою длину, тогда как с массивами проделать это намного сложнее.
- Массивы работают быстрее, но ненамного.
- Массивы поддерживают многомерные индексы.
- Массивы допускают наличие пустых элементов в середине коллекции.

Списки проще в реализации и менее строги (благодаря гибкому изменению длины), поэтому лично я использую списки намного чаще, чем массивы. Это особенно актуально при разработке прототипов игр, когда гибкость особенно важна.

Итоги

Теперь, овладев списками, словарями и массивами, вы легко сможете работать с большим количеством объектов в своих играх. Например, можете вернуться к проекту Hello World из главы 19 «Hello World: ваша первая программа» и добавить `List<GameObject>` в сценарий CubeSpawner, чтобы каждый новый кубик включался в этот список. Это даст вам ссылку на каждый кубик и возможность манипулировать ими после создания. Следующее упражнение показывает, как это реализовать.

Итоговое упражнение

В этом упражнении мы вернемся к проекту Hello World из главы 19 и напишем сценарий, добавляющий каждый новый кубик в список `List<GameObject>` с именем `gameObjectList`. В каждом кадре мы будем уменьшать масштаб кубиков до 95% от их размеров в предыдущем кадре. Когда кубик достигнет масштаба 0,1 или меньше, он будет удаляться из сцены и из списка `gameObjectList`.

Однако удаление элемента из `gameObjectList`, пока цикл `foreach` выполняет итерации, вызовет ошибку. Чтобы избежать этого, кубики, подлежащие удалению, будут временно сохраняться в другом списке, с именем `removeList`, и затем будут выполняться итерации по этому списку и удаляться элементы из `gameObjectList`. (Начав изучать листинг, вы поймете, что я имею в виду.)

Откройте проект Hello World и создайте новую сцену (выбрав в главном меню пункт File > Scene (Файл > Сцена)). Сохраните сцену с именем _Scene_3. Создайте новый сценарий с именем CubeSpawner3 и подключите его к главной камере Main Camera в сцене. Затем откройте CubeSpawner3 в MonoDevelop и введите следующий код:

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class CubeSpawner3 : MonoBehaviour {
 6     public GameObject        cubePrefabVar;
 7     public List<GameObject>  gameObjectList; // Будет хранить все кубики
 8     public float             scalingFactor = 0.95f;
 9     // ^ Коэффициент изменения масштаба каждого кубика в каждом кадре
10     public int               numCubes = 0; // Общее количество кубиков
11
12     // Используйте этот метод для инициализации
13     void Start () {
14         // Инициализация списка List<GameObject>
15         gameObjectList = new List<GameObject>();
16     }
17
18     // Update вызывается в каждом кадре
19     void Update () {
20         numCubes++; // Увеличить количество кубиков                    // a
21
```

```
22          GameObject gObj = Instantiate<GameObject>( cubePrefabVar );     // b
23
24          // Следующие строки устанавливают некоторые значения в новом кубике
25          gObj.name = "Cube "+numCubes; // c
26          Color c = new Color(Random.value, Random.value, Random.value);  // d
27          gObj.GetComponent<Renderer>().material.color = c;
28          // ^ Получить компонент Renderer из gObj и назначить случайный цвет
29          gObj.transform.position = Random.insideUnitSphere;              // e
30
31          gameObjectList.Add (gObj); // Добавить gObj в список кубиков
32
33          List<GameObject> removeList = new List<GameObject>();           // f
34          // ^ Список removeList будет хранить кубики, подлежащие
35          //    удалению из списка gameObjectList
36
37          // Обход кубиков в gameObjectList
38          foreach (GameObject goTemp in gameObjectList) {                 // g
39
40              // Получить масштаб кубика
41              float scale = goTemp.transform.localScale.x;                // h
42              scale *= scalingFactor; // Умножить на коэффициент scalingFactor
43              goTemp.transform.localScale = Vector3.one * scale;
44
45              if (scale <= 0.1f) { // Если масштаб меньше 0.1f...          // i
46                  removeList.Add (goTemp); // ...добавить кубик в removeList
47              }
48          }
49
50          foreach (GameObject goTemp in removeList) {                     // g
51              gameObjectList.Remove (goTemp);                             // j
52              // ^ Удалить кубик из gameObjectList
53              Destroy (goTemp); // Удалить игровой объект кубика
54          }
55      }
56  }
```

a. Оператор инкремента (`++`) используется, чтобы увеличить поле, хранящее общее число созданных кубиков.

b. Создание экземпляра `cubePrefabVar`. Объявление обобщенного типа `<GameObject>` здесь необходимо, потому что `Instantiate()` можно использовать для создания любых объектов (то есть без объявления обобщенного типа компилятор C# не сможет узнать, какого типа данные вернет `Instantiate()`). Объявление `<GameObject>` сообщает компилятору, что функция `Instantiate()` вернет объект типа `GameObject`.

c. Переменная `numCubes` используется с целью создания уникальных имен для кубиков. Первый кубик получит имя *Cube 1*, второй — *Cube 2*, и т. д.

d. Строки 26 и 27 присваивают каждому кубику случайно выбранный цвет. Цвет определяется материалом в компоненте Renderer, присоединенном к игровому объекту `GameObject`, как можно видеть в строке 27.

e. `Random.insideUnitSphere` возвращает случайное местоположение внутри сферы с радиусом 1 и центром в точке [0,0,0]. Этот код создает кубики не в одной точке игрового пространства, а в случайных местах, по соседству с точкой [0,0,0].

f. Как отмечается в комментарии к коду, `removeList` будет использоваться для хранения кубиков, подлежащих удалению из `gameObjectList`. Это необходимо, потому что C# не позволяет удалять элементы из списка, пока цикл `foreach` выполняет итерации по нему. (Например, нельзя вызвать `gameObjectList.Remove()` в теле цикла `foreach`, в строках 38–48, выполняющий итерации по элементам `gameObjectList`.)

g. Этот цикл `foreach` выполняет итерации по всем кубикам в `gameObjectList`. Обратите внимание, что в этом цикле используется временная переменная `goTemp`. Это же имя `goTemp` используется для временной переменной в цикле `foreach` в строке 50, то есть `goTemp` объявляется в обеих строках, 38 и 50. Поскольку в каждом случае область видимости `goTemp` ограничена телом цикла `foreach`, двойное объявление переменной в одной функции `Update()` не вызывает никаких проблем. За дополнительной информацией обращайтесь к разделу «Области видимости переменных» в приложении Б «Полезные идеи».

h. В строках 41–43 определяется текущий масштаб кубика (чтением размера по оси X из `transform.localScale`), затем этот масштаб умножается на 95% и новое значение сохраняется в `transform.localScale`. Операция умножения `Vector3` на число типа `float` (как в строке 43) умножит каждый элемент вектора на это число, то есть [2, 4, 6] * 0.5f вернет [1, 2, 3].

i. Как отмечается в комментарии к коду, если новый масштаб оказался меньше 0.1f, кубик добавляется в `removeList`.

j. Цикл `foreach` в строках 50–54 выполняет итерации по элементам в списке `removeList` и удаляет все кубики, содержащиеся в нем, из списка `gameObjectList`. Так как в данном случае `foreach` выполняет итерации по списку `removeList`, удаление элементов из `gameObjectList` не вызывает ошибок. Удаленный из списка игровой объект кубика продолжит отображаться на экране, пока он не будет уничтожен вызовом метода `Destroy()`. Но даже после этого кубик продолжит храниться в памяти компьютера, потому что он все еще остается элементом списка `removeList`. Однако, так как `removeList` является локальной переменной с областью видимости, ограниченной телом функции `Update()`, после завершения `Update()` переменная `removeList` прекратит существование и все объекты, хранившиеся только в `removeList`, также удалятся из памяти.

Сохраните сценарий и вернитесь в Unity. Если вы действительно хотите создавать кубики, свяжите Cube Prefab из панели Project (Проект) с переменной `cubePrefabVar` в компоненте Main Camera:CubeSpawner3 (Script) в панели Inspector (Инспектор).

После этого щелкните на кнопке Play (Играть) в Unity, и вы увидите, что на экране появится множество кубиков, как в предыдущей версии проекта Hello World. Но

на этот раз кубики будут разных цветов, они уменьшатся с течением времени и, наконец, исчезнут (а не продолжат существовать до бесконечности, как в прежней версии).

Так как сценарий CubeSpawner3 хранит ссылку на каждый кубик в списке `gameObjectList`, он может изменять масштаб кубиков в каждом кадре и затем уничтожить их, когда масштаб окажется меньше 0,1f. При коэффициенте `scalingFactor`, равном 0,95f, каждый кубик будет существовать на протяжении 45 кадров, пока его масштаб не станет меньше 0.1f, отсюда следует, что всегда удаляться и уничтожаться будет 0-й кубик в `gameObjectList`, и свойство `Count` списка `gameObjectList` замрет на величине 45.

Что дальше

В следующей главе вы узнаете, как создавать функции с именами, отличными от `Start()` и `Update()`.

24 Функции и параметры

В этой главе вы научитесь пользоваться огромной силой функций. Вы напишете свои собственные функции, которые принимают разные переменные как входные аргументы и возвращают единственную переменную как результат. Я также расскажу о некоторых особых приемах определения параметров функций, таких как перегрузка функций, необязательные параметры и модификатор `params`, каждый из которых поможет вам писать код более эффективный, модульный, гибкий и поддерживающий многократное использование.

Настройка проекта Function Examples

В приложении А «Стандартная процедура настройки проекта» приводятся подробные инструкции, как настраивать проекты Unity для глав этой книги. В начале описания каждого проекта вы будете видеть врезку, такую как здесь. Выполните рекомендации во врезке, чтобы настроить проект для этой главы.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. За информацией о стандартной процедуре обращайтесь к приложению А.

- **Имя проекта:** Function Examples
- **Имя сцены:** _Scene_Functions
- **Имена сценариев на C#:** CodeExample

Подключите сценарий CodeExample к главной камере Main Camera в сцене _Scene_Functions.

Определение функции

В действительности вы писали функции, начиная с первой программы Hello World, но до этого момента вы просто добавляли новый код во встроенные функции Unity, такие как `Start()` и `Update()`. С этого момента вы начнете писать свои функции.

Функция — это фрагмент кода, выполняющий некоторые действия. Например, чтобы подсчитать количество вызовов функции `Update()`, можно создать новый сценарий на C# со следующим кодом (вы должны добавить строки, выделенные жирным):

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class CodeExample : MonoBehaviour {
 5
 6     public int        numTimesCalled = 0;                                  // a
 7
 8     void Update() {
 9         numTimesCalled++;                                                  // b
10         PrintUpdates();                                                    // c
11     }
12
13     void PrintUpdates() {                                                  // d
14         string outputMessage = "Updates: "+numTimesCalled;                 // e
15         print( outputMessage ); // Пример вывода: "Updates: 1"             // f
16     }
17
18 }
```

a. Объявляется общедоступная переменная `numTimesCalled` и получает начальное значение 0. Поскольку переменная `numTimesCalled` объявлена общедоступной в пределах класса `CodeExample`, но за пределами какой-либо функции, она доступна из любой функции в классе `CodeExample`.

b. Увеличивается значение `numTimesCalled` (то есть к ней прибавляется 1).

c. Строка 10 *вызывает* функцию `PrintUpdates()`. Когда код вызывает функцию, она выполняется. Более подробно об этом я расскажу чуть ниже.

d. Строка 13 *объявляет* функцию `PrintUpdates()`. Объявление функции напоминает объявление переменной. `Void` — это тип значения, возвращаемого функцией, в данном случае функция ничего не возвращает (как я подробнее расскажу ниже). Строки 13–16 все вместе *определяют* функцию. Все строки в коде между открывающей фигурной скобкой `{` в строке 13 и закрывающей `}` в строке 16 являются частью определения `PrintUpdates()`.

   Обратите внимание, что порядок объявления функций в классе не имеет значения. Какая из двух функций будет объявлена первой — `PrintUpdates()` или `Update()`, совершенно неважно, если они находятся в фигурных скобках класса `CodeExample`. Перед выполнением любого кода C# сначала просматривает все объявления в классе. Функцию `PrintUpdates()` можно вызвать в строке 10 и объявить в строке 13, потому что обе функции — `PrintUpdates()` и `Update()` — объявлены в одном классе `CodeExample`.

e. Строка 14 объявляет *локальную* строковую переменную с именем `outputMessage`. Так как она определяется внутри `PrintUpdates()`, ее область видимости ограничивается телом функции `PrintUpdates()`, то есть за пределами функции

`PrintUpdates()` переменная с именем `outputMessage` недоступна. За дополнительной информацией обращайтесь к разделу «Области видимости переменных» в приложении Б, «Полезные идеи».

Строка 14 также определяет переменную `outputMessage` как конкатенацию строки `"Updates: "` и значения общедоступной целочисленной переменной `numTimesCalled`.

`f.` Функция `print()` в Unity принимает единственный *аргумент*, в данном случае `outputMessage`. Она выводит значение `outputMessage` на консоль Unity. Я расскажу об аргументах функций далее в этой главе.

В настоящей игре функция `PrintUpdates()` едва ли пригодится, но она демонстрирует две важные идеи, рассматриваемые в этой главе:

- **Функции инкапсулируют действия:** функцию можно считать именованной коллекцией строк кода. Эти строки выполняются каждый раз, когда вызывается функция. Это продемонстрировали обе функции — `PrintUpdates()` и пример `BuySomeMilk()` из главы 18 «Знакомство с нашим языком: C#».
- **Функции создают свою область видимости:** как рассказывается в разделе «Области видимости переменных» приложения Б «Полезные идеи», переменные, объявленные внутри функции, имеют область видимости, ограниченную телом этой функции. То есть переменная `outputMessage` (объявленная в строке 14) доступна только в функции `PrintUpdates()`. В таких случаях говорят, что «`outputMessage` видима только в функции `PrintUpdates()`» или «`outputMessage` является *локальной* для функции `PrintUpdates()`».

Сравните область видимости локальной переменной `outputMessage` с областью видимости общедоступной переменной `numTimesCalled`, которая доступна в пределах всего класса `CodeExample` и может использоваться в любых функциях в `CodeExample`.

Если запустить этот код в Unity, переменная `numTimesCalled` будет увеличиваться в каждом кадре, и в каждом кадре будет вызываться `PrintUpdates()` (выводящая значение `numTimesCalled` в консоль). Когда вызывается функция, она выполняется и по завершении возвращает управление в точку вызова. То есть в каждом кадре в классе `CodeExample` выполняются следующие действия:

1. В начале каждого кадра движок Unity вызывает функцию `Update()` (строка 8).
2. Строка 9 увеличивает значение `numTimesCalled` на единицу.
3. Строка 10 вызывает `PrintUpdates()`.
4. Выполнение переходит в начало функции `PrintUpdates()` в строке 13.
5. Выполняются строки 14 и 15.
6. Когда выполнение достигает закрывающей фигурной скобки в конце функции `PrintUpdates()` в строке 16, выполнение возвращается в строку 10 (откуда произошел вызов).
7. Выполнение продолжается со строки 11, которая завершает функцию `Update()`.

Оставшаяся часть главы охватывает разные варианты использования функций, простые и сложные, и знакомит с некоторыми непростыми понятиями. По мере продвижения через учебные примеры далее в этой книге вы получите более полное представление об особенностях работы функций и того, для каких целей вы могли бы использовать свои функции, поэтому не переживайте, если что-то в этой главе вам покажется непонятным. Вы всегда можете вернуться сюда, когда будете читать последующие главы.

ИСПОЛЬЗОВАНИЕ КОДА ИЗ ЭТОЙ ГЛАВЫ В UNITY

В первый листинг этой главы были включены все строки, составляющие класс `CodeExample`, но в последующих примерах будет приводиться не весь код. Если вы захотите опробовать примеры, следующие далее, вы должны заключить код в класс. Больше подробностей вы узнаете в главе 26 «Классы», а пока просто добавляйте строки, выделенные жирным в следующем листинге — вокруг кода примеров:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class CodeExample : MonoBehaviour {
 5
       // Вставляйте код из листингов на место этого комментария
16
17 }
```

Например, вот как мог бы выглядеть первый в данной главе листинг без этих строк:

```
 6     public int numTimesCalled = 0;
 7
 8     void Update() {
 9         PrintUpdates();
10     }
11
12     void PrintUpdates() {
13         numTimesCalled++;
14         print( "Updates: "+numTimesCalled ); // Пример: "Updates: 5"
15     }
```

Чтобы ввести строки 6–15 из этого листинга в сценарий на C#, вам нужно добавить вокруг них строки, выделенные жирным в предыдущем сценарии. В результате в MonoDevelop должен получиться сценарий, как показано в первом листинге в этой главе.

В оставшейся части главы нумерация строк в листингах будет начинаться с 6, чтобы показать, что перед этим кодом и за ним должны находиться другие строки.

Параметры и аргументы функций

Некоторые функции вызываются с пустыми круглыми скобками, следующими за их именами (например, `PrintUpdates()` в первом листинге). Другим функциям в круглых скобках можно передавать некоторую информацию (например, `Say("Hello")` в следующем листинге). Когда функция предполагает получение информации извне, тип этой информации определяется объявлением одного или нескольких *параметров*, которые создают локальные переменные функции (указанного типа). В строке 10 в следующем листинге, `void Say ( string sayThis )`, объявляется параметр с именем `sayThis` типа `string`. `sayThis` можно использовать внутри функции `Say()` как обычную локальную переменную.

Когда информация посылается в функцию через ее параметры, говорят, что информация *передается* в функцию. Любое значение, переданное в параметре, называется *аргументом*. Строка 7 в следующем листинге вызывает функцию `Say()` с аргументом `"Hello"`. Также можно сказать, что значение `"Hello"` передается в функцию `Say()`. Передаваемый аргумент должен соответствовать параметру функции, иначе возникнет ошибка.

```
 6     void Awake() {
 7         Say("Hello");                                          // a
 8     }
 9
10     void Say( string sayThis ) {                               // b
11         print(sayThis);
12     }
```

a. Когда в строке 7 вызывается функция `Say()`, ей передается строковый литерал `"Hello"` как аргумент, после этого строка 10 присвоит значение `"Hello"` переменной `sayThis`.

b. Строковая переменная `sayThis` объявляется как параметр функции `Say()`. Это объявление создает локальную переменную `sayThis` с областью видимости, ограниченной телом функции `Say()`, проще говоря, переменная `sayThis` недоступна за пределами функции `Say()`.

МЕТОДЫ AWAKE(), START() И UPDATE()

Как вы узнали в главе 19 «Hello World: ваша первая программа», метод `Update()` каждого игрового объекта `GameObject` вызывается в каждом кадре, а метод `Start()` — один раз, непосредственно перед первым вызовом `Update()`. Во многих сценариях Unity кроме `Start()` и `Update()` используется также метод `Awake()`. Он вызывается один раз, так же как метод `Start()`, но вызов его происходит непосредственно в момент создания игрового объекта. То есть вызов метода `Awake()` каждого игрового объекта всегда происходит непосредственно перед `Start()`.

Метод Awake() в следующем сценарии, подключенном к testPrefab, вызывается до вывода сообщения "After instantiation", тогда как метод Start() объекта testPrefab будет вызван несколькими миллисекундами позже, после завершения функции Test(), непосредственно перед первым вызовом Update().

```
void Test() {
    print( "Before instantiation" );
    Instantiate<GameObject>( testPrefab );
    print( "After instantiation" );
}
```

В функцию Say() в предыдущем листинге был добавлен единственный параметр с именем sayThis. Так же как в объявлении переменной, первое слово, string, определяет тип переменной, а второе слово, sayThis, — ее имя.

Подобно другим локальным переменным функций, параметры исчезают из памяти, как только функция завершится; если попробовать использовать sayThis где-нибудь в функции Awake(), это вызовет ошибку времени компиляции, потому что sayThis доступна только внутри функции Say().

Строка 7 в предыдущем листинге передает в функцию аргумент — строковый литерал "Hello", однако параметр функции мог бы иметь любой другой тип и ей можно было бы передать в аргументе переменную или литерал соответствующего типа (например, строка 7 в следующем листинге передает аргумент this.gameObject в функцию PrintGameObjectName()). Если функция имеет несколько параметров, передаваемые ей аргументы должны разделяться запятыми (см. строку 8 в следующем листинге).

```
6     void Awake() {
7         PrintGameObjectName( this.gameObject );
8         SetColor( Color.red, this.gameObject );
9     }
10
11     void PrintGameObjectName( GameObject go ) {
12         print( go.name );
13     }
14
15     void SetColor( Color c, GameObject go ) {
16         Renderer r = go.GetComponent<Renderer>();
17         r.material.color = c;
18     }
```

ФУНКЦИИ В C# МОЖНО ОПРЕДЕЛЯТЬ В ЛЮБОМ ПОРЯДКЕ. Возможно, вы заметили, что в предыдущем листинге функция Awake() вызывает обе наши функции, PrintGameObject() и SetColor(), в строках 7 и 8, хотя их объявления находятся ниже, в строках 11–18. Это совершенно нормально для C#. Компилятор C# просматривает весь сценарий в поисках имен функций, прежде чем выполнить хоть одну строчку кода, поэтому совершенно неважно, где в сценарии определяются функции.

Возвращаемые значения

Кроме приема входных значений в параметрах функции могут также возвращать единственное значение — *результат* функции, как показано в строке 13 следующего листинга:

```
6     void Awake() {
7         int num = Add( 2, 5 );
8         print( num ); // Выведет в консоль число 7
9     }
10
11     int Add( int numA, int numB ) {
12         int sum = numA + numB;
13         return( sum );
14     }
```

В этом примере функция `Add()` имеет два параметра, целочисленные `numA` и `numB`. Она складывает два целых числа, переданных ей, и возвращает результат. Слово `int` в начале объявления функции в строке 11 указывает, что `Add()` возвращает целое число в результате. Так же как, объявляя переменную, вы должны указать ее тип, объявляя функцию, вы должны указать тип возвращаемого ею результата.

Возвращаемое значение void

Большинство функций, написанных нами до сих пор, имели возвращаемое значение типа `void`, то есть не возвращали ничего[1]. Даже при том, что эти функции ничего не возвращают, внутри них иногда нужно вызвать инструкцию `return`.

Инструкция `return` используется внутри функции, чтобы прервать ее выполнение и вернуть управление обратно в строку, откуда эта функция была вызвана. (Например, инструкция `return` в строке 16 в следующем листинге вернет управление в строку 9.)

Иногда бывает полезно выйти из функции, пропустив оставшийся код. Например, если имеется список с более чем 100 000 игровых объектов (как список `reallyLongList` в следующем листинге) и требуется переместить игровой объект с именем `"Phil"` в начало координат (`Vector3.zero`), а остальные объекты вас не интересуют, вы могли бы написать такую функцию:

```
6     public List<GameObject> reallyLongList; // Определяется в инспекторе // a
7
8     void Awake() {
9         MoveToOrigin("Phil");                                          // b
10    }
11
12    void MoveToOrigin(string theName) {
13        foreach (GameObject go in reallyLongList) {                    // c
```

[1] Слово «void» переводится как «ничего», «пустое место». — *Примеч. пер.*

```
14              if (go.name == theName) {                                   // d
15                  go.transform.position = Vector3.zero;                   // e
16                  return;                                                 // f
17              }
18          }
19      }
```

a. `List<GameObject> reallyLongList` — очень длинный список игровых объектов, который, как предполагается, предопределен в инспекторе Unity. Так как в действительности список `reallyLongList` не существует, этот код не будет работать, если вы не определите его сами.

b. Вызов функции `MoveToOrigin()` со строковым литералом `"Phil"` в виде аргумента.

c. Инструкция `foreach` выполняет обход элементов списка `reallyLongList`.

d. Если найден первый игровой объект с именем `"Phil"` (то есть с именем, указанным в `theName`)...

e. ...он перемещается в начало координат — в точку [0, 0, 0].

f. Строка 16 возвращает управление в строку 9. Это предотвращает обход остальных игровых объектов в списке.

В функции `MoveToOrigin()` нас действительно не интересуют другие игровые объекты, кроме объекта с именем Phil, поэтому лучше прервать работу функции раньше и вернуть управление, не тратя времени на обход остальных элементов списка. Если объект Phil окажется последним в списке, никакой экономии не получится; но если Phil окажется первым в списке, экономия будет огромная.

Обратите внимание, что когда `return` используется в функции с типом возвращаемого значения `void`, ее можно использовать без круглых скобок (и даже когда возвращаемое значение имеется, круглые скобки можно опустить).

Выбор правильных имен для функций

Как вы наверняка помните, имена переменных должны быть достаточно описательными, начинаться с маленькой буквы и следовать правилам верблюжьегоРегистра (каждое следующее слово начинается с заглавной буквы). Например:

```
int       numEnemies;
float     radiusOfPlanet;
Color     colorAlert;
string    playerName;
```

Имена функций должны соответствовать тем же правилам, но начинаться с заглавной буквы, чтобы их проще было отличать от имен переменных в коде. Вот несколько примеров хороших имен функций:

```
void ColorAGameObject( GameObject go, Color c ) {...}
void AlignX( GameObject go0, GameObject go1, GameObject go2 ) {...}
void AlignXList( List<GameObject> goList ) {...}
void SetX( GameObject go, float eX ) {...}
```

Зачем нужны функции?

Функции — это отличный метод инкапсуляции кода и функциональных возможностей в форме, допускающей многократное использование. Обычно, когда возникает необходимость написать одни и те же строки кода более пары раз, это верный признак, что данный код предпочтительнее оформить в виде функции. Для начала посмотрим на листинг, в котором есть повторяющиеся строки кода.

Функция `AlignX()` в следующем листинге принимает три параметра с игровыми объектами, вычисляет среднее значение их координаты X и назначает им всем это среднее значение координаты X:

```
 6     void AlignX( GameObject go0, GameObject go1, GameObject go2 ) {
 7         float avgX = go0.transform.position.x;
 8         avgX += go1.transform.position.x;
 9         avgX += go2.transform.position.x;
10         avgX = avgX/3.0f;
11
12         Vector3 tempPos;
13         tempPos = go0.transform.position;                          // a
14         tempPos.x = avgX;                                          // a
15         go0.transform.position = tempPos;                          // a
16
17         tempPos = go1.transform.position;
18         tempPos.x = avgX;
19         go1.transform.position = tempPos;
20
21         tempPos = go2.transform.position;
22         tempPos.x = avgX;
23         go2.transform.position = tempPos;
24     }
```

a. В строках 13–15 можно видеть, как преодолевается ограничение Unity, не позволяющее напрямую изменять значение `position.x` компонента `transform`. Для этого нужно сначала скопировать текущие координаты в другую переменную (например, `Vector3 tempPos`), затем изменить значение `x` и, наконец, скопировать весь вектор `Vector3` обратно в `transform.position`. Очень утомительно раз за разом писать одни и те же строки; в результате родилась функция `SetX()`, показанная в следующем листинге. Она позволяет установить координату `x` компонента `transform` за один шаг (например, `SetX( this.gameObject, 25.0f )`).

Из-за ограничений на непосредственное изменение значений `x`, `y` и `z` в `transform.position` функция `AlignX()` содержит много повторяющегося кода в строках с 13-й по 23-ю. Ввод такого кода может сильно утомлять, и если позднее что-то понадо-

бится изменить в нем, изменения придется внести в трех местах в функции `AlignX()`. Это одна из главных причин использования функций. В следующем листинге строки 11–23 из предыдущего листинга заменены вызовами новой функции `SetX()`. Жирным выделены строки, изменившиеся в сравнении с предыдущим листингом.

```
 6     void AlignX( GameObject go0, GameObject go1, GameObject go2 ) {
 7         float avgX = go0.transform.position.x;
 8         avgX += go1.transform.position.x;
 9         avgX += go2.transform.position.x;
10         avgX = avgX/3.0f;
11
12         SetX ( go0, avgX );
13         SetX ( go1, avgX );
14         SetX ( go2, avgX );
15     }
16
17     void SetX( GameObject go, float eX ) {
18         Vector3 tempPos = go.transform.position;
19         tempPos.x = eX;
20         go.transform.position = tempPos;
21     }
```

В этом усовершенствованном листинге удаленные строки из предыдущего заменили на определение новой функции `SetX()` (строки 17–21) и три ее вызова (строки 12–14). Теперь, если в процедуру изменения значения `x` потребуется что-то добавить, достаточно будет изменить только одну функцию `SetX()`, а не три фрагмента, как в предыдущем листинге. Конечно, это был очень простой пример, но я надеюсь, что он достаточно наглядно показал, какие возможности открывают функции перед нами как программистами.

В оставшейся части главы рассматриваются некоторые более сложные и интересные подходы к определению функций в C#.

Перегрузка функций

Перегрузка функций — причудливый термин, описывающий возможность функций в C# действовать по-разному в зависимости от типов и количества передаваемых им параметров. В следующем листинге жирным выделены фрагменты, демонстрирующие перегрузку функций.

```
 6     void Awake() {
 7         print( Add( 1.0f, 2.5f ) );
 8         // ^ Выведет: "3.5"
 9         print( Add( new Vector3(1, 0, 0), new Vector3(0, 1, 0) ) );
10         // ^ Выведет "(1.0, 1.0, 0.0)"
11         Color colorA = new Color( 0.5f, 1, 0, 1);
12         Color colorB = new Color( 0.25f, 0.33f, 0, 1);
13         print( Add( colorA, colorB ) );
14         // ^ Выведет "RGBA(0.750, 1.000, 0.000, 1.000)"
15     }
```

```
16
17    float Add( float f0, float f1 ) {                                    // a
18        return( f0 + f1 );
19    }
20
21    Vector3 Add( Vector3 v0, Vector3 v1 ) {                              // a
22        return( v0 + v1 );
23    }
24
25    Color Add( Color c0, Color c1 ) {                                    // a
26        float r, g, b, a;
27        r = Mathf.Min( c0.r + c1.r, 1.0f );                              // b
28        g = Mathf.Min( c0.g + c1.g, 1.0f );                              // b
29        b = Mathf.Min( c0.b + c1.b, 1.0f );                              // b
30        a = Mathf.Min( c0.a + c1.a, 1.0f );                              // b
31        return( new Color( r, g, b, a ) );
32    }
```

a. В этом листинге объявляются и определяются три разные функции `Add()`, вызываемая версия в каждом случае определяется параметрами, переданными в вызов. Когда функции передаются два числа с плавающей точкой, вызывается версия `Add()`, складывающая вещественные числа; когда передаются два вектора `Vector3`, вызывается версия, складывающая векторы; а когда передаются два значения типа `Color`, вызывается версия, складывающая два цвета.

b. Версия `Add()`, складывающая два значения `Color`, заботится о том, чтобы получающиеся значения `r`, `g`, `b` и `a` не превысили 1, потому что значения красного (red), зеленого (green), синего (blue) и альфа (alpha) каналов цвета ограничены величинами между 0 и 1. Для этого используется функция `Mathf.Min()`. `Mathf.Min()` принимает любое количество аргументов и возвращает один из них, имеющий минимальное значение. В предыдущем листинге сумма красных каналов двух цветов получится равной 0,75f, поэтому в возвращаемом результате красный канал будет иметь значение 0,75f; но сумма зеленых каналов получится больше 1,0f, поэтому в возвращаемом результате зеленый канал будет иметь значение 1,0f.

Необязательные параметры

Иногда желательно, чтобы функция имела *необязательные параметры*, которые могут передаваться или не передаваться. В следующем листинге параметр `eX` функции `SetX()` является необязательным. Если задать значение по умолчанию для параметра в определении функции, компилятор будет интерпретировать его как необязательный (как, например, в строке 13 в следующем листинге, где параметр `eX` получает значение по умолчанию 0,0f). Код, выделенный **жирным**, демонстрирует использование необязательного параметра.

```
 6    void Awake() {
 7        SetX( this.gameObject, 25 );                                    // b
 8        print( this.gameObject.transform.position.x ); // Выведет: "25"
```

```
 9          SetX( this.gameObject );                                    // c
10          print( this.gameObject.transform.position.x ); // Выведет: "0"
11      }
12
13      void SetX( GameObject go, float eX=0.0f ) {                     // a
14          Vector3 tempPos = go.transform.position;
15          tempPos.x = eX;
16          go.transform.position = tempPos;
17      }
```

a. Параметр `eX` определен как необязательный со значением по умолчанию 0,0f. Определение значения по умолчанию для параметра `eX` в объявлении функции (часть `=0.0f`) делает параметр `eX` необязательным. Если вызвать функцию без аргумента для параметра `eX`, он получит значение 0,0f.

b. Так как тип `float` может представить любое целочисленное значение[1], передача целого числа в параметре типа `float` вполне допустима. (Например, в строке 7 целочисленный литерал `25` передается как аргумент для параметра `float eX`, объявленного в строке 13.)

c. В строке 9 метод `SetX()` вызывается без аргумента для параметра `eX` (но с аргументом для обязательного параметра `go`). Когда функция вызывается без аргумента для необязательного параметра, используется значение по умолчанию. В данном случае значение по умолчанию для `eX` определено в строке 13 как 0,0f.

Когда `SetX()` первый раз вызывается из `Awake()`, параметр `eX` получает значение 25,0f. Но во втором вызове аргумент для параметра `eX` опущен, поэтому он получает значение по умолчанию 0,0f.

Необязательные параметры всегда должны следовать в объявлении функции после всех обязательных параметров.

Ключевое слово params

Как показано в следующем листинге, в строке 13, с помощью ключевого слова `params` можно позволить функции принимать произвольное количество параметров одного типа. Эти параметры преобразуются в массив указанного типа. Фрагменты, выделенные жирным, демонстрируют применение ключевого слова `params`.

```
 6      void Awake() {
 7          print( Add( 1 ) );          // Выведет: "1"
 8          print( Add( 1, 2 ) );       // Выведет: "3"
```

[1] Точнее говоря, переменная типа `float` может хранить *большинство* значений типа `int`. Как рассказывалось в главе 19 «Переменные и компоненты», тип `float` не способен точно представлять очень большие и очень маленькие числа, поэтому очень большие значения `int` могут округляться до ближайшего числа, которое может быть представлено типом `float`. Проведя эксперимент в Unity, я выяснил, что тип `float` точно представляет целые числа вплоть до 16 777 217, а числа выше представляет с погрешностью.

```
 9          print( Add( 1, 2, 3 ) );    // Выведет: "6"
10          print( Add( 1, 2, 3, 4 ) ); // Выведет: "10"
11      }
12
13      int Add( params int[] ints ) {
14          int sum = 0;
15          foreach (int i in ints) {
16              sum += i;
17          }
18          return( sum );
19      }
```

Теперь функция `Add()` может принимать любое количество целых чисел и возвращать их сумму. Подобно необязательным параметрам, список `params` должен следовать в объявлении функции после всех обязательных параметров (то есть все обязательные параметры должны быть объявлены перед списком `params`).

Это ключевое слово позволяет также переписать предыдущую функцию `AlignX()`, чтобы она принимала любое количество игровых объектов, как показано в следующем листинге.

```
 6      void AlignX( params GameObject[] goArray ) {                    // a
 7          float sumX = 0;
 8          foreach (GameObject go in goArray) {                        // b
 9              sumX += go.transform.position.x;                        // c
10          }
11          float avgX = sumX / goArray.Length;                         // d
12
13          foreach (GameObject go in goArray) {                        // e
14              SetX ( go, avgX );
15          }
16      }
17
18      void SetX( GameObject go, float eX ) {
19          Vector3 tempPos = go.transform.position;
20          tempPos.x = eX;
21          go.transform.position = tempPos;
22      }
```

a. Ключевое слово `params` создает массив и включает в него все игровые объекты, переданные в вызов функции.

b. Цикл `foreach` выполняет обход всех игровых объектов в `goArray`. Переменная `GameObject go` доступна только в теле цикла `foreach`, в строках 8–10, поэтому она не конфликтует с переменной `GameObject go` в цикле `foreach` в строках 13–15.

c. Координата X текущего игрового объекта добавляется к сумме `sumX`.

d. Определяется среднее значение координаты X делением суммы на количество игровых объектов. Обратите внимание, что если в вызов функции не передать ни одного игрового объекта, эта строка вызовет ошибку деления на ноль.

e. Второй цикл `foreach` выполняет обход игровых объектов в `goArray` и вызывает `SetX()`, передавая каждый объект как параметр.

Рекурсивные функции

Иногда функции должны вновь и вновь вызывать самих себя, — такие функции называют *рекурсивными*. Простым примером может служить вычисление факториала числа.

В математике 5! (5 факториал) — это произведение этого числа и всех других натуральных чисел ниже него. (Натуральными называют целые числа больше нуля.)

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

0! — особый случай в математике и равен 1.

```
0! = 1
```

В нашем примере будем возвращать 0 при попытке передать отрицательное число в функцию вычисления факториала:

```
-5! = 0
```

Для вычисления факториала любого целого числа можно написать рекурсивную функцию `Fac()`:

```
 6     void Awake() {
 7         print( Fac(5) );  // Выведет: "120"                          // a
 8         print( Fac(0) );  // Выведет: "1"
 9         print( Fac(-5) ); // Выведет: "0"
10     }
11
12     int Fac( int n ) {                                               // b, d
13         if (n < 0) {     // Обрабатывает случай if n<0
14             return( 0 );
15         }
16         if (n == 0) {    // Это "завершающий случай"                 // e
17             return( 1 );
18         }
19         int result = n * Fac( n-1 );                                 // c, f
20         return( result );                                            // g
21     }
```

a. Когда `Fac()` вызывается с целочисленным параметром 5.

b. Начинается первая итерация из вызовов `Fac()` с n = 5.

c. В строке 19 параметр n (со значением 5) умножается на результат вызова `Fac()` с параметром 4. Этот способ работы функции называется *рекурсией*.

d. Этот вызов начинает вторую итерацию из вызовов `Fac()` с n = 4. Процесс продолжается до шестой итерации, в которой n = 0.

e. Так как n имеет значение 0, в пятую итерацию возвращается значение 1...

f. ...которое затем умножается на 1 (1 * 1)

g. ...и результат — число 1 — возвращается в четвертую итерацию, и так далее, пока не завершатся все рекурсивные вызовы `Fac()` и первая итерация вернет значение 120 в строке 7.

Вот как выглядит цепь всех рекурсивных вызовов `Fac()`:

```
Fac(5)                          // 1 итерация
5 * Fac(4)                        // 2 итерация
5 * 4 * Fac(3)                      // 3 итерация
5 * 4 * 3 * Fac(2)                    // 4 итерация
5 * 4 * 3 * 2 * Fac(1)                  // 5 итерация
5 * 4 * 3 * 2 * 1 * Fac(0)                // 6 итерация
5 * 4 * 3 * 2 * 1 * 1                   // 5 итерация
5 * 4 * 3 * 2 * 1                     // 4 итерация
5 * 4 * 3 * 2                       // 3 итерация
5 * 4 * 6                         // 2 итерация
5 * 24                          // 1 итерация
120                           // Окончательный результат
```

Лучший способ понять, что происходит в рекурсивной функции, — исследовать ее с помощью отладчика — возможности, имеющейся в MonoDevelop, позволяющей просматривать каждый шаг выполнения программы и наблюдать, как разные переменные изменяются в каждом шаге. Процесс отладки станет темой следующей главы.

Итоги

В этой главе вы познакомились с функциями и некоторыми разными способами их использования. Функции составляют основу большинства современных языков программирования, и чем больше вы будете программировать, тем очевиднее для вас будет важность и нужность функций.

Следующая глава, «Отладка», покажет вам, как пользоваться инструментами отладки в Unity. Эти инструменты призваны помогать с поиском проблем в коде, но их также с успехом можно использовать для изучения работы кода. После знакомства с отладкой я советую вернуться сюда и более внимательно исследовать работу функции `Fac()`. Разумеется, вы также можете заняться исследованием любых функций из этой главы с помощью отладчика.

25 Отладка

Для непосвященных отладка может показаться черной магией. Но в действительности это один из важнейших навыков, которым вы должны обладать как разработчик — отладку редко преподают начинающим программистам, и я считаю это трагическим упущением. Все начинающие программисты допускают ошибки, и знание приемов отладки поможет им находить и исправлять их намного быстрее, чем простое изучение кода в надежде, что ошибка покажет себя.

К концу этой главы вы будете понимать разницу между ошибками времени компиляции и времени выполнения, узнаете, как устанавливать точки останова в коде и как выполнять программу по одной строке в поисках трудноуловимых ошибок.

Знакомство с отладкой

Прежде чем начать поиск ошибок, их нужно сначала допустить. В этой главе мы начнем с проекта, созданного в главе 19 «Hello World: ваша первая программа». Если у вас нет этого проекта под рукой, вы всегда можете загрузить его с веб-сайта книги http://book.prototools.net/.

На веб-сайте найдите раздел Chapter 25: "Debugging" и щелкните на нем, чтобы загрузить начальный проект для этой главы.

На протяжении этой главы я неоднократно буду предлагать вам преднамеренно вносить ошибки. Это может показаться странным, но в этой главе моя цель — дать вам некоторый опыт поиска и исправления ошибок, с которыми вы наверняка столкнетесь, работая в Unity. Каждый пример знакомит с одной из потенциальных проблем и показывает, как искать и исправлять их.

→ На протяжении всей главы я буду ссылаться на ошибки по номерам строк. Иногда эти номера будут точно соответствовать строкам с ошибками, иногда номера будут смещены вниз на одну-две строки. Не волнуйтесь, если у вас номера строк не совпадают с моими, просто найдите строки по их содержимому рядом с номерами, на которые я ссылаюсь.

Как упоминалось выше в этой главе, вы должны внести изменения в сценарий CubeSpawner из главы 19. На случай, если вы внесли какие-то свои изменения, вот как должен выглядеть сценарий CubeSpawner для этой главы с номерами строк:

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class CubeSpawner : MonoBehaviour {
 6     public GameObject       cubePrefabVar;
 7
 8     // Используйте этот метод для инициализации
 9     void Start () {
10         // Instantiate( cubePrefabVar );
11     }
12
13     void Update () {
14         Instantiate( cubePrefabVar );
15     }
16 }
```

СОВЕТ ПРОФЕССИОНАЛА

90% ОШИБОК – ПРОСТЫЕ ОПЕЧАТКИ. Я потратил так много времени, помогая студентам исправлять ошибки, что с ходу замечаю опечатки в коде[1]. Вот наиболее распространенные из них:

Орфографические ошибки: даже если вы ошибетесь в единственной букве, компьютер не поймет, что вы хотели сказать ему.

Ошибки в регистре символов: для компилятора C# буквы `A` и `a` — это два разных символа, то есть с его точки зрения слова `variable`, `Variable` и `variAble` — это три совершенно разных слова.

Отсутствие точки с запятой: так же, как в русском языке, почти все предложения должны заканчиваться точкой, почти все инструкции в C# должны заканчиваться точкой с запятой (`;`). Отсутствие точки с запятой часто заставляет компилятор сообщать об ошибке в следующей строке. К вашему сведению: эта роль возложена на точку с запятой по той простой причине, что точка уже используется для отделения целой и дробной частей в вещественных числах, а также в *точечном синтаксисе* ссылок на переменные (например, `varName.x`).

Ошибки времени компиляции

Ошибка времени компиляции — это проблема, которую Unity обнаруживает во время компиляции кода на C# (то есть при попытке интерпретировать код на C# и превратить его в код на обобщенном промежуточном языке (Common Intermediate Language), который позднее будет преобразован средой Unity в машинный код,

[1] Внимательные читатели могут заметить, что этот совет уже давался в главе 17 «Введение в среду разработки Unity». Да, он настолько важен, что не грех повторить его дважды!

понятный компьютеру). Открыв проект Hello World в Unity, выполните следующие действия, чтобы внести и исследовать ошибку времени компиляции:

1. Скопируйте сцену _Scene_1. Для этого щелкните на сцене _Scene_1 в панели Project (Проект), чтобы выбрать ее, и выберите пункт меню Edit > Duplicate (Правка > Дублировать). Unity прекрасно справляется со счетом, поэтому автоматически нарастит номер сцены и даст новой сцене имя _Scene_2.
2. Щелкните дважды на имени _Scene_2, чтобы открыть сцену в панелях Hierarchy (Иерархия) и Scene (Сцена). После того как сцена откроется, в заголовке окна Unity должен появиться текст _Scene_2.unity - Hello World - PC, Mac, & Linux Standalone. Если теперь щелкнуть на кнопке Play (Играть), вы должны увидеть все то же самое, что происходит при запуске сцены _Scene_1.
3. Теперь нужно создать вторую версию класса CubeSpawner, чтобы не повредить класс в сцене _Scene_1. Щелкните на сценарии CubeSpawner в панели Project (Проект), чтобы выделить его, и выберите пункт меню Edit > Duplicate (Правка > Дублировать). В результате будет создан сценарий CubeSpawner1, а в панели Console (Консоль) немедленно появится ошибка (рис. 25.1). Щелкните на ошибке, чтобы увидеть больше информации в нижней половине консоли.

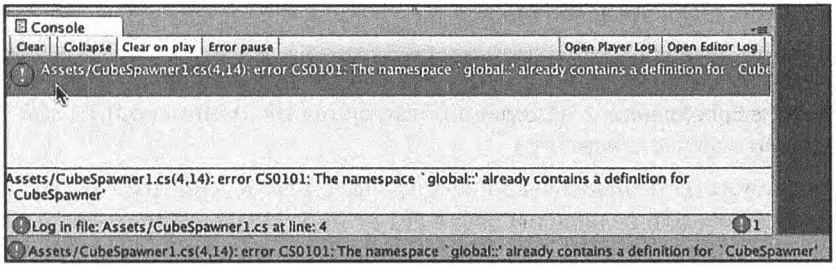


Рис. 25.1. Ваша первая преднамеренно созданная ошибка: ошибка времени компиляции, найденная средой Unity

Это сообщение содержит массу полезной информации, поэтому исследуем его фрагмент за фрагментом.

```
Assets/CubeSpawner1.cs (4,14):
```

Всякое сообщение включает информацию о месте, где среда Unity встретила ошибку. В данном случае Unity сообщает, что ошибка обнаружена в сценарии *CubeSpawner1.cs*, находящемся в папке *Assets* проекта, и что она находится в строке 4, в позиции 14.

```
error CS0101:
```

Второй фрагмент сообщения уточняет, с какого вида ошибкой вы столкнулись. Если вы столкнетесь с ошибкой, которую не смогли понять, поищите в интернете

по словам «Unity error» с кодом ошибки. (В данном случае строка поиска должна иметь вид «Unity error CS0101».) Подобный поиск почти всегда приводит к сообщению на форуме или в другое место, где дается описание вашей проблемы. Как показывает мой опыт, хорошие ответы обычно можно найти на http://forum.unity3d.com и http://answers.unity3d.com, а некоторые из лучших ответов — на http://stackoverflow.com.

```
The namespace 'global::' already contains a definition for 'CubeSpawner'[1]
```

Заключительный фрагмент сообщения об ошибке дает пояснения на простом английском языке. В данном случае сообщается, что имя `CubeSpawner` уже определено где-то в вашем коде. На данный момент оба сценария — CubeSpawner и CubeSpawner1 — определяют класс `CubeSpawner`.

Давайте исправим ошибку:

1. Щелкните дважды на сценарии CubeSpawner1, чтобы открыть его в MonoDevelop. (Также можно щелкнуть на сообщении об ошибке в панели Console (Консоль), после чего Mac откроет сценарий на строке, породившей ошибку; версия для Windows не всегда делает это.)
2. Измените строку 4 в сценарии CubeSpawner1 (строку, объявляющую класс `CubeSpawner`), как показано ниже:

```
5 public class CubeSpawner2 : MonoBehaviour {
```

(Имя класса `CubeSpawner2` намеренно отличается от имени скрипта, так что вы моментально замечаете ошибку.)

3. Сохраните файл и вернитесь в Unity, где вы сможете заметить, что сообщение об ошибке исчезло из панели Console (Консоль).

Всякий раз, когда вы сохраняете сценарий в MonoDevelop, среда Unity обнаруживает это и перекомпилирует сценарий, чтобы убедиться в отсутствии ошибок. Если в сценарии встретится ошибка, вы получите сообщение об ошибке времени компиляции, подобной той, что мы только что исправили. Это наиболее простые в исправлении ошибки, потому что Unity точно определяет местоположение проблемы и сообщает эту информацию вам. Теперь, когда сценарий CubeSpawner определяет класс `CubeSpawner`, а сценарий CubeSpawner1 — класс `CubeSpawner2`, ошибка времени компиляции исчезла.

Ошибки времени компиляции из-за отсутствия точки с запятой

Теперь создадим ошибку времени компиляции другого рода, убрав точку с запятой:

1. Удалите точку с запятой (`;`) в конце строки 14, которая имеет вид:

```
15          Instantiate( cubePrefabVar );
```

[1] Пространство имен 'global::' уже содержит определение 'CubeSpawner'. — *Примеч. пер.*

2. Сохраните сценарий и вернитесь в Unity. В консоли появятся два новых сообщения об ошибке:

```
Assets/CubeSpawner1.cs(15,9): error CS1525: Unexpected symbol '}'[1]
Assets/CubeSpawner1.cs(17,1): error CS8025: Parsing error[2]
```

Главное правило, которому вы должны следовать в подобных случаях: **ошибки всегда должны исправляться, начиная с самой верхней**, поэтому начнем с первой ошибки: «`Assets/CubeSpawner1.cs(15,9): error CS1525: Unexpected symbol '}'`».

Текст ошибки не говорит нам: «Эй, вы забыли поставить точку с запятой», — зато сообщает место, где компилятор встретил ошибку (строка 15, позиция 9). Текст также сообщает, что закрывающая фигурная скобка (}) встречена в неожиданном месте. Опираясь на эту информацию, вы должны осмотреть прилегающую область кода и обнаружить отсутствие точки с запятой.

3. Добавьте точку с запятой в конец строки 14 и сохраните сценарий. Вернувшись в Unity, вы заметите, что исчезли **обе** ошибки.

Ошибки времени компиляции почти всегда обнаруживаются в строке, где фактически находится проблема, или в строке ниже. В данном примере точка с запятой отсутствует в строке 14, а проблема обнаруживается в строке 15. Кроме того, многие ошибки времени компиляции приводят к обнаружению целого каскада ошибок в коде ниже. Если вы всегда будете исправлять ошибки сверху вниз, исправление одной ошибки часто будет устранять множество других ошибок, следующих за ней.

Подключение и удаление сценариев

В окне Unity попробуйте перетащить мышью сценарий CubeSpawner1 на Main Camera в иерархии. На этот раз появится ошибка, как показано на рис. 25.2.

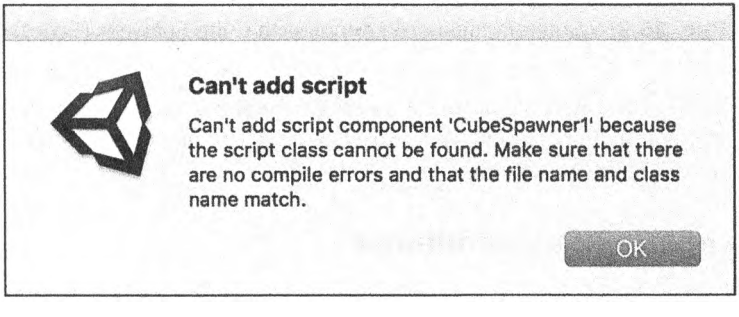


Рис. 25.2. Некоторые ошибки выявляются только при попытке подключить сценарий к игровому объекту

[1] Неожиданный символ '}'. — *Примеч. пер.*
[2] Синтаксическая ошибка. — *Примеч. пер.*

В данном случае Unity сообщает, что имя сценария CubeSpawner1 не соответствует имени класса, определяемого в сценарии: CubeSpawner2. Когда в Unity создается класс, наследующий MonoBehaviour (например, CubeSpawner2 : Monobehaviour), имя такого класса должно совпадать с именем файла, в котором он определен. Чтобы исправить эту ошибку, достаточно просто привести два имени в соответствие.

1. Щелкните на сценарии CubeSpawner1 в панели Project (Проект), чтобы выбрать его, и затем щелкните на нем второй раз, чтобы включить режим переименования. (Можете также нажать клавишу Return в macOS или F2 в Windows.)
2. Измените имя сценария на CubeSpawner2 и снова попробуйте перетащить на Main Camera. На этот раз никаких проблем не должно возникнуть.
3. Щелкните на главной камере Main Camera в иерархии, после этого в инспекторе должны появиться оба подключенных сценария — CubeSpawner и CubeSpawner2.
4. Оба сценария нам не нужны, поэтому щелкните на ярлыке с изображением шестеренки справа от имени Cube Spawner (Script) в инспекторе и выберите в открывшемся меню пункт Remove Component (Удалить компонент), как показано на рис. 25.3. (Также можно щелкнуть правой кнопкой мыши на имени Cube Spawner (Script), чтобы получить то же самое меню.)

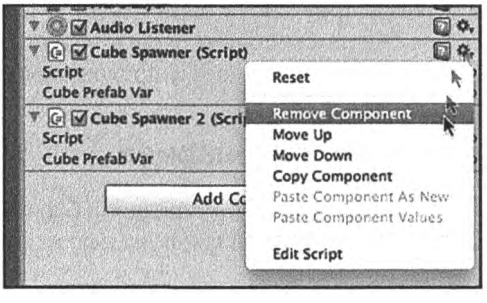


Рис. 25.3. Удаление лишнего компонента Cube Spawner (Script)

Теперь у вас не будет двух разных сценариев, пытающихся одновременно создавать кубики. В следующих нескольких главах мы будем подключать к каждому игровому объекту только по одному сценарию.

Ошибки времени выполнения

Выполните следующие шаги, чтобы исследовать другой вид ошибок:

1. Щелкните на кнопке Play (Играть), чтобы встретиться с еще одним видом ошибок (рис. 25.4).
2. Щелкните на кнопке Pause (Пауза), чтобы приостановить выполнение сценария (кнопка с двумя вертикальными чертами, находится рядом с кнопкой Play (Играть)) и рассмотреть ошибку.

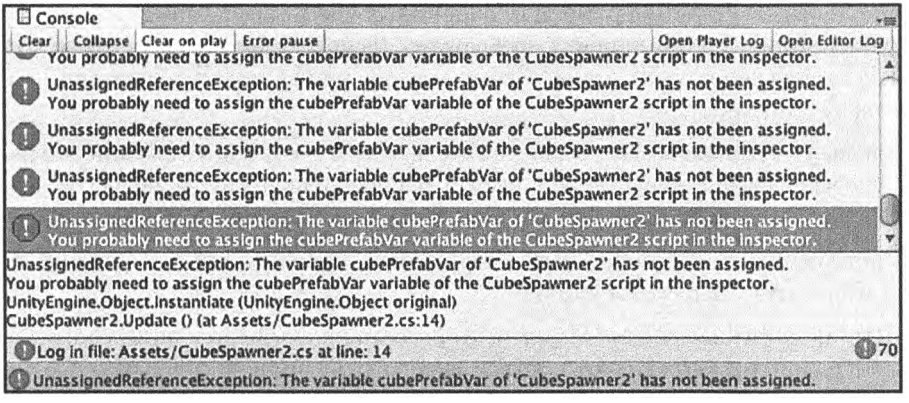


Рис. 25.4. Множество повторов одной и той же ошибки времени выполнения

Это *ошибка времени выполнения*, то есть ошибка, возникающая, только когда Unity фактически пытается запустить проект. Ошибки времени выполнения возникают, когда вы набрали код правильно (с точки зрения компилятора), но что-то идет не так, когда этот код выполняется.

Это сообщение об ошибке выглядит немного иначе, чем предыдущие, что мы видели. Прежде всего, сообщение об ошибке не содержит информации о месте ее появления; но если щелкнуть на одном из сообщений, в нижней половине консоли появится дополнительная информация. Для ошибки времени выполнения последняя строка сообщения подсказывает, где она была замечена. Иногда это место соответствует строке с ошибкой, иногда следующей строке. В данном случае сообщение указывает, что ошибку нужно искать где-то рядом со строкой 14 в файле *CubeSpawner2.cs*.

```
CubeSpawner2.Update () (at Assets/CubeSpawner2.cs:14)
```

Строка 14 в сценарии CubeSpawner2 создает экземпляр `cubePrefabVar`. (Обратите внимание, что у вас номер строки может немного отличаться; в этом нет ничего страшного.)

```
14          Instantiate( cubePrefabVar );
```

С точки зрения компилятора в этой строке нет ошибок. Давайте продолжим исследование сообщения:

UnassignedReferenceException: The variable cubePrefabVar of 'CubeSpawner2' has not been assigned. You probably need to assign the cubePrefabVar variable of the CubeSpawner2 script in the inspector. UnityEngine.Object.Instantiate (UnityEngine. Object original) CubeSpawner2.Update () (at Assets/CubeSpawner2.cs:14)[1]

[1] UnassignedReferenceException: Переменная cubePrefabVar в 'CubeSpawner2' не инициализирована. Возможно, вы должны присвоить переменной cubePrefabVar в сценарии CubeSpawner2 некоторое значение в инспекторе. UnityEngine.Object.Instantiate (UnityEngine. Object оригинал) CubeSpawner2.Update () (в Assets/CubeSpawner2.cs:14). — *Примеч. пер.*

Оно подсказывает, что переменная `cubePrefabVar` не была инициализирована, и если взглянуть на компонент CubeSpawner2 (Script) главной камеры Main Camera в инспекторе (щелкнув на главной камере в иерархии), можно убедиться, что это действительно так.

3. Как вы уже делали в главе 19, щелкните на изображении круглой мишени рядом с `cubePrefabVar` в инспекторе и выберите Cube Prefab в списке ресурсов. Теперь вы должны увидеть в инспекторе, что переменная `cubePrefabVar` имеет значение.
4. Снова щелкните на кнопке Pause (Пауза), чтобы продолжить выполнение; после этого начнут появляться кубики.
5. Щелкните на кнопке Play (Играть), чтобы остановить выполнение. Снова щелкните на кнопке Play (Играть), чтобы запустить выполнение еще раз.

Что случилось?! Снова появилась та же ошибка!

6. Щелкните на кнопке Play (Играть), чтобы опять остановить выполнение.

ИЗМЕНЕНИЯ, ВНЕСЕННЫЕ ВО ВРЕМЯ ПРОИГРЫВАНИЯ, НЕ СОХРАНЯЮТСЯ! Вы еще не раз столкнетесь с этой проблемой. Есть веские причины, почему Unity работает именно так, но такое поведение часто приводит начинающих пользователей в замешательство. Любые изменения, внесенные во время проигрывания или паузы (как, например, присваивание переменной `cubePrefabVar`, которое мы произвели), отменяются после остановки проигрывания. Если нужно, чтобы изменения сохранились, остановите Unity перед добавлением изменений.

7. Теперь, когда среда Unity остановлена, вновь инициализируйте переменную `cubePrefabVar` шаблоном Cube Prefab в инспекторе. На этот раз, так как Unity стоит, изменения сохранятся.
8. Щелкните на кнопке Play (Играть), и все должно заработать.

Пошаговое выполнение кода в отладчике

Кроме инструментов автоматической проверки кода, которые мы уже рассмотрели выше в этой главе, в Unity и MonoDevelop имеется также отладчик, позволяющий выполнять код по одной строке, что может очень пригодиться для исследования происходящего в коде.

Откройте сценарий CubeSpawner2 в MonoDevelop и добавьте в него строки из следующего листинга, выделенные жирным (то есть добавьте строки 14 и 18–27). Если вам понадобится раздвинуть строки в сценарии, просто нажмите Return (Enter в Windows). Код также показан на рис. 25.5.

```
1 using UnityEngine;                                                    // a
2 using System.Collections;
3
4 public class CubeSpawner2 : MonoBehaviour {
5     public GameObject cubePrefabVar;
```

Рис. 25.5. Функция `SpellItOut()` с точкой останова в строке 14

```
 6
 7     // Используйте этот метод для инициализации
 8     void Start () {
 9         // Instantiate( cubePrefabVar );
10     }
11
12     // Update вызывается в каждом кадре
13     void Update () {
14         SpellItOut();                                          // b
15         Instantiate( cubePrefabVar );
16     }
17
18     public void SpellItOut () {                                // c
19         string sA = "Hello World!";
20         string sB = "";
21
22         for (int i=0; i<sA.Length; i++) {                      // d
23             sB += sA[i];                                       // e
24         }
25
26         print(sB);
27     }
28 }
```

a. Обратите внимание, что этот листинг не подключает библиотеку `System.Collections.Generic`, она здесь не нужна, хотя в версиях Unity 5.5+ автоматически подключается во всех сценариях на C#.

b. В строке 14 вызывается функция `SpellItOut()`.

c. Строки 18–27 объявляют и определяют функцию `SpellItOut()`. Эта функция копирует содержимое строки `sA` в строку `sB` по одному символу.

d. Этот цикл `for` выполняет итерации по символам в строке `sA`. Поскольку строка `"Hello World"` содержит 11 символов, цикл выполнит 11 итераций.

e. Строка 23 извлекает *i*-й символ из `sA` и добавляет его в конец `sB`. Это очень неэффективный способ копирования строк, но он хорошо подходит для демонстрации работы отладчика.

Введя весь код и дважды проверив его, щелкните на поле слева от строки 14 (как показано на рис. 25.5). В результате в строке 14 будет создана точка останова, которая отображается как красный кружок. Когда в сценарии имеется точка останова и MonoDevelop находится в режиме отладки, Unity будет останавливать выполнение в этой точке. Давайте проверим.

КАК ПРИНУДИТЕЛЬНО ЗАВЕРШИТЬ ПРИЛОЖЕНИЕ

Прежде чем продолжить углубляться в изучение приемов отладки, вы должны знать, как принудительно завершить приложение (если оно не откликается ни на какой ввод пользователя). Иногда Unity или MonoDevelop перестают реагировать на действия пользователя, и вам может потребоваться принудительно завершить их.

В macOS

Чтобы принудительно завершить приложение, выполните следующие действия:

1. Нажмите комбинацию Command-Option-Esc на клавиатуре. Откроется окно Force Quit (Принудительное завершение).
2. Найдите зависшее приложение, которое нужно завершить. Часто его имя в списке сопровождается текстом (Not responding) (Не отвечает).
3. Щелкните на имени приложения в списке и затем на кнопке Force Quit (Завершить).

В Windows

Чтобы принудительно завершить приложение, выполните следующие действия:

1. Нажмите комбинацию Shift+Ctrl+Esc на клавиатуре. Откроется окно Windows Task Manager (Диспетчер задач Windows).
2. Найдите зависшее приложение.
3. Щелкните на имени приложения и затем на кнопке End Task (Снять задачу).

Принудительно завершив работающую среду Unity, вы потеряете все изменения, что добавили с момента последнего сохранения. Поскольку вы и без того вынуждены постоянно сохранять сценарии на C#, к ним это предупреждение относится меньше всего, но вам, возможно, придется восстанавливать несохраненные изменения в сцене. Это одна из причин, почему я советую сохранять сцены как можно чаще.

Подключение отладчика к Unity

Чтобы MonoDevelop имел возможность следить за выполнением кода в Unity, его нужно подключить к процессу Unity. После подключения отладчика MonoDevelop к Unity он сможет наблюдать за происходящим в вашем коде на C# и приостанавливать выполнение в точках останова (таких, как в строке 14).

1. В MonoDevelop щелкните на кнопке Play (Играть). Она выглядит по-разному в macOS и Windows, как показано в верхней (macOS) и нижней (Windows) частях на рис. 25.6. В обоих случаях кнопка Play (Играть) находится под указателем мыши.

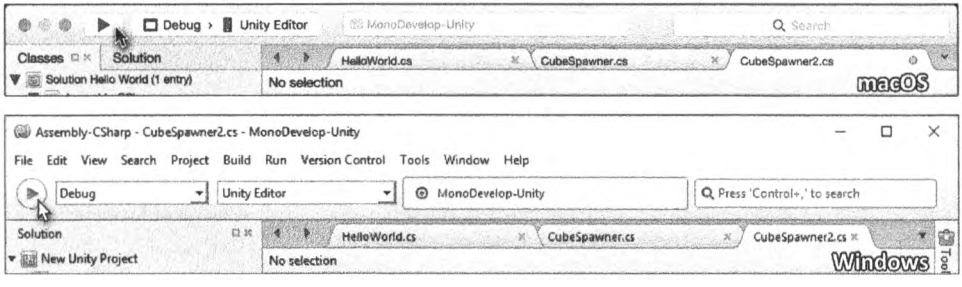


Рис. 25.6. Щелкните на кнопке, чтобы подключить отладчик к процессу Unity Editor

После этого отладчик MonoDevelop автоматически найдет процесс Unity и подключится к нему. В первый раз вам может быть предложено разрешить MonoDevelop выполнить это действие. Дайте такое разрешение.

Когда процесс подключения завершится, вы заметите, что окно MonoDevelop изменится (рис. 25.7). Кнопка Play (Играть) слева вверху превратится в кнопку Stop (Остановить), в нижней части окна MonoDevelop появится пара панелей и несколько кнопок управления отладчиком (рис. 25.8).

У ВАС ОКНО MONODEVELOP МОЖЕТ ВЫГЛЯДЕТЬ НЕМНОГО ИНАЧЕ. В MonoDevelop можно передвигать панели с места на место, как в Unity. Я переместил их у себя для удобства демонстрации происходящего в книжных примерах, поэтому у вас вид окна MonoDevelop может отличаться от моего. У вас должны отображаться те же самые панели, только они могут располагаться чуть иначе.

Исследование кода с помощью отладчика

Теперь, когда отладчик подключен и готов к работе, пришло время посмотреть, как он действует.

1. Вернитесь в Unity и щелкните на кнопке Play (Играть), чтобы запустить сцену. Почти тут же Unity зависнет, и всплывет окно MonoDevelop. Иногда в Windows MonoDevelop не всплывает автоматически, но Unity выглядит зависшим. В этом

случае просто переключитесь на окно MonoDevelop вручную, и вы увидите картину, изображенную на рис. 25.7.

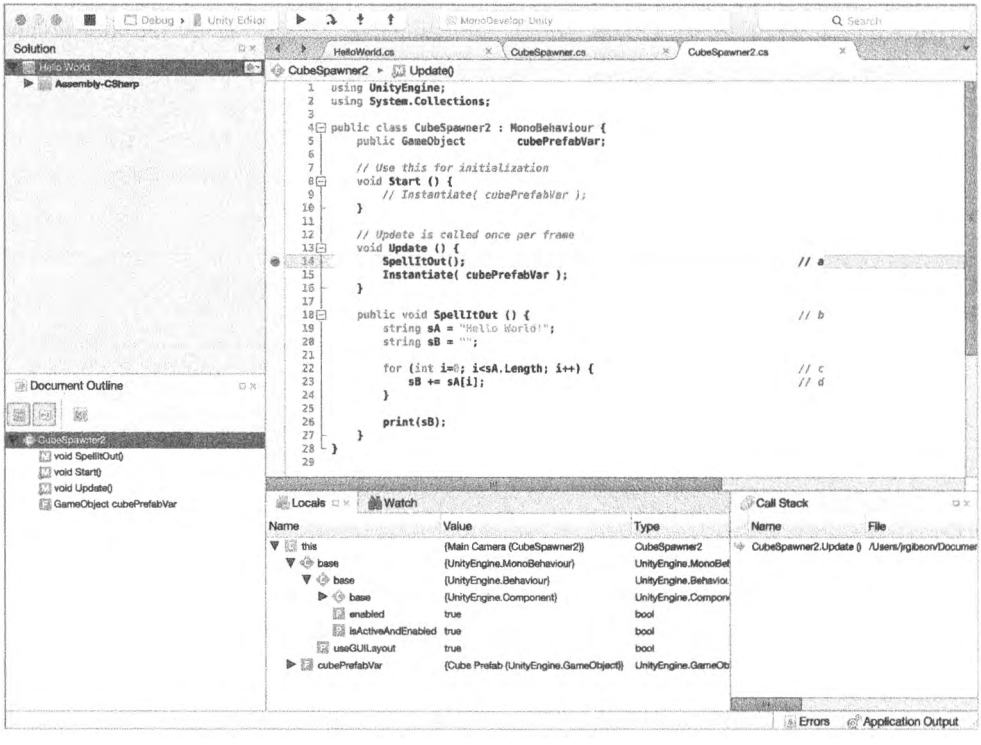


Рис. 25.7. Отладчик остановил выполнение в строке 14

Выполнение функции `Update()` приостановилось в строке 14, где мы поставили точку останова. Серая стрелка в поле слева и желтая подсветка строки показывают текущую выполняемую строку. Когда отладчик приостанавливает выполнение, процесс Unity зависает. Это значит, что вы не сможете переключиться в окно Unity, используя обычные средства, пока выполнение не продолжится.

В режиме отладки меняются некоторые кнопки в верхней части окна MonoDevelop (рис. 25.8).

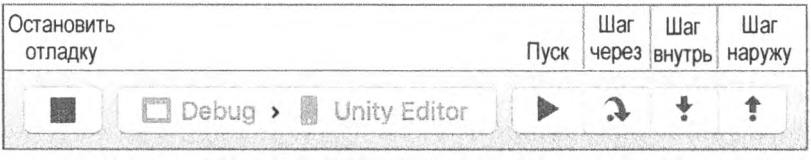


Рис. 25.8. Кнопки управления отладчиком

Следующие шаги описывают работу разных кнопок управления отладчиком. Перед тем как приступить к их выполнению, я советую прочитать врезку «Наблюдение за переменными в отладчике» в конце главы.

2. Щелкните на кнопке Run (Пуск) отладчика в окне MonoDevelop (как показано на рис. 25.8). Она возобновит выполнение сценария в Unity. Когда среда Unity вновь встретит точку останова, подобную той, что мы поставили в строке 14, она снова зависнет, пока вы не возобновите выполнение.

После щелчка на кнопке Run (Пуск) Unity возобновит и продолжит выполнение, пока вновь не достигнет точки останова. После щелчка на кнопке Run (Пуск) Unity выполнит игровой цикл, запустит новый кадр и снова остановится в строке 14 (когда вызовет `Update()`), поэтому вы, возможно, ничего не заметите, кроме того, что моргнет окно MonoDevelop.

В зависимости от типа вашего компьютера вам может понадобиться переключиться обратно в окно Unity (то есть в приложение), чтобы движок Unity фактически перешел к воспроизведению следующего кадра. На некоторых компьютерах Unity автоматически переходит к следующему кадру, пока вы остаетесь в окне MonoDevelop, на некоторых — нет. Если строка с точкой останова не подсвечивается желтым цветом после щелчка на кнопке Run (Пуск), переключитесь в окно Unity — это должно заставить движок начать выполнение следующего кадра и вновь остановиться на точке останова.

Как уже отмечалось выше, когда выполнение кода останавливается в отладчике (то есть когда вы видите серую стрелку и желтую подсветку строки, как показано на рис. 25.7), вы не сможете переключиться в окно Unity. Это нормально и объясняется тем, что Unity полностью останавливается, ожидая, пока вы исследуете код в отладчике. Unity продолжит нормальную работу, когда вы прекратите отладку.

---

3. Когда выполнение вновь остановится в строке 14, щелкните на кнопке Step Over (Шаг через). Желтая подсветка переместится на строку 15 без входа в функцию `SpellItOut()`. Функция `SpellItOut()` запустится и вернет управление, но отладчик перешагнет через нее. Операцию Step Over (Шаг через) удобно использовать, когда не требуется исследовать работу вызываемой функции.

4. Снова щелкните на кнопке Run (Пуск). Unity перейдет к выполнению следующего кадра, и выполнение снова остановится в строке 14.

5. На этот раз щелкните на кнопке Step Into (Шаг внутрь). Желтая подсветка перепрыгнет из строки 14 в строку 19 в функции `SpellItOut()`. Когда вы щелкаете на кнопке Step Into (Шаг внутрь), отладчик входит в вызываемую функцию, тогда как щелчок на Step Over (Шаг через) просто выполняет функцию, не входя в нее.

6. Теперь, находясь внутри функции `SpellItOut()`, щелкните несколько раз на кнопке Step Over (Шаг через), чтобы пройти по строкам в функции `SpellItOut()`.

7. Продолжая щелкать на Step Over (Шаг через), можете понаблюдать, как изменяются `sA` и `sB` в процессе выполнения этой функции (см. врезку «Наблюдение за переменными в отладчике»). В каждой итерации цикла `for` в строках 22–24

в строку sB будет добавляться очередной символ из sA. Наблюдать за изменением содержимого переменных можно в панели отладчика Locals (Текущие данные), которую можно открыть, выбрав в меню MonoDevelop пункт View > Debug Windows > Locals (Вид > Окна отладки > Текущие данные).

8. Если желтая подсветка, соответствующая выполняемой строке, все еще находится в функции SpellItOut(), продолжайте выполнять шаг 7, щелкая на кнопке Step Over (Шаг через) до выхода из функции, затем щелкните на кнопке Run (Пуск) и затем Step Into (Шаг внутрь), чтобы вернуться в функцию SpellItOut().
9. Находясь внутри функции SpellItOut(), щелкните на кнопке Step Out (Шаг наружу). В результате отладчик выйдет из функции SpellItOut() и остановится в строке 15 (следующей за вызовом SpellItOut()). Тогда SpellItOut() выполнится до конца, просто вы не увидите процесса в отладчике. Это удобно, когда нужно пропустить окончание текущей функции, но остаться при этом в режиме пошагового выполнения.
10. Щелкните на кнопке Stop Debug (Остановить отладку), как показано на рис. 25.8, чтобы отключить отладчик MonoDevelop от процесса Unity, остановить отладку и вернуть Unity в нормальный режим выполнения.

Я настоятельно рекомендую воспользоваться отладчиком и исследовать работу рекурсивной функции Fac(), которую мы написали в конце главы 24 «Функции и параметры». Эта функция является отличным примером того, как отладчик помогает лучше понять работу кода.

НАБЛЮДЕНИЕ ЗА ПЕРЕМЕННЫМИ В ОТЛАДЧИКЕ

Одной из наиболее сильных сторон любого отладчика является возможность наблюдения за значениями отдельных переменных. Отладчик MonoDevelop предлагает три способа наблюдения за переменными. Прежде чем опробовать их, выполните инструкции по запуску отладчика, данные в этой главе.

Первый и самый простой способ — навести указатель мыши на любую переменную в панели с программным кодом в окне MonoDevelop. Если навести указатель мыши на имя переменной и задержать его в этом положении примерно на одну секунду, появится всплывающая подсказка со значением этой переменной. Но имейте в виду, что отображаемое значение соответствует точке выполнения в строке, подсвеченной желтым цветом, а не местоположению переменной в коде. Например, переменная sB многократно встречается в функции SpellItOut() и, независимо от того, на какой экземпляр вы наведете указатель мыши, в подсказке будет отображаться одно и то же значение.

Второй способ — найти переменную в панели Locals (Текущие данные) отладчика. Открыть эту панель можно, выбрав в меню MonoDevelop пункт View > Debug Windows > Locals (Вид > Окна отладки > Текущие данные). После этого панель Locals (Текущие данные) появится на экране. Здесь можно видеть список всех локальных переменных, доступных отладчику на данный момент. Шагнув внутрь функции SpellItOut(), как описывалось выше, и оказавшись в строке 19, вы увидите в спи-

ске три локальные переменные: `this`, `sA` и `sB`. Переменные `sA` и `sB` первоначально имеют значение `null`, но после их определения в строках 19 и 20 в панели Locals (Текущие данные) появятся их значения. Выполнив команду Step Over (Шаг через) несколько раз и достигнув строки 22 в отладчике, вы увидите, что была объявлена и определена переменная `i`. Переменная `this` ссылается на текущий экземпляр сценария `CubeSpawner2`. Щелкните на пиктограмме с треугольником рядом с именем переменной, чтобы увидеть список общедоступных полей внутри `this` с переменными `cubePrefabVar` и `base`. Раскройте так же список полей `base`, чтобы увидеть переменные базового класса, унаследованного в `CubeSpawner2`, то есть класса `MonoBehaviour`. Базовые классы, такие как `MonoBehaviour` (их также называют суперклассами, или родительскими классами), обсуждаются в главе 26 «Классы».

Третий способ наблюдения за переменными заключается в том, чтобы явно добавить их в панель Watch (Наблюдение). Чтобы открыть эту панель, выберите в меню MonoDevelop пункт View > Debug Windows > Watch (Вид > Окна отладки > Наблюдение). В панели Watch (Наблюдение) щелкните на пустой строке, чтобы добавить переменные для наблюдения (в поле с текстом «Click here to add a new watch» (Нажмите здесь, чтобы добавить новое наблюдение)). Введите в этом поле имя переменной, и MonoDevelop попробует показать ее значение. Например, введите имя `this.gameObject.name` и нажмите Return, после чего появится строка «Main Camera» — имя игрового объекта, к которому подключен сценарий. Если значение слишком длинное и не умещается по ширине панели Watch (Наблюдение), вы можете щелкнуть на пиктограмме с изображением лупы рядом со значением, чтобы увидеть значение целиком; такое иногда случается при работе с длинными строками текста.

Здесь важно отметить, что иногда ошибки в отладчике (как бы смешно это ни звучало) не позволяют увидеть переменную `this` в панели Locals (Текущие данные). Если с вами такое случится, просто добавьте переменную `this` в панель Watch (Наблюдение), этот прием в отношении `this` срабатывает всегда, даже когда панель Locals (Текущие данные) не отображает эту переменную.

Итоги

Это было краткое введение в отладку. Даже при том, что мы не использовали отладчик для поиска ошибок, вы смогли увидеть, как он помогает понять код. Запомните: всякий раз, когда вы сомневаетесь в работе кода, вы всегда можете выполнить его по шагам в отладчике.

Признаюсь честно: было не особенно приятно заставлять вас намеренно добавлять ошибки в код, но я искренне надеюсь, что смог дать вам опыт встречи с ними и показать, как их анализировать и исправлять, что поможет вам в будущем, когда вы столкнетесь с настоящими ошибками в своем коде. Помните, что всегда можно попробовать поискать в интернете (по тексту ошибки или хотя бы по ее коду) рецепт исправления. Как я писал в начале этой главы, умение пользоваться отладчиком — это один из важнейших навыков, который поможет вам стать компетентным и уверенным в себе программистом.

26 Классы

К концу этой главы вы узнаете, как создавать классы. Класс — это коллекция переменных и функций в одном объекте на C#. Классы являются основными строительными блоками в современных играх и, в более широком смысле, в объектно-ориентированном программировании.

Основы классов

Классы объединяют функции и данные. Говоря иначе, классы состоят из функций и переменных, которые, когда используются в классе, называются *методами* и *полями* соответственно. Классы часто используются для представления объектов в мире вашего игрового проекта. Например, представьте персонажа в типичной ролевой игре. Такой персонаж мог бы иметь несколько полей (или переменных):

```
string      name;      // Имя персонажа
float       health;    // Текущий уровень здоровья персонажа
float       healthMax; // Максимальный уровень здоровья, который он мог бы иметь
List<Item>  inventory; // Список всех предметов, которыми может владеть персонаж
List<Item>  equipped;  // Список предметов, которыми он уже владеет
```

Все эти поля есть у каждого персонажа в ролевой игре, потому что все персонажи имеют определенный уровень здоровья, владеют определенным набором предметов и имеют имена. Кроме того, некоторые методы (функции) могли бы использоваться персонажем или применяться для управления им. (Многоточия [...] в следующем листинге отмечают место, куда нужно добавить код, описывающий работу функций.)

```
void Move(Vector3 newLocation) {...}   // Позволяет персонажу перемещаться
void Attack(Character target) {...}    // Атакует указанного персонажа target
                                       // текущим оружием или заклинанием
void TakeDamage(float damageAmt) {...} // Понижает уровень здоровья персонажа
void Equip(Item newItem) {...}         // Добавляет предмет newItem в список
                                       // equipped предметов, имеющихся в наличии
```

Очевидно, что в настоящей игре персонаж будет обладать большим количеством полей и методов, чем описано здесь, но суть в том, что все персонажи в вашей игре должны иметь эти функции и переменные.

Вы уже использовали классы! Фактически, хотя об этом не говорилось явно, каждый фрагмент кода, написанный вами по этой книге к данному моменту, является частью класса, и вообще, каждый файл с кодом на C#, который вы создадите, можете считать отдельным классом.

Устройство класса (и сценария на C#)

На рис. 26.1 показаны наиболее важные элементы большинства классов. Не все они необходимы в классах, но считаются наиболее типичными.

- **Подключение библиотек** позволяет сценариям на C# использовать разные классы, созданные другими. Подключение выполняется инструкциями `using`. В данном примере подключаются библиотеки со стандартными элементами Unity и с коллекциями, такими как `List`. Инструкции подключения должны находиться в начале сценария.
- **Объявление класса** определяет имя класса и список расширяемых классов (см. раздел «Наследование классов» далее в этой главе). В данном случае класс `Enemy` *расширяет* класс `MonoBehaviour` (`MonoBehaviour` объявляется *суперклассом* для `Enemy`).
- **Поля** — переменные, локальные по отношению к классу, в том смысле, что они доступны всем функциям в классе по их именам. Кроме того, переменные, отмеченные как `public`, доступны всем другим сущностям, которые видят класс (см. раздел «Области видимости переменной» приложения Б «Полезные идеи»).

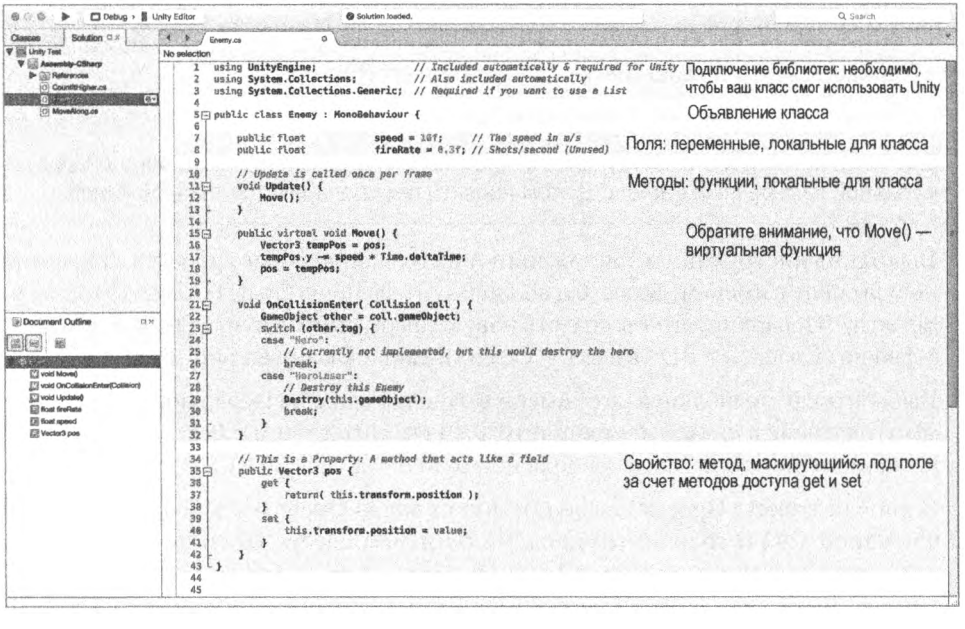


Рис. 26.1. Некоторые важные элементы класса

- **Методы** — это функции, содержащиеся в классе. Они могут обращаться к любым полям в классе, а также иметь свои локальные переменные (например, `Vector3 tempPos` в `Move()`), существующие только внутри функций. Методы определяют поведение классов. Виртуальные (`virtual`) методы — это функции особого вида, о которых подробнее рассказывается в разделе «Наследование классов» далее в этой главе.
- **Свойства** можно считать функциями, маскирующимися под поля за счет использования методов доступа `get` и `set`. Более подробно о свойствах рассказывается в разделе «Свойства» далее в этой главе.

Прежде чем погрузиться в подробности, настроим проект, в котором будем использовать этот код.

Настройка проекта Enemy Class Sample Project

В приложении А «Стандартная процедура настройки проекта» приводятся подробные инструкции, как настраивать проекты Unity для глав этой книги. Выполните рекомендации во врезке, чтобы настроить проект для этой главы.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. За информацией о стандартной процедуре обращайтесь к приложению А.

- **Имя проекта:** Enemy Class Sample Project
- **Имя сцены:** _Scene_0 (Символ подчеркивания в начале имени сцены позволяет поместить ее в начало списка в панели Project (Проект))
- **Имена сценариев на C#:** на этот раз нет

Вам не нужно следовать инструкциям в приложении А в части подключения сценария к главной камере Main Camera. В этом проекте нет сценария для главной камеры.

1. Выполнив инструкции в приложении А по созданию нового проекта, сохраните новую сцену с именем _Scene_0 и воспользуйтесь меню Create (Создать) в панели Hierarchy (Иерархия), чтобы создать новую сферу, выбрав пункт Create > 3D Object > Sphere (Создать > 3D Объект > Сфера), как показано на рис. 26.2.
2. Выберите сферу, щелкнув на ее имени в панели Hierarchy (Иерархия). Затем переместите сферу в начало координат [0, 0, 0] (то есть $x = 0, y = 0, z = 0$), используя компонент Transform (выделен красной рамкой на рис. 26.2).
3. В панели Project (Проект) выберите пункт меню Create > C# Script (Создать > Сценарий C#) и сохраните сценарий с именем Enemy. Щелкните дважды на сценарии, чтобы открыть его в MonoDevelop, и введите следующий код (идентичный изображенному на рис. 26.1). Строки, которые нужно добавить, выделены жирным.

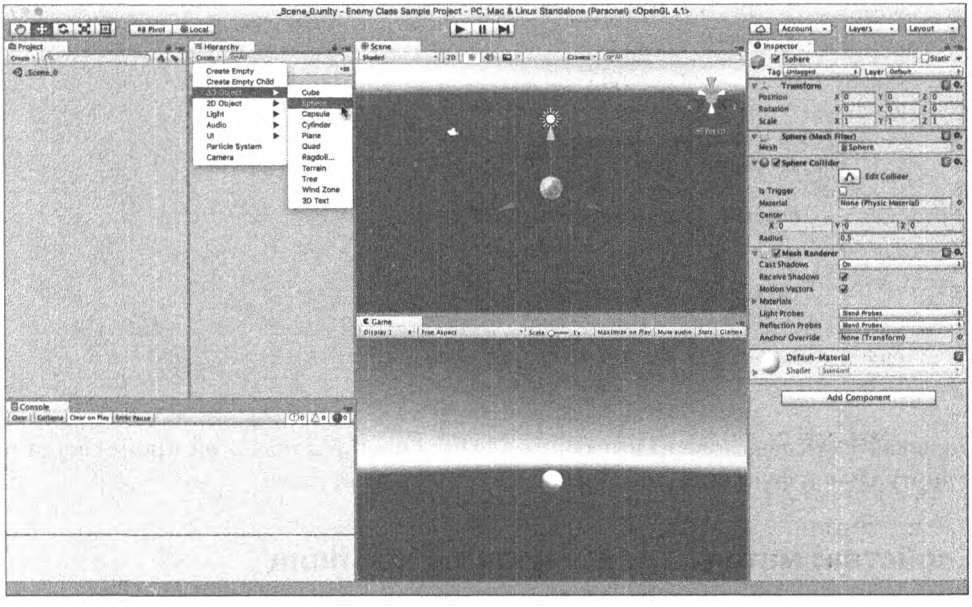


Рис. 26.2. Создание сферы в сцене _Scene_0

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Enemy : MonoBehaviour {

    public float        speed = 10f;     // Скорость в м/с
    public float        fireRate = 0.3f; // Выстрелов в секунду (не используется)

    // Update вызывается в каждом кадре
    void Update() {
        Move();
    }

    public virtual void Move() {
        Vector3 tempPos = pos;
        tempPos.y -= speed * Time.deltaTime;
        pos = tempPos;
    }

    void OnCollisionEnter( Collision coll ) {
        GameObject other = coll.gameObject;
        switch (other.tag) {
            case "Hero":
                // Пока не реализовано, но здесь наносятся повреждения герою
                break;
            case "HeroLaser":
                // Уничтожить этого врага
```

```
29                  Destroy(this.gameObject);
30                  break;
31              }
32          }
33
34          // Это свойство: метод, действующий как поле
35          public Vector3 pos {
36          get {
37              return( this.transform.position );
38          }
39          set {
40              this.transform.position = value;
41          }
42      }
43
44  }
```

Большая часть кода должна выглядеть для вас простой и знакомой, кроме свойства и виртуальной функции, о которых я расскажу в этой главе.

Свойства: методы, действующие как поля

В предыдущем листинге, в строках 16 и 18 в функции `Move()`, можно видеть, что свойство `pos` используется как обычное поле. Это достигается за счет использования методов доступа `get{}` и `set{}`, объявленных в строках 36–41. Они позволяют данному классу выполнять их код при каждой попытке прочитать или присвоить значение свойству `pos`. Каждая операция чтения свойства `pos` вызывает его метод `get{}`, который должен вернуть значение того же типа, что и свойство (то есть `Vector3`). Код в `set{}` выполняется при попытке присвоить значение свойству `pos`, а ключевое слово `value` играет роль *неявной переменной*, хранящей присваиваемое значение. Иными словами, операция присваивания значения `tempPos` свойству `pos` в строке 18 вызывает метод записи `set` свойства `pos` в строке 39; затем в строке 40 значение `tempPos`, хранящееся в переменной `value`, присваивается полю `this.transform.position`. Неявная переменная существует без вашего, как программиста, участия — от вас не требуется объявлять ее. Все методы доступа `set{}` в свойствах имеют неявную переменную `value`. Вы можете создать свойство с единственным методом доступа `get{}`, чтобы сделать свойство доступным только для чтения (или с единственным методом доступа `set{}`, чтобы сделать свойство доступным только для записи).

Свойство `pos` в предыдущем классе `Enemy` используется лишь для того, чтобы избавить вас от ввода лишнего кода при работе с полем `this.transform.position`. Однако в следующем листинге демонстрируется более интересный пример.

1. Создайте новый сценарий на C# с именем CountItHigher.
2. Подключите сценарий CountItHigher к сфере Sphere в сцене.
3. Щелкните дважды на сценарии CountItHigher в панели Project (Проект), чтобы открыть его в MonoDevelop, и введите следующий код:

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 class CountItHigher : MonoBehaviour {
 6     private int        _num = 0;                                    // a
 7
 8     void Update() {
 9         print( nextNum );
10     }
11
12     public int nextNum {                                            // b
13         get {
14             _num++;          // Увеличить значение _num на 1
15             return( _num ); // Вернуть новое значение _num
16         }
17     }
18
19     public int currentNum {                                         // c
20         get { return( _num ); }                                     // d
21         set { _num = value; }                                       // d
22     }
23 }
```

a. Целочисленное поле `_num` объявлено скрытым (спецификатором `private`), поэтому оно доступно только для данного экземпляра класса `CountItHigher`. Скрытые переменные этого экземпляра недоступны для других классов и других экземпляров `CountItHigher` (другие экземпляры `CountItHigher` будут иметь свое поле `_num`, недоступное для этого экземпляра).

b. `nextNum` — это свойство, доступное только для чтения. Так как свойство не имеет метода `set{}`, оно доступно только для чтения (например, `int x = nextNum;`) и недоступно для записи (например, `nextNum = 5;` вызовет ошибку).

c. `currentNum` — это свойство, доступное для чтения и для записи. Обе операции — `int x = currentNum;` и `currentNum = 5;` — будут благополучно выполняться.

d. Методы `get{}` и `set{}` можно записывать в одну строку. Обратите внимание, что в однострочной форме записи точка с запятой (`;`), завершающая инструкцию, ставится перед закрывающей фигурной скобкой (`}`), как показано в строках 20 и 21.

4. Вернитесь в Unity и щелкните на кнопке **Play** (Играть). Так как функция `Update()` вызывается движком Unity в каждом кадре, инструкция `print( nextNum );` будет выводить значения, увеличивающиеся с каждым кадром. Ниже показан вывод, полученный в ходе обработки первых пяти кадров:

```
1
2
3
4
5
```

При каждом чтении свойства `nextNum` (в инструкции `print( nextNum );`) происходит увеличение скрытого поля `_num`, и затем возвращается новое значение (строки 14 и 15 в листинге). Это был очень простой пример, но вообще методы доступа `get` и `set` могут выполнять любые операции, доступные обычным методам, и даже вызывать другие методы и функции.

Аналогично, `currentNum` — это общедоступное свойство, позволяющее читать и изменять значение `_num`. Так как `_num` является скрытым полем, очень удобно иметь общедоступное свойство `currentNum` для работы с ним.

Экземпляры класса как компоненты игрового объекта

Как было показано в предыдущих главах, когда вы перетаскиваете сценарий на игровой объект, он становится компонентом этого игрового объекта, так же как Transform, Rigidbody и другие элементы, которые можно видеть в инспекторе. Это означает, что можно получить ссылку на любой класс, подключенный к игровому объекту, вызвав метод `GameObject.GetComponent<>()` с типом класса в угловых скобках (см. строку 7 в следующем листинге).

1. Создайте новый сценарий на C# с именем MoveAlong.
2. Подключите сценарий MoveAlong к тому же игровому объекту Sphere, что и сценарий CountItHigher.
3. Откройте сценарий MoveAlong в MonoDevelop и введите следующий код, выделенный жирным:

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 class MoveAlong : MonoBehaviour {
 6
 7     void LateUpdate() {                                                    // a
 8         CountItHigher cih=this.gameObject.GetComponent<CountItHigher>(); // b
 9         if ( cih != null ) {                                               // c
10             float tX = cih.currentNum/10f;                                 // d
11             Vector3 tempLoc = pos;                                         // e
12             tempLoc.x = tX;
13             pos = tempLoc;
14         }
15     }
16
17     public Vector3 pos {                                                   // f
18         get { return( this.transform.position ); }
19         set { this.transform.position = value; }
20     }
21
22 }
```

a. `LateUpdate()` — еще одна встроенная функция, которую движок Unity вызывает в каждом кадре. В каждом кадре Unity сначала вызывает метод `Update()` всех классов, подключенных к игровым объектам и затем, когда все методы `Update()` выполнятся, Unity вызывает метод `LateUpdate()` всех объектов. Использование `LateUpdate()` гарантирует, что метод `Update()` класса `CountItHigher` будет вызван перед `LateUpdate()` в классе `MoveAlong`. Таким способом предотвращается состояние, известное как *состояние гонки* (более полная информация приводится в предупреждении ниже).

b. `Cih` — локальная переменная типа `CountItHigher`, она хранит ссылку на экземпляр `CountItHigher`, подключенный к игровому объекту `Sphere`. Вызов `GetComponent<CountItHigher>()` отыщет компонент CountItHigher (Script), подключенный к тому же игровому объекту Sphere, к которому подключен компонент MoveAlong (Script)[1].

c. Если вызвать `GetComponent<>()` и указать тип компонента, не подключенного к игровому объекту, `GetComponent<>()` вернет `null` (значение, обозначающее отсутствие компонента). Проверка `cih` на `null` перед использованием помогает избежать ошибок `NullReferenceException`.

d. Даже при том, что значение `cih.currentNum` имеет тип `int`, когда оно используется в математических операциях со значениями типа `float` (например, `cih.currentNum/10f`) или присваивается переменной типа `float` (обе операции выполняются в строке 10), оно автоматически интерпретируется как значение типа `float`.

e. Строки 11 и 13 используют свойство `pos`, объявленное в строках 17–20.

f. Фактически это то же самое свойство `pos`, как в классе `Enemy`, но для определения его методов доступа `get{}` и `set{}` использована однострочная форма.

Каждый вызов `LateUpdate` в этом коде будет отыскивать компонент CountItHigher (Script) данного игрового объекта и извлекать значение `currentNum` из него. Затем значение `currentNum` делится на 10, и результат присваивается координате X игрового объекта (с помощью свойства `pos`). По мере увеличения `CountItHigher._num` в каждом кадре игровой объект будет перемещаться вдоль оси X.

4. Сохраните оба сценария, этот и `CountItHigher`. В меню MonoDevelop выберите пункт File > Save All (Файл > Сохранить все). Если пункт Save All (Сохранить все) выглядит неактивным (окрашен в серый цвет), значит, вы уже сохранили все открытые сценарии.

---

[1] Вызов `GetComponent()` в каждом кадре влечет неэффективное расходование вычислительных ресурсов, поэтому предпочтительнее сделать `cih` полем класса и устанавливать его значение в методе `Awake()` или `Start()`. Однако на данном этапе эффективность кода для нас менее важна, чем простота и ясность, поэтому `GetComponent()` вызывается в каждом кадре.

5. Щелкните на кнопке Play (Играть) в Unity, чтобы увидеть результаты работы сценариев.
6. Сохраните сцену (выбрав в меню Unity пункт File > Save Scene (Файл > Сохранить сцену)).

ОСТЕРЕГАЙТЕСЬ СОСТОЯНИЯ ГОНКИ! Состояние гонки возникает всякий раз, когда две сущности полагаются друг на друга, но вы не уверены, какая из них выполнится или будет создана первой. Именно поэтому в предыдущем примере используется метод `LateUpdate()`. Если в классе `MoveAlong` использовать метод `Update()`, мы не смогли бы заранее предсказать, какой метод `Update()` будет вызван первым — из класса `CountItHigher` или из класса `MoveAlong`, — поэтому игровой объект мог бы перемещаться до или после изменения `_num`, в зависимости от очередности вызовов. Использование `LateUpdate()` гарантирует, что все методы `Update()` в сцене уже будут вызваны и завершатся перед вызовом любого из методов `LateUpdate()`.

Более подробно о состоянии гонки рассказывается в главе 31 «Прототип 3.5: Space SHMUP Plus».

Наследование классов

Классы обычно расширяют (дополняют) содержимое и поведение других классов (то есть они основаны на других классах). В первом листинге в этой главе класс `Enemy` расширяет `MonoBehaviour`, как и все классы, которые вы видели в этой книге до сих пор. Выполните следующие инструкции, чтобы задействовать класс `Enemy` в вашей игре, а потом мы продолжим обсуждение.

Реализация проекта Enemy Class Sample Project

Выполните следующие шаги:

1. Создайте новую сцену (выберите в главном меню пункт File > New Scene (Файл > Новая сцена)). Сразу же сохраните сцену с именем `_Scene_1`.
2. Создайте в сцене новую сферу (GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера)).
   a. Переименуйте объект Sphere в EnemyGO (GO означает GameObject — игровой объект). Эта новая сфера никак не связана со сферой Sphere в сцене `_Scene_0` (то есть к ней не подключены компоненты сценариев).
   b. Установите координаты X, Y и Z в компоненте Transform объекта EnemyGO в *[0, 4, 0]* в инспекторе.
   c. Перетащите сценарий Enemy, написанный ранее, из панели Project (Проект) на объект EnemyGO в панели Hierarchy (Иерархия) для сцены `_Scene_1`.
   d. Выберите EnemyGO в иерархии; теперь Enemy (Script) должен появиться как компонент игрового объекта EnemyGO.

3. Перетащите EnemyGO из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон с именем EnemyGO. Как описывалось в предыдущих главах, о благополучном создании шаблона можно узнать по появлению элемента с именем EnemyGO и пиктограммой, изображающей синюю коробку, в панели Project (Проект) и окрашиванию в синий цвет имени EnemyGO в панели Hierarchy (Иерархия), что указывает, что это экземпляр шаблона EnemyGO.
4. Выберите главную камеру Main Camera в иерархии и установите ее координаты и другие параметры, как показано на рис. 26.3, в выделенных полях:
   a. Установите координаты местоположения *[0, -15, -10]*.
   b. Выберите в поле Clear Flags значение Solid Color.
   c. Измените значение Perspective в поле Projection на Orthographic.
   d. Введите в поле Size значение *20*.

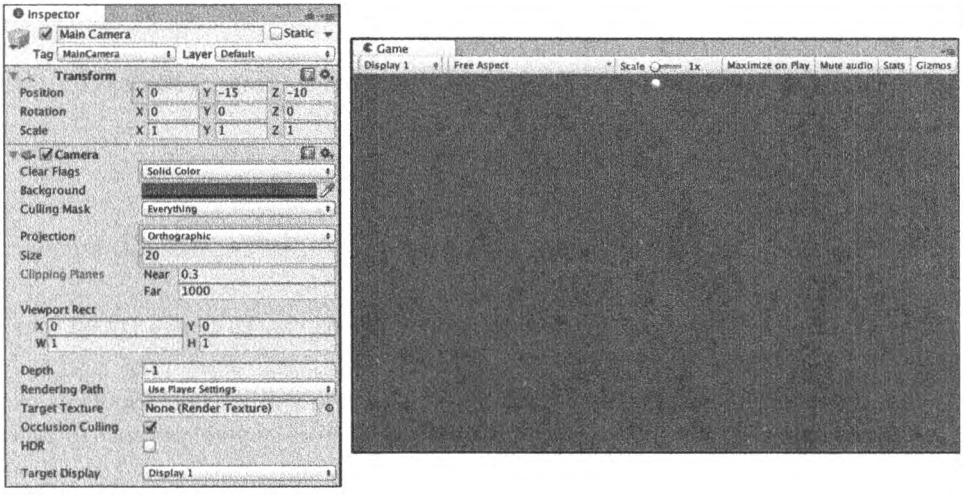


Рис. 26.3. Настройки камеры в сцене _Scene_1 и результат в панели Game (Игра)

Панель Game (Игра), изображенная справа на рис. 26.3, должна показывать сцену такой, как она выглядит через камеру.

5. Щелкните на кнопке Play (Играть). Вы должны увидеть, как экземпляр Enemy перемещается по экрану сверху вниз с постоянной скоростью.
6. Сохраните сцену! Всегда сохраняйте свои сцены.

Суперклассы и подклассы

Термины *суперкласс* и *подкласс* описывают отношение родства между двумя классами, где подкласс *наследует* свойства, поля и методы суперкласса. Например, класс `Enemy` наследует класс `MonoBehaviour`, то есть класс `Enemy` включает не только поля

и методы, объявленные в сценарии Enemy, но также все поля, свойства и методы своего суперкласса, MonoBehaviour и всех классов, которые наследуются классом MonoBehaviour. Именно поэтому сценарии в Unity уже знают о существовании полей, таких как gameObject и transform, и методов, таких как GetComponent<>().

Также можно создать класс, расширяющий Enemy:

1. Создайте новый сценарий на C# в панели Project (Проект) с именем EnemyZig.
2. Откройте сценарий в MonoDevelop, измените имя суперкласса с MonoBehaviour на Enemy и удалите методы Start() и Update(), оставив следующий код.

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class EnemyZig : Enemy {
6     // Удалите весь код по умолчанию, который среда Unity добавила в класс
      // EnemyZig
7 }
```

3. Выберите в панели Hierarchy (Иерархия) пункт меню Create > 3D Object > Cube (Создать > 3D объект > Куб).
   a. Переименуйте его в EnemyZigGO.
   b. Установите координаты EnemyZigGO в значения *[-4, 4, 0]*.
   c. Перетащите сценарий EnemyZig на игровой объект EnemyZigGO в панели Hierarchy (Иерархия).
   d. Перетащите EnemyZigGO из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон EnemyZigGO.
4. Щелкните на кнопке Play (Играть). Заметили, что кубик EnemyZigGO падает вниз с той же скоростью, что и сфера EnemyGO? Это объясняется тем, что класс EnemyZig наследует поведение класса Enemy!
5. Теперь добавьте в EnemyZig новый метод Move() (новые строки выделены жирным):

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class EnemyZig : Enemy {
 6
 7     public override void Move () {
 8         Vector3 tempPos = pos;
 9         tempPos.x = Mathf.Sin(Time.time * Mathf.PI*2) * 4;
10         pos = tempPos; // используется свойство pos суперкласса
11         base.Move();   // вызов метода Move() суперкласса
12     }
13
14 }
```

В этом коде мы *переопределили виртуальную функцию* `Move()`, унаследованную от суперкласса `Enemy`, и заменили ее новой. Метод `Enemy.Move()` должен быть объявлен в суперклассе виртуальным (`virtual`, как в строке 15, в сценарии с классом `Enemy`), чтобы его можно было переопределить в подклассе.

Эта новая функция `Move()` заставляет кубик падать по зигзагообразной траектории, смещая его влево и вправо по синусоиде (синус и косинус часто используются в реализациях циклического поведения, как в этом примере). В этом коде значение для компонента `x` позиции игрового объекта вычисляется как синус от текущего времени (количества секунд, прошедших с момента щелчка на кнопке **Play** (Играть)), умноженного на 2π, что обеспечивает полный цикл синусоиды каждую секунду. Затем значение синуса умножается на 4, чтобы обеспечить изменение координаты X в диапазоне от -4 до 4.

Вызов `base.Move()` в строке 11 требует от `EnemyZig` вызвать версию `Move()` в суперклассе (или «базовом» классе). В результате `EnemyZig.Move()` реализует движение влево и вправо, а `Enemy.Move()` заставляет **EnemyZigGO** падать с той же скоростью, что и **EnemyGO**.

Игровые объекты в этом примере имели в своих именах слово **Enemy** (враг), потому что похожую иерархию классов мы будем использовать в главе 31 для реализации разных врагов.

Итоги

Способность класса объединять данные с функциями дает разработчикам возможность использовать объектно-ориентированный подход, который будет представлен в следующей главе. Объектно-ориентированное программирование позволяет программистам рассуждать о своих классах как об объектах, которые могут двигаться и принимать решения независимо друг от друга. Этот подход очень хорошо сочетается с организацией Unity, основанной на игровых объектах `GameObject`, и помогает легко и быстро писать игры.

27 Объектно-ориентированное мышление

Эта глава рассказывает, как мыслить в терминах объектно-ориентированного программирования (ООП), и логически дополняет обсуждение классов в предыдущей главе.

К концу этой главы вы не только научитесь мыслить в терминах ООП, но и узнаете, как лучше структурировать проекты в среде разработки Unity.

Объектно-ориентированная метафора

Объектно-ориентированный подход проще всего описать с применением метафоры. Представьте стаю птиц. Стая может включать сотни и даже тысячи отдельных птиц, каждая из которых должна уклоняться от препятствий и других птиц, продолжая перемещаться вместе со стаей. Стаи птиц демонстрируют блестяще скоординированное поведение, которое на протяжении многих лет не поддавалось моделированию.

Имитация стаи птиц монолитным способом

До появления объектно-ориентированного программирования (ООП) всякая программа, по сути, была одной большой функцией, выполняющей все, что необходимо[1]. Эта единственная функция управляла всеми данными, перемещала спрайты на экране, обрабатывала ввод с клавиатуры, выполняла логику игры, воспроизводила музыку и отображала графику. В настоящее время такой подход к реализации всего в одной гигантской монолитной функции называется *монолитным программированием*.

Чтобы смоделировать стаю птиц монолитным способом, нужно организовать хранение большого массива птиц и создать программу, которая выполняла бы обход всех птиц и для каждой имитировала бы ее поведение в стае. Такая программа могла бы перемещать каждую птицу из ее местоположения в одном кадре

[1] Конечно, это грубое упрощение, но оно служит прекрасной основой для дальнейших рассуждений.

в новое местоположение в следующем кадре и хранить всю информацию о птицах в массиве.

Подобная монолитная программа была бы очень большой, громоздкой и сложной для отладки. Например, если классы `Enemy` и `EnemyZig` из предыдущей главы объединить в один монолитный класс, управляющий всеми врагами, его реализация могла бы выглядеть примерно так:

```
 1 using System.Collections;
 2 using System.Collections.Generic;
 3 using UnityEngine;
 4
 5 public class MonolithicEnemyController : MonoBehaviour {
 6     // Список всех врагов. Заполняется в инспекторе Unity
 7     public List<GameObject>        enemies;                          // a
 8     public float                   speed = 10f;
 9
10     void Update () {
11         Vector3 tempPos;
12
13         foreach ( GameObject enemy in enemies ) {                     // b
14             tempPos = enemy.transform.position;
15
16             switch ( enemy.name ) {                                   // c
17                 case "EnemyGO":
18                     tempPos.y -= speed * Time.deltaTime;
19                     break;
20                 case "EnemyZigGO":
21                     tempPos.x = 4 * Mathf.Sin(Time.time * Mathf.PI*2);
22                     tempPos.y -= speed * Time.deltaTime;
23                     break;
24             }
25
26             enemy.transform.position = tempPos;
27         }
28     }
```

a. Это список игровых объектов, где хранятся все враги. Ни к одному из врагов не подключен никакой код.

b. Цикл `foreach` в строке 13 перебирает игровые объекты в списке врагов `enemies`.

c. Так как к объектам врагов не подключено никакого кода, пришлось использовать эту инструкцию `switch`, реализующую движение для всех видов врагов.

В этом простом примере код получился довольно коротким и на самом деле не особенно «монолитным», но он утратил изящество и способность к расширению кода из главы 26 «Классы». Если бы кто-то взялся написать игру с 20 разными видами врагов, используя подобный монолитный класс, единственная функция `Update()` легко могла бы вырасти до нескольких сотен строк, потому что для каждого вида врага потребовалось бы добавить свою инструкцию `case`. К счастью, есть более простой и удобный путь. Чтобы добавить 20 дополнительных видов врагов, используя

объектно-ориентированный прием наследования из главы 26, потребовалось бы создать 20 маленьких классов (таких, как `EnemyZig`), каждый из которых получился бы достаточно коротким и простым для понимания и отладки.

Для моделирования стаи птиц с использованием ООП вместо монолитной функции применяется другой подход, заключающийся в моделировании каждой отдельной птицы, ее восприятия окружающего пространства и действий (и все это внутри объекта, представляющего одну птицу).

Имитация стаи птиц с использованием ООП и модели птицы

До 1987 года было предпринято несколько попыток сымитировать поведение птиц в стае и рыб в косяке с использованием приемов монолитного программирования. В то время считалось, что для имитации сложного скоординированного поведения стаи единственная функция должна управлять всеми данными, используемыми в имитации.

Это предубеждение было разрушено публикацией статьи «Flocks, Herds, and Schools: A Distributed Behavioral Model», написанной Крейгом Рейнолдсом в 1987 году[1]. В этой статье Рейнолдс описал невероятно простое объектно-ориентированное решение имитации стайного поведения, получившее название *Boids* (рой). На самом базовом уровне имитация поведения стаи опирается на три простых правила.

1. **Предотвращение столкновений:** предотвращение столкновений с соседними членами стаи.
2. **Согласование скорости:** согласование скорости с соседними членами стаи.
3. **Концентрация вокруг центра стаи:** стремление держаться рядом с центром в группе с соседними членами стаи.

Объектно-ориентированная реализация стаи

В этом учебном примере мы напишем простую реализацию роя Рейнолдса, демонстрирующую возможность создания сложного коллективного поведения с помощью объектно-ориентированного кода. Сначала создайте новый проект, следуя инструкциям во врезке. Когда вы будете выполнять инструкции по реализации этого проекта, я советую вооружиться карандашом и отмечать каждый шаг по его выполнении.

[1] C. W. Reynolds, «Flocks, Herds, and Schools: A Distributed Behavioral Model», Computer Graphics, 21(4), July 1987 (acm SIGGRAPH '87 Proceedings), 25–34.

НАСТРОЙКА ПРОЕКТА BOIDS

Создайте новый проект в Unity, следуя стандартной процедуре настройки, описанной в приложении А «Стандартная процедура настройки проекта».

- **Имя проекта:** Boids
- **Имя сцены:** _Scene_0

Все остальное вы добавите по ходу чтения главы.

Создание простой модели птицы

Чтобы получить модель птицы, создадим комбинацию из вытянутых кубиков. По завершении у нас получится шаблон игрового объекта `Boid`, изображенный на рис. 27.1.

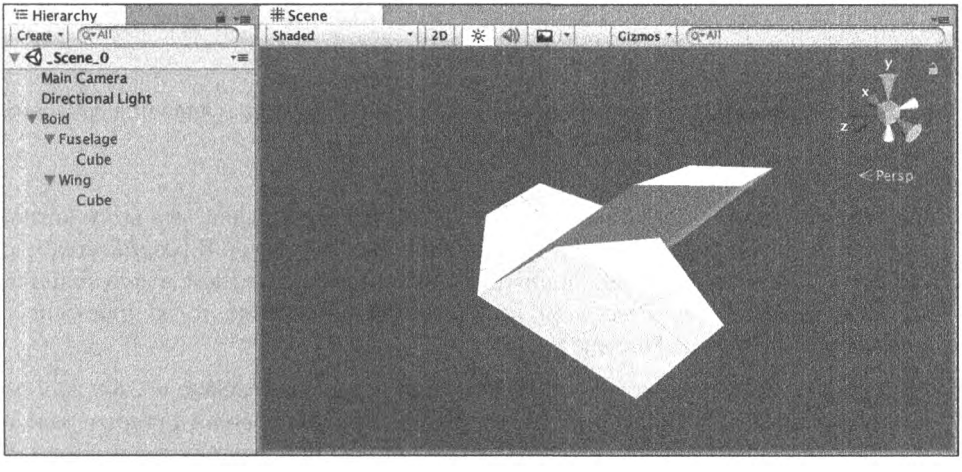


Рис. 27.1. Законченная модель птицы Boid

Выполните следующие шаги:

1. Выберите пункт GameObject > Create Empty (Игровой объект > Создать пустой) в меню Unity.
   a. Переименуйте его, дав имя Boid.
   b. Щелкните на пустом месте в панели Hierarchy (Иерархия), чтобы снять выделение с объекта Boid.
2. Снова выберите пункт GameObject > Create Empty (Игровой объект > Создать пустой) в меню Unity.
   a. Переименуйте новый игровой объект, дав ему имя Fuselage.

b. Наведите указатель мыши на объект **Fuselage** (рис. 27.2A), нажмите левую кнопку и перетащите **Fuselage** на **Boid** в панели **Hierarchy** (Иерархия) (рис. 27.2B).

Это сделает объект **Fuselage** дочерним по отношению к **Boid**. Рядом с элементом **Boid** в панели появится пиктограмма с треугольником, показывающая, что его можно распахнуть и увидеть дочерние объекты (под указателем мыши на рис. 27.2C). Если распахнуть элемент **Boid**, щелкнув на этом треугольнике, иерархия должна выглядеть так, как показано на рис. 27.2C.

Рис. 27.2. Вложенные игровые объекты в иерархии (то есть один является дочерним по отношению к другому)

3. Щелкните правой кнопкой на элементе **Fuselage** и в открывшемся контекстном меню выберите пункт **3D Object > Cube** (3D объект > Куб). В результате будет создан новый объект **Cube**, дочерний по отношению к **Fuselage** (если куб появился в списке не как дочерний объект, принадлежащий **Fuselage**, тогда просто перетащите его мышью на **Fuselage**).
4. Настройте параметры компонента **Transform** объектов **Fuselage** и **Cube**, как показано на рис. 27.3. Сочетание значений масштаба и поворота родительского объекта **Fuselage** приведет к тому, что дочерний **Cube** приобретет вытянутую заостренную форму.

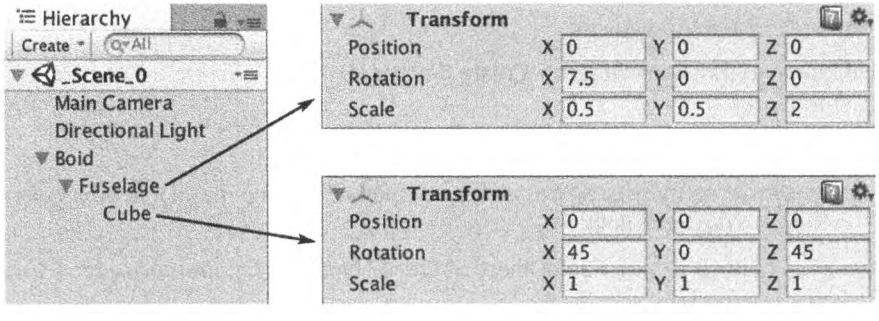


Рис. 27.3. Настройки компонентов Transform для Fuselage и его потомка Cube

5. Выберите объект Cube, принадлежащий Fuselage. Щелкните правой кнопкой на имени компонента Box Collider в панели Inspector (Инспектор) и в открывшемся контекстном меню выберите пункт Remove Component (Удалить компонент). В результате компонент Box Collider будет удален из объекта Cube, что позволит другим объектам беспрепятственно проходить сквозь него. Другая причина, почему мы удалили коллайдер, — форма коллайдера не изменяется с изменением формы куба, поэтому физические границы коллайдера не будут соответствовать видимым границам куба.
6. Выделите объект Fuselage и выберите в главном меню пункт Edit > Duplicate (Правка > Дублировать). В иерархии, в списке дочерних объектов, принадлежащих Boid, должен появиться второй объект Fuselage с именем Fuselage (1).
   a. Переименуйте объект Fuselage (1) в Wing.
   b. Настройте параметры компонента Transform для Wing, как показано на рис. 27.4.

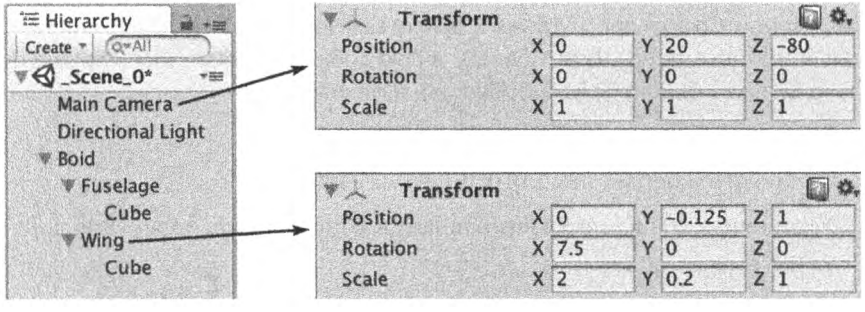


Рис. 27.4. Настройки компонентов Transform для Wing и Main Camera (шаги 6 и 13)

7. Теперь создайте материал, чтобы сформировать след, остающийся за птицей при перемещении в пространстве:
   a. В главном меню Unity выберите пункт Assets > Create > Material (Ресурсы > Создать > Материал) и дайте новому материалу имя TrailMaterial.
   b. Выделите TrailMaterial в панели Project (Проект) и в верхней части панели Inspector (Инспектор) выберите пункт Particles > Additive (Частицы > Аддитивный) в меню Shader.
   c. Справа от раздела Particle Texture в инспекторе появится пустое поле с изображением текстуры и с надписью None (Texture) (Нет текстуры). Щелкните на кнопке Select (Выбрать) в этом поле и в открывшемся окне выберите текстуру Default-Particle. Теперь в поле с изображением должен появиться белый круг с размытыми краями.
8. Щелкните на объекте Boid в панели Hierarchy (Иерархия), чтобы выделить его. Выберите в главном меню пункт Component > Effects > Trail Renderer (Компо-

нент > Эффекты > Визуализатор следа). В результате в Boid добавится компонент Trail Renderer. Выберите компонент Trail Renderer и в панели Inspector (Инспектор):

a. Щелкните на пиктограмме с треугольником рядом с элементом Materials, чтобы распахнуть его.

b. Щелкните на маленьком кружке справа от Element 0 None (Material).

c. В открывшемся списке выберите только что созданный материал TrailMaterial.

d. Установите параметр Time в Trail Renderer равным 1.

e. Установите ширину в поле Width компонента Trail Renderer равной 0.25. Если теперь воспользоваться инструментом Move (Перемещение), чтобы подвинуть Boid в окне Scene (Сцена), в процессе перемещения за объектом должен оставаться светящийся след.

9. Пока объект Boid остается выделенным в иерархии, выберите в главном меню Unity пункт Component > Physics > Sphere Collider (Компонент > Физика > Сферический коллайдер). В результате в Boid добавится компонент Sphere Collider. Выберите компонент Sphere Collider и в инспекторе:

   a. Установите флажок Is Trigger.

   b. Установите координаты [0, 0, 0] в поле Center.

   c. Установите значение Radius равным 4 (оно будет скорректировано далее в коде).

10. Пока объект Boid остается выделенным в иерархии, выберите в главном меню Unity пункт Component > Physics > Rigidbody (Компонент > Физика > Твердое тело). Снимите флажок Use Gravity для компонента Rigidbody в инспекторе.

11. Перетащите Boid из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон с именем Boid. Завершенная модель Boid должна выглядеть так, как показано на рис. 27.1.

12. Удалите синий экземпляр Boid из панели Hierarchy (Иерархия). Теперь, когда в панели Project (Проект) имеется шаблон Boid, объект в иерархии стал не нужен.

13. Выберите главную камеру Main Camera в иерархии и установите параметры ее компонента Transform так, как показано на рис. 27.4. Это отдалит главную камеру от центра сцены и позволит видеть сразу несколько птиц.

14. Выберите в главном меню пункт GameObject > Create Empty (Игровой объект > Создать пустой). Дайте новому игровому объекту имя BoidAnchor. Этот пустой игровой объект BoidAnchor будет служить родителем для всех экземпляров Boid, которые будут добавлены в сцену, и поможет сократить объем информации, отображаемой в панели Hierarchy (Иерархия).

15. Сохраните сцену. Мы внесли много изменений, и было бы обидно потерять все.

Сценарии на C#

В этом проекте будет использоваться пять разных сценариев на C#, каждый из которых выполняет важную работу.

- **Boid** — этот сценарий мы подключим к шаблону **Boid**. Он будет управлять движением отдельных объектов **Boid**. Поскольку программа является объектно-ориентированной, каждый объект **Boid** будет «мыслить» самостоятельно и реагировать, опираясь на свое видение окружающего мира.
- **Neighborhood** — этот сценарий мы также подключим к шаблону **Boid**. Он будет следить за соседями по стае. Ключом к пониманию мира каждой птицей **Boid** является знание о существовании других птиц, находящихся по соседству.
- **Attractor** — птицам в стае нужен некий «центр притяжения», вокруг которого они будут концентрироваться, и этот простой сценарий будет подключен к игровому объекту, играющему роль такого центра.
- **Spawner** — этот сценарий будет подключен к главной камере **Main Camera**. **Spawner** содержит поля (переменные), используемые всеми экземплярами, которые он создает из шаблона **Boid**.
- **LookAtAttractor** — также будет подключен к главной камере **Main Camera**. Этот сценарий будет поворачивать камеру в сторону **Attractor** в каждом кадре.

Конечно, то же самое можно было бы сделать с меньшим количеством сценариев, но тогда каждый из них получился бы больше, чем хотелось бы. Этот пример написан в духе расширения объектно-ориентированного программирования, известного как *компонентно-ориентированное проектирование*. Более подробную информацию вы найдете во врезке.

КОМПОНЕНТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

Шаблон проектирования «Компонент» (Component) был формализован в 1994 году, в книге «Design Patterns: Elements of Reusable Object-Oriented Software»[1], написанной «бандой четырех». Основная идея шаблона «Компонент» заключается в объединении связанных функций и данных в общий класс и в то же время в сохранении каждого класса как можно более маленьким и узкоспециализированным[2].

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides, «Design Patterns: Elements of Reusable Object-Oriented Software» (Reading, MA: Addison-Wesley, 1994). В этой книге также был формализован шаблон проектирования «Одиночка» (Singleton) и некоторые другие, используемые в моей книге (*Гамма Эрих, Хелм Ричард, Джонсон Ральф, Влиссидес Джон.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2016. — *Примеч. пер.*).

[2] Полное описание шаблона «Компонент» намного сложнее, но и такого вполне достаточно для наших целей.

По названию нетрудно догадаться, что в Unity мы постоянно используем такие компоненты. Каждый игровой объект в Unity — это очень маленький класс, способный выступать в роли контейнера для разных компонентов, каждый из которых решает конкретную — и независимую — задачу. Например:

- Transform управляет местоположением, поворотом, масштабом и иерархией.
- Rigidbody управляет движением и физическими аспектами.
- Коллайдеры определяют факт столкновения и управляют объемом, в котором эти столкновения определяются.

Несмотря на наличие определенных связей между собой, все они достаточно независимы, чтобы быть отдельными компонентами. Выделение отдельных компонентов также упрощает их расширение в будущем: отделение коллайдеров от твердого тела Rigidbody позволяет с легкостью создавать новые виды коллайдеров, например конический коллайдер `ConeCollider`, — при этом нет необходимости менять реализацию твердого тела Rigidbody, чтобы приспособить ее для работы с новым типом коллайдера.

Это, безусловно, важно для разработчиков игрового движка, но что это значит для нас как дизайнеров игр и разработчиков прототипов? Самое важное — компонентно-ориентированное мышление позволяет нам писать короткие классы. Короткие сценарии проще писать, делиться ими с другими, повторно использовать и отлаживать — все это более чем достойные цели.

Единственная отрицательная черта компонентно-ориентированного дизайна — для его реализации требуется величайшая предусмотрительность, что в определенной степени противоречит философии прототипирования, главной целью которой является как можно быстрее получить действующий прототип. В результате в нескольких первых главах части III этой книги вы столкнетесь с традиционным стилем прототипирования, цель которого — как можно быстрее получить хоть что-то работающее, а в главе 35 «Прототип 6: Dungeon Delver» мы исследуем более компонентно-ориентированный подход.

За дополнительной информацией о разных шаблонах проектирования обращайтесь к разделу «Шаблоны проектирования программного обеспечения» в приложении Б «Полезные идеи».

Сценарий Attractor

Начнем со сценария Attractor. Attractor — это объект, вокруг которого будут концентрироваться члены стаи Boid. Без него члены стаи могли бы стремиться держаться рядом друг с другом, но сама стая разлетится за края экрана.

1. Выберите в главном меню Unity пункт GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера), чтобы создать новую сферу, и переименуйте ее в Attractor.

2. Выберите Attractor. В инспекторе щелкните правой кнопкой на имени компонента Sphere Collider и в открывшемся контекстном меню выберите пункт Remove

Component (Удалить компонент), чтобы удалить компонент Sphere Collider из объекта Attractor.

3. Установите масштаб в поле Scale компонента Transform объекта Attractor равным S:[4, 0.1, 4] (то есть X=4, Y=0.1 и Z=4).
4. Выберите в главном меню Unity пункт Component > Effects > Trail Renderer (Компонент > Эффекты > Визуализатор следа). Выберите компонент Trail Renderer объекта Attractor и в инспекторе:
   a. Щелкните на пиктограмме с треугольником в разделе Materials, чтобы раскрыть его.
   b. Щелкните на маленьком кружке справа от Element 0 None (Material).
   c. Выберите в открывшемся списке материалов Sprites-Default.
   d. Установите параметр Time в Trail Renderer равным 4.
   e. Установите ширину в поле Width компонента Trail Renderer равной 0,25.
5. Щелкните на кнопке Add Component (Добавить компонент) в нижней части инспектора и выберите в открывшемся меню пункт New Script (Новый сценарий). Дайте новому сценарию имя Attractor и щелкните на кнопке Create and Add (Создать и добавить), чтобы создать сценарий и добавить его в объект Attractor одним действием.
6. Откройте сценарий Attractor в MonoDevelop и введите код из следующего листинга. Строки, которые нужно добавить, выделены жирным.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Attractor : MonoBehaviour {
    static public Vector3 POS = Vector3.zero;                          // a

    [Header("Set in Inspector")]
    public float        radius = 10;
    public float        xPhase = 0.5f;
    public float        yPhase = 0.4f;
    public float        zPhase = 0.1f;

    // FixedUpdate вызывается при каждом пересчете физики (50 раз в секунду)
    void FixedUpdate () {                                              // b
        Vector3 tPos = Vector3.zero;
        Vector3 scale = transform.localScale;
        tPos.x = Mathf.Sin(xPhase * Time.time) * radius * scale.x;     // c
        tPos.y = Mathf.Sin(yPhase * Time.time) * radius * scale.y;
        tPos.z = Mathf.Sin(zPhase * Time.time) * radius * scale.z;
        transform.position = tPos;
        POS = tPos;
    }

```

a. POS, как статическая переменная, используется всеми экземплярами Attractor (хотя в данном случае у нас будет только один экземпляр класса Attractor). Когда поле объявляется статическим (static), оно принадлежит самому классу, а не его экземплярам, то есть POS — это переменная класса, а не поле экземпляра. Это значит, что пока POS и класс Attractor являются общедоступными, любой экземпляр любого класса сможет обратиться к переменной POS по имени Attractor.POS. Она будет использоваться всеми экземплярами Boid для определения местоположения экземпляра Attractor.

b. Метод FixedUpdate() похож на метод Update(), но вызывается в кадрах пересчета физики, а не изображения. Дополнительную информацию вы найдете во врезке.

c. Как упоминалось в предыдущей главе, синус часто используется для реализации циклических движений. Здесь поля, соответствующие разным фазам (например, xPhase), заставляют Attractor перемещаться по сцене вокруг каждой оси (X, Y и Z) немного не в фазе друг с другом.

7. Сохраните сценарий Attractor, вернитесь в Unity и щелкните на кнопке Play (Играть). Вы должны увидеть, как объект Attractor движется в игровом объеме по синусу с учетом радиуса radius и масштаба transform.localScale объекта Attractor.

ФИКСИРОВАННЫЕ ОБНОВЛЕНИЯ И ФИЗИЧЕСКИЙ ДВИЖОК

Unity старается работать как можно быстрее, поэтому она отображает новый кадр, как только это становится возможно. Это означает, что Time.deltaTime между соседними вызовами Update() может изменяться в пределах от менее чем 1/400 секунды на быстрых компьютерах до более 1 секунды на медленных мобильных устройствах. Кроме того, частота вызовов Update() может существенно меняться от кадра к кадру на одном и том же компьютере в зависимости от множества факторов, поэтому величина Time.deltaTime между соседними вызовами Update() всегда будет получаться разной.

Физические движки, такие как NVIDIA PhysX, используемый в Unity, полагаются на постоянную периодичность, чего Update() не в состоянии обеспечить. Как результат, Unity всегда выполняет физические расчеты с постоянной частотой независимо от мощности компьютера. Частота вызовов FixedUpdate() определяется статическим полем Time.fixedDeltaTime. По умолчанию Time.fixedDeltaTime получает значение 0.02f (то есть 1/50), иными словами движок PhysX выполняет расчеты и вызывает FixedUpdate() 50 раз в секунду.

Как результат, метод FixedUpdate() лучше использовать для задач, которые касаются движущихся объектов с твердым телом Rigidbody (именно поэтому мы использовали его для управления движением объектов Attractor и Boid). Он также хорошо подходит для случаев, когда требуется постоянная периодичность, не зависящая от быстродействия компьютера.

FixedUpdate() вызывается непосредственно перед тем, как движок PhysX приступит к расчетам.

Имейте также в виду, что методы класса Input — GetKeyDown(), GetKeyUp(), GetButtonDown() и GetButtonUp() — никогда не должны вызываться в FixedUpdate(), потому что они работают с единственным вызовом Update() после события. Например, GetKeyDown() вернет true только в одном вызове Update() после нажатия клавиши, поэтому если между двумя вызовами FixedUpdates() несколько раз выполнится Update(), вызывающий Input.GetKeyDown(), внутри FixedUpdate() он вернет true, только если нажатие случится после последнего вызова Update() и перед вызовом FixedUpdate(). Независимо от того, имеет ли это замечание смысл прямо сейчас, просто запомните: никогда не вызывайте Input.GetKeyDown() и другие методы класса Input с именами, заканчивающимися на ...Down() или ...Up() внутри FixedUpdate(). Другие методы класса Input, такие как GetAxis(), GetKey() и GetButton(), прекрасно работают и в FixedUpdate(), и в Update(). Упомянутые методы класса Input мы будем использовать в части III книги.

Сценарий LookAtAttractor

Далее мы должны заставить главную камеру Main Camera следовать за перемещениями объекта Attractor.

1. Выберите главную камеру Main Camera в иерархии.
2. Создайте сценарий на C# с именем LookAtAttractor и подключите его к Main Camera (используя любой из способов, которые вы уже знаете).
3. Откройте сценарий LookAtAttractor в MonoDevelop и введите следующий код:

```
5 public class LookAtAttractor : MonoBehaviour {
6
7     void Update () {
8         transform.LookAt(Attractor.POS); // Да, просто добавьте эту строку!
9     }
10
11 }
```

4. Сохраните сценарий, вернитесь в Unity и щелкните на кнопке Play (Играть).

Теперь камера должна постоянно смотреть в сторону объекта Attractor.

Сценарий Boid — часть 1

Поскольку на класс Boid будет ссылаться несколько других сценариев, мы сейчас только создадим его, но не будем добавлять в него код. Это позволит другим сценариям ссылаться на класс Boid и компилироваться без ошибок (и слово Boid не будет подчеркиваться красной линией в MonoDevelop).

1. Выберите шаблон Boid в панели Project (Проект).
2. Щелкните на кнопке Add Component (Добавить компонент) в нижней части инспектора и выберите в открывшемся меню пункт New Script (Новый сценарий). Дайте новому сценарию имя Boid и щелкните на кнопке Create and Add (Создать и добавить).

Пока это все, что нам нужно от сценария Boid. Пойдем дальше.

Сценарий Spawner — часть 1

Сценарий Spawner будет подключаться к главной камере Main Camera, и вы сможете редактировать общедоступные поля класса Spawner в инспекторе Unity. Благодаря этому вы получите центральное место для настройки всех параметров, управляющих движением объектов Boid.

1. Выберите главную камеру Main Camera в иерархии.
2. Создайте сценарий с именем Spawner и подключите его к главной камере Main Camera любым известным вам способом.
3. Откройте сценарий Spawner в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Spawner : MonoBehaviour {
    // Этот класс реализует шаблон проектирования "Одиночка". Существовать может
    // только один экземпляр Spawner, поэтому сохраним его в статической
    // переменной S.
    static public Spawner       S;                                      // a
    static public List<Boid>    boids;                                  // b

    // Эти поля позволяют настраивать порядок создания объектов Boid
    [Header("Set in Inspector: Spawning")]
    public GameObject           boidPrefab;                             // c
    public Transform            boidAnchor;
    public int                  numBoids = 100;
    public float                spawnRadius = 100f;
    public float                spawnDelay = 0.1f;

    // Эти поля позволяют настраивать стайное поведение объектов Boid
    [Header("Set in Inspector: Boids")]
    public float                velocity = 30f;
    public float                neighborDist = 30f;
    public float                collDist = 4f;
    public float                velMatching = 0.25f;
    public float                flockCentering = 0.2f;
    public float                collAvoid = 2f;
    public float                attractPull = 2f;
    public float                attractPush = 2f;
    public float                attractPushDist = 5f;

```

```
31      void Awake () {
32          // Сохранить этот экземпляр Spawner в S
33          S = this;                                                    // d
34          // Запустить создание объектов Boid
35          boids = new List<Boid>();
36          InstantiateBoid();
37      }
38
39      public void InstantiateBoid() {
40          GameObject go = Instantiate(boidPrefab);
41          Boid b = go.GetComponent<Boid>();
42          b.transform.SetParent(boidAnchor);                           // e
43          boids.Add( b );
44          if (boids.Count < numBoids) {
45              Invoke( "InstantiateBoid", spawnDelay );                 // f
46          }
47      }
48
```

a. Поле `S` — это ссылка на объект-одиночку — один из шаблонов проектирования программного обеспечения, описанных в приложении Б «Полезные идеи». Шаблон «Одиночка» используется тогда, когда должен существовать только один экземпляр конкретного класса. Поскольку будет существовать только один экземпляр класса `Spawner`, его можно сохранить в статическом поле `S`. Соответственно, так же как с общедоступным статическим полем `POS` класса `Attractor`, в любой точке кода вы сможете использовать `Spawner.S` для ссылки на единственный экземпляр `Spawner`.

b. Список `List<Boid> boids` хранит ссылки на все экземпляры `Boid`, созданные экземпляром `Spawner`.

c. Значения в полях `boidPrefab` и `boidAnchor` этого сценария вы должны установить в инспекторе (как описывается в шагах 5 и 6 ниже).

d. Здесь `this` — это ссылка на экземпляр `Spawner`, хранящийся в `S`. В коде реализации класса `this` ссылается на текущий экземпляр этого класса. В сценарии `Spawner` ссылка `this` указывает на экземпляр `Spawner`, подключенный к главной камере Main Camera, который является единственным экземпляром `Spawner` в сцене _Scene_0.

e. Подчинение всех экземпляров `Boid` одному игровому объекту помогает поддержать порядок в панели Hierarchy (Иерархия). Эта строка подчиняет их все одному родителю — `boidAnchor` (в шаге 6 мы поместим в поле `boidAnchor` игровой объект BoidAnchor с помощью инспектора Unity). При желании увидеть все экземпляры `Boid` в иерархии вы сможете просто раскрыть родительский игровой объект BoidAnchor, щелкнув на пиктограмме с треугольником рядом с ним.

f. Первоначально метод `InstantiateBoid()` однократно вызывается в методе `Awake()`, а затем он использует функцию `Invoke()`, чтобы вызвать себя снова, если количество экземпляров `Boid` не сравнялось со значением `numBoids`. Функ-

ция `Invoke` принимает два аргумента: имя метода для вызова (в виде строки `"InstantiateBoid"`) и задержку во времени в секундах (`spawnDelay`, или 0.1 секунды).

4. Сохраните сценарий **Spawner**, вернитесь в Unity и выберите главную камеру **Main Camera** в панели **Hierarchy** (Иерархия).
5. Перетащите шаблон **Boid** из панели **Project** (Проект) в поле **boidPrefab** компонента **Spawner (Script)** главной камеры **Main Camera** в инспекторе.
6. Перетащите игровой объект **BoidAnchor** из панели **Hierarchy** (Иерархия) в поле **boidAnchor** компонента **Spawner (Script)** главной камеры **Main Camera** в инспекторе.

Щелкните на кнопке **Play** (Играть) в Unity. Вы увидите, как каждую десятую долю секунды **Spawner** создает новые экземпляры **Boid** и подчиняет их объекту **BoidAnchor**, но пока экземпляры **Boid** просто накапливаются в точке нахождения **BoidAnchor** в центре сцены, ничего не делая. Пришло время вернуться к сценарию **Boid**.

Сценарий Boid — часть 2

Вернитесь к сценарию **Boid** и выполните следующие шаги:

1. Откройте сценарий **Boid** в MonoDevelop и введите код, выделенный жирным.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Boid : MonoBehaviour {

    [Header("Set Dynamically")]
    public Rigidbody        rigid;                                    // a

    // Используйте этот метод для инициализации
    void Awake () {
        rigid = GetComponent<Rigidbody>();                            // a

        // Выбрать случайную начальную позицию
        pos = Random.insideUnitSphere * Spawner.S.spawnRadius;        // b

        // Выбрать случайную начальную скорость
        Vector3 vel = Random.onUnitSphere * Spawner.S.velocity;       // c
        rigid.velocity = vel;

        LookAhead();                                                  // d

        // Окрасить птицу в случайный цвет, но не слишком темный      // e
        Color randColor = Color.black;
        while ( randColor.r + randColor.g + randColor.b < 1.0f ) {
            randColor = new Color(Random.value, Random.value, Random.value);
        }
        Renderer[] rends = gameObject.GetComponentsInChildren<Renderer>();// f
        foreach ( Renderer r in rends ) {
```

```
30              r.material.color = randColor;
31          }
32          TrailRenderer tRend = GetComponent<TrailRenderer>();
33          tRend.material.SetColor("_TintColor", randColor);
34      }
35
36      void LookAhead() {                                              // d
37          // Ориентировать птицу клювом в направлении полета
38          transform.LookAt(pos + rigid.velocity);
39      }
40
41      public Vector3 pos {                                            // b
42          get { return transform.position; }
43          set { transform.position = value; }
44      }
45
46
```

a. Вызов `GetComponent<>()` требует довольно много времени, поэтому ради высокой производительности мы кэшируем ссылку (то есть сохраним ее, чтобы потом можно было быстро обратиться к ней) на компонент **Rigidbody**. Поле `rigid` позволит нам избавиться от необходимости вызывать `GetComponent<>()` в каждом кадре.

b. Статическое свойство `insideUnitSphere` класса `Random`, доступное только для чтения, генерирует случайный `Vector3` с координатами внутри сферы с радиусом, равным 1. Затем элементы вектора умножаются на общедоступное поле `spawnRadius` объекта-одиночки `Spawner`, чтобы поместить новый экземпляр `Boid` в случайно выбранную точку на расстоянии `spawnRadius` от начала координат (позиция [0, 0, 0]). Получившийся вектор `Vector3` присваивается свойству `pos`, которое определяется в конце этого листинга.

c. Статическое свойство `Random.onUnitSphere` генерирует `Vector3` с координатами точки где-то на поверхности сферы с радиусом, равным 1. Проще говоря, оно возвращает вектор `Vector3` с длиной, равной 1, и указывающий в случайном направлении. Мы умножаем его на поле `velocity` объекта-одиночки `Spawner` и присваиваем результат полю `velocity` компонента **Rigidbody** экземпляра **Boid**.

d. `LookAhead()` ориентирует **Boid** (птицу) клювом в направлении вектора `rigid.velocity`.

e. Строки 24–33, по большому счету, не особенно нужны, но они делают сцену более красочной. Эти строки выбирают для каждой птицы случайный цвет (достаточно ярко выглядящий на экране).

f. Вызов `gameObject.GetComponentsInChildren<Renderer>()` возвращает массив всех компонентов **Renderer**, подключенных к данному игровому объекту **Boid**, и всех его потомков. В данном случае он вернет компоненты **Renderer** кубов — игровых объектов **Fuselage** и **Wing**.

2. Сохраните сценарий, вернитесь в Unity и щелкните на кнопке **Play** (Играть).

Теперь птицы создаются в разных точках, летят в разных направлениях и окрашены в разные цвета, но они по-прежнему не реагируют ни на что в окружающем мире.

3. Вернитесь в MonoDevelop и добавьте в сценарий Boid следующие строки, выделенные жирным. Обратите внимание, что в следующем листинге некоторые строки пропущены. На протяжении оставшейся части книги я буду использовать многоточие (...) для обозначения пропущенных строк. Не удаляйте строки, скрывающиеся под многоточием.

```
 5 public class Boid : MonoBehaviour {
 ...                                                                    // a
41     public Vector3 pos {
42         get { return transform.position; }
43         set { transform.position = value; }
44     }
45
46     // FixedUpdate вызывается при каждом пересчете физики (50 раз в секунду)
47     void FixedUpdate () {
48         Vector3 vel = rigid.velocity;                                  // b
49         Spawner spn = Spawner.S;                                       // c
50
51         // ПРИТЯЖЕНИЕ - организовать движение в сторону объекта Attractor
52         Vector3 delta = Attractor.POS - pos;                           // d
53         // Проверить, куда двигаться, в сторону Attractor или от него
54         bool attracted = (delta.magnitude > spn.attractPushDist);
55         Vector3 velAttract = delta.normalized * spn.velocity;          // e
56
57         // Применить все скорости
58         float fdt = Time.fixedDeltaTime;
59
60         if (attracted) {                                               // f
61             vel = Vector3.Lerp(vel, velAttract, spn.attractPull*fdt);
62         } else {
63             vel = Vector3.Lerp(vel, -velAttract, spn.attractPush*fdt);
64         }
65
66         // Установить vel в соответствии с velocity в объекте-одиночке Spawner
67         vel = vel.normalized * spn.velocity;                           // g
68         // В заключение присвоить скорость компоненту Rigidbody
69         rigid.velocity = vel;
70         // Повернуть птицу клювом в сторону нового направления движения
71         LookAhead();
72     }
73 }
```

a. Многоточие (...) отмечает несколько пропущенных строк, потому что они не изменились по сравнению с предыдущим сценарием Boid.

b. Эта еще одна переменная `Vector3 vel`, отличная от переменной `Vector3 vel` в методе `Awake()`, потому что все локальные переменные существуют только в том методе, где они объявлены.

c. Я добавил локальную переменную `spn` для хранения `Spawner.S`, потому что строки, обращающиеся к `Spawner.S`, не умещались по ширине книжной страницы.

d. Здесь мы извлекаем позицию объекта **Attractor** из общедоступного статического поля `Attractor. POS`. Вычитая `pos` (позицию текущей птицы **Boid**) из позиции объекта **Attractor**, мы получаем вектор `Vector3`, указывающий из точки с координатами **Boid** в точку с координатами **Attractor**. Затем, проверяя близость птицы к центру притяжения, определяем, должна она приближаться к нему или отдаляться от него. В строке 54 можно видеть пример присваивания логического результата сравнения с переменной (вместо использования результата сравнения в инструкции `if`).

e. Вектор `delta`, направленный в сторону **Attractor**, нормализуется (то есть приводится к единичной длине) и умножается на `spn.velocity`, чтобы получить `velAttract` той же длины, что и `vel`.

f. Если птица достаточно далеко от центра притяжения **Attractor** и стремится к нему, вызовом `Lerp()` выполняется *линейная интерполяция* `vel` в направлении `velAttract`. Поскольку векторы `vel` и `velAttract` имеют одинаковую магнитуду (длину), интерполяция производит равномерное взвешивание. Если птица оказалась слишком близко от точки `Attractor.POS`, тогда выполняется линейная интерполяция `vel` в направлении, противоположном `velAttract`.

   Метод линейной интерполяции `Lerp()` принимает два вектора `Vector3` и создает новый вектор `Vector3` — взвешенную смесь двух входных векторов. Доля участия каждого входного вектора в результате определяется третьим аргументом. Если в третьем аргументе передать 0, новый вектор `vel` будет равен исходному вектору `vel`; если передать 1, новый вектор `vel` получит значение `velAttract`. Поскольку мы умножаем `spn.attractPull` на `fdt` (сокращенный вариант записи умножения `Spawner.S.attractPull` на `Time.fixedDeltaTime`), третий аргумент будет иметь значение `Spawner.S.attractPull/50`. Больше информации о линейной интерполяции вы найдете в разделе «Интерполяция» приложения Б «Полезные идеи».

g. Все это время, работая с векторами равной длины, мы определяли направление движения `vel` с определенной скоростью. Теперь `vel` нормализован, и его нужно умножить на скорость в поле `velocity` объекта-одиночки `Spawner`, чтобы получить окончательную скорость данной птицы — объекта **Boid**.

4. Сохраните сценарий, вернитесь в Unity и щелкните на кнопке **Play** (Играть).

Теперь все птицы будут притягиваться к объекту **Attractor**. Когда **Attractor** изменит направление движения, птицы также изменят направление полета, устремившись вслед за объектом **Attractor**. Уже неплохо, но мы можем кое-что усовершенствовать. Для этого нам нужна информация о соседних птицах.

Сценарий Neighborhood

Сценарий Neighborhood будет определять, какие птицы находятся по соседству к текущей, и снабжать нас информацией о них, включая координаты центральной точки и среднюю скорость в группе соседей, а также информировать о птицах, находящихся слишком близко.

1. Создайте новый сценарий на C# с именем Neighborhood и подключите его к шаблону Boid в панели Project (Проект).
2. Откройте сценарий Neighborhood в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Neighborhood : MonoBehaviour {
    [Header("Set Dynamically")]
    public List<Boid>       neighbors;
    private SphereCollider  coll;

    void Start() {                                                    // a
        neighbors = new List<Boid>();
        coll = GetComponent<SphereCollider>();
        coll.radius = Spawner.S.neighborDist/2;
    }

    void FixedUpdate() {                                              // b
        if (coll.radius != Spawner.S.neighborDist/2) {
            coll.radius = Spawner.S.neighborDist/2;
        }
    }

    void OnTriggerEnter(Collider other) {                             // c
        Boid b = other.GetComponent<Boid>();
        if (b != null) {
            if (neighbors.IndexOf(b) == -1) {
                neighbors.Add(b);
            }
        }
    }

    void OnTriggerExit(Collider other) {                              // d
        Boid b = other.GetComponent<Boid>();
        if (b != null) {
            if (neighbors.IndexOf(b) != -1) {
                neighbors.Remove(b);
            }
        }
    }

    public Vector3 avgPos {                                           // e
        get {
            Vector3 avg = Vector3.zero;
```

```
43              if (neighbors.Count == 0) return avg;
44
45              for (int i=0; i<neighbors.Count; i++) {
46                  avg += neighbors[i].pos;
47              }
48              avg /= neighbors.Count;
49
50              return avg;
51          }
52      }
53
54      public Vector3 avgVel {                                          // f
55          get {
56              Vector3 avg = Vector3.zero;
57              if (neighbors.Count == 0) return avg;
58
59              for (int i=0; i<neighbors.Count; i++) {
60                  avg += neighbors[i].rigid.velocity;
61              }
62              avg /= neighbors.Count;
63
64              return avg;
65          }
66      }
67
68      public Vector3 avgClosePos {                                     // g
69          get {
70              Vector3 avg = Vector3.zero;
71              Vector3 delta;
72              int nearCount = 0;
73              for (int i=0; i<neighbors.Count; i++) {
74                  delta = neighbors[i].pos - transform.position;
75                  if (delta.magnitude <= Spawner.S.collDist) {
76                      avg += neighbors[i].pos;
77                      nearCount++;
78                  }
79              }
80              // Если нет соседей, летящих слишком близко, вернуть Vector3.zero
81              if (nearCount == 0) return avg;
82
83              // Иначе координаты центральной точки
84              avg /= nearCount;
85              return avg;
86          }
87      }
88
89  }
```

a. Этот метод `Start()` создает экземпляр списка `List neighbors`, получает ссылку на коллайдер `SphereCollider` этого игрового объекта (напомню, что этим игровым объектом является экземпляр шаблона **Boid**, к которому также подключен компонент **SphereCollider**) и устанавливает радиус коллайдера **SphereCollider** равным половине значения поля `neighborDist` в объекте-одиночке `Spawner`. Половине, потому что `neighborDist` определяет расстояние, на котором игровые объекты

Boid должны видеть друг друга, и если радиус каждого коллайдера будет равен половине этого расстояния, они будут обнаруживать друг друга, едва соприкоснувшись, находясь точно на расстоянии `neighborDist`.

b. В каждом вызове `FixedUpdate()` класс `Neighborhood` проверяет изменение величины `neighborDist` и, если она изменилась, изменяет радиус в SphereCollider. Изменение радиуса в SphereCollider вызывает массивные вычисления в движке PhysX, поэтому мы присваиваем новое значение, только если это необходимо.

c. `OnTriggerEnter()` вызывается, когда что-то другое оказывается в зоне действия триггера этого коллайдера SphereCollider (триггер — это коллайдер, позволяющий другим объектам проходить через него). Экземпляры Boid должны быть единственными объектами, имеющими прикрепленные к ним коллайдеры, но для большей уверенности мы вызываем метод `GetComponent<Boid>()` коллайдера `other` и продолжаем, только если результат не равен `null`. Если обнаруженный поблизости экземпляр Boid еще не включен в список соседей `neighbors`, добавляем его.

d. Аналогично, когда другой экземпляр Boid выходит из соприкосновения с триггером этого экземпляра Boid, вызывается `OnTriggerExit()`, и мы удаляем отдалившийся экземпляр `Boid` из списка `neighbors`.

e. `avgPos` — свойство, доступное только для чтения, — просматривает все экземпляры Boid в списке `neighbors` и вычисляет координаты центральной точки в группе. Обратите внимание, как при этом используется общедоступное свойство `pos` каждого экземпляра Boid. Если список соседей пуст, возвращается `Vector3.zero`.

f. Аналогично, свойство `avgVel` возвращает среднюю скорость всех соседних экземпляров Boid.

g. `avgClosePos` — свойство, доступное только для чтения, — просматривает список `neighbors` и выявляет соседей, находящихся на расстоянии меньше `collDist` (хранится в объекте-одиночке `Spawner`) и находит координаты центральной точки для этой группы.

3. Сохраните сценарий Neighborhood и вернитесь в среду Unity, чтобы дать ей возможность скомпилировать сценарий и отобразить возможные ошибки.

Сценарий Boid — часть 3

Теперь, когда у нас появился компонент Neighborhood, подключенный к шаблону Boid, можно завершить реализацию класса `Boid`.

1. Откройте сценарий Boid в MonoDevelop и введите новые строки, выделенные жирным в следующем листинге. Когда будете вводить код, имейте в виду, что у вас номера строк могут не совпадать с моими. В этом нет ничего страшного, главное, чтобы сам код совпадал. Язык C# интерпретирует все *пробельные*

символы (пробелы, переводы строк, табуляции и т. д.) как одно и то же, поэтому лишние пустые строки здесь или там не имеют никакого значения. Для ясности я включил в книгу полный сценарий Boid.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Boid : MonoBehaviour {

    [Header("Set Dynamically")]
    public Rigidbody        rigid;

    private Neighborhood neighborhood;

    // Используйте этот метод для инициализации
    void Awake () {
        neighborhood = GetComponent<Neighborhood>();
        rigid = GetComponent<Rigidbody>();

        // Выбрать случайную начальную позицию
        pos = Random.insideUnitSphere * Spawner.S.spawnRadius;

        // Выбрать случайную начальную скорость
        Vector3 vel = Random.onUnitSphere * Spawner.S.velocity;
        rigid.velocity = vel;

        LookAhead();

        // Окрасить птицу в случайный цвет, но не слишком темный
        Color randColor = Color.black;
        while ( randColor.r + randColor.g + randColor.b < 1.0f ) {
            randColor = new Color(Random.value, Random.value, Random.value);
        }
        Renderer[] rends = gameObject.GetComponentsInChildren<Renderer>();
        foreach ( Renderer r in rends ) {
            r.material.color = randColor;
        }
        TrailRenderer trend = GetComponent<TrailRenderer>();
        trend.material.SetColor("_TintColor", randColor);
    }

    void LookAhead() {
        // Ориентировать птицу клювом в направлении полета
        transform.LookAt(pos + rigid.velocity);
    }

    public Vector3 pos {
        get { return transform.position; }
        set { transform.position = value; }
    }

    // FixedUpdate вызывается при каждом пересчете физики (50 раз в секунду)
```

```
50      void FixedUpdate () {
51          Vector3 vel = rigid.velocity;
52          Spawner spn = Spawner.S;
53
54          // ПРЕДОТВРАЩЕНИЕ СТОЛКНОВЕНИЙ - избегать близких соседей
55          Vector3 velAvoid = Vector3.zero;
56          Vector3 tooClosePos = neighborhood.avgClosePos;
57          // Если получен вектор Vector3.zero, ничего предпринимать не надо
58          if (tooClosePos != Vector3.zero) {
59              velAvoid = pos - tooClosePos;
60              velAvoid.Normalize();
61              velAvoid *= spn.velocity;
62          }
63
64          // СОГЛАСОВАНИЕ СКОРОСТИ - попробовать согласовать скорость с соседями
65          Vector3 velAlign = neighborhood.avgVel;
66          // Согласование требуется, только если velAlign не равно Vector3.zero
67          if (velAlign != Vector3.zero) {
68              // Нас интересует только направление, поэтому нормализуем скорость
69              velAlign.Normalize();
70              // и затем преобразуем в выбранную скорость
71              velAlign *= spn.velocity;
72          }
73
74          // КОНЦЕНТРАЦИЯ СОСЕДЕЙ - движение в сторону центра группы соседей
75          Vector3 velCenter = neighborhood.avgPos;
76          if (velCenter != Vector3.zero) {
77              velCenter -= transform.position;
78              velCenter.Normalize();
79              velCenter *= spn.velocity;
80          }
81
82          // ПРИТЯЖЕНИЕ – организовать движение в сторону объекта Attractor
83          Vector3 delta = Attractor.POS - pos;
84          // Проверить, куда двигаться, в сторону Attractor или от него
85          bool attracted = (delta.magnitude > spn.attractPushDist);
86          Vector3 velAttract = delta.normalized * spn.velocity;
87
88          // Применить все скорости
89          float fdt = Time.fixedDeltaTime;
90          if (velAvoid != Vector3.zero) {
91              vel = Vector3.Lerp(vel, velAvoid, spn.collAvoid*fdt);
92          } else {
93              if (velAlign != Vector3.zero) {
94                  vel = Vector3.Lerp(vel, velAlign, spn.velMatching*fdt);
95              }
96              if (velCenter != Vector3.zero) {
97                  vel = Vector3.Lerp(vel, velAlign, spn.flockCentering*fdt);
98              }
99              if (velAttract != Vector3.zero) {
100                 if (attracted) {
101                     vel = Vector3.Lerp(vel, velAttract, spn.attractPull*fdt);
102                 } else {
103                     vel = Vector3.Lerp(vel, -velAttract, spn.attractPush*fdt);
```

```
104                 }
105             }
106         }
107
108         // Установить vel в соответствии с velocity в объекте-одиночке Spawner
109         vel = vel.normalized * spn.velocity;
110         // В заключение присвоить скорость компоненту Rigidbody
111         rigid.velocity = vel;
112         // Повернуть птицу клювом в сторону нового направления движения
113         LookAhead();
114     }
115 }
```

2. Сохраните сценарий, вернитесь в Unity и щелкните на кнопке Play (Играть).

Теперь птицы должны проявлять более выраженное стайное поведение. Можете выбрать главную камеру Main Camera в иерархии и поиграть с разными значениями настроек для Boid в объекте-одиночке Spawner. В табл. 27.1 перечислены некоторые любопытные версии значений.

Таблица 27.1 Значения Boids

	По умолчанию	Разрозненная стая	Маленькие группы	Сплоченная стая
velocity	30	30	30	30
neighborDist	30	30	8	30
collDist	4	10	2	10
velMatching	0,25	0,25	0,25	10
flockCentering	0,2	0.2	8	0,2
collAvoid	2	4	10	4
attractPull	2	1	1	3
attractPush	2	2	20	2
attractPushDist	5	20	20	1

Итоги

В этой главе вы познакомились с идеей объектно-ориентированного программирования, которая будет эксплуатироваться в оставшейся части книги. Благодаря своей организации с использованием игровых объектов и компонентов, Unity прекрасно подходит для воспитания объектно-ориентированного мышления. Вы также познакомились с компонентно-ориентированным проектированием — шаблоном

проектирования программ, прекрасно работающим в Unity, и хотя концептуально такой подход существенно сложнее, он может помочь вам упростить код и сделать его более управляемым.

Наряду с идеей компонентно-ориентированного подхода в ООП также нашла развитие идея *модульности*. Во многих отношениях модульный код является полной противоположностью монолитному. Модульный подход фокусируется на создании маленьких, многократно используемых функций и классов, которые делают что-то одно, но очень хорошо. Поскольку модульные классы и функции очень малы (обычно меньше 500 строк), в них проще разбираться и их легче отлаживать. Модульный код также способствует многократному его использованию.

Далее начинается часть III книги, где будет представлена серия учебных примеров, которые познакомят вас с прототипами разных видов игр. Я надеюсь, что они помогут вам начать свой путь в качестве дизайнера, создателя прототипов и разработчика.

ЧАСТЬ III

Прототипы игр и примеры

Глава 28. Прототип 1: Apple Picker
Глава 29. Прототип 2: Mission Demolition
Глава 30. Прототип 3: SPACE SHMUP
Глава 31. Прототип 3.5: SPACE SHMUP PLUS
Глава 32. Прототип 4: PROSPECTOR SOLITAIRE
Глава 33. Прототип 5: BARTOK
Глава 34. Прототип 6: Word Game
Глава 35. Прототип 7: DUNGEON DELVER

28 Прототип 1: Apple Picker

Итак, сегодня вы создадите свой первый цифровой прототип игры.

Поскольку это ваш первый прототип, он достаточно прост. Но с каждой последующей главой прототипы становятся все сложнее и используют все больше возможностей Unity.

К концу этой главы у вас будет работающий прототип простой аркадной игры.

Цель создания цифрового прототипа

Пока мы не приступили к прототипу Apple Picker, вероятно, сейчас самый подходящий момент, чтобы подумать о целях создания цифровых прототипов. В части I книги проводилось широкое обсуждение бумажных прототипов и рассказывалось, в чем их польза. Бумажные прототипы помогают:

- тестировать, отвергать и/или быстро совершенствовать механику и правила игры;
- исследовать динамическое поведение игры и понять непредсказуемость, создаваемую правилами;
- определить, насколько понятны игрокам правила и элементы игрового процесса;
- понять эмоциональный отклик игроков в ответ на игру.

Цифровые прототипы также добавляют фантастическую возможность попробовать игру, фактически в этом их главная цель. Вы можете часами описывать игровую механику своему собеседнику, но гораздо эффективнее (и интереснее) просто дать ему сыграть в игру и увидеть ее вживую. Эта тема подробно обсуждается в книге Стива Свинка «Game Feel»[1].

В этой главе вы создадите действующую игру, и конечный результат вы вполне сможете показать друзьям и коллегам. Дав им поиграть, попросите оценить сложность игры: слишком простая, слишком сложная или такая, как надо. Воспользуйтесь их

[1] Steve Swink, *Game Feel: A Game Designer's Guide to Virtual Sensation* (Boston: Elsevier, 2009).

мнением, чтобы скорректировать значения переменных в игре и настроить уровень сложности для каждого из них.

А теперь приступим к созданию Apple Picker.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. Чтобы освежить в памяти стандартную процедуру настройки проекта, обращайтесь к приложению А «Стандартная процедура настройки проекта».

- **Имя проекта:** Apple Picker Prototype
- **Имя сцены:** _Scene_0
- **Имена сценариев на C#:** ApplePicker, Apple, AppleTree и Basket

Сценарии на C# не нужно ни к чему подключать.

Подготовка

К счастью, вы уже многое сделали при подготовке к этому прототипу в главе 16, «Цифровое мышление», когда мы анализировали Apple Picker и классическую игру *Kaboom!*. Как отмечалось в той главе, у Apple Picker та же игровая механика, что и у *Kaboom!*. Уделите немного времени, вернитесь к главе 16 и проверьте, полностью ли вы понимаете блок-схемы работы каждого элемента: Яблони (AppleTree), Яблока (Apple) и Корзины (Basket).

Когда вы будете выполнять инструкции по реализации этого проекта, я советую вооружиться карандашом и отмечать каждый шаг по его выполнении.

Начало: художественные ресурсы

Как прототип, эта игра не нуждается в фантастической графике, она просто должна работать. Искусство, которое вы будете творить в процессе чтения книги, известно как искусство программирования, искусство подготовки почвы для фактического игрового искусства, создаваемого художниками. Цель этого искусства, как и почти всего другого в прототипе, — как можно быстрее перейти от идеи к действующему прототипу. Хорошо, если вы уже владеете искусством программирования, хотя это совершенно необязательно.

AppleTree

Начнем с яблони.

1. В главном меню Unity выберите пункт GameObject > 3D Object > Cylinder (Игровой объект > 3D объект > Цилиндр). Этот игровой объект будет служить стволом

яблони. Переименуйте Cylinder в Trunk, выбрав его в иерархии и щелкнув на поле с именем объекта в верхней части панели Inspector (Инспектор). Установите параметры его компонента Transform, как показано на рис. 28.1.

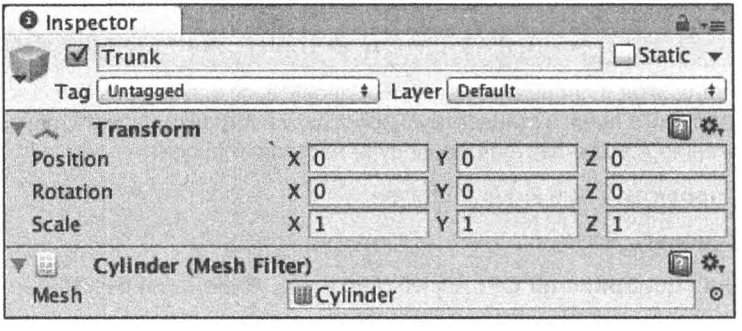


Рис. 28.1. Параметры компонента Transform объекта Cylinder с именем Trunk

В учебных примерах в этой книге я буду использовать следующую форму записи параметров компонентов Transform игровых объектов:

```
Trunk (Cylinder)     P:[ 0, 0, 0 ]     R:[ 0, 0, 0 ]     S:[ 1, 1, 1 ]
```

Предыдущая строка предлагает установить следующие параметры компонента Transform игрового объекта с именем Trunk: в разделе Position X=0, Y=0 и Z=0; в разделе Rotation X=0, Y=0 и Z=0; и в разделе Scale X=1, Y=1 и Z=1. Слово Cylinder в круглых скобках сообщает тип игрового объекта. Этот формат также будет встречаться вам в середине абзацев, например: P:[0, 0, 0] R:[0, 0, 0] S:[1, 1, 1].

2. Теперь выберите пункт меню GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера). Переименуйте объект Sphere в Leaves и настройте его компонент Ttransform, как показано ниже:

```
Leaves (Sphere) P:[ 0, 0.5, 0 ] R:[ 0, 0, 0 ] S:[ 3, 2, 3 ]
```

Вместе объекты Leaves и Trunk должны (немного) напоминать дерево, но сейчас это два отдельных объекта. Вы должны создать пустой игровой объект, который будет играть роль их родителя и объединять в один объект.

3. Выберите пункт меню GameObject > Create Empty (Игровой объект > Создать пустой). В результате будет создан пустой игровой объект. Настройте его компонент Ttransform, как показано ниже:

```
GameObject (Empty) P:[ 0, 0, 0 ] R:[ 0, 0, 0 ] S:[ 1, 1, 1 ]
```

Пустой игровой объект включает только компонент Transform и может служить прекрасным контейнером для других игровых объектов.

4. В панели Hierarchy (Иерархия) измените имя игрового объекта на AppleTree. То же самое можно сделать, щелкнув на имени игрового объекта, чтобы выделить его,

подождать секунду и либо нажать клавишу Return на клавиатуре (F2 в Windows), либо щелкнуть на имени второй раз.

5. Перетащите по отдельности игровые объекты Trunk и Leaves в AppleTree (вы увидите тот же указатель мыши в форме изогнутой стрелки, который появляется при подключении сценария на C# к игровому объекту (рис. 19.4)), чтобы в иерархии они оказались в списке подчиненных объектов AppleTree. Вы сможете увидеть их, щелкнув на пиктограмме с треугольником рядом со словом AppleTree. Теперь панель Hierarchy (Иерархия) и содержимое AppleTree должны выглядеть как на рис. 28.2.

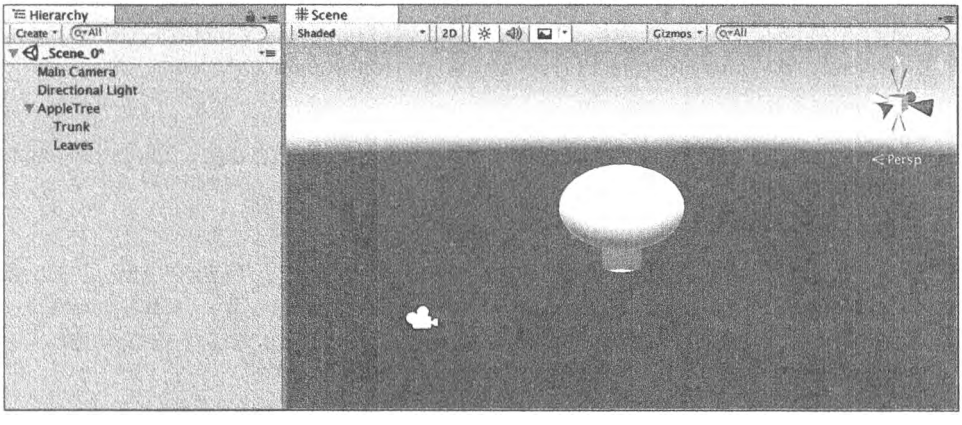


Рис. 28.2. Объект AppleTree в панелях Hierarchy (Иерархия) и Scene (Сцена) с подчиненными объектами Leaves и Trunk. Звездочка (*) справа от _Scene_0 в верхней части панели Hierarchy (Иерархия) указывает, что я еще не сохранил измененную сцену. Я должен это сделать!

Теперь, когда игровые объекты Trunk и Leaves подчинены AppleTree, если попробовать передвинуть, масштабировать или повернуть AppleTree, вместе с ним передвинутся, изменят масштаб или повернутся оба объекта — Trunk и Leaves. Попробуйте поэкспериментировать с параметрами компонента Transform объекта AppleTree.

6. Поиграв с настройками компонента Transform объекта AppleTree, установите их, как показано ниже:

```
AppleTree P:[ 0, 0, 0 ] R:[ 0, 0, 0 ] S:[ 2, 2, 2 ]
```

Согласно им, AppleTree переместится в центр сцены и увеличится в два раза по сравнению с начальными размерами.

7. Добавьте компонент твердого тела Rigidbody в AppleTree, выделив его в иерархии и выбрав в главном меню Unity пункт Component > Physics > Rigidbody (Компонент > Физика > Твердое тело).

8. В инспекторе для компонента Rigidbody объекта AppleTree снимите флажок Use Gravity. Если оставить его, яблоня будет падать вниз при проигрывании сцены.

Как рассказывалось в главе 20 «Переменные и компоненты», компонент Rigidbody гарантирует синхронное обновление коллайдеров объектов Trunk и Sphere при перемещении AppleTree в сцене.

Простые материалы для AppleTree

Хотя сейчас мы обсуждаем исключительно *искусство программирования*, это не значит, что все объекты должны быть исходного белого цвета. Давайте добавим немного красок в сцену.

1. Выберите пункт меню Assets > Create > Material (Ресурсы > Создать > Материал). В результате в панели Project (Проект) появится новый материал.
   a. Переименуйте его в Mat_Wood.
   b. Перетащите материал Mat_Wood на объект Trunk в сцене или в панели Hierarchy (Иерархия).
   c. Выберите Mat_Wood в панели Project (Проект).
   d. В инспекторе для Mat_Wood, в разделе Main Maps, выберите в поле Albedo коричневый цвет, какой вам понравится[1]. При желании можете также подвигать ползунки Metallic и Smoothness, позволяющие придать металлический вид и изменить гладкость материала соответственно[2].
2. Проделайте то же самое и создайте материал с именем Mat_Leaves.
   a. Перетащите материал Mat_Leaves на объект Leaves в сцене или в панели Hierarchy (Иерархия).
   b. Выберите в поле Albedo для материала Mat_Leaves зеленый цвет, напоминающий цвет листьев.
3. Перетащите AppleTree из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон. Как вы могли наблюдать в предыдущих главах, эта операция создаст шаблон AppleTree в панели Project (Проект) и окрасит в синий цвет имя AppleTree в иерархии.

---

[1] Для стандартного шейдера Unity (Unity Standard Shader) параметр Albedo определяет основной цвет. Чтобы узнать больше о стандартном шейдере, поищите по фразе «Standard Shader» в руководстве к Unity (http://docs.unity.com). Это довольно глубокая тема с несколькими подразделами в оглавлении в левой колонке онлайн-документации к Unity. Если вас интересует только параметр Albedo, поищите по фразе «Albedo Color and Transparency».

[2] Я выбрал значения Metallic=0 и Smoothness=0.25. Двигая эти ползунки, вы можете наблюдать их влияние на изображении Trunk в панели Scene (Сцена). Также эффект можно видеть в поле предварительного просмотра материала в нижней части инспектора. Если поле предварительного просмотра не видно, щелкните на темно-сером прямоугольнике Mat_Wood внизу инспектора.

4. По умолчанию сцены в Unity уже включают Directional Light (Направленное освещение). Установите координаты, поворот и масштаб элемента Directional Light в иерархии, как показано ниже:

```
Directional Light P:[ 0, 20, 0 ] R:[ 50, -30, 0 ] S:[ 1, 1, 1 ]
```

   В результате в сцене должно получиться приятное для глаз диагональное освещение. Здесь стоит отметить, что координаты направленного освещения не играют важной роли — свет будет распространяться в том же направлении, независимо от местоположения, но я задал позицию [0, 20, 0], чтобы убрать *значок* (gizmo), изображающий освещение, из середины сцены. Поэкспериментировав с поворотом Directional Light, вы увидите, что первый источник направленного освещения соответствует положению солнца на небе Unity. Мы не будем изменять поворот в Apple Picker, но вообще с его помощью можно создавать отличные эффекты в трехмерных играх.

5. Чтобы сместить AppleTree вверх, выберите объект AppleTree в иерархии и измените его координаты на P:[0, 10, 0]. При этом объект может исчезнуть из панели Scene (Сцена), выйдя за пределы видимой части, но вы можете изменить масштаб сцены, чтобы увидеть яблоню, покрутив колесико мыши.

Apple

Закончив создание яблони AppleTree, можно переходить к созданию шаблона яблок, которые будут падать с яблони.

1. Выберите пункт меню GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера). Переименуйте созданный объект в Apple и настройте его компонент Transform, как показано ниже:

```
Apple (Sphere) P:[ 0, 0, 0 ] R:[ 0, 0, 0 ] S:[ 1, 1, 1 ]
```

2. Создайте новый материал с именем Mat_Apple и выберите в поле Albedo красный цвет (или светло-зеленый, если вам больше нравятся зеленые яблоки).

3. Перетащите Mat_Apple на Apple в иерархии.

Добавление физических характеристик яблока

Как рассказывалось в главе 17 «Введение в среду разработки Unity», компонент Rigidbody позволяет моделировать поведение объектов в соответствии с законами физики (например, падать или сталкиваться с другими объектами).

1. Выберите Apple в панели Hierarchy (Иерархия). В главном меню Unity выберите пункт Component > Physics > Rigidbody (Компонент > Физика > Твердое тело).

2. Щелкните на кнопке Play (Играть), и Apple упадет вниз, подчиняясь закону тяготения.

3. Щелкните на кнопке **Play** (Играть) еще раз, чтобы остановить воспроизведение, и **Apple** вернется в начальную позицию.

Добавление тега «Apple» в объект Apple

Позднее вам понадобится получить массив всех игровых объектов **Apple** на экране, а чтобы упростить эту задачу, снабдим их специальным тегом.

1. Выберите **Apple** в иерархии, щелкните на кнопке в инспекторе рядом с надписью **Tag** (на которой в данный момент отображается надпись **Untagged** (Без тега)) и в открывшемся списке выберите **Add Tag** (Добавить тег), как показано в части A на рис. 28.3. В результате откроется диспетчер тегов и слоев **Tags & Layers**.
2. Возможно, вам понадобится щелкнуть на пиктограмме с треугольником рядом с надписью **Tags**, чтобы увидеть диспетчер, как показано в части B на рис. 28.3. Щелкните на значке **+**, чтобы добавить новый тег.
3. Введите в поле **New Tag Name** (Имя нового тега) текст **Apple** (C) и щелкните на кнопке **Save** (Сохранить). В списке **Tags** (Теги) появится элемент **Apple** (D).

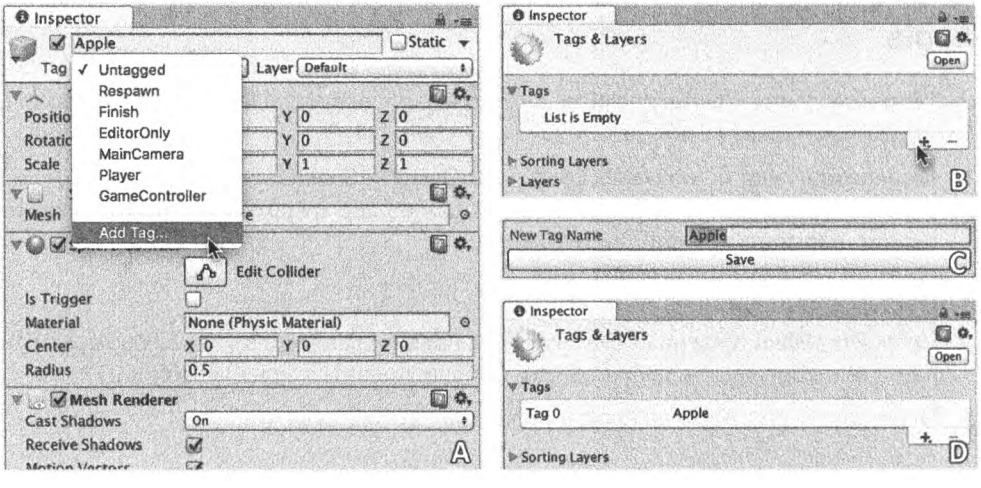


Рис. 28.3. Шаги 1, 2 и 3 добавления тега Apple в список тегов

4. Щелкните на объекте **Apple** в иерархии, чтобы вернуть отображение настроек этого объекта в инспекторе.
5. Щелкните на кнопке рядом с надписью **Tag** еще раз; теперь в списке вы увидите тег **Apple**. Выберите пункт **Apple** в списке тегов. Теперь все игровые объекты **Apple** будут автоматически снабжаться тегом **Apple**, что позволит быстро выявлять и выбирать их.

Преобразование Apple в шаблон

Выполните следующие шаги, чтобы преобразовать игровой объект Apple в шаблон:

1. Перетащите Apple из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон[1].
2. После создания шаблона Apple в панели Project (Проект) щелкните на экземпляре Apple в панели Hierarchy (Иерархия) и удалите его (выбрав пункт Delete (Удалить) в контекстном меню — щелчком правой кнопки мыши — или нажав комбинацию клавиш Command-Delete (просто Delete в Windows) на клавиатуре). Поскольку яблоки в игре будут создаваться из шаблона Apple в панели Project (Проект), в самом начале в сцене не должно присутствовать никаких яблок.

Basket

Искусство программирования корзины, как и других объектов, не содержит ничего сложного.

1. Выберите в главном меню Unity пункт GameObject > 3D Object > Cube (Игровой объект > 3D объект > Куб). Переименуйте в Basket и настройте его компонент Ttransform, как показано ниже:

   ```
   Basket (Cube) P:[ 0, 0, 0 ] R:[ 0, 0, 0 ] S:[ 4, 1, 4 ]
   ```

   В результате должен получиться плоский широкий прямоугольник.
2. Создайте новый материал с именем Mat_Basket, окрасьте его в ненасыщенный желтый цвет (например, соломенный) и примените материал к корзине.
3. Добавьте в Basket компонент Rigidbody. Выделите Basket в иерархии и выберите в главном меню Unity пункт Component > Physics > Rigidbody (Компонент > Физика > Твердое тело).
   a. В инспекторе для компонента Rigidbody объекта Basket снимите флажок Use Gravity.
   b. Там же установите флажок Is Kinematic.
4. Перетащите Basket из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон, и удалите оставшийся экземпляр Basket из иерархии (как мы проделали это с экземпляром Apple).
5. Сохраните сцену.

Теперь панели Hierarchy (Иерархия) и Project (Проект) должны выглядеть как на рис. 28.4.

[1] Более подробное обсуждение шаблонов вы найдете в главе 19 «Hello World: ваша первая программа».

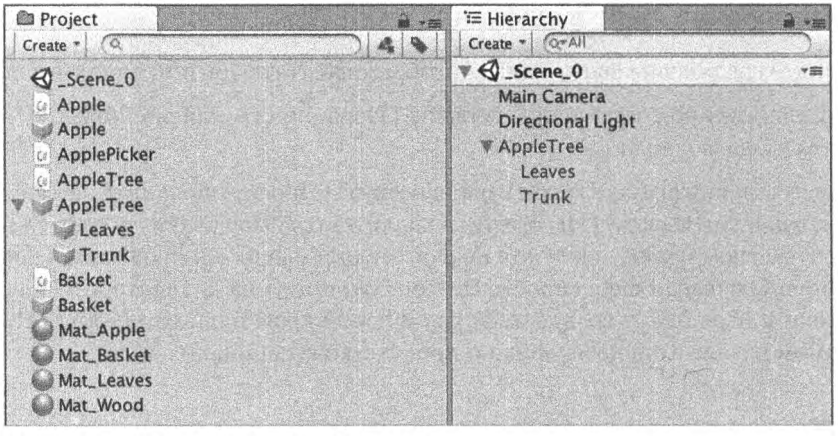


Рис. 28.4. Панели Hierarchy (Иерархия) и Project (Проект) в данный момент разработки прототипа. Сценарии Apple, ApplePicker, AppleTree и Basket вы должны были создать на этапе настройки проекта, в начале главы

Настройка камеры

Одним из важнейших аспектов любой игры является выбор правильного местоположения камеры. В Apple Picker нам нужно, чтобы камера отображала игровое поле приличного размера. Поскольку игра по сути является двумерной, также нужно настроить камеру, чтобы она давала ортографическую проекцию вместо перспективной.

ОРТОГРАФИЧЕСКАЯ И ПЕРСПЕКТИВНАЯ ПРОЕКЦИИ

Ортографическая и *перспективная* — это названия двух проекций, отображаемых в играх виртуальными трехмерными камерами (рис. 28.5).

Камера с *перспективной* проекцией действует как человеческий глаз: после прохождения света через линзы объекты, находящиеся ближе к камере, выглядят больше, а объекты, находящиеся дальше, выглядят меньше. Соответственно, *поле зрения* камеры (проекция) приобретает форму усеченного конуса (или, точнее, квадратной усеченной пирамиды). Чтобы убедиться в этом, щелкните на главной камере Main Camera в иерархии и затем уменьшите масштаб в панели Scene (Сцена). Появится «проволочный» каркас *в форме усеченной пирамиды*, простирающийся от камеры и изображающий поле зрения камеры.

Для камеры с *ортогональной* проекцией размеры объектов не зависят от удаленности. Ортогональное поле зрения имеет не пирамидальную, а прямоугольную форму. Чтобы убедиться в этом, выберите главную камеру Main Camera в панели Hierarchy (Иерархия). Найдите компонент Camera в инспекторе и измените перспективную проекцию (Perspective) на ортогональную (Orthogonal). Теперь каркас поля зрения будет иметь форму трехмерного прямоугольника, а не пирамиды.

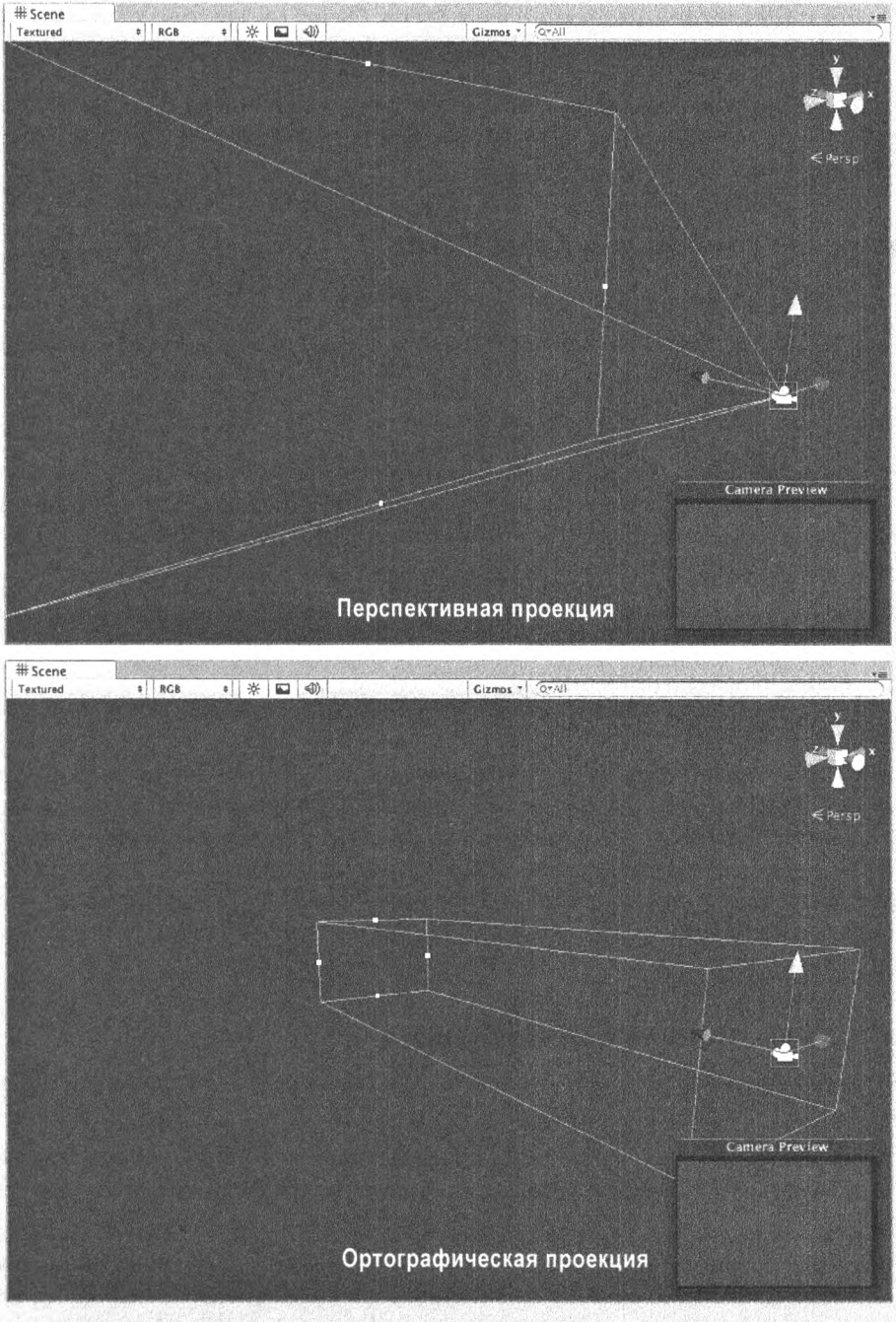


Рис. 28.5. Сравнение перспективной и ортографической проекций

Также иногда полезно настроить панель Scene (Сцена) как ортогональную вместо перспективной. Для этого щелкните на слове <Persp под значком с изображением осей в правом верхнем углу сцены (внимательно рассмотрите изображения на рис. 28.5), чтобы переключить перспективное отображение на изометрическое (обозначается надписью =Iso, «изометрическая» — еще одно название ортографической проекции).

Настройка камеры для Apple Picker

Теперь настроим камеру для прототипа Apple Picker:

1. Выберите Main Camera в панели Hierarchy (Иерархия) и настройте компонент Ttransform, как показано ниже:

```
Main Camera (Camera) P:[ 0, 0, -10 ] R:[ 0, 0, 0 ] S:[ 1, 1, 1 ]
```

   В результате точка зрения камеры сместится вниз на 1 метр (одна единица в системе координат Unity равна 1 метру), на высоту 0. Так как в системе координат Unity единицей измерения является метр, в этой книге я иногда буду сокращать «1 единица в системе координат Unity» до «1 метр».

2. В инспекторе для компонента Camera установите следующие значения (как показано на рис. 28.6):

   a. В поле Projection выберите значение Orthographic.

   b. В поле Size установите значение 16.

В результате AppleTree получит оптимальные размеры в панели Game (Игра), и останется достаточно места для падения яблок, которые игрок должен ловить в корзины. Часто достаточно установить примерные настройки камеры и корректировать их потом, когда появится возможность поиграть в игру. Как и все остальное в разработке игр, поиск оптимальных настроек камеры — итеративный процесс. Окончательные настройки главной камеры Main Camera в инспекторе должны выглядеть как на рис. 28.6.

Настройка панели Game (Игра)

Еще одним важным фактором игры является соотношение сторон панели Game (Игра).

1. Вверху панели Game (Игра) имеется раскрывающийся список, в данный момент отображающий Free Aspect. Это список выбора соотношения сторон.

2. Щелкните на списке и выберите пункт 16:9. Это стандартное соотношение сторон широкоформатных телевизионных экранов и компьютерных мониторов, поэтому данный выбор даст оптимальное изображение, если запустить игру в полноэкранном режиме. Также снимите флажок Low Resolution Aspect Ratios, если вы работаете в macOS.

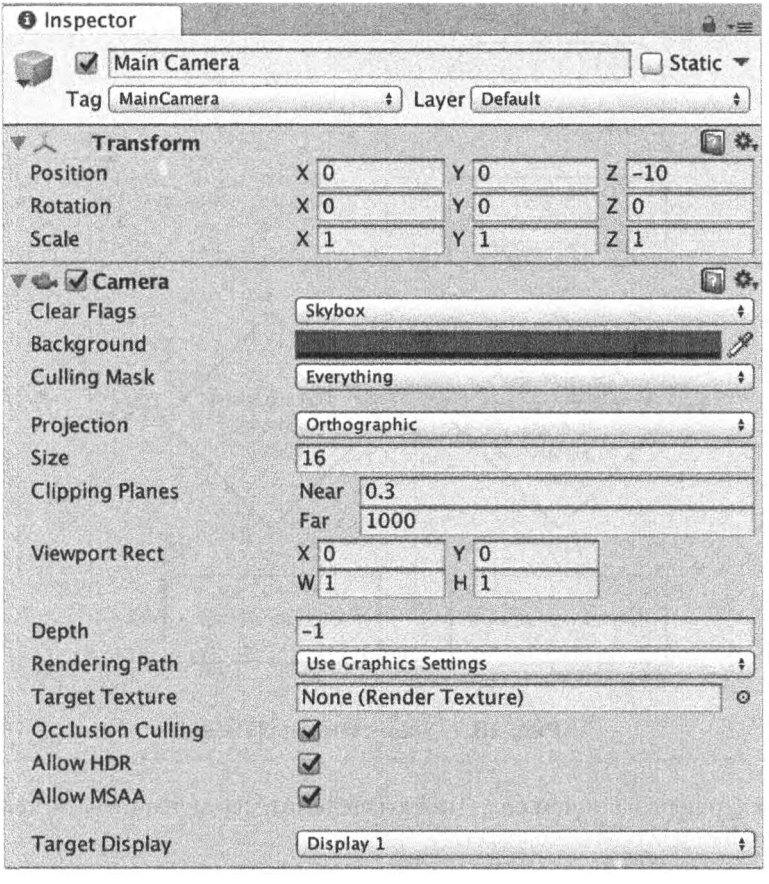


Рис. 28.6. Настройки Main Camera в инспекторе

Программирование прототипа Apple Picker

Теперь займемся реализацией программного кода, который будет приводить в движение прототип этой игры. На рис. 28.7 изображена блок-схема работы AppleTree из главы 16 «Цифровое мышление».

Код, реализующий поведение AppleTree, должен выполнять следующие действия:

- В каждом кадре перемещать яблоню с заданной скоростью.
- Изменять направление движения по достижении края игрового поля.
- Случайно изменять направление движения, исходя из заданной вероятности.
- Сбрасывать яблоко каждую секунду.

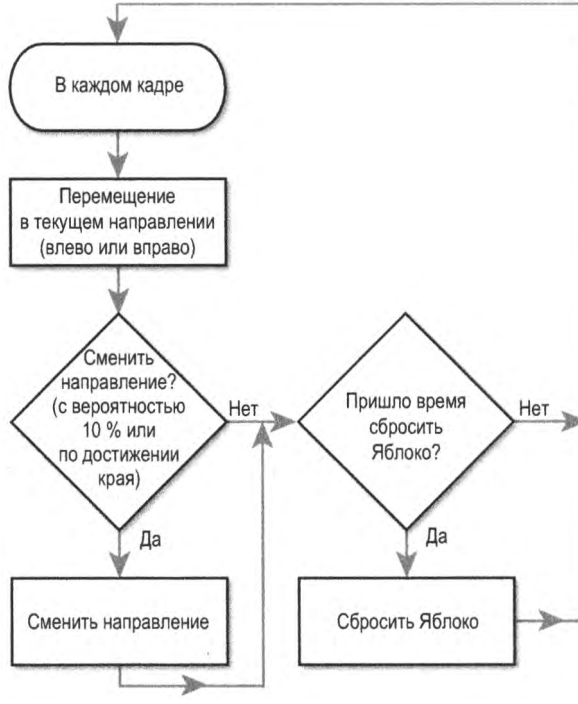


Рис. 28.7. Блок-схема AppleTree

Вот и все! А теперь напишем код. Дважды щелкните на сценарии AppleTree в панели Project (Проект), чтобы открыть его.

1. Нам понадобятся некоторые переменные для хранения настроек; чтобы добавить их, откройте класс AppleTree в MonoDevelop и введите следующий код. Код, который нужно фактически ввести, выделен жирным.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AppleTree : MonoBehaviour {
    [Header("Set in Inspector")]
    // Шаблон для создания яблок
    public GameObject   applePrefab;

    // Скорость движения яблони
    public float        speed = 1f;

    // Расстояние, на котором должно изменяться направление движения яблони
    public float        leftAndRightEdge = 10f;

    // Вероятность случайного изменения направления движения
```

```
    public float        chanceToChangeDirections = 0.1f;

    // Частота создания экземпляров яблок
    public float        secondsBetweenAppleDrops = 1f;

    void Start () {
        // Сбрасывать яблоки раз в секунду
    }

    void Update () {
        // Простое перемещение
        // Изменение направления
    }
}
```

Возможно, вы заметили, что в предыдущем листинге нет номеров строк, которые присутствовали в листингах в предыдущих главах. Листинги в этой части в основном будут приводиться без номеров строк, потому что они могут отличаться у вас и у меня из-за разного числа переводов строк в длинных листингах[1]. Сохраните сценарий AppleTree в MonoDevelop и вернитесь в Unity.

2. Чтобы этот код действительно что-то делал, его нужно подключить к игровому объекту AppleTree.
   a. Перетащите сценарий AppleTree из панели Project (Проект) на шаблон AppleTree в той же панели Project (Проект).
   b. Щелкните на экземпляре AppleTree в панели Hierarchy (Иерархия); в результате сценарий добавится не только в шаблон AppleTree, но и во все его экземпляры.
   c. Выберите AppleTree в иерархии; в инспекторе, в разделе с компонентом AppleTree (Script), должны появиться все только что объявленные переменные.
3. Попробуйте переместить AppleTree в сцене, изменяя координаты X и Y в компоненте Transform в инспекторе, чтобы определить наилучшее местоположение AppleTree по высоте (position.y) и уточнить левую и правую границы для перемещения. У меня наилучшими оказались значение 12 для position.y и значения −20 и 20 для position.x левой и правой границ, находясь в которых яблоня все еще хорошо видна в панели Game (Игра).
   a. Установите координаты AppleTree в P:[0, 12, 0].
   b. В переменную `leftAndRightEdge` компонента AppleTree (Script) в инспекторе введите значение 20.

[1] Еще одна причина, почему я отказался от номеров строк, состоит в том, что теперь мне нужен каждый символ, чтобы уместить длинные строки по ширине книжной страницы.

РУКОВОДСТВО ПО РАЗРАБОТКЕ СЦЕНАРИЕВ ДЛЯ ДВИЖКА UNITY

Прежде чем глубоко погрузиться в этот проект, очень важно, чтобы вы заглянули в руководство по разработке сценариев для движка Unity, если у вас есть вопросы относительно кода, представленного здесь. Есть два способа добраться до руководства:

1. Выберите в главном меню Unity пункт Help > Scripting Reference (Справка > Руководство по разработке сценариев). В результате запустится веб-браузер и откроет страницу руководства, хранящуюся локально на вашей машине, то есть вы сможете читать руководство, даже не имея подключения к интернету. Вы можете ввести имя любого класса или функции в поле поиска слева и найти дополнительную информацию о нем (о ней).

   Введите *MonoBehaviour* в поле поиска на странице с руководством и нажмите Return. Щелкните на верхнем результате, чтобы увидеть все методы, доступные в классе `MonoBehaviour` (и во всех сценариях, наследующих его, которые вы пишете и подключаете к игровым объектам в Unity).

2. Находясь в MonoDevelop, выделите текст, о котором вы хотели бы узнать подробнее, и выберите в меню пункт Help > Unity API Reference (Справка > Справочник Unity API). В этом случае откроется онлайн-версия руководства, поэтому этот прием не будет работать без подключения к интернету, но в конечном итоге вы получите ту же информацию, что имеется в локальном руководстве, которое открывается первым способом.

При первом посещении страницы руководства вам может быть предложено выбрать язык — C# или JS, щелкнув на одной из двух кнопок справа вверху в окне. Щелкните на кнопке с надписью C#, чтобы выбрать язык C# как предпочтительный. Большинство примеров кода доступно на обоих языках — C# и JavaScript, хотя некоторые старые примеры могут быть написаны только на JavaScript.

Простое перемещение

Теперь добавим перемещение яблони:

1. Добавьте в метод `Update()` сценария AppleTree строки из следующего листинга, выделенные жирным. Обратите внимание на многоточия в листинге (`...`). Они указывают, что я пропустил некоторые строки в этом листинге для экономии места. Не удаляйте эти строки!

```
public class AppleTree : MonoBehaviour {
    ...                                                          // a
    void Update () {
        // Простое перемещение
        Vector3 pos = transform.position;                        // b
        pos.x += speed * Time.deltaTime;                         // c
        transform.position = pos;                                // d

        // Изменение направления
    }
}
```

Комментарии // справа служат ссылками на следующую дополнительную информацию.

a. Во всех главах с учебными примерами я использую в листингах многоточия (...) взамен фрагментов кода, которые я пропустил. Без этого в некоторых последующих главах листинги оказались бы чересчур длинными. Увидев многоточия, как в этом примере, не изменяйте код, который под ними скрывается; просто оставьте его как есть и сосредоточьте внимание на новом коде (выделенном жирным). В данном случае не требуется ничего менять в строках между объявлением класса `AppleTree` и методом `Update()`, поэтому я заменил многоточием строки, оставшиеся без изменения.

b. Эта строка определяет локальную переменную `Vector3 pos` для хранения текущей позиции яблони.

c. Компонент `x` переменной `pos` увеличивается на произведение скорости `speed` и интервала времени `Time.deltaTime` (количество секунд, прошедших после отображения предыдущего кадра). Благодаря этому яблоня будет перемещаться *с учетом реального времени*, что очень важно при программировании игр (см. врезку «Привязка игр к реальному времени»).

d. Измененное значение `pos` присваивается обратно свойству `transform.position` (что вызывает перемещение яблони `AppleTree` в новое местоположение). Если не выполнить присваивание `pos` свойству `transform.position`, яблоня не переместится.

Возможно, вам интересно, почему это изменение запрограммировано в трех строках вместо одной. Почему нельзя написать код, как показано ниже?

```
transform.position.x += speed * Time.deltaTime;
```

Дело в том, что `transform.position` — это свойство, метод, маскирующийся под поле (то есть функция, маскирующаяся под переменную) за счет использования методов доступа `get{}` и `set{}` (о которых рассказывалось в главе 26 «Классы»). Прочитать значение компонента свойства можно, но выполнить запись в него нельзя. Проще говоря, прочитать значение `transform.position.x` можно, но записать новое значение непосредственно в `transform.position.x` не получится. По этой причине пришлось создать промежуточную переменную `Vector3 pos`, изменить ее и записать обратно в `transform.position`.

2. Сохраните сценарий, вернитесь в Unity и щелкните на кнопке **Play** (Играть). Теперь вы должны увидеть, как яблоня `AppleTree` очень медленно перемещается по экрану. Попробуйте присвоить переменной `speed` разные значения в инспекторе и подберите такое, которое покажется вам лучшим. Мне лично больше понравилось значение 10 для `speed`, соответствующее скорости перемещения 10 м/с (10 метров в секунду, или 10 единиц в системе координат Unity в секунду). Остановите воспроизведение игры в Unity и задайте в переменной `speed` значение 10 в инспекторе.

ПРИВЯЗКА ИГР К РЕАЛЬНОМУ ВРЕМЕНИ

Когда перемещения объектов в игре привязаны к течению реального времени, их скорость не зависит от частоты кадров. Реализовать такую привязку можно с помощью переменной `Time.deltaTime`, которая сообщает время в секундах, прошедшее с момента воспроизведения предыдущего кадра. `Time.deltaTime` обычно имеет очень маленькое значение. Для игры, действующей со скоростью 25 кадров в секунду, `Time.deltaTime` возвращает значение 0,04f, то есть на воспроизведение каждого кадра затрачивается 4/100 секунды. Если строка `// b` в сценарии выше будет выполняться с частотой 25 кадров в секунду, результат будет выглядеть примерно так:

```
pos.x += speed * Time.deltaTime;
pos.x += 1.0f * 0.04f;
pos.x += 0.04f;
```

То есть за 1/25 секунды величина `pos.x` увеличится на 0,04 м. За одну полную секунду значение `pos.x` увеличится на 0,04 м * 25 кадров, то есть на 1 метр.

Если игра будет выполняться с частотой 100 кадров в секунду, тогда:

```
pos.x += speed * Time.deltaTime;
pos.x += 1.0f * 0.01f;
pos.x += 0.01f;
```

То есть за 1/100 секунды `pos.x` увеличится на 0,01. За одну полную секунду значение `pos.x` увеличится на 0,01 м × 100 кадров, или на 1 метр.

Привязка к реальному времени гарантирует перемещение элементов в игре с одинаковой скоростью при любой частоте кадров, благодаря чему ваши игры будут доставлять удовольствие и тем, кто играет на самом новом аппаратном обеспечении, и тем, кто пользуется старыми машинами. Программирование с привязкой к реальному времени также очень важно при создании игр для мобильных устройств, потому что вычислительная мощность и быстродействие мобильных устройств изменяются в очень широких пределах.

Изменение направления

Теперь, когда яблоня **AppleTree** перемещается с приличной скоростью, она быстро достигнет края экрана и исчезнет за ним. Давайте организуем изменение направления движения, когда яблоня отдаляется от центра на расстояние `leftAndRightEdge`. Измените сценарий **AppleTree**, как показано ниже:

```
public class AppleTree : MonoBehaviour {
    ...
    void Update () {
        // Простое перемещение
        ...
        // Изменение направления
        if ( pos.x < -leftAndRightEdge ) {                              // a
            speed = Mathf.Abs(speed);           // Начать движение вправо // b
```

```
        } else if ( pos.x > leftAndRightEdge ) {                          // c
            speed = -Mathf.Abs(speed);          // Начать движение влево  // c
        }
    }
}
```

a. Проверить, не оказалось ли значение `pos.x`, только что установленное в предыдущих строках, меньше отрицательного значения предела `leftAndRightEdge`.

b. Если величина `pos.x` оказалась слишком маленькой, переменной `speed` присваивается результат вызова `Mathf.Abs(speed)`, который возвращает абсолютное положительное значение `speed` и тем самым гарантирует, что в следующем кадре начнется перемещение яблони вправо.

c. Если величина `pos.x` оказалась больше `leftAndRightEdge`, переменной `speed` присваивается отрицательное значение результата вызова `Mathf.Abs(speed)`, благодаря чему в следующем кадре AppleTree начнет движение влево.

Сохраните сценарий, вернитесь в Unity и щелкните на кнопке Play (Играть), чтобы увидеть, что из этого получилось.

Случайное изменение направления с заданной вероятностью

Чтобы добавить случайное изменение направления перемещения яблони, выполните следующие шаги:

1. Добавьте в сценарий строки, выделенные жирным в следующем листинге:

```
public class AppleTree : MonoBehaviour {
    ...
    void Update () {
        // Простое перемещение
        ...
        // Изменение направления
        if ( pos.x < -leftAndRightEdge ) {
            speed = Mathf.Abs(speed);           // Начать движение вправо
        } else if ( pos.x > leftAndRightEdge ) {
            speed = -Mathf.Abs(speed);          // Начать движение влево
        } else if ( Random.value < chanceToChangeDirections ) {          // a
            speed *= -1; // Change direction                             // b
        }
    }
}
```

a. `Random.value` возвращает случайное число типа `float` между 0 и 1 (включая 0 и 1 как возможные значения). Если случайное значение меньше `chanceToChangeDirections`...

b. ... изменить направление движения AppleTree можно, поменяв знак переменной `speed` на противоположный (например, с 1 на −1).

2. Щелкнув на кнопке Play (Играть), вы увидите, что для значения по умолчанию 0,1 в переменной `chanceToChangeDirections` направление меняется слишком часто. Измените в инспекторе значение `chanceToChangeDirections` на 0,02, и теперь ситуация должна выглядеть намного лучше.

   В продолжение дискуссии во врезке «Привязка игр к реальному времени» вероятность смены направления фактически не привязана к реальному времени. В каждом кадре есть вероятность 2 %, что AppleTree изменит направление движения. На быстром компьютере эта вероятность может проверяться до 400 раз в секунду (и в среднем вызывать смену направления 8 раз в секунду), тогда как на медленном компьютере вероятность будет проверяться лишь 30 раз в секунду (и в среднем вызывать смену направления 0,6 раза в секунду).

3. Чтобы исправить этот недостаток, перенесем код, управляющий сменой направления из `Update()` (который вызывается со скоростью отображения кадров) в `FixedUpdate()` (который вызывается точно 50 раз в секунду, независимо от быстродействия компьютера).

```
public class AppleTree : MonoBehaviour {
    ...
    void Update () {
        // Простое перемещение
        ...
        // Изменение направления
        if ( pos.x < -leftAndRightEdge ) {
            speed = Mathf.Abs(speed);            // Начать движение вправо
        } else if ( pos.x > leftAndRightEdge ) {
            speed = -Mathf.Abs(speed);           // Начать движение влево
        }                                                                  // a
    }

    void FixedUpdate() {
        // Теперь случайная смена направления привязана ко времени,
        // потому что выполняется в FixedUpdate()
        if ( Random.value < chanceToChangeDirections ) {                   // b
            speed *= -1; // Change direction
        }
    }
}
```

   a. Две строки с комментариями `// a` и `// b`, добавленные в сценарий на шаге 1, были вырезаны и заменены закрывающей фигурной скобкой...

   b. ...и вставлены сюда.

Теперь AppleTree будет случайно менять направление в среднем 1 раз в секунду (50 вызовов `FixedUpdate` в секунду × вероятность 0,02 = в среднем 1 раз в секунду).

Сбрасывание яблок

Пришел черед сбрасывать яблоки:

1. Выберите AppleTree в иерархии и посмотрите на параметры компонента Apple Tree (Script) в инспекторе. В настоящее время поле `applePrefab` имеет значение

None (Game Object), то есть оно еще не установлено (слово GameObject в круглых скобках указывает, что поле `applePrefab` имеет тип `GameObject`). Этому полю нужно присвоить шаблон Apple из панели Project (Проект). Сделать это можно любым из двух способов:

- щелкнуть на маленьком изображении мишени справа от Apple Prefab None (Game Object) в инспекторе и в открывшемся окне на вкладке Assets (Ресурсы) выбрать Apple

  или

- перетащить шаблон Apple из панели Project (Проект) на поле `applePrefab` в панели Inspector (Инспектор). Графически этот процесс изображен на рис. 19.4 в главе 19 «Hello World: ваша первая программа».

2. Вернитесь в MonoDevelop и добавьте код из следующего листинга, выделенный жирным, в класс `AppleTree`:

```
public class AppleTree : MonoBehaviour {
    ...
    void Start () {
        // Сбрасывать яблоки раз в секунду
        Invoke( "DropApple", 2f );                                    // a
    }

    void DropApple() {                                                // b
        GameObject apple = Instantiate<GameObject>( applePrefab );    // c
        apple.transform.position = transform.position;                // d
        Invoke( "DropApple", secondsBetweenAppleDrops );              // e
    }
    void Update () { ... }                                            // f
    ...
}
```

a. Функция `Invoke()` вызывает функцию, заданную именем, через указанное число секунд. В данном случае вызывается новая функция `DropApple()`. Второй параметр, `2f`, сообщает методу `Invoke()`, что тот должен подождать 2 секунды перед вызовом `DropApple()`.

b. `DropApple()` — это наша функция. Она создает экземпляр Apple в точке, где находится AppleTree.

c. `DropApple()` создает экземпляр `applePrefab` и присваивает его переменной `apple` типа `GameObject`.

d. Местоположение этого нового игрового объекта `apple` устанавливается равным местоположению яблони AppleTree.

e. Снова вызывается `Invoke()`. На этот раз он вызовет функцию `DropApple()` через `secondsBetweenAppleDrops` секунд (в данном случае через 1 секунду, согласно настройкам по умолчанию в инспекторе). Так как при каждом вызове `DropApple()` планирует следующий вызов самой себя, каждую секунду будет сбрасываться новое яблоко.

f. Фигурные скобки с многоточием `{ ... }` в этой строке показывают, что я опустил тело метода `Update()`. Вы ничего не должны менять в методе `Update()`, увидев многоточие, как здесь.

3. Сохраните сценарий **AppleTree**, вернитесь в Unity, щелкните на кнопке **Play** (Играть) и посмотрите, что получится.

   Ожидали ли вы, что яблоки и яблоня будут отлетать друг от друга, как при столкновении? Сейчас происходит то же самое, как в примере «Hello World», в главе 19, где кубики летали по всему игровому пространству. Я нарочно подвел вас к этой проблеме, чтобы показать, как ее исправить, если вам придется столкнуться с ней в своих играх. Прежде всего нужно сделать твердое тело **Rigidbody** в **AppleTree** *кинематическим* — оно будет доступно для управления из программного кода, но не будет реагировать на столкновения с другими объектами.

4. В компоненте **Rigidbody** игрового объекта **AppleTree** установите флажок **Is Kinematic** в инспекторе.

5. Снова щелкните на кнопке **Play** (Играть), и вы увидите, что проблема с яблоней исчезла, но с яблоками осталась.

Проблема с самой яблоней **AppleTree** была исправлена, но яблоки продолжают сталкиваться с яблоней и отскакивать от нее влево, вправо или вниз с более высокой скоростью, чем при воздействии одной только силы тяжести. Чтобы исправить эту проблему, вам нужно поместить яблоки в *физический слой*, где они не будут сталкиваться с яблоней. Физические слои — это группы объектов, которые могут сталкиваться между собой или игнорировать друг друга. Если игровые объекты **AppleTree** и **Apple** поместить в два разных физических слоя и для этих физических слоев настроить игнорирование друг друга, тогда объекты **AppleTree** и **Apple** перестанут сталкиваться друг с другом.

Настройка физических слоев

Сначала создадим несколько новых физических слоев. Необходимые шаги иллюстрирует рис. 28.8.

1. Щелкните на **AppleTree** в иерархии и в раскрывающемся списке **Layer**, в инспекторе, выберите пункт **Add Layer...** (Добавить слой...). В результате откроется диспетчер тегов и слоев **Tags & Layers**, где в разделе **Layers** (Слои) можно настроить имена физических слоев (убедитесь, что делаете это не в разделе **Tags** (Теги) или **Sorting Layers** (Слои сортировки)). Обратите внимание, что слои **Builtin Layers** с **0** по **7** отображаются серым цветом и неактивны, но можно править слои **User Layer** с **8** по **31**.

2. Дайте слою 8 имя **AppleTree**, слою 9 — имя **Apple** и слою 10 — имя **Basket**.

3. В главном меню Unity выберите пункт **Edit > Project Settings > Physics** (Правка > Параметры проекта > Физика). В результате в инспекторе откроется диспетчер настроек физического движка **Physics Manager** (рис. 28.9). В матрице столкно-

Рис. 28.8. Процедура создания новых физических слоев (шаги 1 и 2) и их привязки (шаг 5)

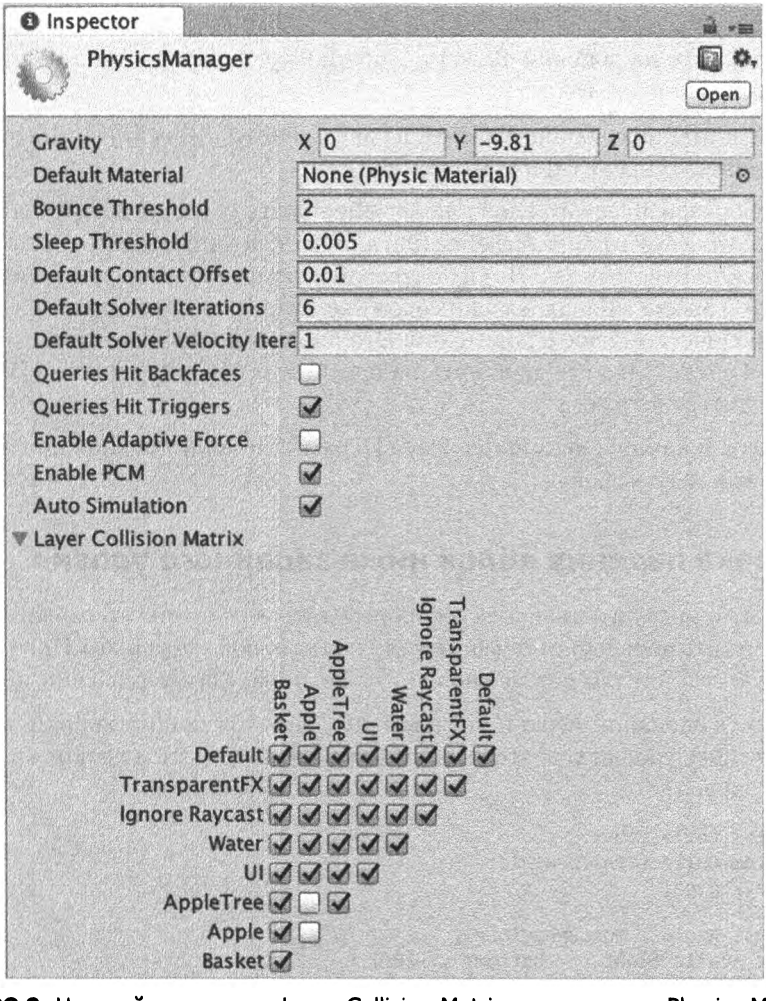


Рис. 28.9. Настройки матрицы Layer Collision Matrix в диспетчере Physics Manager

вений слоев Layer Collision Matrix, состоящей из флажков, которая находится внизу раздела Physics Manager, установите флажки на пересечении физических слоев, игровые объекты в которых будут поддерживать столкновения друг с другом (в том числе и объекты, находящиеся в одном физическом слое).

4. Экземпляры Apple должны обнаруживать столкновения с объектами Basket, но не с объектом AppleTree и другими экземплярами Apple. Для этого матрица столкновений слоев Layer Collision Matrix должна выглядеть как на рис. 28.9.
5. После настройки матрицы Layer Collision Matrix можно заняться распределением важных игровых объектов по физическим слоям.
   a. Щелкните на шаблоне Apple в панели Project (Проект) и выберите пункт Apple в раскрывающемся списке Layer, в верхней части панели Inspector (Инспектор).
   b. Щелкните на шаблоне Basket в панели Project (Проект) и выберите пункт Basket в списке Layer.
   c. Щелкните на шаблоне AppleTree в панели Project (Проект) и выберите пункт AppleTree в списке Layer (см. рис. 28.8).

После выбора физического слоя для AppleTree Unity спросит, хотите вы изменить слой только для экземпляра AppleTree или для его дочерних объектов тоже. Конечно же, вы должны ответить Yes (Да) и подтвердить изменение слоя для дочерних объектов Trunk и Sphere, принадлежащих AppleTree, потому что они должны находиться в одном физическом слое с родителем. Это изменение затронет также экземпляр AppleTree в сцене. Можете щелкнуть на AppleTree в панели Hierarchy (Иерархия), чтобы убедиться в этом.

Если теперь щелкнуть на кнопке Play (Играть), яблоки должны просто падать с яблони, как задумывалось.

Остановка падения яблок ниже заданного уровня

Если запустить игру и оставить ее воспроизводиться в течение некоторого времени, можно заметить появление большого количества яблок в иерархии. Причина в том, что каждую секунду код создает новое яблоко, но не удаляет упавшие яблоки.

1. Откройте сценарий Apple и добавьте код, уничтожающий яблоки, когда они упадут ниже уровня `transform.position.y == -20` (то есть скроются за нижним краем экрана):

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Apple : MonoBehaviour {
    public static float     bottomY = -20f;                                    // a

    void Update () {
```

```
        if ( transform.position.y < bottomY ) {
            Destroy( this.gameObject );                                    // b
        }
    }
}
```

a. Выделенная жирным строка `public static float` объявляет и определяет статическую переменную с именем `bottomY`. Как отмечалось в главе 26 «Классы», статические переменные совместно используются всеми экземплярами класса, поэтому все экземпляры Apple будут иметь одно и то же значение `bottomY`. Если `bottomY` изменится в одном экземпляре, она одновременно изменится во всех экземплярах. Также важно заметить, что статические поля, такие как `bottomY`, не отображаются в инспекторе.

b. Функция `Destroy()` удаляет указанный объект из игры и может использоваться для удаления компонентов и игровых объектов. В данном случае должен использоваться вызов `Destroy(this.gameObject)`, потому что вызов `Destroy(this)` удалит компонент Apple (Script) из экземпляра Apple. Во всех сценариях ссылка `this` указывает на текущий экземпляр класса C#, выполняющий вызов (в этом листинге `this` ссылается на экземпляр компонента Apple (Script)), а не на сам игровой объект. Всякий раз, когда требуется удалить игровой объект целиком из подключенного к нему класса компонента, следует использовать вызов `Destroy( this.gameObject )`.

2. Сохраните сценарий Apple.

3. Подключите сценарий Apple к шаблону Apple в панели Project (Проект), чтобы задействовать этот код в игре. Вы уже знаете, что нужно перетащить сценарий на игровой объект, чтобы подключить его, но есть еще один способ сделать то же самое:

   a. Выберите Apple в панели Project (Проект).

   b. Прокрутите содержимое в панели Inspector (Инспектор) до самого низа и щелкните на кнопке Add Component (Добавить компонент).

   c. В открывшемся меню выберите Scripts > Apple (Сценарии > Apple).

Если теперь щелкнуть на кнопке Play (Играть) в Unity и уменьшить масштаб сцены, можно увидеть, как яблоки падают вниз, достигают уровня Y, равного –20, и исчезают.

Это все, что имеет отношение к яблокам.

Создание экземпляров корзин

В процессе реализации поведения корзин я познакомлю вас с идеей, с которой мы снова и снова будем сталкиваться в прототипах. Как вы уже знаете, объектно-ориентированное мышление побуждает дизайнеров создавать независимые классы для всех игровых объектов (как мы только что сделали это для AppleTree и Apple), но часто бывает полезно иметь сценарий, который управляет игрой в целом.

1. Подключите сценарий ApplePicker к главной камере Main Camera в иерархии. Я часто подключаю такие сценарии, управляющие игрой, к Main Camera, потому что главная камера гарантированно присутствует в любой сцене.
2. Откройте сценарий ApplePicker в MonoDevelop, введите следующий код и сохраните его:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ApplePicker : MonoBehaviour {
    [Header("Set in Inspector")]                                    // a
    public GameObject       basketPrefab;
    public int              numBaskets = 3;
    public float            basketBottomY = -14f;
    public float            basketSpacingY = 2f;

    void Start () {
        for (int i=0; i<numBaskets; i++) {
            GameObject tBasketGO = Instantiate<GameObject>( basketPrefab );
            Vector3 pos = Vector3.zero;
            pos.y = basketBottomY + ( basketSpacingY * i );
            tBasketGO.transform.position = pos;
        }
    }
}
```

a. Эта строка добавляет раздел в инспектор, где будут отображаться переменные. В последующих листингах для переменных, значения которых вычисляются в процессе игры, я буду использовать заголовок "Set Dynamically".

Этот код создает три экземпляра шаблона Basket, располагая их на экране вертикально, друг над другом.

3. В Unity щелкните на Main Camera в панели Hierarchy (Иерархия) и в инспекторе выберите для поля basketPrefab шаблон Basket из панели Project (Проект). Щелкните на кнопке Play (Играть), и вы увидите, как этот код создаст три корзины в нижней части экрана.

Перемещение корзин мышью

Далее нам нужно написать код, перемещающий каждую корзину (экземпляр Basket) вслед за указателем мыши.

1. Подключите сценарий Basket к шаблону Basket в панели Project (Проект).
2. Откройте сценарий Basket в MonoDevelop, введите следующий код и сохраните его:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class Basket : MonoBehaviour {

    void Update () {
        // Получить текущие координаты указателя мыши на экране из Input
        Vector3 mousePos2D = Input.mousePosition;                          // a

        // Координата Z камеры определяет, как далеко в трехмерном пространстве
        //   находится указатель мыши
        mousePos2D.z = -Camera.main.transform.position.z;                  // b

        // Преобразовать точку на двумерной плоскости экрана в трехмерные
        // координаты игры
        Vector3 mousePos3D = Camera.main.ScreenToWorldPoint( mousePos2D ); // c

        // Переместить корзину вдоль оси X в координату X указателя мыши
        Vector3 pos = this.transform.position;
        pos.x = mousePos3D.x;
        this.transform.position = pos;
    }
}
```

a. Переменной `mousePos2D` присваивается значение `Input.mousePosition`. Это экранные координаты, то есть расстояние в пикселах от левого верхнего угла экрана до указателя мыши. Координата Z в `Input.mousePositon` всегда равна 0, потому что экран, по сути, это двумерная плоскость.

b. Эта строка присваивает координате Z в `mousePos2D` значение координаты Z главной камеры с обратным знаком. В игре координата Z главной камеры равна −10, соответственно `mousePos2D.z` получит значение 10. Тем самым мы сообщаем последующему вызову функции `ScreenToWorldPoint()`, как далеко от камеры должна находиться точка `mousePos3D` в трехмерном пространстве, фактически помещая ее на плоскость Z=0.

c. `ScreenToWorldPoint()` преобразует экранные координаты `mousePoint2D` в координаты в трехмерном игровом пространстве. Если значение `mousePos2D.z` оставить равным 0, точка `mousePos3D` получит координату Z, равную −10 (координата Z главной камеры). Присвоив `mousePos2D.z` значение 10, мы поместили точку `mousePos3D` в трехмерном пространстве на удалении 10 метров от главной камеры, благодаря чему поле `mousePos3D.z` получило значение 0. В игре Apple Picker это не имеет большого значения, но в будущих играх координата Z указателя мыши будет играть более важную роль. Если что-то для вас осталось неясным, я советую заглянуть в описание `Camera.ScreenToWorldPoint()` руководства по разработке сценариев для Unity[1].

Теперь, после щелчка на кнопке Play (Играть) в Unity корзины будут перемещаться вслед за указателем мыши, и вы сможете ловить ими яблоки, хотя на самом деле яблоки пока не ловятся.

[1] Руководство доступно по адресу https://docs.unity3d.com/ScriptReference/. Обязательно щелкните на кнопке C#, чтобы видеть примеры на языке C# (а не на JavaScript).

Ловля яблок

Перейдем к реализации ловли яблок:

1. Добавьте в сценарий Basket строки из следующего листинга, выделенные жирным:

```
public class Basket : MonoBehaviour {

    void Update () { … }

    void OnCollisionEnter( Collision coll ) {                    // a
        // Отыскать яблоко, попавшее в эту корзину
        GameObject collidedWith = coll.gameObject;               // b
        if ( collidedWith.tag == "Apple" ) {                     // c
            Destroy( collidedWith );
        }
    }
}
```

   a. Метод `OnCollisionEnter` вызывается всякий раз, когда какой-то другой объект сталкивается с этой корзиной. В аргументе `Collision` передается информация о столкновении, включая ссылку на игровой объект, столкнувшийся с коллайдером корзины.

   b. Эта строка присваивает локальной переменной `collidedWith` ссылку на игровой объект, столкнувшийся с корзиной.

   c. Проверяется, что `collidedWith` является яблоком, для чего поле `tag` яблока сравнивается с тегом `"Apple"`, который присваивается всем экземплярам Apple. Если `collidedWith` — это яблоко, оно уничтожается.

   Если теперь яблоко коснется корзины, оно будет уничтожено.

2. Сохраните сценарий Basket, вернитесь в Unity и щелкните на кнопке Play (Играть).

   Теперь игра действует очень похоже на классическую игру *Kaboom!*. В ней пока нет элементов *графического пользовательского интерфейса* (ГПИ), отображающих набранные очки или количество оставшихся жизней, но даже без них Apple Picker может считаться успешно реализованным прототипом. В текущем виде этот прототип позволяет настроить некоторые аспекты игры и подобрать оптимальный уровень сложности.

3. Сохраните сцену.

4. Создайте копию текущей сцены для балансировки настроек игры.

   a. Щелкните на сцене _Scene_0 в панели Project (Проект), чтобы выбрать ее.

   b. Нажмите комбинацию Command-D на клавиатуре (Ctrl+D в Windows), чтобы создать копию сцены, или выберите в главном меню пункт Edit > Duplicate (Правка > Дублировать). В результате будет создана новая сцена с именем _Scene_1.

   c. Щелкните дважды на _Scene_1, чтобы открыть ее.

Будучи точной копией _Scene_0, игра в этой новой сцене будет действовать без всяких изменений.

Щелкните на AppleTree в иерархии, чтобы настроить переменные в _Scene_1, не касаясь при этом переменных в _Scene_0 (любые изменения, выполняемые в игровых объектах, выделенных в панели Project (Проект), будут применяться к обеим сценам). Попробуйте сделать игру сложнее. Настроив сложность игры в _Scene_1, сохраните ее и откройте сцену _Scene_0. Если вы забыли, какая сцена открыта в текущий момент, просто взгляните на текст в заголовке окна Unity или в верхнюю часть панели Hierarchy (Иерархия). И там и там вы увидите название сцены.

ГИП и управление игрой

Последнее, что мы добавим в нашу игру, — графический интерфейс пользователя (ГИП) и управление игрой, которые сделают этот прототип более похожим на настоящую игру. Далее мы добавим один элемент ГИП — счетчик очков и элементы управления — уровни и жизни.

Счетчик очков

Счетчик очков дает игроку возможность видеть его достижения в игре.

1. Откройте сцену _Scene_0, дважды щелкнув на ней в панели Project (Проект).
2. В меню Unity выберите пункт GameObject > UI > Text (Игровой объект > ПИ > Текст)[1].

   Так как это первый элемент графического интерфейса пользователя Unity в этой сцене, в панели Hierarchy (Иерархия) появится несколько объектов. Первый из них — Canvas. Объект Canvas представляет двумерный холст, на который будут помещаться элементы ГИП. Взгляните на панель Scene (Сцена), в ней вы должны увидеть очень большой двумерный прямоугольник, далеко простирающийся от его начала вдоль осей X и Y.
3. Дважды щелкните на Canvas в иерархии, чтобы уменьшить масштаб и увидеть весь объект целиком. Этот объект примет размеры в соответствии с размерами панели Game (Игра), поэтому если для панели Game (Игра) установлено соотношение сторон 16:9, холст Canvas также получит это соотношение сторон. Также можно щелкнуть на кнопке 2D в верхней части панели Scene (Сцена), чтобы включить двумерный вид, в котором проще работать с холстом Canvas.

[1] Как-то раз, когда я тестировал этот раздел с описанием учебного примера, пункт GameObject > UI > Text (Игровой объект > ПИ > Текст) в меню Unity оказался неактивным. Если то же самое случится у вас, щелкните правой кнопкой мыши на пустом месте в панели Hierarchy (Иерархия) и выберите в появившемся контекстном меню пункт UI > Text (ПИ > Текст).

Другой игровой объект, добавленный на верхний уровень иерархии, — EventSystem. Он позволяет добавлять кнопки, ползунки и другие интерактивные элементы ГИП; однако в этом прототипе мы не будем использовать его.

В списке дочерних объектов Canvas вы увидите игровой объект Text. Если вы его не видите, щелкните на пиктограмме с треугольником рядом с объектом Canvas в иерархии, чтобы распахнуть его. Дважды щелкните на игровом объекте Text в панели Hierarchy (Иерархия), чтобы увеличить масштаб. Весьма вероятно, что по умолчанию для текста будет выбран черный цвет, который плохо различим на фоне в панели Scene (Сцена).

4. Выберите игровой объект Text в иерархии и в панели Inspector (Инспектор) измените его имя на HighScore.

5. Выполните следующие инструкции, чтобы настроить объект HighScore в инспекторе, как показано на рис. 28.10:

   a. В компоненте RectTransform объекта HighScore:

      - В разделе Anchors установите координаты Min X = 0, Min Y = 1, Max X = 0 и Max Y = 1.
      - В разделе Pivot установите координаты: X = 0 и Y = 1.
      - Установите координаты Pos X = 10, Pos Y = –6 и Pos Z = 0.
      - Установите ширину и высоту: Width = 256 и Height = 32.

   После этого снова дважды щелкните на HighScore в иерархии, чтобы переместить вид в панели Scene (Сцена) в центр объекта.

   b. В компоненте Text (Script) объекта HighScore, в панели Inspector (Инспектор):

      - В поле Text введите текст «High Score: 1000» (без кавычек).
      - В поле Font Style выберите Bold.
      - В поле Font Size введите число 28.
      - В поле Color выберите белый цвет, чтобы сделать текст более заметным на фоне в панели Game (Игра).

6. Щелкните правой кнопкой мыши на HighScore в иерархии и в контекстном меню выберите пункт Duplicate (Дублировать)[1].

7. Выделите новый игровой объект HighScore (1) и измените его имя на ScoreCounter.

8. Измените настройки компонентов RectTransform и Text (Script) игрового объекта ScoreCounter в инспекторе, как показано на рис. 28.10. Не забудьте изменить значения Anchors и Pivot в компоненте RectTransform, а также значение Alignment в компоненте Text. Обратите внимание, что при изменении значения Anchors или Pivot в RectTransform Unity автоматически изменяет Pos X, чтобы объект

[1] Возможно, вам придется щелкнуть на панели Hierarchy (Иерархия), чтобы в ней появилась копия HighScore (1).

ScoreCounter остался на том же месте внутри Canvas. Чтобы предотвратить это, щелкните на кнопке R в разделе RectTransform для объекта ScoreCounter, которая на рис. 28.10 находится под указателем мыши.

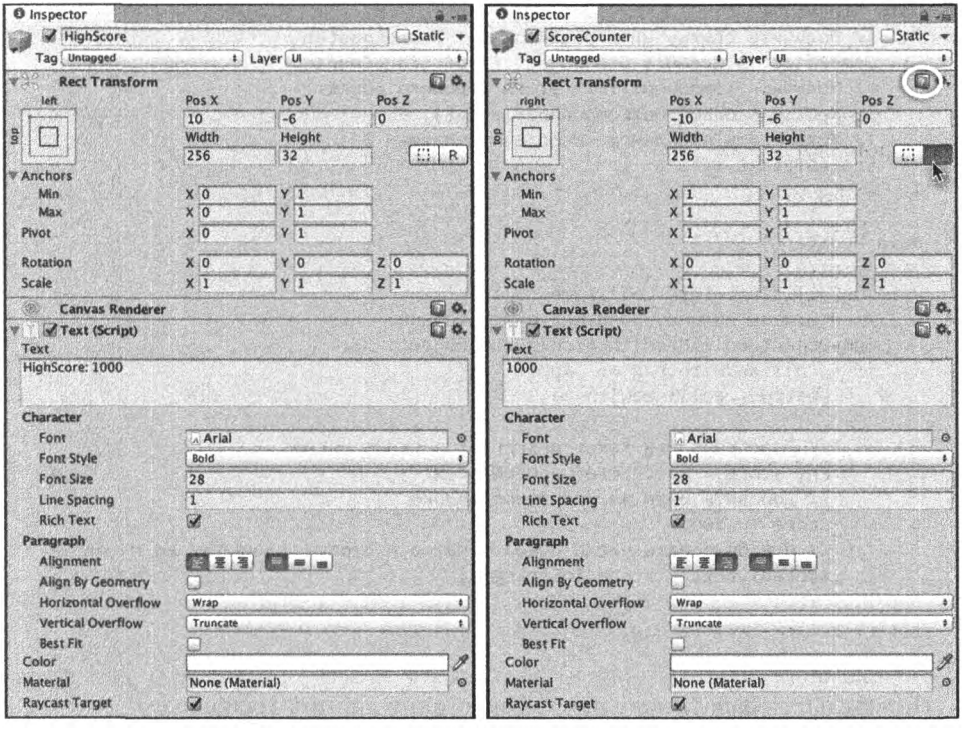


Рис. 28.10. Настройки компонентов RectTransform и Text в объектах HighScore и ScoreCounter

Добавление очков за пойманное яблоко

Когда яблоко сталкивается с корзиной, об этом событии извещаются два сценария: Apple и Basket. В сценарии Basket уже имеется метод `OnCollisionEnter()`, поэтому далее описывается, как изменить этот сценарий, чтобы начислить игроку очки за ловли яблок. Сто очков за одно яблоко выглядят вполне разумным числом (хотя я всегда считал, что немного смешно добавлять лишние нули справа от заработанных очков).

1. Откройте сценарий Basket в MonoDevelop и добавьте строки из следующего листинга, выделенные жирным:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
using UnityEngine.UI; // Эта строка подключает библиотеку для работы с ГИП // a

public class Basket : MonoBehaviour {
    [Header("Set Dynamically")]
    public Text            scoreGT;                                          // a

    void Start() {
        // Получить ссылку на игровой объект ScoreCounter
        GameObject scoreGO = GameObject.Find("ScoreCounter");                // b
        // Получить компонент Text этого игрового объекта
        scoreGT = scoreGO.GetComponent<Text>();                              // c
        // Установить начальное число очков равным 0
        scoreGT.text = "0";
    }

    void Update () { … }

    void OnCollisionEnter( Collision coll ) {
        // Отыскать яблоко, попавшее в эту корзину
        GameObject collidedWith = coll.gameObject;
        if ( collidedWith.tag == "Apple" ) {
            Destroy( collidedWith );

            // Преобразовать текст в scoreGT в целое число
            int score = int.Parse( scoreGT.text );                           // d
            // Добавить очки за пойманное яблоко
            score += 100;
            // Преобразовать число очков обратно в строку и вывести ее на экран
            scoreGT.text = score.ToString();
        }
    }
}
```

a. Не забудьте добавить эти строки. Они стоят в листинге особняком.

b. `GameObject.Find("ScoreCounter")` отыскивает в сцене игровой объект с именем `"ScoreCounter"` и возвращает ссылку на него, которая тут же присваивается локальной переменной `scoreGO`. Обратите внимание, что имя `"ScoreCounter"` не должно содержать пробелов ни в коде, ни в панели Hierarchy (Иерархия).

c. `scoreGO.Getcomponent<Text>()` отыскивает компонент `Text` игрового объекта `scoreGO` и возвращает ссылку на него, которая присваивается общедоступному полю `scoreGT`. Затем, в следующей строке, в это поле записывается начальное количество очков, равное нулю. Без строки `using UnityEngine.UI;` в начале сценария вы не смогли бы использовать компонент `Text` в коде на C#. По мере совершенствования приемов программирования в Unity Technology все шире используют модель, согласно которой вы должны сами подключать необходимые библиотеки функций.

d. `int.Parse( scoreGT.text )` получает текст, отображаемый в ScoreCounter, и преобразует его в целое число `score`. Затем к этому числу добавляется 100 очков, вызовом `score.ToString()` результат преобразуется обратно в строку и записывается в свойство `text` объекта `scoreGT`.

Уведомление сценария ApplePicker, что яблоко не было поймано

Другой аспект, делающий прототип Apple Picker более похожим на игру, — завершение раунда и удаление корзины, если яблоко не было поймано. В данный момент яблоки уничтожаются автоматически, и это здорово, но они должны уведомлять сценарий ApplePicker об этом, чтобы можно было завершить раунд и уничтожить остальные яблоки. Для этого необходимо, чтобы один сценарий вызывал функцию в другом сценарии.

1. Для начала в MonoDevelop внесите следующие изменения в сценарий Apple:

```
public class Apple : MonoBehaviour {
    [Header("Set in Inspector")]
    public static float     bottomY = -20f;

    void Update () {
        if ( transform.position.y < bottomY ) {                              // a
            Destroy( this.gameObject );

            // Получить ссылку на компонент ApplePicker главной камеры Main Camera
            ApplePicker apScript = Camera.main.GetComponent<ApplePicker>();// b
            // Вызвать общедоступный метод AppleDestroyed() из apScript
            apScript.AppleDestroyed();                                       // c
        }
    }
}
```

   a. Обратите внимание, что все новые строки находятся внутри инструкции `if`.

   b. Этот вызов вернет ссылку на компонент ApplePicker (Script) главной камеры Main Camera. Поскольку класс `Camera` имеет встроенную статическую переменную `Camera.main`, которая ссылается на главную камеру Main Camera, нет необходимости использовать вызов `GameObject.Find("Main Camera")`, чтобы получить эту ссылку. Следующий вызов `GetComponent<ApplePicker>()` вернет ссылку на компонент ApplePicker (Script) главной камеры, которая тут же будет присвоена локальной переменной `apScript`. После этого появляется возможность напрямую обращаться к переменным и методам экземпляра класса `ApplePicker`, подключенного к главной камере.

   c. Это вызов несуществующего метода `AppleDestroyed()` класса `ApplePicker`. Так как метод пока не существует, MonoDevelop подчеркнет эту строку красной волнистой линией, и вы не сможете запустить игру в Unity, пока не определите `AppleDestroyed()`.

2. Нам нужно добавить общедоступный метод `AppleDestroyed()` в сценарий ApplePicker, поэтому откройте этот сценарий в MonoDevelop и добавьте следующие строки, выделенные жирным:

```
public class ApplePicker : MonoBehaviour {
    ...
```

```
    void Start () { ... }

    public void AppleDestroyed() {                                    // a
        // Удалить все упавшие яблоки
        GameObject[] tAppleArray=GameObject.FindGameObjectsWithTag("Apple"); // b
        foreach ( GameObject tGO in tAppleArray ) {
            Destroy( tGO );
        }
    }
}
```

a. Метод `AppleDestroyed()` должен быть объявлен общедоступным (`public`), чтобы его можно было вызывать из других классов (таких, как `Apple`). По умолчанию все методы объявляются скрытыми (`private`) и недоступны (не могут вызываться) другим классам.

b. `GameObject.FindGameObjectsWithTag("Apple")` вернет массив всех существующих игровых объектов Apple[1]. Последующий цикл `foreach` выполняет обход всех найденных объектов и уничтожает их.

Сохраните ВСЕ сценарии в MonoDevelop. После определения метода `AppleDestroyed()` снова появится возможность запустить игру в Unity.

Уничтожение корзины после потери яблока

Последний фрагмент кода, который мы добавим в эту сцену, удаляет одну корзину при потере яблока и останавливает игру после удаления последней корзины. Добавьте следующие изменения в сценарий ApplePicker (на этот раз, на всякий случай, сценарий приводится целиком):

```
using System.Collections;
using System.Collections.Generic;                                     // a
using UnityEngine;
using UnityEngine.SceneManagement;                                    // b

public class ApplePicker : MonoBehaviour {
    [Header("Set in Inspector")]
    public GameObject       basketPrefab;
    public int              numBaskets = 3;
    public float            basketBottomY = -14f;
    public float            basketSpacingY = 2f;
    public List<GameObject> basketList;

    void Start () {
        basketList = new List<GameObject>();                          // c
        for (int i=0; i<numBaskets; i++) {
```

[1] На самом деле `GameObject.FindGameObjectsWithTag()` требует довольно много времени для работы, поэтому я не рекомендую использовать его внутри `Update()` или `FixedUpdate()`. Но так как вызов происходит, только когда игрок теряет корзину Basket (и игра замедляется в этой точке), в такой ситуации этот вызов вполне допустим.

```
            GameObject tBasketGO = Instantiate<GameObject>( basketPrefab );
            Vector3 pos = Vector3.zero;
            pos.y = basketBottomY + ( basketSpacingY * i );
            tBasketGO.transform.position = pos;
            basketList.Add( tBasketGO );                                    // d
        }
    }

    public void AppleDestroyed() {
        // Удалить все упавшие яблоки
        GameObject[] tAppleArray=GameObject.FindGameObjectsWithTag("Apple");
        foreach ( GameObject tGO in tAppleArray ) {
            Destroy( tGO );
        }

        // Удалить одну корзину                                             // e
        // Получить индекс последней корзины в basketList
        int basketIndex = basketList.Count-1;
        // Получить ссылку на этот игровой объект Basket
        GameObject tBasketGO = basketList[basketIndex];
        // Исключить корзину из списка и удалить сам игровой объект
        basketList.RemoveAt( basketIndex );
        Destroy( tBasketGO );
    }
}
```

a. Игровые объекты Basket будут храниться в списке `List`, поэтому необходимо подключить библиотеку `System.Collections.Generic`, которая, начиная с версии Unity 5.5, автоматически подключается во всех новых сценариях. (Дополнительная информация о списках приводится в главе 23 «Коллекции в C#».) Общедоступное поле `List<GameObject> basketList` объявлено в начале класса, а определяется и инициализируется в первой строке метода `Start()`.

b. Эта библиотека будет использоваться на шаге 3, в разделе «Добавление высшего достижения» далее в этой главе.

c. Эта строка определяет `basketList` как новый список типа `List<GameObject>`. Переменная была объявлена в строке с комментарием `// b`, но после объявления она получает значение `null`. Эта строка создает фактический список для использования в последующем.

d. Новая строка в конце цикла `for` добавляет каждую вновь созданную корзину в список `basketList`. Корзины добавляются в порядке создания, то есть снизу вверх.

e. Новый фрагмент в методе `AppleDestroyed()` уничтожает одну из корзин. Так как корзины добавляются в список в порядке снизу вверх, важно, чтобы первой уничтожалась последняя корзина в списке (то есть уничтожение производится сверху вниз).

Если теперь запустить игру и потерять все три корзины, Unity возбудит исключение `IndexOutOfRange`.

Добавление высшего достижения

Теперь задействуем игровой объект HighScore, созданный ранее:

1. Создайте новый сценарий на C# с именем HighScore и подключите его к игровому объекту HighScore в панели Hierarchy (Иерархия).
2. Откройте сценарий HighScore в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI; // Напомню, эта библиотека нужна для работы с ГИП

public class HighScore : MonoBehaviour {
    static public int    score = 1000;                                    // a

    void Update () {                                                      // b
        Text gt = this.GetComponent<Text>();
        gt.text = "High Score: "+score;
    }
}
```

   a. Объявление переменной `int score` не только общедоступной (`public`), но еще и статической (`static`) дает возможность обращаться к ней из других сценариев по полному имени `HighScore.score`. Это одна из замечательных особенностей статических переменных, которые мы будем использовать во всех прототипах в этой книге.

   b. Код в методе `Update()` просто отображает значение `score` в компоненте `Text`. В данном случае можно было не вызывать метод `ToString()` переменной `score`, потому что при использовании оператора `+` для конкатенации (объединения) строки с данными другого типа (в данном случае строковый литерал `"High Score: "` объединяется с целочисленной переменной `score`) метод `ToString()` вызывается неявно (то есть автоматически).

3. Откройте сценарий Basket и добавьте следующие строки, выделенные жирным, которые используют статическую переменную `score`:

```
public class Basket : MonoBehaviour {
    ...
    void OnCollisionEnter( Collision coll ) {
        ...
        if ( collidedWith.tag == "Apple" ) {
            ...
            // Преобразовать число очков обратно в строку и вывести ее на экран
            scoreGT.text = score.ToString();

            // Запомнить высшее достижение
            if (score > HighScore.score) {
                HighScore.score = score;
            }
        }
    }
}
```

Теперь переменная `HighScore.score` будет обновляться всякий раз, когда количество очков, заработанное пользователем, превысит ее текущее значение.

4. Откройте сценарий **ApplePicker** и добавьте следующие строки, возвращающие игру в исходное состояние, когда игрок потеряет все корзины. Этот код предотвращает появление исключения `IndexOutOfRange`, упомянутого выше.

```
public class ApplePicker : MonoBehaviour {
    ...
    public void AppleDestroyed() {
        ...
        // Исключить корзину из списка и удалить сам игровой объект
        basketList.RemoveAt( basketIndex );
        Destroy( tBasketGO );

        // Если корзин не осталось, перезапустить игру
        if ( basketList.Count == 0 ) {
            SceneManager.LoadScene( "_Scene_0" );                    // a
        }
    }
}
```

   a. `SceneManager.LoadScene( "_Scene_0" )` вновь загружает сцену **_Scene_0**. Этот метод не получится вызвать, если не добавить строку `using UnityEngine.SceneManagement;`, как описывалось в разделе «Уничтожение корзины после потери яблока» выше. Повторная загрузка сцены фактически сбрасывает игру в исходное состояние[1].

5. К данному моменту мы изменили несколько сценариев. Не забыли сохранить их? Если еще не сохранили — или не помните, как это часто случается со мной, — выберите в меню MonoDevelop пункт **File > Save All** (Файл > Сохранить все), чтобы сохранить все изменившиеся сценарии. Если пункт **Save All** (Сохранить все) отображается серым цветом, как неактивный, тогда все в порядке — все ваши сценарии уже сохранены.

---

[1] Во всех последних версиях Unity, по крайней мере вплоть до Unity 2017, при повторной загрузке уровня часто возникает известная ошибка. Если вы увидели, что сцена потемнела после повторной загрузки с помощью `SceneManager`, значит, вы столкнулись с этой ошибкой. Пока в Unity не исправят эту ошибку, можно использовать временное решение — запретить автоматическое запекание освещения (суть которого заключается в предварительном вычислении освещенности сцены). Для этого откройте панель **Lighting** (Освещение), выбрав в главном меню Unity пункт **Window > Lighting > Settings** (Окно > Освещение > Настройки). Снимите флажок **Auto Generate** (Генерировать автоматически) в нижней части панели **Lighting** (Освещение) рядом с кнопкой **Generate Lighting** (Сгенерировать освещение). Затем щелкните на кнопке **Generate Lighting** (Сгенерировать освещение), чтобы вручную пересчитать освещение, дождитесь, когда расчеты закончатся, и закройте окно **Lighting** (Освещение). Эта ошибка проявляется только в редакторе Unity и не должна возникать в автономных приложениях или в приложениях WebGL после их сборки.

Сохранение высшего достижения в хранилище PlayerPrefs

Так как `HighScore.score` — это статическая переменная, она не сбрасывается и сохраняет свое значение на протяжении всего времени игры. То есть высшее достижение переходит из одного раунда в другой. Но если игру остановить (щелкнув на кнопке Play (Играть) еще раз), значение `HighScore.score` *сбросится*. Эту проблему можно решить, воспользовавшись хранилищем `PlayerPrefs` в Unity. Хранилище `PlayerPrefs` сохраняет информацию из сценариев на компьютере так, что ее можно восстановить позже, и она не исчезает после остановки игры. Хранилище `PlayerPrefs` доступно из редактора Unity, скомпилированных сборок и сборок для WebGL, поэтому высшее достижение, полученное вами, будет доступно другим игрокам, играющим в игру на том же компьютере.

1. Откройте сценарий HighScore и добавьте следующие изменения, выделенные жирным:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI; // Напомню, эта библиотека нужна для работы с ГИП.

public class HighScore : MonoBehaviour {
    static public int    score = 1000;

    void Awake() {                                                    // a
        // Если значение HighScore уже существует в PlayerPrefs, прочитать его
        if (PlayerPrefs.HasKey("HighScore")) {                        // b
            score = PlayerPrefs.GetInt("HighScore");
        }
        // Сохранить высшее достижение HighScore в хранилище
        PlayerPrefs.SetInt("HighScore", score);                       // c
    }

    void Update () {
        Text gt = this.GetComponent<Text>();
        gt.text = "High Score: "+score;
        // Обновить HighScore в PlayerPrefs, если необходимо
        if (score > PlayerPrefs.GetInt("HighScore")) {                // d
            PlayerPrefs.SetInt("HighScore", score);
        }
    }
}
```

   a. `Awake()` — это встроенный в Unity метод класса `MonoBehaviour` (как `Start()` или `Update()`), который вызывается при создании экземпляра класса `HighScore` (то есть `Awake()` всегда вызывается перед `Start()`).

   b. `PlayerPrefs` — это словарь значений, на которые можно ссылаться по ключам (то есть уникальным строкам). В этом случае используется ключ `"HighScore"`. Эта строка проверяет наличие целочисленного значения с ключом `"HighScore"` в хранилище `PlayerPrefs` и, если оно имеется, читает его.

Для каждого проекта/приложения создается свое, отдельное хранилище `PlayerPrefs`, поэтому можно не опасаться, что имя `HighScore` будет конфликтовать с именем `HighScore` в хранилище `PlayerPrefs` другого проекта.

c. Последняя строка в методе `Awake()` сохраняет текущее значение `score` в хранилище `PlayerPrefs` с ключом `"HighScore"`. Если ключ `"HighScore"` уже имеется в хранилище, его значение будет затерто значением `score`; но если ключ отсутствует, эта инструкция гарантирует его создание.

d. Теперь, после добавления этих строк, метод `Update()` будет проверять текущее значение `HighScore.score` в каждом кадре и, если оно превысит значение, хранящееся в `PlayerPrefs`, запишет его в хранилище.

Использование хранилища `PlayerPrefs` позволит игре Apple Picker запоминать высшее достижение на этом компьютере и восстанавливать его после повторного запуска игры и даже после перезагрузки компьютера.

2. Снова *сохраните все* сценарии в MonoDevelop, просто для уверенности. Вернитесь в Unity и щелкните на кнопке **Play** (Играть).

Теперь вы сможете играть в игру, зарабатывать очки и сохранять высшее достижение. Достигнув нового рекорда, попробуйте остановить игру и вновь запустить ее — вы увидите, что ваш рекорд сохранился.

Итоги

Теперь у вас есть прототип игры, очень похожий на классическую игру *Kaboom!* компании Activision. В нашей игре не хватает, например, последовательного увеличения сложности, начального и конечного экранов, но вы можете добавить все это самостоятельно, когда у вас появится больше опыта.

Следующие шаги

Вот несколько идей, которые вы могли бы воплотить в этот прототип в будущем. Самый лучший способ обучения программированию — выполнить инструкции в руководстве, подобные тем, что были представлены в этой главе, а затем попробовать добавить свои изменения.

- **Начальный экран:** вы можете создать начальный экран в отдельной сцене, поместить в него изображение заставки и кнопку **Start** (Пуск). По щелчку на кнопке **Start** (Пуск) можно вызвать `SceneManager.LoadScene ( "_Scene_0" );`, чтобы запустить игру. Не забудьте, что для доступа к `SceneManager` нужно добавить в начало сценария строку `using UnityEngine.SceneManagement;`.
- **Конечный экран:** также можно создать экран **Game Over** (Игра окончена). На этом экране можно показать количество очков, набранное игроком, и сообщить ему, установил ли он новый рекорд. Этот экран обязательно должен содержать

кнопку Play Again (Играть опять), щелчок на которой вызывает `SceneManager.LoadScene( "_Scene_0" )`.

- **Увеличение сложности:** в следующих прототипах мы обсудим вопрос создания уровней сложности, но если вы решите реализовать их в этом прототипе, имеет смысл добавить список для хранения таких параметров AppleTree, как скорость (`speed`), вероятность смены направления (`chanceToChangeDirections`) и время в секундах между падением яблок (`secondsBetweenAppleDrops`). Каждый элемент такого списка мог бы представлять отдельный уровень сложности, где 0-й элемент соответствует самому простому уровню, а последний — самому сложному. В процессе игры можно наращивать некий счетчик и использовать его как индекс в таком списке, то есть для значения 0 счетчика использовать значения для переменных из 0-го элемента списка.

Если вы решите добавить в игру любой из дополнительных экранов — начальный или конечный, — не забудьте включить соответствующие сцены в список сборки в Build Settings. Для этого откройте нужную сцену в Unity и выберите в меню пункт File > Build Settings... (Файл > Параметры сборки...). В открывшемся окне Build Settings (Параметры сборки) щелкните на кнопке Add Open Scenes (Добавить открытые сцены), и имя текущей открытой сцены добавится в список Scenes in Build (Сцены для сборки). При сборке автономной версии игры имейте в виду, что после ее запуска первой будет загружена и показана сцена с номером 0 в имени.

29 Прототип 2: Mission Demolition

Игры, учитывающие законы физики, являются одними из наиболее популярных, и именно эта черта обеспечивает широкую известность таким играм, как *Angry Birds*. В этой главе вы создадите свою игру с поддержкой физики в духе *Angry Birds* и всех других физических игр, предшествовавших ей, таких как *Crossbows and Catapults*, *Worms*, *Scorched Earth* и т. д.

Эта глава охватывает следующие темы: физика, столкновения, взаимодействия с мышью, уровни и управление состоянием игры.

Начало: прототип 2

Так как это наш второй прототип и у вас уже есть некоторый опыт, в этой главе мы будем идти вперед немного быстрее, не останавливаясь на том, что вы уже знаете. Конечно, я продолжу детально освещать новые темы. Я советую вооружиться карандашом и отмечать каждый шаг в этой главе по мере его выполнения.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. Чтобы освежить в памяти стандартную процедуру настройки проекта, обращайтесь к приложению А «Стандартная процедура настройки проекта».

- **Имя проекта:** Mission Demolition Prototype
- **Имя сцены:** _Scene_0
- **Имена сценариев на C#:** пока нет

Идея прототипа игры

В этой игре игрок будет стрелять из рогатки по замку, стараясь разрушить его. В каждом замке имеется особая зона, попав в которую игрок переходит на следующий уровень.

Вот как выглядит желаемая последовательность событий:

1. Когда игрок наводит указатель мыши на рогатку, она должна подсвечиваться.
2. Если игрок нажимает левую кнопку мыши (в Unity это кнопка с номером 0), когда рогатка подсвечена, в области указателя мыши должен создаваться снаряд.
3. При перемещении указателя мыши с нажатой левой кнопкой снаряд должен следовать за указателем, но оставаясь в пределах сферического коллайдера рогатки.
4. От каждого рога рогатки к снаряду должна тянуться белая линия, чтобы рогатка в игре выглядела как настоящая.
5. Когда игрок отпускает левую кнопку мыши (с номером 0), снаряд должен запускаться из рогатки.
6. Цель игрока — разрушить замок, находящийся в нескольких метрах, и попасть в особую область внутри него.
7. Игроку дается право выполнить три выстрела, чтобы поразить цель. Каждый сделанный выстрел оставляет след на экране, чтобы игрок мог сориентироваться и скорректировать следующий выстрел.

Многие пункты в предыдущем списке относятся к механике, но один из них носит исключительно эстетический характер: пункт под номером 4. Все другие пункты, упоминающие художественные элементы, используют их в игровой механике, но единственной целью пункта 4 является улучшение эстетики игры, поэтому он наименее важен для прототипа. Составляя список идей для игры, важно помнить об этом. Это не означает, что вы не должны уделять внимание эстетике в прототипе; просто нужно уметь правильно расставлять приоритеты, отдавая предпочтение элементам, имеющим самое непосредственное влияние на механику игры. Для экономии времени и места в книге основное внимание в этом прототипе будет уделено другим элементам, а реализацию пункта 4 я оставлю вам как самостоятельное упражнение.

Художественные ресурсы

Прежде чем приступить к программному коду, нужно создать несколько ресурсов художественного оформления.

Земля

Выполните следующие шаги, чтобы создать землю:

1. Откройте сцену _Scene_0. Проверьте, что содержимое сцены _Scene_0 видно в панели Hierarchy (Иерархия): в списке должны присутствовать главная камера Main Camera и источник направленного освещения Directional Light (если эти элементы не видны в списке, щелкните на пиктограмме с треугольником рядом с элементом _Scene_0 в панели Hierarchy (Иерархия)).

2. Создайте куб (выберите в меню пункт GameObject > 3D Object > Cube (Игровой объект > 3D объект > Куб)). Переименуйте куб в Ground. Чтобы превратить куб в сплошной прямоугольник, вытянутый во всю ширину сцены вдоль оси X, установите параметры компонента Transform, как показано ниже:

```
Ground (Cube) P:[ 0, -10, 0 ] R:[ 0, 0, 0 ] S:[ 100, 1, 4 ]2
```

3. Создайте новый материал (Assets > Create > Material (Ресурсы > Создать > Материал)) и дайте ему имя Mat_Ground.
   a. Выберите в поле Albedo для материала Mat_Ground коричневый цвет.
   b. Установите ползунок Smoothness в позицию 0 (земля не должна блестеть).
   c. Подключите Mat_Ground к игровому объекту Ground в иерархии (в предыдущей главе рассказывалось, как это сделать).
4. Сохраните сцену.

Направленное освещение

В последних версиях Unity источник направленного освещения Directional Light добавляется в сцену по умолчанию, но мы должны правильно настроить его для данного проекта.

1. Выберите Directional Light в панели Hierarchy (Иерархия). Одной из особенностей источников направленного освещения является полная независимость сцен от их местоположения; значение имеет только поворот источника света. Учитывая это, переместите источник в сторону, установив параметры его компонента Transform, как показано ниже:

```
Directional Light P:[ -10, 0, 0 ] R:[ 50, -30, 0 ] S:[ 1, 1, 1 ].
```

2. Сохраните сцену.

Настройка камеры

Шаги для настройки камеры:

1. Выберите главную камеру Main Camera в иерархии и переименуйте ее в _MainCamera.
2. Установите следующие параметры компонента Transform камеры _MainCamera (обязательно установите координату Y в разделе Position равной 0):

```
_MainCamera P:[ 0, 0, -10 ] R:[ 0, 0, 0 ] S:[ 1, 1, 1 ].
```

3. Установите следующие параметры компонента Camera камеры _MainCamera:
   a. В раскрывающемся списке Clear Flags выберите пункт Solid Color.
   b. В поле Background выберите цвет, близкий к небесно-голубому.

c. В поле Projection выберите значение Orthographic.

d. В поле Size введите число 10.

После этого настройки должны выглядеть, как показано на рис. 29.1. Обратите внимание, что полоса внизу в поле Background должна быть белой, а не черной. Если она черная, то значение альфа-канала (то есть прозрачность) для цвета равно 0 (полностью прозрачный/невидимый). Чтобы исправить это, щелкните на поле с выбранным цветом и установите в компоненте A цвета значение 255[1].

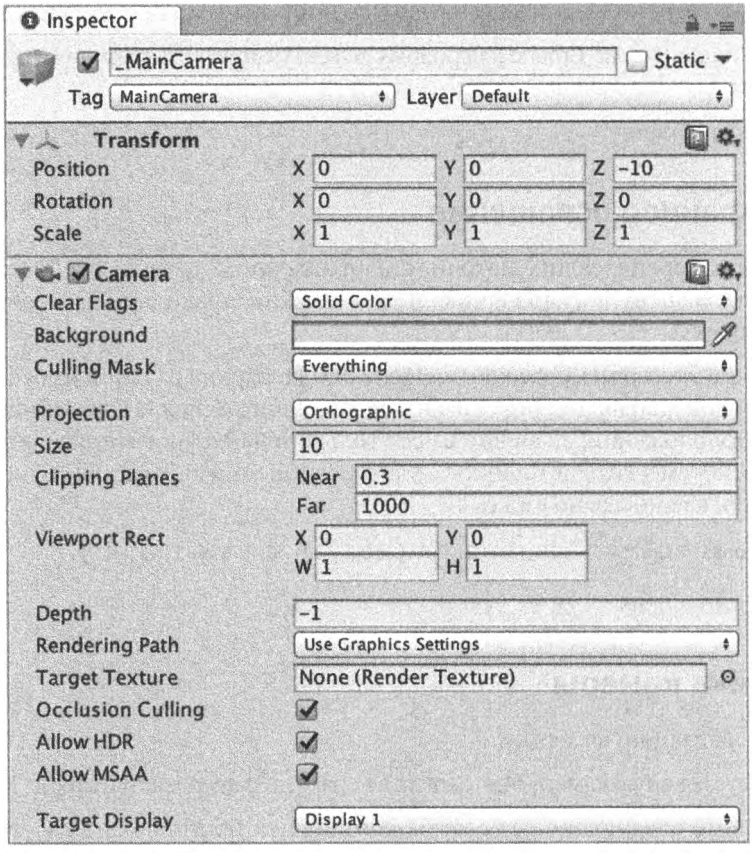


Рис. 29.1. Настройки компонентов Transform и Camera в _MainCamera

[1] Корректировка альфа-канала для фонового цвета Background в действительности не нужна, но при определении цветов в Unity для альфа-канала часто по умолчанию выбирается значение 0, и в прошлом у меня возникали проблемы из-за этого, поэтому я хочу приучить вас проверять значение альфа-канала для любых цветов в инспекторе.

Раньше вы уже использовали камеру с ортографической проекцией, но я не пояснял значение параметра Size. В ортографической проекции параметр Size определяет расстояние от центра до нижнего края поля зрения камеры, то есть Size — это половина высоты области, видимой для камеры. Как показано на рис. 29.2, где изображена панель Game (Игра), прямоугольник Ground находится в позиции с координатой Y, равной −10, и делится ровно пополам нижним краем окна Game (Игра).[1] Попробуйте изменить соотношение сторон панели Game (Игра) в раскрывающемся списке, выделенном рамкой на рис. 29.2 и находящемся под указателем мыши. Вы увидите, что, независимо от выбранного соотношения сторон, центр прямоугольника Ground совпадает с нижним краем панели Game (Игра).

4. После экспериментов выберите соотношение сторон 16:9, как показано на рис. 29.2.
5. Сохраните сцену. Никогда не забывайте сохранять свои сцены.

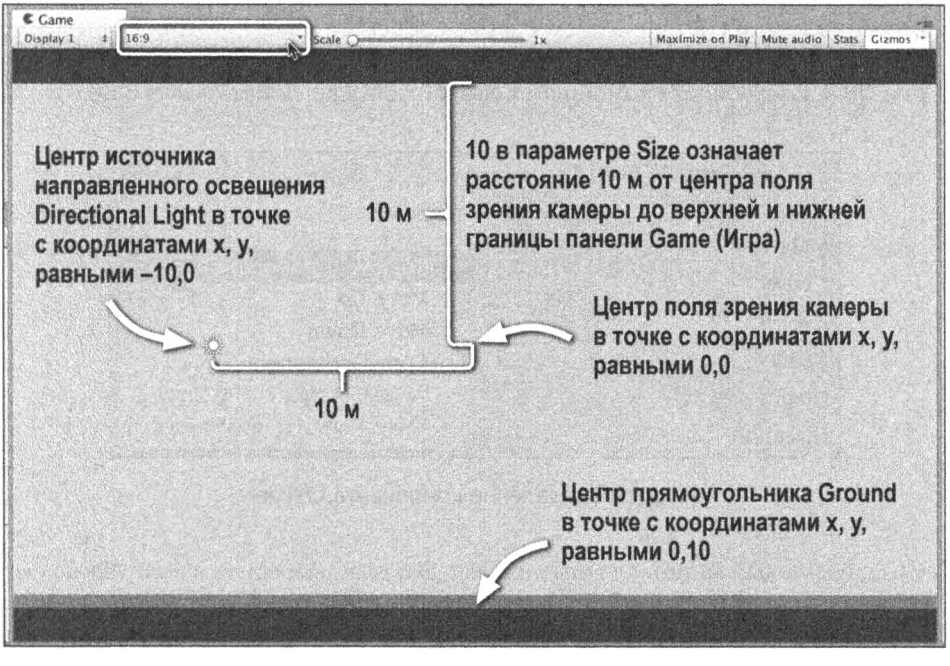


Рис. 29.2. Демонстрация смысла значения 10 в параметре Size для камеры с ортографической проекцией

[1] Если вы не видите прямоугольник Ground внизу в панели Game (Игра), проверьте еще раз координату Y в разделе Position главной камеры _MainCamera — она должна иметь значение 0, а координата Y в разделе Position игрового объекта Ground должна быть равна −10. Если вы не видите источник направленного освещения Directional Light в панели Game (Игра), щелкните на кнопке Gizmos (Значки), чтобы сделать его видимым.

Рогатка

Теперь создадим простую рогатку из трех цилиндров:

1. Создайте пустой игровой объект (GameObject > Create Empty (Игровой объект > Создать пустой)). Измените его имя на Slingshot и установите параметры его компонента Transform, как показано ниже:

```
Slingshot (Empty) P:[ 0, 0, 0 ] R:[ 0, 0, 0 ] S:[ 1, 1, 1 ].
```

2. Создайте новый цилиндр (GameObject > 3D Object > Cylinder (Игровой объект > 3D объект > Цилиндр)) и измените его имя на Base. В иерархии перетащите Base на Slingshot, сделав объект Slingshot его родителем. Щелкните на пиктограмме с треугольником рядом с объектом Slingshot и выберите Base. Установите параметры компонента Transform объекта Base, как показано ниже:

```
Base (Cylinder) P:[ 0, 1, 0 ] R:[ 0, 0, 0 ] S:[ 0.5, 1, 0.5 ]
```

3. Не снимая выделения с объекта Base, щелкните на ярлыке с изображением шестеренки рядом с компонентом Capsule Collider в инспекторе и в открывшемся меню выберите пункт Remove Component (Удалить компонент), как показано на рис. 29.3. В результате компонент Collider будет удален из объекта Base.

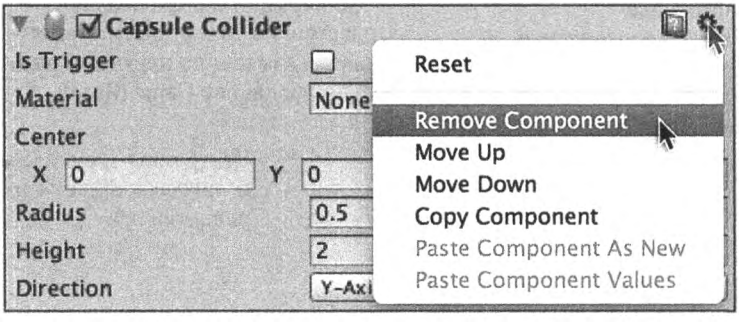


Рис. 29.3. Удаление компонента Collider

4. Создайте новый материал с именем Mat_Slingshot, выберите в поле Albedo светло-желтый цвет (или любой другой, какой пожелаете) и установите ползунок Smoothness в позицию 0. Перетащите материал Mat_Slingshot на объект Base, чтобы применить к нему материал.

5. Выберите объект Base в панели Hierarchy (Иерархия) и создайте его копию, нажав комбинацию Command-D на клавиатуре (Ctrl+D в Windows) или выбрав в главном меню пункт Edit > Duplicate (Правка > Дублировать). Создавая копию, вы гарантируете, что она также будет дочерним объектом для Slingshot, сохранит материал Mat_Slingshot и не будет иметь коллайдера.

6. Измените имя новой копии с Base (1) на LeftArm. Установите параметры компонента Transform объекта LeftArm, как показано ниже:

```
LeftArm (Cylinder) P:[ 0, 3, 1 ] R:[ 45, 0, 0 ] S:[ 0.5, 1.414, 0.5 ]
```

Этот объект будет служить левым рогом рогатки.

7. Выберите объект LeftArm в иерархии и создайте его копию (Command-D или Ctrl+D). Переименуйте этот экземпляр в RightArm. Установите параметры компонента Transform объекта RightArm, как показано ниже:

```
RightArm (Cylinder) P:[ 0, 3, -1 ] R:[ -45, 0, 0 ] S:[ 0.5, 1.414, 0.5 ].
```

8. Выберите объект Slingshot в иерархии. Добавьте в него компонент сферического коллайдера (Component > Physics > Sphere Collider (Компонент > Физика > Сферический коллайдер)). Установите параметры компонента Sphere Collider, как показано на рис. 29.4 (Is Trigger = true, Center = [0, 4, 0], Radius = 3).

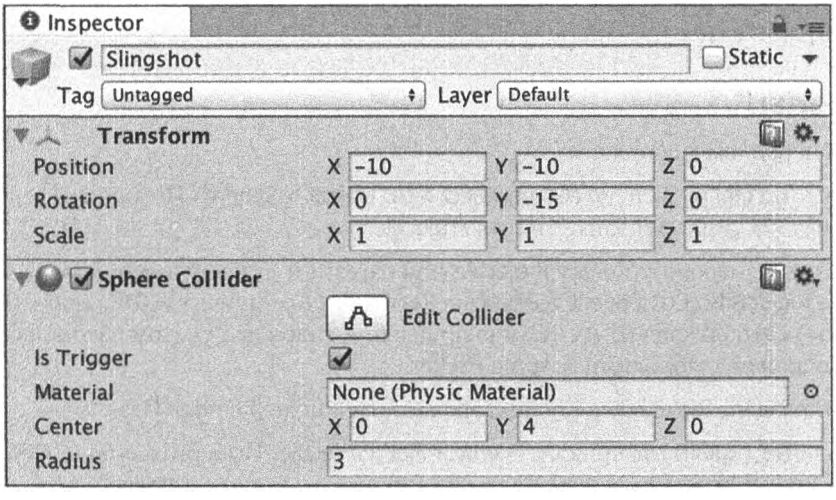


Рис. 29.4. Настройки компонента Sphere Collider объекта Slingshot

Как вы уже знаете, коллайдер с установленным флажком Is Trigger превращается в *триггер*. Триггеры участвуют в имитации законов физики в Unity и посылают уведомления, когда другие коллайдеры или триггеры оказываются в их зоне действия. Однако другие объекты не отскакивают от триггеров, как от обычных коллайдеров. Этот большой сферический коллайдер будет использоваться для взаимодействия Slingshot с мышью.

9. Установите параметры компонента Transform объекта Slingshot, как показано ниже:

```
Slingshot (Empty) P:[ -10, -10, 0 ] R:[ 0, -15, 0 ] S:[ 1, 1, 1 ]
```

Благодаря этим значениям рогатка будет касаться левого края экрана, а угол поворота −15° относительно оси Y придаст глубину даже при использовании камеры с ортографической проекцией.

10. Добавьте точку выстрела рогатки, чтобы определить место, откуда будут вылетать снаряды. Щелкните правой кнопкой на Slingshot в иерархии и выберите в контекстном меню пункт Create Empty (Создать пустой), чтобы создать новый, пустой игровой объект, вложенный в Slingshot. Переименуйте этот игровой объект в LaunchPoint. Установите параметры компонента Transform объекта LaunchPoint, как показано ниже:

```
LaunchPoint (Empty) P:[ 0, 4, 0 ] R:[ 0, 15, 0 ] S:[ 1, 1, 1 ]
```

Поворот на 15° относительно оси Y сориентирует LaunchPoint по направлениям осей XYZ в мировых координатах (то есть компенсирует поворот рогатки на −15°). Если выбрать инструмент Move (Перемещение), нажав клавишу W на клавиатуре, вы сможете увидеть местоположение и ориентацию LaunchPoint в панели Scene (Сцена).[1]

11. Сохраните сцену.

Снаряды

Теперь перейдем к снарядам.

1. Создайте сферу в сцене (GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера)) и дайте ей имя Projectile.
2. Выберите игровой объект Projectile в иерархии и присоедините к нему компонент Rigidbody (Component > Physics > Rigidbody (Компонент > Физика > Твердое тело)). Это обеспечит имитацию поведения снаряда с учетом законов физики по аналогии с яблоками в Apple Picker.

   a. В разделе Rigidbody в инспекторе введите число 5 в поле Mass.

3. Создайте новый материал с именем Mat_Projectile. Выберите в поле Albedo для Mat_Projectile темно-серый цвет. Установите ползунок Metallic в значение 0,5 и ползунок Smoothness в значение 0,65, чтобы придать снаряду вид металлического шарика. Примените материал Mat_Projectile к игровому объекту Projectile в иерархии.
4. Перетащите Projectile из панели Hierarchy (Иерархия) в панель Project (Проект), чтобы создать шаблон. Удалите экземпляр Projectile, оставшийся в панели Hierarchy (Иерархия).

---

[1] В левом верхнем углу окна Unity есть две кнопки: одна переключается между значениями Pivot (В опорной точке) и Center (В центре), а другая — между значениями Local (Локальная) и Global (Глобальная). Кнопка Local/Global определяет вид ориентации маркеров в локальных или глобальных координатах. Попробуйте выбрать инструмент Move (W), предварительно выделив повернутый объект (например, Slingshot), и попереключать эти кнопки, чтобы увидеть, какой эффект они оказывают на позиционирование маркеров.

После выполнения перечисленных действий содержимое панелей Project (Проект) и Hierarchy (Иерархия) должно выглядеть так, как показано на рис. 29.5.

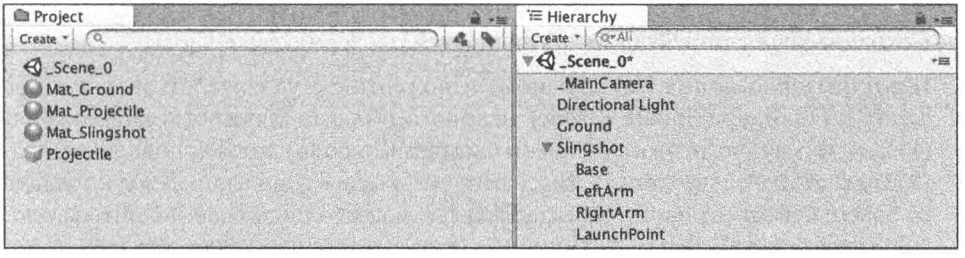


Рис. 29.5. Содержимое панелей Project (Проект) и Hierarchy (Иерархия) в данный момент. Звездочка рядом с именем сцены _Scene_0 подсказывает, что ее нужно сохранить

5. Сохраните сцену.

Программный код прототипа

Теперь, когда все художественные ресурсы на месте, можно начинать добавлять в проект программный код. В первую очередь займемся сценарием для рогатки Slingshot, который будет реагировать на события от мыши, создавать снаряды и стрелять ими. Писать сценарий мы будем с использованием итеративного подхода: сначала добавим небольшой фрагмент кода, протестируем его и только потом приступим к добавлению следующего фрагмента. Начав писать свои сценарии, вы по достоинству оцените этот фантастический подход: реализовать что-то небольшое и простое, протестировать, реализовать что-то еще, повторить.

Создание класса Slingshot

Выполните следующие шаги, чтобы создать класс `Slingshot`:

1. Создайте новый сценарий на C# с именем Slingshot (Assets > Create > C# Script (Ресурсы > Создать > Сценарий C#)). Подключите его к игровому объекту Slingshot в иерархии и откройте в MonoDevelop. Введите следующий код и удалите все лишние строки, которые добавляются по умолчанию:

```
using UnityEngine;
using System.Collections;

public class Slingshot : MonoBehaviour {

    void OnMouseEnter() {
        print("Slingshot:OnMouseEnter()");
    }
```

```
    void OnMouseExit() {
        print("Slingshot:OnMouseExit()");
    }
}
```

2. Сохраните сценарий Slingshot в MonoDevelop и вернитесь в Unity.
3. Щелкните на кнопке Play (Играть) и подвигайте указателем мыши в пределах коллайдера Sphere Collider игрового объекта Slingshot в панели Game (Игра). В результате этого в панели Console (Консоль) должен появиться текст `"Slingshot:OnMouseEnter()"`. Выведите указатель мыши за пределы коллайдера Sphere Collider игрового объекта Slingshot; в консоли должен появиться текст `"Slingshot:OnMouseExit()"`. Функции `OnMouseEnter()` и `OnMouseExit()` в этом сценарии вызываются автоматически любыми коллайдерами или триггерами.

Это лишь первый шаг на пути создания сценария, который будет выстреливать снарядами, и мы не будем спешить, потому что важно идти к намеченной цели последовательно и небольшими шагами.

Демонстрация активного состояния рогатки

Далее добавим подсветку, чтобы игрок мог видеть, когда рогатка находится в активном состоянии:

1. Выберите LaunchPoint в иерархии. Добавьте в LaunchPoint компонент Halo (Component > Effects > Halo (Компонент > Эффекты > Гало)), который создаст эффект светящегося ореола вокруг LaunchPoint. Введите число 3 в поле Size эффекта гало и выберите светло-серый цвет в поле Color, чтобы сделать ореол видимым (я у себя выбрал значения [r:191, g:191, b:191, a:255]).
2. Теперь добавьте в сценарий Slingshot следующий код. Кстати, сейчас самый удобный момент закомментировать инструкции `print()`, добавленные для тестирования на предыдущем шаге:

```
public class Slingshot : MonoBehaviour {
    public GameObject         launchPoint;

    void Awake() {
        Transform launchPointTrans = transform.Find("LaunchPoint");        // a
        launchPoint = launchPointTrans.gameObject;
        launchPoint.SetActive( false );                                    // b
    }

    void OnMouseEnter() {
        //print("Slingshot:OnMouseEnter()");
        launchPoint.SetActive( true );                                     // b
    }

    void OnMouseExit() {
        //print("Slingshot:OnMouseExit()");
        launchPoint.SetActive( false );                                    // b
    }
}
```

a. `transform.Find("LaunchPoint")` найдет дочерний объект с именем `LaunchPoint`, вложенный в Slingshot, и вернет его компонент Transform. Следующая строка получит игровой объект `GameObject`, владеющий этим компонентом Transform, и сохранит ссылку на него в поле `launchPoint`.

b. Метод `SetActive()` игровых объектов, таких как `launchPoint`, сообщает игре, должна ли она игнорировать их. Подробнее об этом я расскажу чуть ниже.

3. Сохраните сценарий Slingshot, вернитесь в Unity и щелкните на кнопке Play (Играть). Если теперь попробовать вводить указатель мыши в область действия сферического коллайдера-триггера объекта Slingshot и выводить из нее, будет включаться и выключаться ореол, показывающий границы, в которых игрок может взаимодействовать с рогаткой.

Как было сказано в подпункте *b* выше, метод `SetActive()` игровых объектов, таких как `launchPoint`, сообщает игре, должна ли она игнорировать их. Неактивный игровой объект не отображается на экране и его функции, такие как `Update()` или `OnCollisionEnter()`, не вызываются игрой. Сам объект все также будет находиться в памяти, просто будет исключен из активной части игры. Флажок в инспекторе, в верхней его части, слева от имени игрового объекта, показывает, является ли данный игровой объект активным (рис. 29.6).

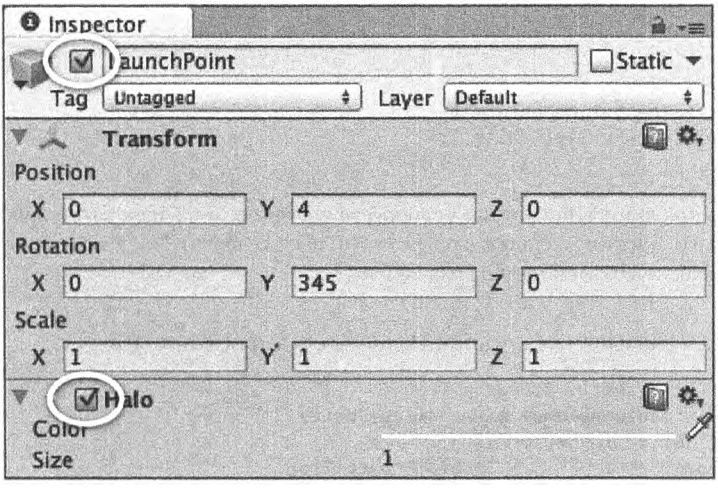


Рис. 29.6. Флажки, управляющие активностью игрового объекта и компонента

Компоненты имеют похожий флажок. Он определяет активность компонента. Для большинства компонентов (например, Renderer и разных коллайдеров) этим признаком можно управлять программно (например, `Renderer.enabled = false`), но по каким-то причинам компонент Halo недоступен программному коду, то есть им нельзя управлять из сценария на C#. Вам иногда придется сталкиваться с подобными несогласованностями и использовать обходное решение. В данном

случае мы не можем выключить компонент Halo непосредственно, поэтому деактивируем игровой объект, содержащий его.

4. Сохраните сцену.

Создание снаряда

Следующий шаг — создание снаряда, когда пользователь нажмет кнопку мыши с номером 0.

НЕ ИЗМЕНЯЙТЕ СОДЕРЖИМОЕ МЕТОДОВ OnMouseEnter() И OnMouseExit() В СЛЕДУЮЩЕМ ЛИСТИНГЕ! Об этом уже говорилось в предыдущих главах, но я повторю.

В листинге с кодом сценария Slingshot вы увидите инструкции `OnMouseEnter()` и `OnMouseExit()`, сопровождаемые символами `{ ... }` (фигурные скобки и многоточие между ними). Чем более сложные игры мы пишем, тем длиннее получаются сценарии. Всякий раз, увидев имя функции, существовавшей прежде, за которым следуют символы `{ ... }`, имейте в виду, что весь код, имевшийся внутри этих фигурных скобок в предыдущем листинге, должен остаться без изменений. В этом примере функции `OnMouseEnter()` и `OnMouseExit()` должны оставаться в своем прежнем виде:

```
void OnMouseEnter() {
    //print("Slingshot:OnMouseEnter()");
    launchPoint.SetActive( true );
}

void OnMouseExit() {
    //print("Slingshot:OnMouseExit()");
    launchPoint.SetActive( false );
}
```

Помните это правило. Многоточие в любом месте в коде означает, что я использовал его, чтобы сократить листинг и скрыть код, который вы уже ввели. Последовательность `{ ... }` не является фактическим кодом на языке C#.

1. Добавьте в Slingshot следующий код, выделенный жирным:

```
public class Slingshot : MonoBehaviour {
    // поля, устанавливаемые в инспекторе Unity
    [Header("Set in Inspector")]                                    // a
    public GameObject          prefabProjectile;

    // поля, устанавливаемые динамически
    [Header("Set Dynamically")]                                     // a
    public GameObject          launchPoint;
    public Vector3             launchPos;                           // b
    public GameObject          projectile;                          // b
    public bool                aimingMode;                          // b

    void Awake() {
        Transform launchPointTrans = transform.FindChild("LaunchPoint");
        launchPoint = launchPointTrans.gameObject;
```

```
        launchPoint.SetActive( false );
        launchPos = launchPointTrans.position;                    // c
    }

    void OnMouseEnter() { ... } // Не изменяйте OnMouseEnter()

    void OnMouseExit() { ... }  // Не изменяйте OnMouseExit()

    void OnMouseDown() {                                          // d
        // Игрок нажал кнопку мыши, когда указатель находился над рогаткой
        aimingMode = true;
        // Создать снаряд
        projectile = Instantiate( prefabProjectile ) as GameObject;
        // Поместить в точку launchPoint
        projectile.transform.position = launchPos;
        // Сделать его кинематическим
        projectile.GetComponent<Rigidbody>().isKinematic = true;
    }
}
```

a. Код в квадратных скобках, как здесь, называется *атрибутом компилятора* и содержит определенные инструкции для Unity или компилятора. В данном случае атрибут `Header` сообщает движку Unity, что он должен создать заголовок для этого сценария в инспекторе. Сохраните этот код, выберите Slingshot в панели Hierarchy (Иерархия) и посмотрите на компонент Slingshot (Script). Вы увидите заголовки, делящие общедоступные поля на два раздела: один для полей, которые устанавливаются в инспекторе, а другой для полей, которые устанавливаются динамически, в процессе игры.

   В этом примере, прежде чем запустить игру, вы должны установить значение в поле `prefabProjectile` (ссылку на шаблон, из которого будут создаваться все снаряды) в инспекторе Unity. Значения других полей будут устанавливаться динамически. Заголовки позволяют легко отличать подобные поля в инспекторе.

b. Взгляните теперь на другие новые поля:

   - `launchPos` хранит трехмерные мировые координаты `launchPoint`.
   - `Projectile` — это ссылка на вновь созданный экземпляр Projectile.
   - `aimingMode` — обычно имеет значение `false` и получает значение `true`, когда игрок нажимает кнопку мыши с номером 0, когда указатель находится над объектом Slingshot. Эта переменная управляет поведением остального кода. В следующем разделе мы напишем код в методе `Update()` класса `Slingshot`, который будет выполняться только при выполнении условия `aimingMode == true`.

c. В метод `Awake()` добавлена еще одна строка, устанавливающая координаты `launchPos`.

d. Метод `OnMouseDown()` содержит большую часть изменений в этом листинге.

Метод `OnMouseDown()` вызывается только в одном кадре, следующем сразу за нажатием кнопки мыши над компонентом коллайдера игрового объекта Slingshot, то есть этот метод может быть вызван, только если указатель мыши находится в допустимой начальной позиции. Он создает экземпляр `prefabProjectile` и сохраняет ссылку на него в `projectile`. Затем `projectile` помещается в точку `launchPos`. Наконец, свойству `isKinematic` твердого тела Rigidbody снаряда присваивается значение `true`. Когда твердое тело объявляется кинематическим, оно не перемещается физическим движком автоматически, но все еще участвует в имитации (то есть кинематическое твердое тело не перемещается под действием силы тяжести или в результате столкновений, но может вызывать автоматическое перемещение других, некинематических твердых тел).

2. Сохраните сценарий и вернитесь в Unity. Выберите Slingshot в панели Hierarchy (Иерархия) и в поле `prefabProjectile` выберите шаблон Projectile из панели Project (Проект), щелкнув на пиктограмме с изображением мишени, справа от `prefabProjectile`, или перетащив шаблон Projectile из панели Project (Проект) на поле `prefabProjectile` в инспекторе.

3. Щелкните на кнопке Play (Играть), наведите указатель мыши на область рогатки и щелкните по ней — на экране появится экземпляр Projectile.

4. Не будем останавливаться на достигнутом и продолжим. Добавьте в класс `Slingshot` следующее поле и метод `Update()`:

```
public class Slingshot : MonoBehaviour {
    // поля, устанавливаемые в инспекторе Unity
    [Header("Set in Inspector")]
    public GameObject          prefabProjectile;
    public float               velocityMult = 8f;                    // a

    // поля, устанавливаемые динамически
    [Header("Set Dynamically")]
    ...
    public bool                aimingMode;

    private Rigidbody projectileRigidbody;                           // a

    void Awake() { ... }
    ...

    void OnMouseDown() {
        ...
        // Сделать его кинематическим
        projectileRigidbody = projectile.GetComponent<Rigidbody>();  // a
        projectileRigidbody.isKinematic = true;                      // a
    }

    void Update() {
        // Если рогатка не в режиме прицеливания, не выполнять этот код
        if (!aimingMode) return;                                     // b

        // Получить текущие экранные координаты указателя мыши
```

```
        Vector3 mousePos2D = Input.mousePosition;                          // c
        mousePos2D.z = -Camera.main.transform.position.z;
        Vector3 mousePos3D = Camera.main.ScreenToWorldPoint( mousePos2D );

        // Найти разность координат между launchPos и mousePos3D
        Vector3 mouseDelta = mousePos3D-launchPos;
        // Ограничить mouseDelta радиусом коллайдера объекта Slingshot     // d
        float maxMagnitude = this.GetComponent<SphereCollider>().radius;
        if (mouseDelta.magnitude > maxMagnitude) {
            mouseDelta.Normalize();
            mouseDelta *= maxMagnitude;
        }

        // Передвинуть снаряд в новую позицию
        Vector3 projPos = launchPos + mouseDelta;
        projectile.transform.position = projPos;
        if ( Input.GetMouseButtonUp(0) ) {                                 // e
            // Кнопка мыши отпущена
            aimingMode = false;
            projectileRigidbody.isKinematic = false;
            projectileRigidbody.velocity = -mouseDelta * velocityMult;
            projectile = null;
        }
    }
}
```

a. Обязательно добавьте все три этих изменения. Две последние строки в методе `OnMouseDown()` заменяют строку, введенную в предыдущем листинге.

b. Если рогатка не в режиме прицеливания, вернуть управление и не выполнять оставшийся код.

c. Преобразовать координаты указателя мыши из экранных координат в мировые. Происходящее в этих строках описывалось в предыдущей главе.

d. Этот код ограничивает перемещение снаряда так, чтобы его центр оставался в пределах коллайдера Sphere Collider игрового объекта Slingshot. Подробнее об этом рассказывается ниже.

e. `Input.GetMouseButtonUp(0)` — еще один способ прочитать состояние кнопок мыши.

Большая часть из всего этого объясняется в комментариях в коде; но я считаю, что мы должны подробнее остановиться на векторной математике, используемой в вычислениях.

Как показано на рис. 29.7, сложение и вычитание векторов производится покомпонентно. На рисунке изображена двумерная система координат, но это правило действует также в отношении трехмерных векторов. Вычитание компонентов X и Y векторов A и B выполняется по отдельности, в результате получается новый двумерный вектор `Vector2`, определяемый как `Vector2( 2-5, 8-3 )`, или `Vector2( -3, 5 )`. Рисунок показывает, что разность A–B дает векторное расстояние между A и B, которое одновременно определяет расстояние и направление

для перемещения из точки B в точку A. Для простоты запоминания, что на что указывает, используйте мнемонику: «A минус B дает направление на A».

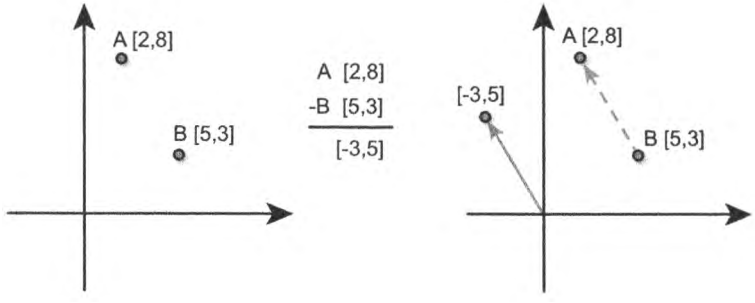


Рис. 29.7. Вычитание двумерных векторов: *A минус B дает направление на A*

Это важное обстоятельство для метода `Update()`, потому что снаряд `projectile` нужно переместить вдоль вектора `mouseDelta` из позиции `launchPos` в текущую позицию указателя мыши `mousePos3D`. Однако расстояние, на которое может переместиться снаряд вдоль `mouseDelta`, ограничено значением `maxMagnitude` — величиной радиуса коллайдера `SphereCollider` объекта Slingshot (сейчас радиус установлен в инспекторе равным 3 м).

Если вектор `mouseDelta` длиннее `maxMagnitude`, его длина усекается до `maxMagnitude`. Это достигается первым вызовом `mouseDelta.Normalize()` (который устанавливает длину вектора `mouseDelta` равной 1, но сохраняет его направление) и умножением `mouseDelta` на `maxMagnitude`.

Снаряд `projectile` перемещается в новое местоположение, и если сейчас запустить игру, вы увидите, что снаряд перемещается вслед за указателем мыши, при этом дальность перемещения ограничивается заданным радиусом.

`Input.GetMouseButtonUp(0)` возвращает `true` только в первом кадре, следующем за событием отпускания левой кнопки мыши (кнопки с номером 0).[1] Это означает, что инструкция `if` в конце `Update()` выполнится в кадре, в котором была отпущена кнопка мыши. В этом кадре:

[1] Вот почему `Input.GetMouseButtonUp()`, `Input.GetKeyDown()` и другие похожие методы класса `Input`, имена которых оканчиваются на `Up` или `Down`, не следует использовать внутри `FixedUpdate()`. Метод `FixedUpdate` вызывается точно 50 раз в секунду, тогда как `Update` (или кадр изображения) может вызываться до 400 раз в секунду. Если так случится, что между двумя вызовами `FixedUpdates` произойдет несколько вызовов `Update` и в любом из них, кроме самого последнего, `Input.GetKeyDown()` примет значение `true`, в последующем вызове `FixedUpdate()` он вернет `false`. В этом случае может сложиться впечатление, что клавиатура или мышь работает с перебоями, потому что игра будет откликаться не на каждое нажатие. Перенесите все вызовы методов `...Up` и `...Down` класса `Input` в метод `Update()`, и вы избавитесь от этой проблемы.

- Признак `aimingMode` получит значение `false`.
- Твердое тело Rigidbody снаряда станет некинематическим, что позволит ему перемещаться в игровом пространстве под действием физических сил.
- Твердое тело Rigidbody снаряда получит начальную скорость, пропорциональную расстоянию снаряда от `launchPos`.
- Наконец, переменной `projectile` будет присвоено значение `null`. Эта операция не удалит созданный экземпляр `Projectile`, но освободит поле `projectile` для записи в него ссылки на другой экземпляр, когда пользователь решит произвести очередной выстрел.

5. Щелкните на кнопке Play (Играть) и посмотрите, как действует рогатка. Достаточно ли высокую скорость получает снаряд? Поэкспериментируйте со значением `velocityMult` в инспекторе и подберите оптимальное, на ваш взгляд, значение. Я выбрал значение 10. Не забудьте остановить игру в Unity и снова ввести подобранное значение, чтобы зафиксировать его.
6. Сохраните сцену.

На данном этапе экземпляры Projectile очень быстро улетают за границы экрана. Давайте заставим камеру следить за полетом снаряда.

Слежение за полетом снаряда

Наша следующая задача — заставить главную камеру _MainCamera сопровождать полет снаряда, но решить ее немного сложнее, чем кажется. Фактически мы должны реализовать следующее поведение:

A. Камера находится в начальной позиции и никуда не перемещается, пока рогатка Slingshot не в режиме прицеливания.

B. После выстрела камера следует за снарядом (с небольшим отставанием для большей гладкости изображения).

C. С перемещением камеры вверх должно увеличиваться значение `Camera.orthographicSize`, чтобы земля оставалась в поле зрения.

D. Когда снаряд достигает цели, камера должна прекратить следование за ним и вернуться в начальную позицию.

Выполните следующие шаги:

1. Создайте новый сценарий на C# (**Assets > Create > C# Script** (Ресурсы > Создать > Сценарий C#)) с именем FollowCam.
2. Перетащите сценарий FollowCam на главную камеру _MainCamera в панели Hierarchy (Иерархия), чтобы сделать его компонентом камеры.
3. Щелкните дважды на сценарии FollowCam, чтобы открыть его, и введите следующий код:

```
using UnityEngine;
using System.Collections;

public class FollowCam : MonoBehaviour {
    static public GameObject POI; // Ссылка на интересующий объект         // a

    [Header("Set Dynamically")]
    public float             camZ; // Желаемая координата Z камеры

    void Awake() {
        camZ = this.transform.position.z;
    }

    void FixedUpdate () {
        // Однострочная версия if не требует фигурных скобок
        if (POI == null) return; // выйти, если нет интересующего объекта  // b

        // Получить позицию интересующего объекта
        Vector3 destination = POI.transform.position;
        // Принудительно установить значение destination.z равным camZ, чтобы
        //   отодвинуть камеру подальше
        destination.z = camZ;
        // Поместить камеру в позицию destination
        transform.position = destination;
    }
}
```

a. `POI` — это «point of interest», интересующий объект, за которым должна следить камера (например, экземпляр Projectile). Так как поле объявлено статическим и общедоступным (`static public`), одно и то же значение `POI` будет совместно использоваться всеми экземплярами класса `FollowCam` и станет доступно из любой точки в коде как `FollowCam.POI`. Так проще из сценария Slingshot сообщить главной камере _MainCamera, за каким экземпляром Projectile она должна следить.

b. Если `POI` имеет значение `null` (по умолчанию), метод `FixedUpdate()` тут же завершается и не выполняет остальной код.

Поле `camZ` хранит начальное значение координаты Z камеры. В `FixedUpdate()` камера перемещается в позицию `POI`, за исключением координаты Z, которая в каждом кадре получает значение `camZ`. (Это предотвращает чрезмерное приближение камеры к наблюдаемому объекту `POI`, когда тот может заполнить весь кадр или стать невидимым.) В данном случае решено использовать `FixedUpdate()` вместо `Update()`, потому что слежение осуществляется за снарядом, которым управляет физический движок PhysX, а как мы помним, обновление физики происходит синхронно с вызовами метода `FixedUpdate()`.

4. Откройте сценарий Slingshot и добавьте единственную строку, выделенную жирным в конце метода `Update()`:

```
public class Slingshot : MonoBehaviour {
    ...
    void Update() {
```

```
        ...
        if ( Input.GetMouseButtonUp(0) ) {
            ...
            projectileRigidbody.velocity = -mouseDelta * velocityMult;
            FollowCam.POI = projectile;
            projectile = null;
        }
    }
}
```

Эта новая строка записывает в общедоступное поле `FollowCam.POI` ссылку на вновь выпущенный снаряд. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и попробуйте щелкнуть на кнопке Play (Играть), чтобы увидеть, как все это выглядит.

Сразу бросаются в глаза следующие проблемы:

A. Если достаточно уменьшить масштаб, можно заметить, что снаряд улетает за край земли.

B. Если выстрелить в землю, снаряд не отскочит от нее и не прекратит полет, когда попадет в нее. Если приостановить игру сразу после выстрела, выбрать снаряд в иерархии (чтобы выделить его и вывести маркеры ориентации) и затем возобновить игру, вы увидите, что снаряд катится по земле без остановки.

C. После выстрела камера сразу же подпрыгивает на уровень снаряда, что создает неприятное ощущение.

D. Когда снаряд окажется на определенной высоте — или пересечет край земли, — вы будете видеть только небо, и вам будет сложно понять, насколько высоко летит снаряд.

Выполните следующие шаги, чтобы исправить описанные проблемы друг за другом (расположены в порядке от простых в исправлении к сложным).

Сначала устраним проблему **A**, определив параметры компонента Transform объекта Ground, как: P:[100, -10, 0] R:[0, 0, 0] S:[400, 1, 4]. Благодаря этому прямоугольник, представляющий землю, будет простираться далеко вправо.

Чтобы устранить проблему **B**, нужно добавить ограничения в компонент Rigidbody и физические свойства материала шаблона снаряда Projectile:

1. Выберите шаблон Projectile в панели Project (Проект).
2. В компоненте Rigidbody выберите в раскрывающемся списке способов определения столкновений Collision Detection значение Continuous, соответствующее непрерывному определению. Чтобы получить дополнительную информацию о способах определения столкновений, щелкните на кнопке вызова справки в правом верхнем углу раздела компонента Rigidbody. Если говорить в двух словах, непрерывное определение столкновений — Continuous — увеличивает нагрузку на процессор по сравнению с дискретным, но обеспечивает более высокую точность для объектов, движущихся с высокой точностью, таких как снаряд.

3. Также в компоненте Rigidbody объекта Projectile:
   a. Щелкните на пиктограмме с изображением треугольника рядом с названием подраздела Constraints, чтобы раскрыть его.
   b. Установите флажок Freeze Position Z.
   c. Установите флажки Freeze Rotation X, Y и Z.

   Флажок Freeze Position Z предотвратит приближение снаряда к камере или удаление от нее (удержит снаряд на том же расстоянии от камеры, на каком находятся также земля и замок, который мы добавим далее). Флажки Freeze Rotation X, Y и Z предотвратят вращение снаряда.
4. Сохраните сцену, щелкните на кнопке Play (Играть) и попробуйте выстрелить еще раз.

   Настройки, выполненные в компоненте Rigidbody, предотвратят бесконечное качение снаряда, но его поведение все еще выглядит неестественно. Все мы с рождения живем в мире, подчиняющемся законам физики, и интуитивно замечаем неестественность поведения, не соответствующего реальному физическому миру. То же относится к нашим игрокам, а это значит, что, несмотря на сложность математического аппарата, применяемого для моделирования физической системы, если физика вашей игры будет выглядеть достаточно реалистичной для игроков, вам не придется объяснять им эту математику. Добавление физического материала поможет сделать физическое поведение ваших объектов более реалистичным.
5. В меню Unity выберите пункт Assets > Create > Physic Material (Ресурсы > Создать > Физический материал).
6. Дайте этому физическому материалу имя PMat_Projectile.
7. Щелкните на PMat_Projectile и в инспекторе введите в поле bounciness число 1.
8. В панели Project (Проект) перетащите PMat_Projectile на шаблон Projectile, чтобы применить его к Projectile.SphereCollider.
9. Сохраните сцену, щелкните на кнопке Play (Играть) и попробуйте выстрелить снова.

Выбрав Projectile, вы должны увидеть в инспекторе, что PMat_Projectile назначен как материал коллайдера Sphere Collider. Теперь после выстрела снаряд подскакивает и останавливается, а не просто скользит по поверхности земли.

Проблема **C** имеет два решения: замедление реакции камеры посредством интерполяции и добавление ограничения на ее местоположение. Применим оба:

1. Чтобы замедлить реакцию камеры, добавьте в FollowCam следующие строки, выделенные жирным:

```
public class FollowCam : MonoBehaviour {
    static public GameObject POI; // The static point of interest

    [Header("Set in Inspector")]
```

```
    public float            easing = 0.05f;

    [Header("Set Dynamically")]
    ...
    void FixedUpdate () {
        // Однострочная версия if не требует фигурных скобок
        if (POI == null) return; // выйти, если нет интересующего объекта

        // Получить позицию интересующего объекта
        Vector3 destination = POI.transform.position;
        // Определить точку между текущим местоположением камеры и destination
        destination = Vector3.Lerp(transform.position, destination, easing);
        // Принудительно установить значение destination.z равным camZ, чтобы
        //   отодвинуть камеру подальше
        destination.z = camZ;
        // Поместить камеру в позицию destination
        transform.position = destination;
    }
}
```

Метод `Vector3.Lerp()` выполняет интерполяцию между двумя точками, возвращая взвешенное среднее. Если полю `easing` присвоить 0, `Lerp()` вернет первую точку (`transform.position`); если полю `easing` присвоить 1, `Lerp()` вернет вторую точку (`destination`). Если полю `easing` присвоить любое другое значение в диапазоне между 0 и 1, `Lerp()` вернет точку, находящуюся между заданными (при значении 0,5 в поле `easing` будет возвращена точка, лежащая точно посередине). Определение `easing = 0.05f` означает, что в каждом вызове `FixedUpdate` (то есть при каждом обновлении, выполняемом физическим движком, которое происходит с частотой 50 раз в секунду) камера должна перемещаться примерно на 5% от расстояния между текущим ее местоположением и местоположением `POI`. Так как `POI` постоянно перемещается, камера плавно следует за ним. Попробуйте поэкспериментировать со значением `easing`, чтобы понять, как оно влияет на поведение камеры. Такое применение метода `Lerp()` является очень упрощенной формой *линейной интерполяции*. Больше информации о линейной интерполяции вы найдете в приложении Б «Полезные идеи».

2. Добавьте ограничения на позицию камеры, включив в FollowCam следующие строки, выделенные жирным:

```
public class FollowCam : MonoBehaviour {
    ...
    [Header("Set in Inspector")]
    public float                easing = 0.05f;
    public Vector2              minXY = Vector2.zero;

    [Header("Set Dynamically")]
    ...
    void FixedUpdate () {
        // Однострочная версия if не требует фигурных скобок
        if (POI == null) return; // выйти, если нет интересующего объекта

        // Получить позицию интересующего объекта
        Vector3 destination = POI.transform.position;
```

```
        // Ограничить X и Y минимальными значениями
        destination.x = Mathf.Max( minXY.x, destination.x );
        destination.y = Mathf.Max( minXY.y, destination.y );
        // Определить точку между текущим местоположением камеры и destination
        ...
    }
}
```

По умолчанию вектор `Vector2 minXY` имеет значение [0, 0], которое прекрасно соответствует нашим потребностям. Функция `Mathf.Max()` выбирает максимальное из двух полученных значений. В момент выстрела снаряд имеет отрицательную координату X, поэтому `Mathf.Max()` гарантирует, что камера никогда не сместится левее X = 0 в отрицательную область. Аналогично, второй вызов `Mathf.Max()` не позволит камере опуститься ниже плоскости Y = 0, когда координата Y снаряда станет отрицательной. (Напомню, что координата Y объекта Ground равна –10.)

Чтобы устранить проблему **D**, требуется динамически корректировать размер проекции `orthographicSize` камеры.

1. Добавьте в сценарий FollowCam следующие строки, выделенные жирным:

```
public class FollowCam : MonoBehaviour {
    ...
    void FixedUpdate () {
        ...
        // Поместить камеру в позицию destination
        transform.position = destination;
        // Изменить размер orthographicSize камеры, чтобы земля
        //   оставалась в поле зрения
        Camera.main.orthographicSize = destination.y + 10;
    }
}
```

Мы знаем, что только что добавленные строки `Mathf.Max()` не позволят значению `destination.y` стать меньше 0. Соответственно, минимальное значение, которое может принимать `orthographicSize`, равно 10, при этом поле зрения камеры будет при необходимости расширяться, чтобы земля всегда оставалась в зоне видимости.

2. Щелкните дважды на объекте Ground в иерархии, чтобы уменьшить масштаб в панели Scene (Сцена) и отобразить в ней объект Ground целиком.
3. Выберите _MainCamera, щелкните на кнопке Play (Играть) и произведите выстрел вертикально вверх. В панели Scene (Сцена) вы увидите, как постепенно расширяется поле зрения камеры с увеличением высоты полета снаряда.
4. Сохраните сцену.

Создание иллюзии движения

Сейчас FollowCam отлично перемещает камеру, но все еще сложно оценить скорость полета снаряда, особенно на большой высоте. Чтобы устранить эту проблему, воспользуемся приемом создания иллюзии движения. Иллюзия движения возникает,

когда мы видим другие быстро перемещающиеся предметы, и в двумерных играх основана на идее многоуровневого скроллинга. Суть многоуровневого скроллинга состоит в том, чтобы в двумерных играх близкие предметы перемещать относительно камеры с большей скоростью, а удаленные — с меньшей. Полноценное обсуждение идеи многоуровневого скроллинга выходит далеко за рамки этого учебного примера, однако мы можем создать простую иллюзию движения, добавив и разместив в случайном порядке несколько облаков на небе. Когда снаряд будет пролетать мимо них, у игрока будет создаваться иллюзия движения.

Создание облаков

Чтобы воплотить идею в жизнь, создадим несколько простых облаков:

1. Создайте новую сферу (**GameObject > 3D Object > Sphere** (Игровой объект > 3D объект > Сфера)).
   a. Наведите указатель мыши на имя компонента **Sphere Collider** сферы в инспекторе. Щелкните правой кнопкой мыши и в контекстном меню выберите пункт **Remove Component** (Удалить компонент).
   b. Установите координаты сферы в **Transform** равными P:[0, 0, 0], чтобы она была видима в обеих панелях, **Game** (Игра) и **Scene** (Сцена).
   c. Переименуйте объект **Sphere** в **CloudSphere**.
2. Создайте новый материал с именем **Mat_Cloud** (**Assets > Create > Material** (Ресурсы > Создать > Материал)).
   a. Перетащите **Mat_Cloud** на **CloudSphere** и затем выберите **Mat_Cloud** в панели **Project** (Проект).
   b. В раскрывающемся списке **Shader** в инспекторе выберите значение **Legacy Shaders > Self-Illumin > Diffuse**. Материалы с этим шейдером не только отражают направленный свет, но и излучают свой.
   c. Щелкните на цветовой шкале в поле **Main Color** в инспекторе и выберите 50% серый цвет (в формате RGBA, в цветовой палитре Unity, он определяется как [128, 128, 128, 255]). Это должно придать облаку **CloudSphere** в панели **Game** (Игра) чуть более темный оттенок слева снизу и создать иллюзию облака при солнечном освещении.
   d. Перетащите **CloudSphere** в панель **Project** (Проект), чтобы создать шаблон.
   e. Удалите экземпляр **CloudSphere** из иерархии.
3. Создайте в сцене пустой игровой объект (**GameObject > Create Empty** (Игровой объект > Создать пустой)) с именем **Cloud**.
   a. Выберите **Cloud** в иерархии и установите координаты в компоненте **Transform**: P:[0, 0, 0].

b. В инспекторе для Cloud щелкните на кнопке Add Component (Добавить компонент) и выберите в открывшемся меню пункт New Script (Новый сценарий).

c. Дайте новому сценарию имя Cloud, не забыв при этом выбрать в поле Language язык C Sharp, и щелкните на кнопке Create and Add (Создать и добавить). В результате будет создан новый сценарий, уже подключенный к объекту Cloud.

Мы не будем создавать разные облака вручную, а сгенерируем их программно (то есть будем случайно создавать их в коде сценария). Именно так игры, подобные Minecraft, создают свои миры. Наш код создания облаков будет намного проще, чем аналогичный код в Minecraft, но он поможет вам обрести опыт случайного создания объектов программным способом.

4. Откройте сценарий Cloud в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Cloud : MonoBehaviour {
    [Header("Set in Inspector")]                                          // a
    public GameObject           cloudSphere;
    public int                  numSpheresMin = 6;
    public int                  numSpheresMax = 10;
    public Vector3              sphereOffsetScale = new Vector3(5,2,1);
    public Vector2              sphereScaleRangeX = new Vector2(4,8);
    public Vector2              sphereScaleRangeY = new Vector2(3,4);
    public Vector2              sphereScaleRangeZ = new Vector2(2,4);
    public float                scaleYMin = 2f;

    private List<GameObject>    spheres;                                  // b

    void Start () {
        spheres = new List<GameObject>();

        int num = Random.Range(numSpheresMin, numSpheresMax);             // c
        for (int i=0; i<num; i++) {
            GameObject sp = Instantiate<GameObject>( cloudSphere );       // d
            spheres.Add( sp );
            Transform spTrans = sp.transform;
            spTrans.SetParent( this.transform );

            // Выбрать случайное местоположение
            Vector3 offset = Random.insideUnitSphere;                     // e
            offset.x *= sphereOffsetScale.x;
            offset.y *= sphereOffsetScale.y;
            offset.z *= sphereOffsetScale.z;
            spTrans.localPosition = offset;                               // f

            // Выбрать случайный масштаб
            Vector3 scale = Vector3.one;                                  // g
            scale.x = Random.Range(sphereScaleRangeX.x, sphereScaleRangeX.y);
            scale.y = Random.Range(sphereScaleRangeY.x, sphereScaleRangeY.y);
            scale.z = Random.Range(sphereScaleRangeZ.x, sphereScaleRangeZ.y);

            // Скорректировать масштаб y по расстоянию x от центра
```

```
            scale.y *= 1 - (Mathf.Abs(offset.x) / sphereOffsetScale.x);    // h
            scale.y = Mathf.Max( scale.y, scaleYMin );

            spTrans.localScale = scale;                                    // i

        }
    }

    // Update вызывается в каждом кадре
    void Update () {
        if (Input.GetKeyDown(KeyCode.Space)) {                             // j
            Restart();
        }
    }

    void Restart() {                                                       // k
        // Удалить старые сферы, составляющие облако
        foreach (GameObject sp in spheres) {
            Destroy(sp);
        }
        Start();
    }
}
```

a. Все эти поля используются для настройки процедуры случайного создания облаков.

   - `numSpheresMin` / `numSpheresMax` — минимальное и максимальное (в действительности на 1 больше фактического) количество создаваемых экземпляров CloudSphere.
   - `sphereOffsetScale` — максимальное расстояние (положительное или отрицательное) CloudSphere от центра Cloud вдоль каждой оси.
   - `sphereScaleRangeX` / `Y` / `Z` — диапазон масштаба для каждой оси. По умолчанию создаются экземпляры CloudSphere, ширина которых больше высоты.
   - `scaleYMin` — в конце функции `Start()` масштаб по оси Y каждого экземпляра CloudSphere уменьшается в зависимости от удаленности от центра вдоль оси X. Благодаря этому толщина облаков будет уменьшаться к краям. `scaleYMin` — это наименьший возможный масштаб по оси Y (чтобы избежать появления слишком тонких облаков).

b. Список `List<GameObject> spheres` хранит ссылки на все экземпляры CloudSphere, созданные для данного облака Cloud.

c. Выбирается случайное количество экземпляров CloudSphere для включения в облако Cloud.

d. Создаются экземпляры CloudSphere и добавляются в список `spheres`. Затем свойство `transform` каждого экземпляра CloudSphere присваивается переменной `spTrans` и родителем компонента Transform каждого экземпляра CloudSphere устанавливается компонент Transform данного объекта Cloud. Вы-

ражение `this.transform` в данном случае идентично выражению `transform`; то есть ссылку `this` можно опустить.

e. Выбирается случайная точка внутри единичной сферы (то есть точка где-то внутри сферы с радиусом, равным единице, и с центром в начале координат: [0, 0, 0]). Каждая координата (X, Y, Z) этой точки затем умножается на соответствующее значение `sphereOffsetScale`.

f. Свойству `CloudSphere.transform.localPosition` присваивается смещение `offset`. `transform.position` всегда выражается в мировых координатах, тогда как `transform.localPosition` — в координатах относительно центра родителя (в данном случае объекта Cloud).

g. Случайно выбирается масштаб для каждой оси в отдельности. Вектор `sphereScaleRange` хранит в поле `x` минимальное значение, а в поле `y` — максимальное.

h. После выбора масштаба по осям изменяется масштаб по оси Y, в зависимости от смещения CloudSphere от центра Cloud вдоль оси X. Чем дальше сфера от центра облака, тем меньше масштаб по оси Y.

i. Значение `scale` присваивается свойству `localScale` экземпляра CloudSphere. Масштаб всегда определяется относительно родительского компонента Transform, поэтому нет поля `transform.scale`. Только `localScale` и `lossyScale`. Свойство `lossyScale`, доступное только для чтения, пытается возвращать масштаб в мировых координатах, при этом вы должны понимать, что это всего лишь оценка.

j. Этот фрагмент кода добавлен только для тестирования. Он обеспечивает вызов метода `Restart()` (см. примечание `// k`) в ответ на нажатие клавиши пробела.

k. Метод `Restart()` уничтожает все дочерние сферы CloudSphere и вызывает `Start()`, чтобы сгенерировать новое облако.

5. Сохраните сценарий Cloud и вернитесь в Unity.

6. Выберите Cloud в иерархии и в инспекторе присвойте полю cloudSphere сценария Cloud (Script) шаблон CloudSphere.

Щелкните на кнопке Play (Играть), и вы увидите случайно сгенерированное облако. При любом нажатии клавиши пробела будет вызван метод `Restart()`, уничтожено текущее облако и создано новое. Это позволит нажимать клавишу пробела и проверять разные настройки Cloud (Script) в инспекторе. Поэкспериментируйте и подберите настройки, которые покажутся вам оптимальными.

Как не потерять значения в инспекторе, подобранные в процессе экспериментов

Как вы уже видели в предыдущих главах, в момент остановки игры (повторным щелчком на кнопке Play (Играть)) для подготовки к запуску новой все измененные

параметры Cloud (Script) в инспекторе примут исходные значения. При желании этого можно избежать.

1. Пока игра продолжается, щелкните на кнопке с изображением шестеренки справа от имени компонента Cloud (Script). В открывшемся меню выберите пункт Copy Component (Копировать компонент).
2. Остановите воспроизведение (снова щелкнув на кнопке Play (Играть)).
3. Снова щелкните на кнопке с изображением шестеренки справа от имени компонента Cloud (Script) и в открывшемся меню выберите пункт Paste Component Values (Вставить значения в компонент).

В результате значения, подобранные в инспекторе в процессе игры, заменят исходные.

Комментирование тестового кода

Возможность нажимать пробел и создавать новые облака действительно нужна для тестирования, но после выбора оптимальных значений параметров Cloud (Script) в инспекторе тестовый код нужно удалить. Позднее он вам может еще пригодиться, поэтому не будем удалять его совсем, а просто закомментируем.

1. Откройте сценарий Cloud в MonoDevelop.
2. Закомментируйте все строки в методе `Update()`.

```
public class Cloud : MonoBehaviour {
    ...
    void Update () {
//      if (Input.GetKeyDown(KeyCode.Space)) {
//          Restart();
//      }
    }
    ...
}
```

Так как метод `Restart()` теперь не будет вызываться, его можно не комментировать.

Создание множества облаков

Теперь, научившись создавать одно облако, создадим несколько.

1. Создайте шаблон Cloud, перетащив игровой объект Cloud из иерархии в панель Project (Проект). Удалите экземпляр Cloud из иерархии.
2. Создайте новый, пустой игровой объект с именем CloudAnchor (GameObject > Create Empty (Игровой объект > Создать пустой)). Это даст нам игровой объект, который будет служить родителем для всех экземпляров Cloud, чтобы предотвратить переполнение панели Hierarchy (Иерархия) в процессе игры. Установите координаты в компоненте Transform объекта CloudAnchor: P:[0, 0, 0].

3. Создайте новый сценарий на C# с именем CloudCrafter и подключите его к _MainCamera. Это будет второй компонент сценария в _MainCamera — Unity допускает это при условии, что два сценария не конфликтуют друг с другом (например, не пытаются установить координаты одного и того же игрового объекта в каждом кадре). Так как FollowCam управляет перемещением камеры, а CloudCrafter просто добавляет на небо облака, они не должны конфликтовать между собой.

4. Откройте CloudCrafter в MonoDevelop и введите следующий код:

```
using UnityEngine;
using System.Collections;

public class CloudCrafter : MonoBehaviour {
    [Header("Set in Inspector")]
    public int            numClouds = 40;         // Число облаков
    public GameObject     cloudPrefab;            // Шаблон для облаков
    public Vector3        cloudPosMin = new Vector3(-50,-5,10);
    public Vector3        cloudPosMax = new Vector3(150,100,10);
    public float          cloudScaleMin = 1;      // Мин. масштаб каждого облака
    public float          cloudScaleMax = 3;      // Макс. масштаб каждого облака
    public float          cloudSpeedMult = 0.5f; // Коэффициент скорости облаков

    private GameObject[] cloudInstances;

    void Awake() {
        // Создать массив для хранения всех экземпляров облаков
        cloudInstances = new GameObject[numClouds];
        // Найти родительский игровой объект CloudAnchor
        GameObject anchor = GameObject.Find("CloudAnchor");
        // Создать в цикле заданное количество облаков
        GameObject cloud;
        for (int i=0; i<numClouds; i++) {
            // Создать экземпляр cloudPrefab
            cloud = Instantiate<GameObject>( cloudPrefab );
            // Выбрать местоположение для облака
            Vector3 cPos = Vector3.zero;
            cPos.x = Random.Range( cloudPosMin.x, cloudPosMax.x );
            cPos.y = Random.Range( cloudPosMin.y, cloudPosMax.y );
            // Масштабировать облако
            float scaleU = Random.value;
            float scaleVal = Mathf.Lerp( cloudScaleMin, cloudScaleMax, scaleU );
            // Меньшие облака (с меньшим значением scaleU) должны быть ближе
            // к земле
            cPos.y = Mathf.Lerp( cloudPosMin.y, cPos.y, scaleU );
            // Меньшие облака должны быть дальше
            cPos.z = 100 - 90*scaleU;
            // Применить полученные значения координат и масштаба к облаку
            cloud.transform.position = cPos;
            cloud.transform.localScale = Vector3.one * scaleVal;
            // Сделать облако дочерним по отношению к anchor
            cloud.transform.SetParent( anchor.transform );
            // Добавить облако в массив cloudInstances
            cloudInstances[i] = cloud;
```

```
        }
    }

    void Update() {
        // Обойти в цикле все созданные облака
        foreach (GameObject cloud in cloudInstances) {
            // Получить масштаб и координаты облака
            float scaleVal = cloud.transform.localScale.x;
            Vector3 cPos = cloud.transform.position;
            // Увеличить скорость для ближних облаков
            cPos.x -= scaleVal * Time.deltaTime * cloudSpeedMult;
            // Если облако сместилось слишком далеко влево...
            if (cPos.x <= cloudPosMin.x) {
                // Переместить его далеко вправо
                cPos.x = cloudPosMax.x;
            }
            // Применить новые координаты к облаку
            cloud.transform.position = cPos;
        }
    }
}
```

5. Сохраните сценарий CloudCrafter и вернитесь в Unity.
6. Перетащите шаблон Cloud из панели Project (Проект) в поле cloudPrefab компонента CloudCrafter (Script) главной камеры _MainCamera в инспекторе. Во всех остальных полях можно оставить значения по умолчанию.
7. Сохраните сцену.

   Метод `Awake()` класса `CloudCrafter` создает все облака и размещает их в сцене. Метод `Update()` смещает каждое облако чуть влево в каждом кадре. Когда облако оказывается левее точки `cloudPosMin.x`, оно переносится вправо, в позицию `cloudPosMax.x`.
8. Щелкните на кнопке Play (Играть), и вы увидите, как будет создано несколько облаков, перемещающихся поперек экрана.

Уменьшите масштаб в панели Scene (Сцена) и понаблюдайте за движением облаков. Теперь полет снаряда на фоне облаков после выстрела должен создавать более полную иллюзию движения.

Организация панели Project (Проект)

Теперь, после создания большого количества разных ресурсов, самое время поговорить об организации панели Project (Проект). В настоящий момент содержимое панели должно выглядеть так, как показано на рис. 29.8 слева.

Справа на рис. 29.8 можно видеть, что я добавил папки для организации содержимого панели Project (Проект). Обычно я поступаю так сразу после создания проекта, но теперь специально отложил организацию до этого проекта, чтобы вы сами могли почувствовать, насколько удобнее работать с ним, когда он организован.

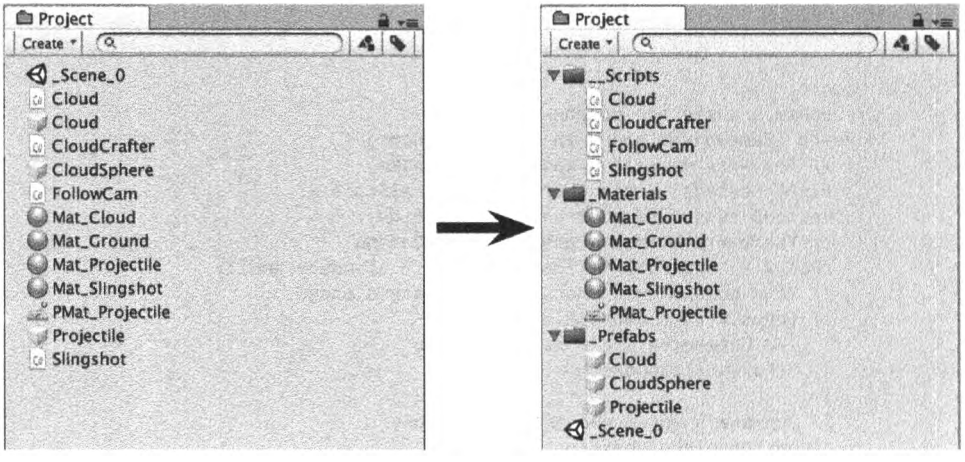


Рис. 29.8. Неорганизованная (*слева*) и организованная (*справа*) панель Project (Проект)

1. Создайте три папки (**Assets > Create > Folder** (Ресурсы > Создать > Папку)) с именами **__Scripts**, **_Materials** и **_Prefabs**. Подчеркивания в начале их имен помогут отсортировать их и поместить в списке выше любых других ресурсов, не являющихся папками, а два символа подчеркивания в начале имени папки **__Scripts** помогут перенести ее в начало списка в панели **Project** (Проект). При этом одновременно будут созданы папки на жестком диске внутри папки *Assets* проекта, то есть такое решение организует не только содержимое панели **Project** (Проект), но и папки *Assets*.
2. Перетащите мышью ресурсы в соответствующие папки в панели **Project** (Проект). В папку **_Materials** перетащите материалы обоих видов, физические и обычные.

Двухколоночный макет панели **Project** (Проект) в Unity автоматически предлагает похожую организацию, но мне никогда не нравилось, что в этом макете по умолчанию ресурсы отображаются в виде пиктограмм, а кроме того, двухколоночное представление не предполагает организации папки *Assets* на жестком диске, как в случае с созданием папок.

Строительство замка

В игре *Mission Demolition* игрок должен что-то разрушать, поэтому построим замок, который будет служить этой цели. На рис. 29.10 показано, как выглядит готовый замок.

1. Отрегулируйте панель **Scene** (Сцена) так, чтобы видеть сцену сзади в изометрической проекции, щелкнув на стрелке напротив оси Z, на значке с изображениями осей (см. рис. 29.9 слева). Если рядом со словом **Back** (Вид сзади) появился символ **<**, щелкните на нем, и вместо него отобразится значок с тремя

параллельными линиями, указывающий, что выполнен переход из перспективной в изометрическую (ортографическую) проекцию.

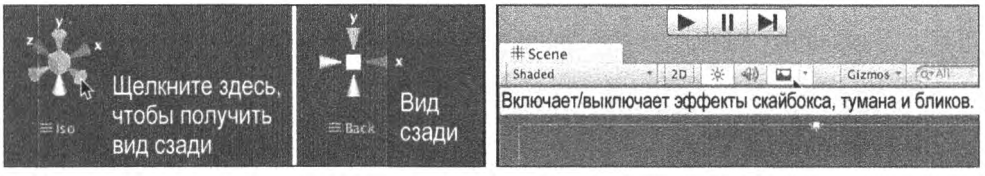


Рис. 29.9. Выбор вида сзади

2. Кроме того, сейчас самое время избавиться от скайбокса (Skybox — изображение неба) в панели **Scene** (Сцена). Для этого щелкайте на кнопке с изображением гористого ландшафта, справа от кнопки с динамиком, в верхней части панели **Scene** (Сцена), — на рис. 29.9 она находится под указателем мыши, — пока фон сцены не окрасится в равномерный серый цвет.
3. Щелкните дважды на **_MainCamera** в иерархии, чтобы отрегулировать масштаб панели **Scene** (Сцена), удобный для строительства замка.

Изготовление стен и перекрытий

Начнем с создания шаблонов игровых объектов, из которых будет строиться замок:

1. Создайте дубликат материала **Mat_Cloud** и дайте ему имя **Mat_Stone**[1].
   a. Выберите **Mat_Cloud** в панели **Project** (Проект).
   b. Выберите в меню Unity пункт **Edit > Duplicate** (Правка > Дублировать).
   c. Измените имя **Mat_Cloud 1** на **Mat_Stone**.
   d. Выделите **Mat_Stone** и в поле **Main Color** выберите 25 % серый цвет (RGBA: [64, 64, 64, 255]).
2. Создайте новый куб (**GameObject > 3D Object > Cube** (Игровой объект > 3D объект > Куб)) и переименуйте его в **Wall**.
   a. Настройте компонент **Transform** объекта **Wall**: P:[0, 0, 0] R:[0, 0, 0] S:[1, 4, 4].
   b. Добавьте в объект **Wall** компонент **Rigidbody** (**Component > Physics > Rigidbody** (Компонент > Физика > Твердое тело)).
   c. Ограничьте координату Z объекта **Wall**, установив флажок **FreezePosition Z** в настройках компонента **Rigidbody**.

[1] Когда я преподавал в Университете штата Мичиган, у меня был замечательный ассистент по имени Мэтт Стоун (Matt Stone). Если вы встретитесь с ним на конференции разработчиков игр, передайте ему привет от меня.

d. Ограничьте поворот, установив флажки FreezeRotation X и Y в настройках компонента Rigidbody.

e. Введите в поле Rigidbody.mass число 4.

f. Перетащите материал Mat_Stone на объект Wall, чтобы окрасить его в серый цвет.

3. Создайте в папке __Scripts новый сценарий с именем RigidbodySleep и введите следующий код:

```
using UnityEngine;

public class RigidbodySleep : MonoBehaviour {
    void Start () {
        Rigidbody rb = GetComponent<Rigidbody>();
        if (rb != null) rb.Sleep();
    }
}
```

Это заставит твердое тело Rigidbody стены Wall предполагать, что оно не должно никуда двигаться. Это обеспечит устойчивость замка (в некоторых версиях Unity наблюдались проблемы, когда замки падали еще до того, как снаряд попадал в них).

4. Подключите сценарий RigidbodySleep к объекту Wall.

5. Перетащите Wall в панель Project (Проект), чтобы создать шаблон (поместите его в папку _Prefabs), и после этого удалите экземпляр Wall из панели Hierarchy (Иерархия).

6. Выберите шаблон Wall в папке _Prefabs в панели Project (Проект) и создайте его копию.

a. Переименуйте копию Wall 1 в Slab.

b. Выберите Slab в папке _Prefabs и настройте масштаб в компоненте Transform: S:[4, 0.5, 4].

Строительство замка из стен и перекрытий

Теперь построим замок из стен и перекрытий:

1. Создайте пустой игровой объект, который будет служить корневым узлом замка (GameObject > Create Empty (Игровой объект > Создать пустой)).

a. Дайте ему имя Castle.

b. Настройте компонент Transform: P:[0, –9.5, 0] R:[0, 0, 0] S:[1, 1, 1]. Эти настройки поместят объект в удобное для строительства место с нижней границей точно на объекте Ground.

2. Перетащите Wall из папки _Prefabs в иерархию на объект Castle, чтобы сделать его дочерним по отношению к Castle.

3. Создайте три копии Wall и определите для них следующие местоположения:

```
Wall P:[ -6, 2, 0 ]   Wall (1) P:[ -2, 2, 0 ]   Wall (2) P:[ 2, 2, 0 ]
Wall (3) P:[ 6, 2, 0 ]
```

4. Перетащите Slab из папки _Prefabs в панели Project (Проект) в иерархию на объект Castle, чтобы сделать их дочерними по отношению к Castle.

5. Создайте две копии Slab и определите для них следующие местоположения:

```
Slab P:[ -4, 4.25, 0 ]   Slab (1) P:[ 0, 4.25, 0 ]   Slab (2) P:[ 4, 4.25, 0 ]
```

6. Чтобы построить второй этаж замка, выберите мышью три соседних стены Wall и два перекрытия Slab над стенами. Создайте их копии (Command-D или Ctrl+D) и, удерживая нажатой клавишу Command (Ctrl на PC), перетащите их вверх, сформировав второй этаж[1]. Вам потребуется вручную настроить их местоположения; окончательные координаты новых стен и перекрытий должны быть следующими:

```
Wall (4) P:[ -4, 6.5, 0 ]    Wall (5) P:[ 0, 6.5, 0 ]    Wall (6) P:[ 4, 6.5, 0 ]
Slab (3) P:[ -2, 8.75, 0 ]   Slab (4) P:[ 2, 8.75, 0 ]
```

7. Повторите прием с копированием, чтобы построить третий и четвертый этажи, добавив еще три вертикальных стены и одно горизонтальное перекрытие:

```
Wall (7) P:[ -2, 11, 0 ]    Wall (8) P:[ 2, 11, 0 ]    Slab (5) P:[ 0, 13.25, 0 ]
Wall (9) P:[ 0, 15.5, 0 ]
```

   Одно из основных преимуществ строительства замка из шаблонов, как в этом примере, состоит в том, что вы легко сможете изменить внешний вид всех перекрытий Slab сразу, изменив шаблон Slab.

8. Выберите шаблон Slab в панели Project (Проект) и введите в поле transform.scale.x число 3.5. Изменение должно отразиться на всех экземплярах Slab, присутствующих в замке. Теперь ваш замок должен выглядеть так, как показано на рис. 29.10, кроме зеленой области прицеливания.

Создание области прицеливания

Последним игровым объектом, который мы должны добавить в замок, является область прицеливания, которую должен поразить игрок.

1. Создайте куб с именем Goal и:

   a. Сделайте его дочерним по отношению к объекту Castle.

   b. Настройте компонент Transform объекта Goal: P:[0, 2, 0] R:[0, 0, 0] S:[3, 4, 4].

[1] Удержание нажатой клавиши Command (Ctrl на PC) при перемещении в Unity обеспечивает выравнивание перемещаемых объектов по сетке, из-за чего может потребоваться вручную настроить их местоположения.

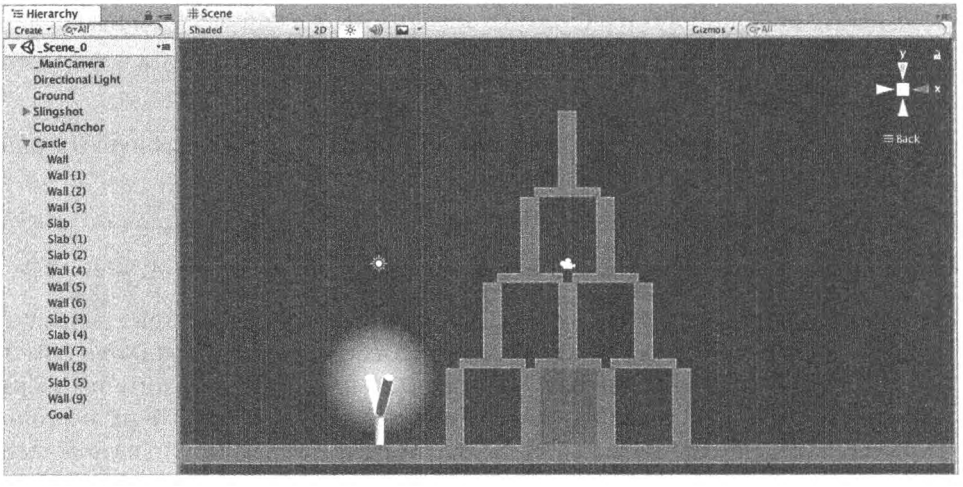


Рис. 29.10. Законченный замок

c. Установите флажок BoxCollider.isTrigger в настройках объекта Goal в инспекторе.

d. Перетащите Goal в папку _Prefabs в панели Project (Проект), чтобы создать шаблон.

2. Создайте в папке _Materials новый материал с именем Mat_Goal.

a. Перетащите Mat_Goal на шаблон Goal в папке _Prefabs в панели Project (Проект).

b. Выделите Mat_Goal в панели Project (Проект) и выберите шейдер Legacy Shaders > Transparent > Diffuse.

c. В поле Main Color для материала Mat_Goal выберите ярко-зеленый цвет с непрозрачностью 25 % (в формате RGBA в цветовой палитре Unity он определяется как [0, 255, 0, 64]).

Тестирование замка

Для тестирования замка выполните следующие шаги:

1. Настройте компонент Transform объекта Castle: P:[50, –9.5, 0] и затем щелкните на кнопке Play (Играть). Возможно, вам придется перезапустить игру несколько раз, но вы должны поразить замок снарядом.

2. Сохраните сцену.

Возврат для другого выстрела

Теперь, когда у нас есть замок для разрушения, пришло время добавить дополнительную игровую логику. После приземления снаряда камера должна вернуть фокус на рогатку:

1. Прежде чем сделать что-то еще, добавьте тег Projectile в шаблон снаряда.
   a. Выберите шаблон Projectile в панели Project (Проект).
   b. В инспекторе выберите пункт Add Tag (Добавить тег) в раскрывающемся списке Tag. В результате в инспекторе откроется диспетчер тегов и слоев Tags & Layers.
   c. Щелкните на значке + справа внизу в пустом списке тегов Tags.
   d. Введите в поле New Tag Name (Имя нового тега) текст Projectile и щелкните на кнопке Save (Сохранить).
   e. Снова выберите шаблон Projectile в панели Project (Проект).
   f. Присвойте ему тег Projectile, выбрав пункт Projectile в раскрывающемся списке Tag в инспекторе.
2. Откройте сценарий FollowCam в MonoDevelop и измените следующие строки:

```
public class FollowCam : MonoBehaviour {

    ...
    void FixedUpdate () {
//--    // Однострочная версия if не требует фигурных скобок                // a
//--    if (POI == null) return; // выйти, если нет интересующего объекта
//--
//--    // Получить позицию интересующего объекта
//--    Vector3 destination = POI.transform.position;

      Vector3 destination;
      // Если нет интересующего объекта, вернуть P:[ 0, 0, 0 ]
      if (POI == null) {
          destination = Vector3.zero;
      } else {
          // Получить позицию интересующего объекта
          destination = POI.transform.position;
          // Если интересующий объект - снаряд, убедиться, что он остановился
          if (POI.tag == "Projectile") {
              // Если он стоит на месте (то есть не двигается)
              if ( POI.GetComponent<Rigidbody>().IsSleeping() ) {
                  // Вернуть исходные настройки поля зрения камеры
                  POI = null;
                  // в следующем кадре
                  return;
              }
          }
      }

      // Ограничить X и Y минимальными значениями
      destination.x = Mathf.Max( minXY.x, destination.x );
      ...
    }
}
```

   a. Все строки, начинающиеся с последовательности символов //--, должны быть удалены или закомментированы.

Теперь, когда снаряд прекратит полет и остановится (то есть когда метод `Rigidbody.IsSleeping()` вернет значение правда), сценарий FollowCam присвоит `null` полю `POI`, и камера вернется в исходную позицию. Однако для остановки снаряда наверняка понадобится немало времени. Давайте настроим физический движок так, чтобы он прекращал моделирование физического поведения объекта немного раньше, чем обычно.

3. Скорректируйте значение Sleep Threshold в диспетчере физики PhysicsManager:
   a. Откройте диспетчер физики PhysicsManager (Edit > Project Settings > Physics (Правка > Параметры проекта > Физика)).
   b. Измените значение 0,005 в поле Sleep Threshold на 0,02. Параметр Sleep Threshold определяет минимальное расстояние, которое должно пройти твердое тело Rigidbody в течение одного физического кадра, чтобы движок продолжил моделирование его поведения в следующем кадре. Если в течение одного кадра объект пройдет меньшее расстояние (в данном случае меньше 2 см), движок PhysX остановит объект с этим компонентом Rigidbody и прекратит моделировать его поведение (то есть прекратит перемещать игровой объект), пока в игре не случится что-то, что снова заставит его двигаться.
4. Сохраните сцену. Если теперь попробовать сыграть в игру, камера будет возвращаться в исходную позицию, и вы сможете произвести новый выстрел.

Добавление следа, остающегося за снарядом

В Unity уже есть встроенный эффект визуализации следа Trail Renderer, но он нам не подойдет, потому что нам требуется больший контроль, чем может дать встроенный эффект. Поэтому будем использовать Line Renderer, на котором основан Trail Renderer:

1. Сначала создайте пустой игровой объект (GameObject > Create Empty (Игровой объект > Создать пустой)) с именем ProjectileLine.
   a. Добавьте в него компонент Line Renderer (Component > Effects > Line Renderer (Компонент > Эффекты > Визуализатор линии)).
   b. В настройках ProjectileLine в инспекторе распахните раздел Materials. Настройте компонент Line Renderer, как показано на рис. 29.11.
2. Создайте сценарий на C# (Assets > Create > C# Script (Ресурсы > Создать > Сценарий C#)) в папке _Scripts. Дайте ему имя ProjectileLine и подключите к игровому объекту ProjectileLine. Откройте сценарий ProjectileLine в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ProjectileLine : MonoBehaviour {
    static public ProjectileLine S; // Одиночка
```

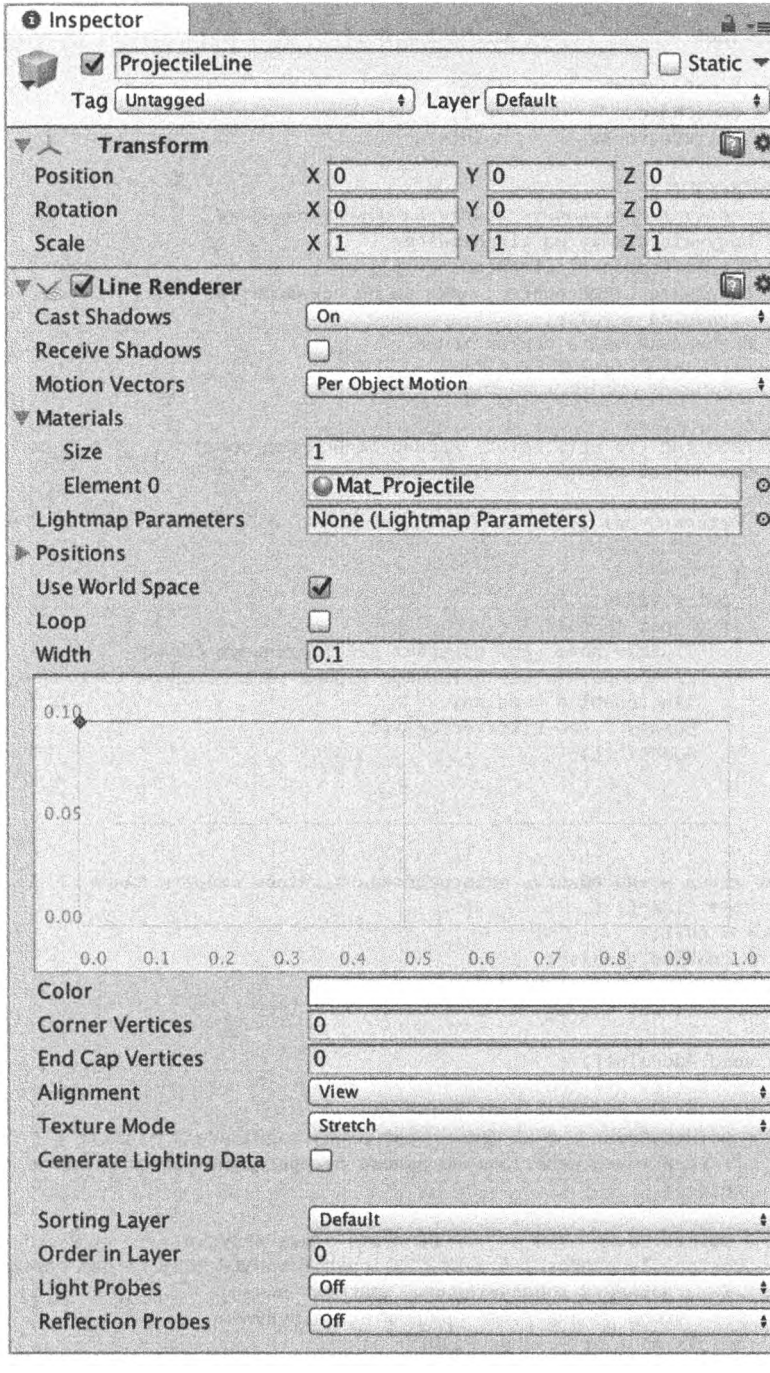


Рис. 29.11. Настройки ProjectLine

```
[Header("Set in Inspector")]
public float               minDist = 0.1f;

private LineRenderer       line;
private GameObject         _poi;
private List<Vector3>      points;

void Awake() {
    S = this; // Установить ссылку на объект-одиночку
    // Получить ссылку на LineRenderer
    line = GetComponent<LineRenderer>();
    // Выключить LineRenderer, пока он не понадобится
    line.enabled = false;
    // Инициализировать список точек
    points = new List<Vector3>();
}

// Это свойство (то есть метод, маскирующийся под поле)
public GameObject poi {
    get {
        return( _poi );
    }
    set {
        _poi = value;
        if ( _poi != null ) {
            // Если поле _poi содержит действительную ссылку,
            //   сбросить все остальные параметры в исходное состояние
            line.enabled = false;
            points = new List<Vector3>();
            AddPoint();
        }
    }
}

// Этот метод можно вызвать непосредственно, чтобы стереть линию
public void Clear() {
    _poi = null;
    line.enabled = false;
    points = new List<Vector3>();
}

public void AddPoint() {
    // Вызывается для добавления точки в линии
    Vector3 pt = _poi.transform.position;
    if ( points.Count > 0 && (pt - lastPoint).magnitude < minDist ) {
        // Если точка недостаточно далека от предыдущей, просто выйти
        return;
    }
    if ( points.Count == 0 ) { // Если это точка запуска...
        Vector3 launchPosDiff = pt - Slingshot.LAUNCH_POS; // Для определения
        // ...добавить дополнительный фрагмент линии,
        //   чтобы помочь лучше прицелиться в будущем
        points.Add( pt + launchPosDiff );
        points.Add(pt);
        line.positionCount = 2;
```

```
            // Установить первые две точки
            line.SetPosition(0, points[0] );
            line.SetPosition(1, points[1] );
            // Включить LineRenderer
            line.enabled = true;
        } else {
            // Обычная последовательность добавления точки
            points.Add( pt );
            line.positionCount = points.Count;
            line.SetPosition( points.Count-1, lastPoint );
            line.enabled = true;
        }
    }

    // Возвращает местоположение последней добавленной точки
    public Vector3 lastPoint {
        get {
            if (points == null) {
                // Если точек нет, вернуть Vector3.zero
                return( Vector3.zero );
            }
            return( points[points.Count-1] );
        }
    }

    void FixedUpdate () {
        if ( poi == null ) {
            // Если свойство poi содержит пустое значение, найти интересующий
            // объект
            if (FollowCam.POI != null) {
                if (FollowCam.POI.tag == "Projectile") {
                    poi = FollowCam.POI;
                } else {
                    return; // Выйти, если интересующий объект не найден
                }
            } else {
                return; // Выйти, если интересующий объект не найден
            }
        }
        // Если интересующий объект найден,
        //   попытаться добавить точку с его координатами в каждом FixedUpdate
        AddPoint();
        if ( FollowCam.POI == null ) {
            // Если FollowCam.POI содержит null, записать nulll в poi
            poi = null;
        }
    }
}
```

3. В сценарий Slingshot нужно также добавить свойство LAUNCH_POS, чтобы позволить методу AddPoint() ссылаться на начальную launchPoint:

```
public class Slingshot : MonoBehaviour {
    static private Slingshot S;                                          // a
    // поля, устанавливаемые в инспекторе Unity
```

```
    [Header("Set in Inspector")]
    ...
    private Rigidbody        projectileRigidbody;

    static public Vector3 LAUNCH_POS {                                    // b
        get {
            if (S == null) return Vector3.zero;
                return S.launchPos;
        }
    }

    void Awake() {
        S = this;                                                         // c
        Transform launchPointTrans = transform.FindChild("LaunchPoint");
        ...
    }
    ...
}
```

a. Это скрытый статический экземпляр `Slingshot`, который будет играть роль объекта-одиночки, а так как он объявлен скрытым (`private`), обращаться к нему смогут только экземпляры класса `Slingshot`.

b. Статическое общедоступное свойство, использующее статический скрытый экземпляр `Slingshot` в `S`, открывает доступ только для чтения к полю `launchPos` класса `Slingshot`. Если поле `S` содержит `null`, оно вернет [0, 0, 0].

c. Здесь экземпляр `this` класса `Slingshot` присваивается полю `S`. Так как `Awake()` — это самый первый метод, который вызывается после создания экземпляра любого класса, наследующего `MonoBehaviour`, поле `S` будет установлено до того, как какой-то другой код попробует обратиться к свойству `LAUNCH_POS`.

Если теперь вы попробуете сыграть в игру, то заметите хорошо видимый серый след, оставляемый снарядом. След будет стираться и заменяться новым при каждом последующем выстреле.

4. Сохраните сцену.

Поражение области прицеливания

Область прицеливания в замке должна реагировать на попадание в нее снаряда:

1. Создайте новый сценарий на C# с именем Goal и подключите его к шаблону Goal в папке _Prefabs в панели Project (Проект). Затем введите в сценарий Goal следующий код.

```
using UnityEngine;
using System.Collections;

public class Goal : MonoBehaviour {
```

```
    // Статическое поле, доступное любому другому коду
    static public bool        goalMet = false;

    void OnTriggerEnter( Collider other ) {
        // Когда в область действия триггера попадает что-то,
        //   проверить, является ли это "что-то" снарядом
        if ( other.gameObject.tag == "Projectile" ) {
            // Если это снаряд, присвоить полю goalMet значение true
            Goal.goalMet = true;
            // Также изменить альфа-канал цвета, чтобы увеличить непрозрачность
            Material mat = GetComponent<Renderer>().material;
            Color c = mat.color;
            c.a = 1;
            mat.color = c;
        }
    }
}
```

Если теперь поразить область прицеливания снарядом, она окрасится в ярко-зеленый цвет. Может потребоваться несколько выстрелов, чтобы разрушить часть замка и добраться до нее. Чтобы упростить задачу на время тестирования, можно выбрать мешающие стены и деактивировать их, сняв флажок в инспекторе, находящийся сразу под словом Inspector (Инспектор) в заголовке панели. Но не забудьте снова активировать их, закончив тестирование области прицеливания Goal.

2. Сохраните сцену.

Добавление дополнительных замков

Один замок — хорошо, а больше — лучше, поэтому добавим еще несколько замков.

1. Переименуйте игровой объект Castle в Castle_0.
2. Создайте шаблон Castle_0, перетащив игровой объект в папку _Prefabs в панели Project (Проект). Убедившись, что шаблон Castle_0 создан, удалите экземпляр Castle_0 из иерархии[1].
3. Создайте копию шаблона Castle_0 в панели Project (Проект) — она автоматически получит имя Castle_1.
4. Перетащите Castle_1 в панель Scene (Сцена) и измените его конструкцию. Удалив одну из стен, вы почти наверняка получите сообщение «Break Prefab Instance»

[1] При создании шаблона Castle_0 все экземпляры Slab и Wall потеряют связь с их шаблонами Slab и Wall. Разработчики Unity работают над этой проблемой, и в Unity 2017 они реализовали поддержку вложенных шаблонов, но пока не полностью. Если вам поведение ваших шаблонов покажется странным, обращайтесь за информацией, которую я буду постоянно обновлять, по адресу http://book.prototools.net.

(экземпляр не соответствует шаблону). Это нормально. Смело продолжайте конструировать Castle_1 по своему усмотрению[1].

5. Завершив строительство замка Castle_1, выберите Castle_1 в иерархии и щелкните на кнопке Apply (Применить) в верхней части инспектора (она находится в одном ряду с кнопками Select (Выбрать) и Revert (Отменить), правее заголовка Prefab (Шаблон)). Щелчок на кнопке Apply (Применить) перенесет изменения, произведенные в этом экземпляре, обратно в шаблон Castle_1.
6. Теперь можно удалить экземпляр Castle_1 из иерархии.

   Повторите описанную процедуру и создайте еще несколько замков. На рис. 29.12 показаны замки, которые создал я.

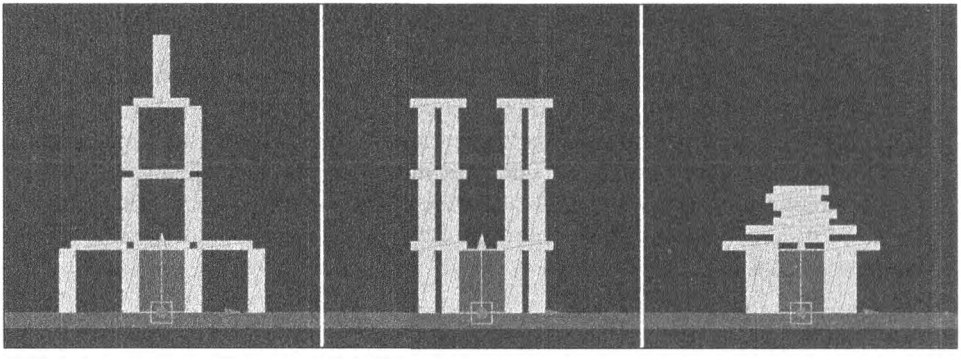


Рис. 29.12. Дополнительные замки

7. После создания замков проверьте, что их экземпляров не осталось в иерархии, и сохраните сцену.

Добавление в сцену пользовательского интерфейса

Выполните следующие шаги, чтобы добавить пользовательский интерфейс в сцену:

1. Добавьте в сцену элемент пользовательского интерфейса UI Text (GameObject > UI > Text (Игровой объект > ПИ > Текст)) и дайте ему имя UIText_Level.

[1] Если удерживать нажатой клавишу Command в macOS (или Ctrl в Windows), перемещение стен и перекрытий мышью будет осуществляться по сетке с шагом 0,5 м, что может упростить строительство замка. Кроме того, старайтесь размещать блоки замка так, чтобы они не пересекались друг с другом, иначе они будут расталкивать друг друга, как мы видели это в главе 19 «Hello World: ваша первая программа». Так как функция `RigidbodySleep` с самого начала принудительно останавливает моделирование поведения блоков, эффект «расталкивания» не проявится до попадания снаряда в замок, но потом может возникнуть интересный эффект взрывающегося замка, — если, конечно, это то, что вы хотите.

2. Создайте второй элемент UI Text с именем UIText_Shots.
3. Настройте оба элемента, как показано на рис. 29.13.

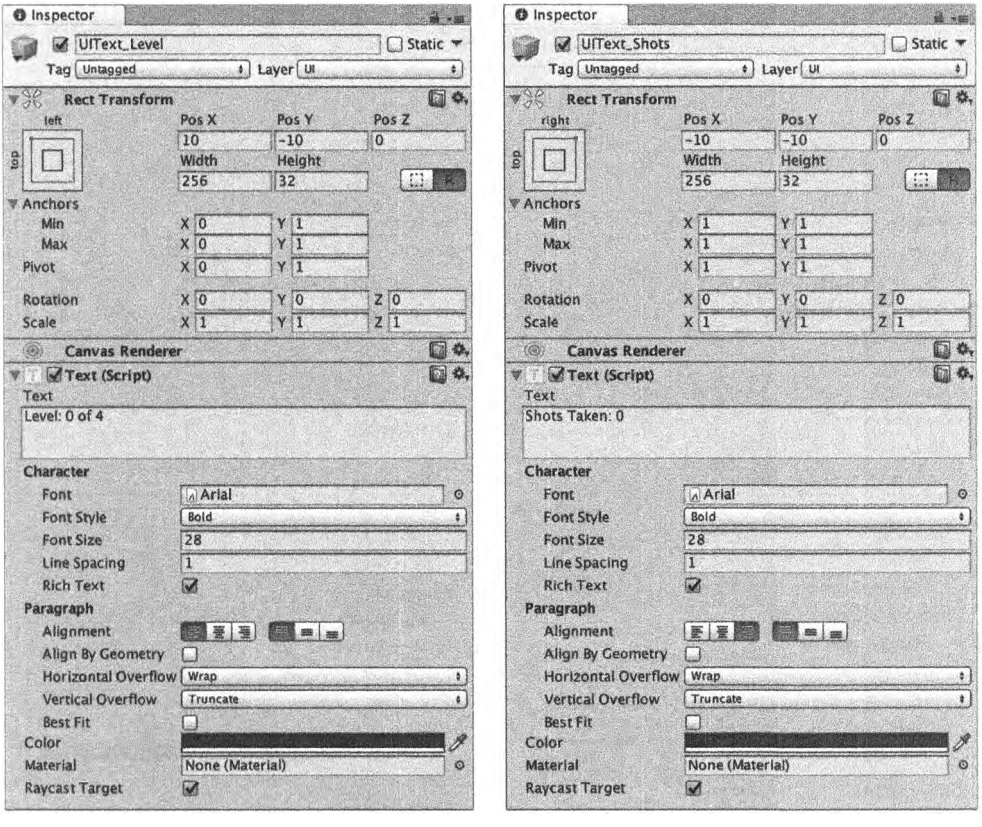


Рис. 29.13. Настройки элементов пользовательского интерфейса UIText_Level и UIText_Shots

4. Создайте UI Button (GameObject > UI > Button (Игровой объект > ПИ > Кнопка)). Дайте кнопке имя UIButton_View.
5. Настройте компонент Rect Transform кнопки UIButton_View, как показано на рис. 29.14, но пока не трогайте настройки за пределами компонента Rect Transform.
6. В иерархии щелкните на пиктограмме с изображением треугольника рядом с UIButton_View, выберите дочерний объект Text и настройте компонент Text (Script), как показано на рис. 29.14. Не меняйте ничего за пределами раздела Character. По завершении сохраните сцену.

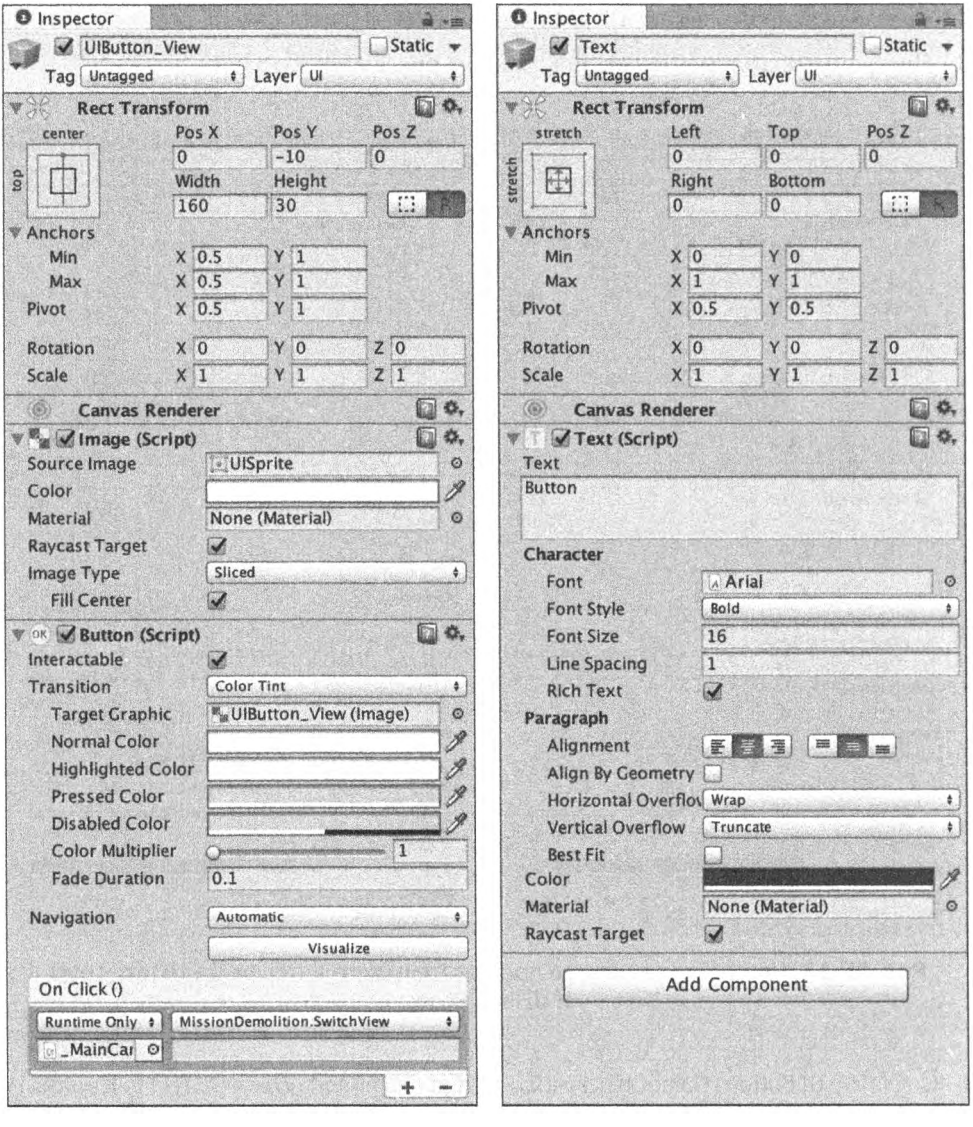


Рис. 29.14. Настройки UIButton_View и его дочернего объекта Text

Совершенствование управления игрой

В первую очередь нам необходимо найти местоположение для камеры, откуда будут видны сразу рогатка и замок.

1. Создайте новый, пустой игровой объект (**GameObject > Create Empty** (Игровой объект > Создать пустой)) с именем **ViewBoth**. Настройте компонент **Transform**

объекта ViewBoth: P:[25, 25, 0] R:[0, 0, 0] S:[1, 1, 1]. Он будет служить точкой POI для камеры, когда у вас появится желание увидеть сразу рогатку и замок.

2. Создайте в папке _Scripts новый сценарий на C# с именем MissionDemolition и подключите его к главной камере _MainCamera. Он будет играть роль диспетчера состояний игры. Откройте сценарий MissionDemolition и введите следующий код:

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;                                                    // a

public enum GameMode {                                                   // b
    idle,
    playing,
    levelEnd
}

public class MissionDemolition : MonoBehaviour {
    static private MissionDemolition S; // скрытый объект-одиночка

    [Header("Set in Inspector")]
    public Text                 uitLevel;  // Ссылка на объект UIText_Level
    public Text                 uitShots;  // Ссылка на объект UIText_Shots
    public Text                 uitButton; // Ссылка на дочерний объект Text
в UIButton_View
    public Vector3              castlePos; // Местоположение замка
    public GameObject[]         castles;   // Массив замков

    [Header("Set Dynamically")]
    public int                  level;     // Текущий уровень
    public int                  levelMax;  // Количество уровней
    public int                  shotsTaken;
    public GameObject           castle;    // Текущий замок
    public GameMode             mode = GameMode.idle;
    public string               showing = "Show Slingshot"; // Режим FollowCam

    void Start() {
        S = this; // Определить объект-одиночку

        level = 0;
        levelMax = castles.Length;
        StartLevel();
    }

    void StartLevel() {
        // Уничтожить прежний замок, если он существует
        if (castle != null) {
            Destroy( castle );
        }

        // Уничтожить прежние снаряды, если они существуют
        GameObject[] gos = GameObject.FindGameObjectsWithTag("Projectile");
        foreach (GameObject pTemp in gos) {
            Destroy( pTemp );
```

```
        }

        // Создать новый замок
        castle = Instantiate<GameObject>( castles[level] );
        castle.transform.position = castlePos;
        shotsTaken = 0;

        // Переустановить камеру в начальную позицию
        SwitchView("Show Both");
        ProjectileLine.S.Clear();

        // Сбросить цель
        Goal.goalMet = false;

        UpdateGUI();

        mode = GameMode.playing;
    }

    void UpdateGUI() {
        // Показать данные в элементах ПИ
        uitLevel.text = "Level: "+(level+1)+" of "+levelMax;
        uitShots.text = "Shots Taken: "+shotsTaken;
    }

    void Update() {
        UpdateGUI();

        // Проверить завершение уровня
        if ( (mode == GameMode.playing) && Goal.goalMet ) {
            // Изменить режим, чтобы прекратить проверку завершения уровня
            mode = GameMode.levelEnd;
            // Уменьшить масштаб
            SwitchView("Show Both");
            // Начать новый уровень через 2 секунды
            Invoke("NextLevel", 2f);
        }
    }

    void NextLevel() {
        level++;
        if (level == levelMax) {
            level = 0;
        }
        StartLevel();
    }

    public void SwitchView( string eView = "" ) {                          // c
        if (eView == "") {
            eView = uitButton.text;
        }
        showing = eView;
        switch (showing) {
            case "Show Slingshot":
```

```
                FollowCam.POI = null;
                uitButton.text = "Show Castle";
                break;

            case "Show Castle":
                FollowCam.POI = S.castle;
                uitButton.text = "Show Both";
                break;

            case "Show Both":
                FollowCam.POI = GameObject.Find("ViewBoth");
                uitButton.text = "Show Slingshot";
                break;
        }
    }

    // Статический метод, позволяющий из любого кода увеличить shotsTaken
    public static void ShotFired() {                                    // d
        S.shotsTaken++;
    }
}
```

a. Строку `using UnityEngine.UI;` нужно добавить, чтобы получить возможность использовать любые классы пользовательского интерфейса, такие как `Text` и `Button`.

b. Первый экземпляр перечисления в этой части книги. Подробнее о перечислениях рассказывается во врезке «Перечисления» ниже.

c. Общедоступный метод `SwitchView()` будет вызываться этим экземпляром `MissionDemolition` и экземпляром `Button` пользовательского интерфейса (который мы вскоре реализуем). Код `string eView = ""` определяет значение по умолчанию `""` для параметра `eView`, то есть, вызывая этот метод, можно не передавать ему строку. Иными словами, `SwitchView()` можно вызвать как `SwitchView("Show Both")` или как `SwitchView()`. Если вызвать метод без параметра, первая инструкция `if` в методе запишет в `eView` текущий текст на кнопке Button.

d. `ShotFired()` — статический общедоступный метод, который вызывается сценарием Slingshot, чтобы уведомить MissionDemolition о произведенном выстреле.

ПЕРЕЧИСЛЕНИЯ

Перечисление (`enum`) — это способ определить именованные числа в C#. Определение `enum` в начале сценария MissionDemolition объявляет тип перечисления `GameMode` с тремя возможными значениями: `idle`, `playing` и `levelEnd`. После определения перечисления можно объявить переменную с типом перечисления.

```
public GameMode mode = GameMode.idle;
```

Эта строка создаст новую переменную с именем mode и типом GameMode и присвоит ей значение GameMode.idle.

Перечисления обычно используются, когда имеется лишь несколько конкретных вариантов значений для переменных и желательно, чтобы эти варианты легко распознавались человеком. Режим игры можно было бы также определять строками (например, "idle", "playing" или "levelEnd"), но перечисление дает способ более выразительный, более устойчивый к орфографическим ошибкам и позволяющий использовать функцию автоматического дополнения в среде разработки.

Более подробную информацию о перечислениях вы найдете в приложении Б «Полезные идеи».

3. Теперь, после определения статического метода `ShotFired()` в классе `MissionDemolition`, можно добавить его вызов в класс `Slingshot`. Вставьте следующую строку, выделенную жирным, в сценарий Slingshot:

```
public class Slingshot : MonoBehaviour {
    ...
    void Update() {
        ...
        if ( Input.GetMouseButtonUp(0) ) {
            // Кнопка мыши отпущена
            ...
            FollowCam.POI = projectile;
            projectile = null;
            MissionDemolition.ShotFired();                    // a
            ProjectileLine.S.poi = projectile;                // b
        }
    }
}
```

   a. Так как метод `ShotFired()` в `MissionDemolition` объявлен как статический, к нему можно обращаться с использованием имени самого класса `MissionDemolition`, а не через конкретный экземпляр. Когда `MissionDemolition.ShotFired()` вызывается сценарием Slingshot, он увеличивает значение `MissionDemolition.S.shotsTaken`.

   b. Эта строка заставляет `ProjectileLine` следовать за новым снарядом Projectile после выстрела.

4. Сохраните все сценарии и вернитесь в Unity.

5. Выберите UIButton_View в иерархии, найдите в инспекторе список On Click(), в конце раздела Button (Script) и щелкните на кнопке +.

   a. Под кнопкой Runtime Only (Только во время выполнения) имеется поле, в котором в данный момент отображается None (Object).

b. Щелкните на изображении маленькой круглой мишени справа от поля с текстом None (Object) и в появившемся окне выберите _MainCamera (выполните двойной щелчок на _MainCamera). В результате главная камера _MainCamera будет выбрана как игровой объект, получающий вызовы от UIButton_View.

c. Щелкните на раскрывающемся списке правее, в котором сейчас отображается текст No Function, и выберите MissionDemolition > SwitchView(String)[1].

В результате этих настроек если теперь щелкнуть на кнопке UIButton_View, она вызовет общедоступный метод `SwitchView()` экземпляра MissionDemolition, подключенного к главной камере _MainCamera. Настройки в разделе Button (Script) инспектора должны теперь выглядеть так, как показано на рис. 29.14.

6. Выберите _MainCamera в иерархии. В настройках компонента MissionDemolition (Script) в инспекторе задайте значения для нескольких переменных.

   a. Настройте поле `castlePos` как [50, –9.5, 0], расположив замки на достаточном расстоянии от рогатки.

   b. Чтобы установить значение в поле `uitLevel`, щелкните в инспекторе на изображении мишени справа от поля, и в открывшемся диалоге выберите UIText_Level на вкладке Scene (Сцена).

   c. Щелкните на изображении мишени справа от поля `uitShots` и выберите UIText_Shots на вкладке Scene (Сцена).

   d. Щелкните на изображении мишени справа от поля `uitButton` и выберите Text на вкладке Scene (Сцена); это единственный текстовый элемент пользовательского интерфейса в сцене, и он соответствует элементу Text с надписью на кнопке UIButton_View.

   e. Щелкните на пиктограмме с треугольником рядом со списком `castles` и введите в поле Size количество замков, созданных прежде. (В примере на рис. 29.15 я создал четыре замка.)

   f. Перетащите каждый из пронумерованных шаблонов Castle в соответствующий элемент в массиве `castles`, чтобы определить замки для разных уровней в игре. Постарайтесь упорядочить их по возрастанию сложности.

7. Сохраните сцену!

Теперь можно играть в игру, преодолевая несколько уровней сложности, и следить за количеством произведенных выстрелов. Можно также щелкать на кнопке вверху для переключения вида.

---

[1] В серое поле, которое появится под кнопкой MissionDemolition.SwitchView, можно ввести строку для передачи в вызов метода `SwitchView`. Оставьте поле пустым, чтобы обеспечить передачу в `SwitchView` пустой строки `""`, совпадающей со значением по умолчанию для необязательного параметра `eView`, — то есть вам не нужно изменять это поле.

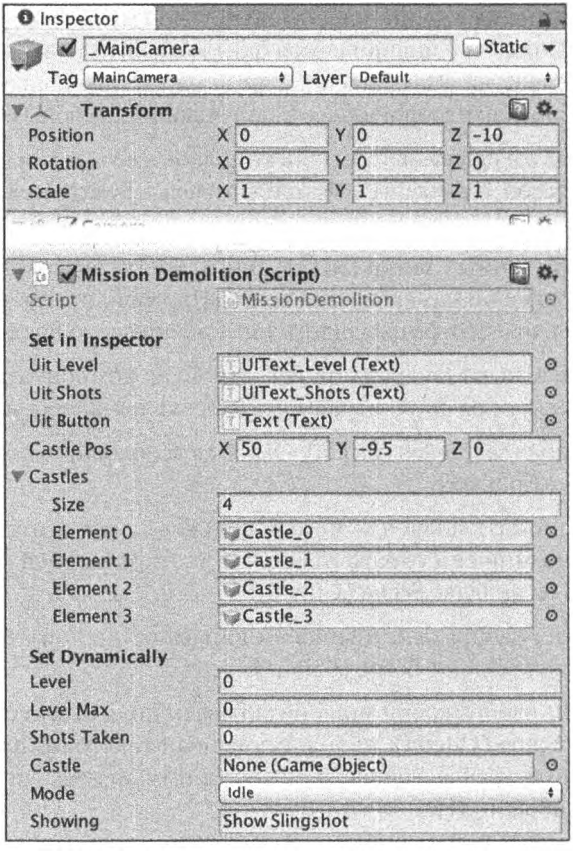


Рис. 29.15. Заключительные настройки (с массивом Castles) _MainCamera:MissionDemolition

Итоги

На этом мы заканчиваем разработку прототипа *Mission Demolition*. В одной-единственной главе нам удалось создать игру, учитывающую законы физики подобно *Angry Birds*, которую вы можете самостоятельно продолжить, развивать и расширять. Этот и все последующие учебные примеры призваны служить основой, поверх которой вы сможете конструировать свои игры.

Следующие шаги

Вы можете добавить в прототип десятки новых возможностей, например:

- Использовать `PlayerPrefs` для хранения высшего достижения на каждом уровне, как это было сделано в Apple Picker.

- Создать разные части замка из разных материалов, с большей или меньшей массой. Некоторые материалы могут даже ломаться от достаточно сильного удара.
- Показать траектории нескольких последних выстрелов, а не только самого последнего.
- Использовать Line Renderer для отображения резиновой ленты рогатки.
- Реализовать настоящий многоуровневый скроллинг фоновых облаков и добавить больше фоновых элементов, таких как горы или здания и сооружения.
- Любые другие, какие только захотите!

После реализации других прототипов в этой книге вернитесь сюда и подумайте, что бы вы могли добавить. Создайте свой проект, покажите его людям и постепенно, шаг за шагом, улучшайте игру. Помните, что развитие проекта — всегда итеративный процесс. Внеся изменения, которые вам не понравились, не опускайте руки, — просто уберите их и попробуйте что-нибудь еще.

30 Прототип 3: SPACE SHMUP

К жанру SHMUP (или «shoot 'em up» — игра-стрелялка, шутер) относятся такие игры, как классические *Galaga* и *Galaxian* из 1980-х, и современный шедевр *Ikaruga*.

В этой главе мы создадим шутер и используем некоторые приемы программирования, которые послужат вам в вашей карьере программиста и разработчика прототипов. К ним относятся наследование классов, использование статических полей и методов и шаблон «Одиночка» (Singleton). Большинство из этих приемов вы уже видели прежде, но в этом прототипе мы будем использовать их особенно широко.

Начало: прототип 3

В этом проекте вы создадите прототип классического космического шутера. К концу этой главы вы получите простой прототип примерно одного уровня с прототипами из двух предыдущих глав, а в следующей главе мы посмотрим, как реализовать несколько дополнительных особенностей. На рис. 30.1 показано, как будет выглядеть законченный прототип после двух глав. На этих изображениях можно видеть космический корабль игрока, окруженный зеленым защитным полем, а также несколько типов врагов и несколько типов бонусов — подбираемых элементов, улучшающих характеристики корабля (кубики с надписями B, O и S).

Импортирование пакета ресурсов для Unity

В процесс настройки этого прототипа добавился новый шаг — загрузка и импортирование пакета ресурсов для Unity. Создание сложных ресурсов и графики для игр выходит далеко за рамки этой книги, поэтому я создал пакет с некоторыми простыми ресурсами, необходимыми для создания всех визуальных эффектов в этой игре. Конечно, как уже неоднократно упоминалось в этой книге, для прототипа гораздо важнее, как он играет и как воспринимается, чем как он выглядит, но не менее важно знать, как обращаться с художественными ресурсами.

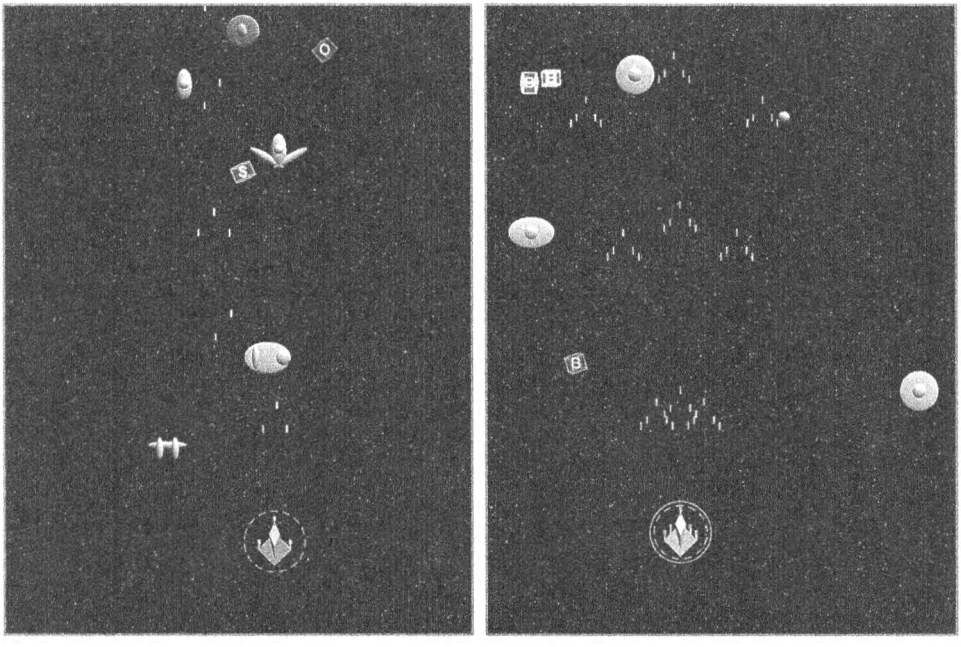


Рис. 30.1. Два изображения прототипа игры Space SHMUP. Слева игрок стреляет из стандартного бластера, а справа — из многоствольных пушек

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. Чтобы освежить в памяти стандартную процедуру настройки проекта, обращайтесь к приложению А «Стандартная процедура настройки проекта».

- **Имя проекта:** Space SHMUP Prototype
- **Имя сцены:** _Scene_0
- **Имена сценариев на C#:** пока нет
- **Папки проекта:** __Scripts (два подчеркивания перед «Scripts»), _Materials, _Prefabs
- **Загрузите и импортируйте пакет:** найдите раздел Chapter 30 по адресу http://book.prototools.net
- **Имена сценариев на C#:** пока нет
- **Переименуйте:** главную камеру Main Camera в _MainCamera

Чтобы загрузить и установить пакет, упомянутый во врезке «Настройка проекта для этой главы», сначала перейдите по указанному URL (http://book.prototools.net) и найдите раздел для этой главы. Загрузите файл C30_Space_SHMUP_Starter.unitypackage на свой компьютер (обычно загружаемые файлы сохраняются в папке *Downloads*). Откройте проект в Unity и выберите пункт меню Assets > Import Package > Custom Package (Ресурсы > Импортировать пакет > Свой пакет). Найдите и выберите файл C30_Space_SHMUP_Starter.unitypackage в вашей папке *Downloads*. Откроется диалог импортирования, как показано на рис. 30.2.

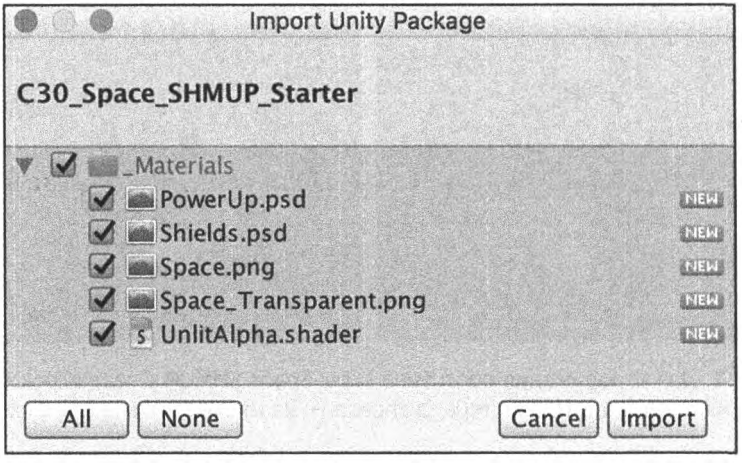


Рис. 30.2. Диалог импортирования пакета .unitypackage

Выберите все файлы, как показано на рис. 30.2 (щелкнув на кнопке All (Все)), и щелкните на кнопке Import (Импортировать). В результате четыре новые *текстуры* и один *шейдер* будут сохранены в папке *_Materials*. Текстуры — это обычные файлы изображений. Создание текстур выходит далеко за рамки этой книги, однако этой теме посвящено большое количество книг и онлайн-руководств. Наиболее широко используемым, пожалуй, инструментом редактирования графики является *Adobe Photoshop*, но он стоит довольно дорого. Распространенная альтернатива с открытым исходным кодом — *Gimp* (http://www.gimp.org), а также очень хороший и удивительно недорогой коммерческий конкурент — *Affinity Photo* (https://affinity.serif.com/photo).

Мы не будем рассматривать создание *шейдеров* в этой книге. Шейдеры — это программы, сообщающие компьютеру, как отображать текстуры, присоединенные к игровым объектам. Они могут придать сцене более реалистичный, или мультяшный вид, или любой другой, какой пожелаете, и являются важной частью графики любой современной игры.

В Unity используется свой уникальный язык шейдеров с названием ShaderLab. Если у вас появится желание узнать больше о нем, я советую начать с руковод-

ства по программированию шейдеров «Shader Reference» (http://docs.unity3d.com/Documentation/Components/SL-Reference.html).

В пакет включен простой шейдер, который не использует многих возможностей, доступных шейдерам, и только отображает на экране цветную, неосвещенную фигуру. Шейдер *UnlitAlpha.shader* отлично подходит для отображения ярких цветных экранных элементов. Также UnlitAlpha поддерживает наложение цветов и полупрозрачность, что помогает отображать кубики бонусов, подбором которых игрок может улучшать характеристики корабля.

Настройка сцены

Выполните следующие шаги, чтобы настроить сцену (возьмите карандаш и отмечайте галочками шаги по мере их выполнения):

1. Выберите источник направленного света Directional Light в иерархии и настройте его компонент Transform:

```
P:[ 0, 20, 0 ] R:[ 50, -30, 0 ] S:[ 1, 1, 1 ]
```

2. Переименуйте главную камеру Main Camera в _MainCamera (о чем уже говорилось выше, во врезке с инструкциями по настройке проекта). Выберите главную камеру _MainCamera и настройте ее компонент Transform:

```
P:[ 0, 0, -10 ] R:[ 0, 0, 0 ] S:[ 1, 1, 1 ]
```

3. В компоненте Camera главной камеры _MainCamera настройте следующие поля, а потом сохраните сцену:
   - В поле Clear Flags выберите значение Solid Color.
   - В поле Background выберите черный цвет (со значением 255 в альфа-канале; RGBA:[0, 0, 0, 255]).
   - В поле Projection выберите значение Orthographic.
   - В поле Size введите число 40 (после настройки поля Projection).
   - В поле Near Clipping Plane введите число 0.3.
   - В поле Far Clipping Plane введите число 100.
4. Поскольку наша будущая игра — это шутер с вертикальной ориентацией игрового поля, для панели Game (Игра) нужно установить соотношение сторон с книжной ориентацией. Для этого в панели Game (Игра) щелкните на раскрывающемся списке выбора соотношения сторон, в котором в данный момент должен отображаться текст Free Aspect (Свободное соотношение, как показано на рис. 30.3). Найдите внизу списка кнопку с символом +. Щелкните на ней, чтобы добавить новое предопределенное соотношение сторон. Введите значения, как показано на рис. 30.3, и щелкните на кнопке Add (Добавить). Выберите в списке вновь появившийся пункт Portrait (3:4) (Книжная (3:4)).

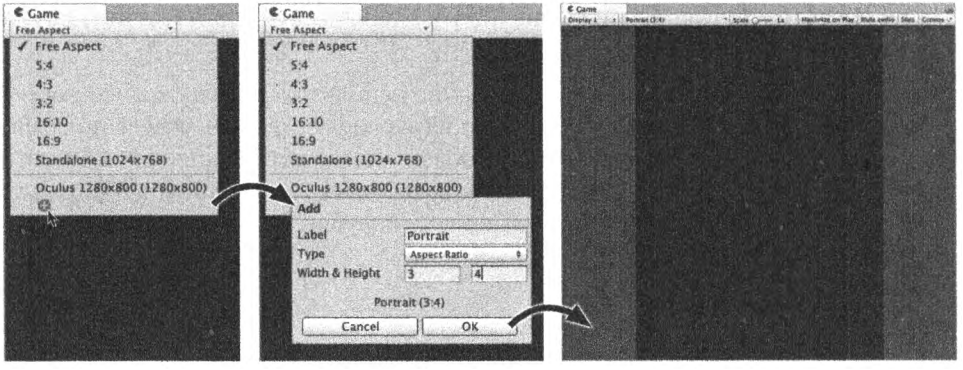


Рис. 30.3. Добавление предопределенного соотношения сторон в панель Game (Игра)

Создание космического корабля игрока

В этой главе мы будем чередовать работу с художественными ресурсами и программным кодом, отказавшись от прежнего подхода, когда сначала создавались все художественные ресурсы, а потом разрабатывался весь программный код. Чтобы создать космический корабль игрока, выполните следующие шаги:

1. Создайте пустой игровой объект с именем _Hero (GameObject > Create Empty (Игровой объект > Создать пустой)). Настройте его компонент Transform: P:[0, 0, 0] R:[0, 0, 0] S:[1, 1, 1].

2. Создайте куб (GameObject > 3D Object > Cube (Игровой объект > 3D объект > Куб)) и перетащите его на объект _Hero, сделав его дочерним объектом по отношению к _Hero. Дайте кубу имя Wing и настройте его компонент Transform: P:[0, -1, 0] R:[0, 0, 45] S:[3, 3, 0.5].

3. Создайте пустой игровой объект с именем Cockpit и сделайте его дочерним по отношению к _Hero.

4. Создайте куб и сделайте его дочерним по отношению к Cockpit (для этого можно щелкнуть правой кнопкой мыши на Cockpit и выбрать в контекстном меню пункт 3D Object > Cube (3D объект > Куб)). Настройте его компонент Transform: P:[0, 0, 0] R:[315, 0, 45] S:[1, 1, 1].

5. Снова выберите объект Cockpit и настройте его компонент Transform: P:[0, 0, 0] R:[0, 0, 180] S:[1, 3, 1]. В данном случае используется тот же трюк, с которым вы познакомились в главе 27 «Объектно-ориентированное мышление», позволяющий быстро создать корабль вытянутой формы.

6. Выберите объект _Hero в иерархии и щелкните на кнопке Add Component (Добавить компонент) в инспекторе. Выберите в открывшемся меню пункт New Script (Новый сценарий). Дайте вновь созданному сценарию имя Hero, установите флажок Language is C Sharp (Язык C Sharp) и щелкните на кнопке Create and Add

(Создать и добавить). Это еще один способ создать новый сценарий и подключить его к игровому объекту. В панели Project (Проект) перетащите сценарий Hero в папку __Scripts.

7. Добавьте в _Hero компонент Rigidbody, выбрав _Hero в иерархии, и затем выберите в меню пункт Add Component > Physics > Rigidbody (Добавить компонент > Физика > Твердое тело), щелкнув на кнопке Add Component (Добавить компонент) в инспекторе. Настройте следующие поля в компоненте Rigidbody объекта _Hero:
   - Снимите флажок Use Gravity.
   - Установите флажок isKinematic.
   - В разделе Constraints: установите флажки Freeze Position Z и Freeze Rotation X, Y и Z.

   Позднее мы еще вернемся к объекту _Hero, но пока этих настроек достаточно.
8. Сохраните сцену! Не забывайте сохранять сцену после любых изменений в ней. Позже я проверю, как вы это запомнили.

Метод Update() в сценарии Hero

Метод `Update()` в следующем листинге сначала читает из `InputManager` значения на горизонтальной и вертикальной осях (см. врезку «Input.GetAxis() и InputManager»), сохраняет полученные в диапазоне между –1 и 1 в переменные `xAxis` и `yAxis`. Вторая половина метода `Update()` выполняет перемещение корабля с учетом значения скорости `speed` и реального времени.

Последняя строка (с комментарием `// c`) поворачивает корабль согласно вводу. Хотя мы зафиксировали угол поворота в компоненте Rigidbody объекта _Hero, его все еще можно изменить вручную, если был установлен флажок isKinematic компонента. (Как рассказывалось в предыдущей главе, `isKinematic = true` означает, что твердое тело Rigidbody все так же обрабатывается физической системой, но не перемещается автоматически в соответствии с `Rigidbody.velocity`.) Этот поворот придает движению корабля выразительность и динамизм, то есть делает его более «сочным»[1].

Откройте сценарий Hero в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Hero : MonoBehaviour {
    static public Hero      S; // Одиночка                                    // a
```

[1] Термин *сочность* в отношении процесса игры был придуман в 2005 году Кайлом Габлером (Kyle Gabler) и другими участниками проекта «Experimental Gameplay Project» в центре развлекательных технологий Университета Карнеги-Меллона. Сочными они называли элементы, дающие «постоянную и обширную обратную связь». Подробнее об этом можно прочитать в их статье «How to Prototype a Game in Under 7 Days» на сайте Gamasutra.

```
    [Header("Set in Inspector")]
    // Поля, управляющие движением корабля
    public float            speed = 30;
    public float            rollMult = -45;
    public float            pitchMult = 30;

    [Header("Set Dynamically")]
    public float            shieldLevel = 1;

    void Awake() {
        if (S == null) {
            S = this; // Сохранить ссылку на одиночку                    // a
        } else {
            Debug.LogError("Hero.Awake() - Attempted to assign second Hero.S!");
        }
    }

    void Update () {
        // Извлечь информацию из класса Input
        float xAxis = Input.GetAxis("Horizontal");                       // b
        float yAxis = Input.GetAxis("Vertical");                         // b

        // Изменить transform.position, опираясь на информацию по осям
        Vector3 pos = transform.position;
        pos.x += xAxis * speed * Time.deltaTime;
        pos.y += yAxis * speed * Time.deltaTime;
        transform.position = pos;

        // Повернуть корабль, чтобы придать ощущение динамизма           // c
        transform.rotation = Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);
    }
}
```

a. Реализация шаблона проектирования «Одиночка» в классе `Hero` (см. раздел «Шаблоны проектирования программного обеспечения» в приложении Б). Код в методе `Awake()` выводит сообщение об ошибке в панель Console (Консоль), обнаружив повторную попытку присвоить значение полю `Hero.S` (что может произойти, если в одной сцене создать два экземпляра игрового объекта с подключенным сценарием Hero или если подключить два компонента сценария Hero к одному игровому объекту).

b. Эти две строки используют класс `Input`, встроенный в Unity, чтобы извлечь информацию из диспетчера ввода Unity InputManager. Подробнее о диспетчере ввода рассказывается во врезке ниже.

c. Строка `transform.rotation...` под этим комментарием меняет угол поворота корабля, исходя из скорости его движения, что придает ощущение отзывчивости и сочности.

Попробуйте сыграть в игру и подвигать корабль клавишами со стрелками или буквами WASD, чтобы увидеть, как он себя ведет. Я выбрал значения для переменных `speed`, `rollMult` и `pitchMult`, которые показались мне оптимальными, но это ваша

игра, и вам могут понравиться другие значения. Поэкспериментируйте с ними в инспекторе, выбрав объект _Hero.

Дополнительные приятные ощущения придает кажущаяся инерция корабля. Когда вы отпускаете клавишу управления движением, кораблю требуется некоторое время, чтобы замедлить полет. Аналогично, после нажатия клавиши требуется некоторое время, чтобы корабль разогнался. Эта кажущаяся инерционность определяется настройками Sensitivity и Gravity осей, которые есть во врезке ниже. Изменение этих настроек в диспетчере ввода InputManager будет влиять на движение и маневренность корабля _Hero.

INPUT.GETAXIS() И INPUTMANAGER

Большая часть кода в `Hero.Update()` должна быть знакомой, однако здесь впервые в книге вы видите метод `Input.GetAxis()`. Диспетчер ввода InputManager в Unity позволяет настраивать разные оси ввода и читать их значения с помощью `Input.GetAxis()`. Оси, поддерживаемые по умолчанию классом `Input`, можно увидеть, выбрав в меню пункт Edit > Project Settings > Input (Правка > Параметры проекта > Ввод).

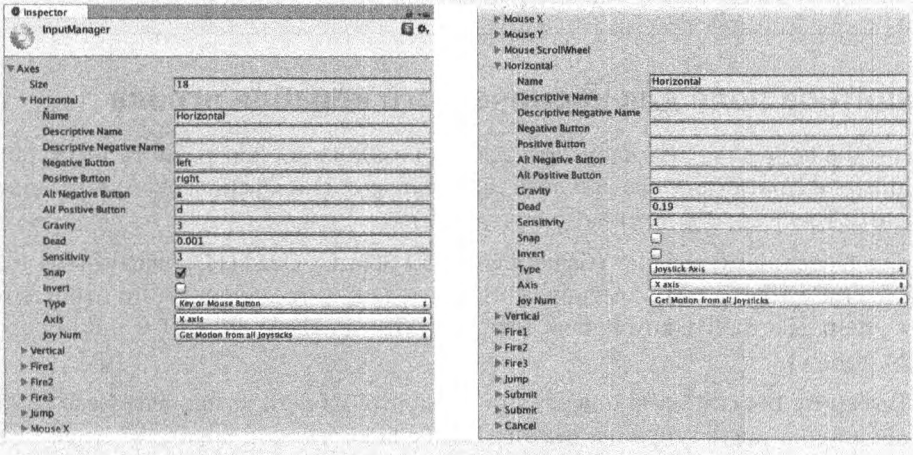


Рис. 30.4. Диспетчер ввода InputManager в Unity имеет множество настроек по умолчанию (здесь разбиты на две половины)

Первое, что бросается в глаза на рис. 30.4, — имена некоторых осей встречаются дважды (например, Horizontal, Vertical и Jump). Как можно судить по двум раскрытым разделам Horizontal на рисунке, это позволяет управлять осью Horizontal либо нажатием клавиш на клавиатуре (слева на рис. 30.4), либо джойстиком (справа). Это одно из главных преимуществ осей ввода — одной осью можно управлять с применением разных устройств ввода. Благодаря этому прочитать значение оси в игре можно единственной строкой кода, вместо того чтобы писать отдельную

строку для обработки джойстика, отдельную — для клавиш со стрелками и отдельную — для клавиш A и D, управляющих перемещением по горизонтали.

Каждый вызов `Input.GetAxis()` возвращает значение `float` между –1 и 1 (и значение 0 по умолчанию). Каждая ось в диспетчере ввода InputManager также включает значения Sensitivity (чувствительность) и Gravity (тяжесть), хотя они влияют только на ввод с клавиатуры и мыши (Key or Mouse Button) (см. изображение слева на рис. 30.4). Чувствительность (Sensitivity) и тяжесть (Gravity) обеспечивают гладкую интерполяцию при нажатии и отпускании клавиши (то есть ось ввода не сразу принимает конечное значение — оно плавно изменяется с течением времени от текущего до конечного). Для оси Horizontal, как показано на рисунке, значение 3 в поле Sensitivity означает, что когда игрок нажимает на клавишу со стрелкой вправо, значение оси изменится от 0 до 1 за 1/3 секунды. Значение 3 в поле Gravity означает, что когда игрок отпускает клавишу со стрелкой вправо, значение оси изменится в обратном направлении, от текущего значения до нуля, за 1/3 секунды. Чем выше значения Sensitivity или Gravity, тем быстрее значение оси достигнет конечного значения.

Узнать больше о диспетчере ввода InputManager, как и о многом другом в Unity, можно, щелкнув на кнопке Help (Справка) с изображением синей книги и знаком вопроса, которая находится между именем InputManager и кнопкой с изображением шестеренки в верхней части инспектора.

Защитное поле для космического корабля игрока

Защитное поле для _Hero формируется как комбинация прозрачного текстурированного квадрата (для визуального представления) и сферического коллайдера Sphere Collider (для обработки столкновений):

1. Создайте новый квадрат (GameObject > 3D Object > Quad (Игровой объект > 3D объект > Квадрат)) с именем Shield и сделайте его дочерним по отношению к _Hero. Настройте компонент Transform объекта Shield: P:[0, 0, 0], R:[0, 0, 0], S:[8, 8, 8].
2. Выберите объект Shield в иерархии и удалите из него компонент Mesh Collider, щелкнув в инспекторе на пиктограмме с шестеренкой справа от имени Mesh Collider и выбрав в открывшемся меню пункт Remove Component (Удалить компонент). Добавьте компонент Sphere Collider (Component > Physics > Sphere Collider (Компонент > Физика > Сферический коллайдер)).
3. Создайте новый материал (Assets > Create > Material (Ресурсы > Создать > Материал)) с именем Mat_Shield и поместите его в папку _Materials в панели Project (Проект). Перетащите Mat_Shield на объект Shield в _Hero в иерархии, чтобы связать его с объектом квадрата Shield.
4. Выберите Shield в иерархии, теперь материал Mat_Shield появится в инспекторе, в разделе с настройками объекта Shield. В поле Shader материала Mat_Shield выберите ProtoTools > UnlitAlpha. Под открывшимся окном выбора шейдера для

Mat_Shield должна появиться область, позволяющая выбрать основной цвет материала, а также текстуру. (Если свойства шейдера не видны, щелкните один раз на имени Mat_Shield в инспекторе, и они должны появиться.)

5. Щелкните на кнопке Select (Выбрать) в правом нижнем углу квадратика с изображением текстуры и выберите текстуру с именем Shield. Щелкните на цветовой шкале в поле Main Color и выберите ярко-зеленый цвет (RGBA:[0, 255, 0, 255]). Затем введите значения в поля:

   - 0,2 в Tiling.x.
   - 0,4 в Offset.x.
   - В поле Tiling.y оставьте значение 1,0.
   - В поле Offset.y оставьте значение 0.

   Текстура Shields по задумке разбита на пять сегментов по горизонтали. Значение 0,2 в поле Tiling.x заставляет Mat_Shield использовать только 1/5 от всей текстуры Shields по оси X, а поле Offset.x определяет, какой сегмент из пяти выбрать. Попробуйте использовать значения 0, 0,2, 0,4, 0,6 и 0,8 в поле Offset.x, чтобы увидеть изображение защитного поля с разными уровнями мощности.

6. Создайте новый сценарий на C# с именем Shield (Assets > Create > C# Script (Ресурсы > Создать > Сценарий C#)). Перетащите его в папку __Scripts в панели Project (Проект) и затем перетащите его же на объект Shield в иерархии, чтобы создать компонент сценария в игровом объекте Shield.

7. Откройте сценарий Shield в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Shield : MonoBehaviour {
    [Header("Set in Inspector")]
    public float    rotationsPerSecond = 0.1f;

    [Header("Set Dynamically")]
    public int      levelShown = 0;

    // Скрытые переменные, не появляющиеся в инспекторе
    Material        mat;                                                    // a

    void Start() {
        mat = GetComponent<Renderer>().material;                            // b
    }

    void Update () {
        // Прочитать текущую мощность защитного поля из объекта-одиночки Hero
        int currLevel = Mathf.FloorToInt( Hero.S.shieldLevel );             // c
        // Если она отличается от levelShown...
        if (levelShown != currLevel) {
            levelShown = currLevel;
```

```
            // Скорректировать смещение в текстуре, чтобы отобразить поле с другой
            // мощностью
            mat.mainTextureOffset = new Vector2( 0.2f*levelShown, 0 );      // d
        }
        // Поворачивать поле в каждом кадре с постоянной скоростью
        float rZ = -(rotationsPerSecond*Time.time*360) % 360f;              // e
        transform.rotation = Quaternion.Euler( 0, 0, rZ );
    }
}
```

a. Поле `mat` типа `Material` объявлено как скрытое, то есть оно не будет отображаться в инспекторе и не будет доступно за пределами этого класса `Shield`.

b. В методе `Start()` переменная `mat` определяется как материал компонента Renderer этого игрового объекта (Shield в иерархии). Это позволит быстро настроить текстуру в строке с комментарием `// d`.

c. Значение `Hero.S.shieldLevel` округляется вниз до ближайшего целого, и результат присваивается `currLevel`. Округление `shieldLevel` гарантирует, что переход к новому смещению в текстуре по оси X произойдет точно к границе между сегментами, а не между двумя границами.

d. Эта строка корректирует смещение по оси X в текстуре для материала Mat_Shield, чтобы показать защитное поле заданной мощности.

e. Эта и следующая строки медленно поворачивают игровой объект Shield относительно оси Z.

Сохранение корабля _Hero на экране

Теперь движение корабля _Hero выглядит достаточно привлекательно, а вращающееся защитное поле создает красивый эффект, но в данный момент легко можно вывести корабль за границы экрана. Для решения этой проблемы мы создадим компонент сценария многократного пользования[1]. Подробнее о компонентно-ориентированном проектировании рассказывается в главе 27 «Объектно-ориентированное мышление», а также в разделе «Шаблоны проектирования программного обеспечения» приложения Б «Полезные идеи». Вкратце, компонент — это небольшой фрагмент кода, предназначенный для расширения возможностей игровых объектов, не конфликтующий с другим кодом, подключенным к этому объекту. Все компоненты в Unity, с которыми вы работали в инспекторе (такие, как Renderer, Transform и т. д.), следуют этому правилу. Теперь мы сделаем то же самое с небольшим сценарием, который должен предотвратить выход корабля _Hero за границы экрана. Имейте в виду, что этот сценарий работает только с ортографическими камерами.

[1] В первом издании этой книги была реализована намного более сложная система предотвращения выхода игровых объектов за границы экрана, во многом слишком избыточная для этой главы и слишком запутанная. Во втором издании я заменил ее новой версией, чтобы упростить главу и закрепить идею использования компонентов.

1. Выберите _Hero в иерархии и, щелкнув на кнопке Add Component (Добавить компонент) в инспекторе, выберите в открывшемся меню пункт New Script (Новый сценарий). Дайте сценарию имя BoundsCheck и щелкните на кнопке Create and Add (Создать и добавить). Перетащите сценарий BoundsCheck в папку __Scripts в панели Project (Проект).

2. Откройте сценарий BoundsCheck и добавьте в него следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// В каждой из четырех следующих строк сначала введите /// и затем нажмите Tab.
/// <summary>
/// Предотвращает выход игрового объекта за границы экрана.
/// Важно: работает ТОЛЬКО с ортографической камерой Main Camera в [ 0, 0, 0 ].
/// </summary>
public class BoundsCheck : MonoBehaviour {                                    // a
    [Header("Set in Inspector")]
    public float    radius = 1f;

    [Header("Set Dynamically")]
    public float    camWidth;
    public float    camHeight;

    void Awake() {
        camHeight = Camera.main.orthographicSize;                             // b
        camWidth = camHeight * Camera.main.aspect;                            // c
    }

    void LateUpdate () {                                                      // d
        Vector3 pos = transform.position;

        if (pos.x > camWidth - radius) {
            pos.x = camWidth - radius;
        }
        if (pos.x < -camWidth + radius) {
            pos.x = -camWidth + radius;
        }

        if (pos.y > camHeight - radius) {
            pos.y = camHeight - radius;
        }
        if (pos.y < -camHeight + radius) {
            pos.y = -camHeight + radius;
        }

        transform.position = pos;
    }

    // Рисует границы в панели Scene (Сцена) с помощью OnDrawGizmos()
    void OnDrawGizmos () {                                                    // e
        if (!Application.isPlaying) return;
        Vector3 boundSize = new Vector3(camWidth*2, camHeight*2, 0.1f);
```

```
        Gizmos.DrawWireCube(Vector3.zero, boundSize);
    }
}
```

a. Поскольку этот сценарий предназначен для многократного использования, будет полезно добавить в него внутреннюю документацию с описанием. Строки перед объявлением класса, начинающиеся с `///`, обнаруживаются встроенной в C# системой документирования[1]. Следующий далее текст между тегами `<summary>` интерпретируется как краткое описание целей использования класса. После ввода сценария наведите указатель мыши на имя класса `BoundsCheck` в строке с комментарием `// a`, и на экране должна появиться всплывающая подсказка с этим описанием.

b. `Camera.main` открывает доступ к первой камере с тегом `MainCamera` в сцене. Далее, если камера ортографическая, `.orthographicSize` вернет число из поля Size в инспекторе (в данном случае 40). То есть переменной `camHeight` будет присвоено расстояние от начала мировых координат (позиция [0, 0, 0]) до верхнего или нижнего края экрана.

c. `Camera.main.aspect` — это отношение ширины к высоте поля зрения камеры, как определяет отношение сторон панели Game (Игра) — в настоящий момент Portrait (3:4). Умножив `camHeight` на `.aspect`, можно получить расстояние от центра до левой или правой границы экрана.

d. `LateUpdate()` вызывается в каждом кадре после вызова методов `Update()` всех игровых объектов. Если бы этот код находился в функции `Update()`, он мог бы выполниться до или после вызова `Update()` в сценарии Hero. Поместив этот код в `LateUpdate()`, мы исключили вероятность *состояния гонки* между двумя функциями `Update()` и гарантировали, что `Hero.Update()` в каждом кадре будет перемещать игровой объект _Hero в новое местоположение до вызова этой функции, ограничивающей перемещение _Hero границами экрана.

e. `OnDrawGizmos()` — встроенный метод класса `MonoBehaviour`, который может рисовать на поверхности панели Scene (Сцена).

Состояние гонки возникает, когда порядок выполнения двух фрагментов кода (то есть A перед B или B перед A) имеет значение, но нет возможности управлять этим порядком. Например, если бы `BoundsCheck.LateUpdate()` в этом сценарии выполнялся перед `Hero.Update()`, игровой объект _Hero мог бы выйти за границы экрана (потому что сначала была бы выполнена проверка выхода за границы, а потом перемещение корабля). Использование `LateUpdate()` в сценарии BoundsCheck гарантирует правильный порядок выполнения двух фрагментов кода.

3. Щелкните на кнопке Play (Играть) и попробуйте поперемещать корабль по экрану. Со значением радиуса `radius` по умолчанию корабль будет частично заходить за границу экрана так, что не менее метра корпуса корабля будет оставаться на

[1] Чтобы узнать больше, выполните поиск в интернете по фразе «C# XML документация».

экране. Если в инспекторе присвоить полю BoundsCheck.radius значение 4, корабль всегда будет находиться на экране целиком. Если присвоить полю radius значение –4, корабль сможет выходить за границу экрана, но будет оставаться рядом с ней, не удаляясь, пока вы не решите вернуть его обратно. Остановите игру и введите в поле radius значение 4.

Добавление вражеских кораблей

В главе 26 «Классы» рассказывалось немного о классе Enemy и его подклассах в играх, подобных этой. Там вы узнали, что можно создать суперкласс, представляющий всех врагов, который можно расширить посредством подклассов. В этой игре мы тоже реализуем такое расширение (в следующей главе), но сначала создадим графические ресурсы.

Графические изображения вражеских кораблей

Поскольку корабль игрока имеет угловатые формы, все вражеские корабли будут сконструированы из сфер, как показано на рис. 30.5.

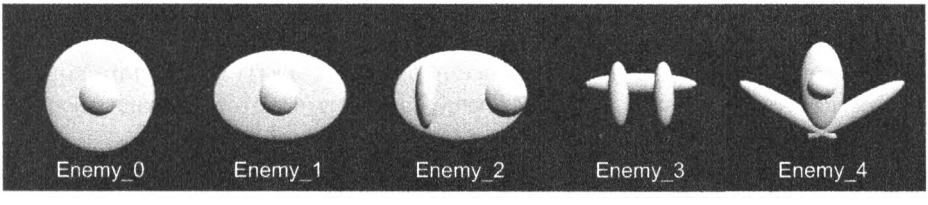


Рис. 30.5. Все пять типов вражеских кораблей (освещенность в Unity будет немного отличаться)

Чтобы создать графический ресурс с изображением Enemy_0, выполните следующие шаги:

1. Создайте пустой игровой объект с именем Enemy_0 и настройте его компонент Transform: P:[-20, 10, 0], R:[0, 0, 0], S:[1, 1, 1]. Местоположение выбрано так, чтобы этот корабль не перекрывался кораблем игрока _Hero, пока мы будем его конструировать.
2. Создайте сферу с именем Cockpit, сделайте ее дочерней по отношению к Enemy_0 и настройте ее компонент Transform: P:[0, 0, 0], R:[0, 0, 0], S:[2, 2, 1].
3. Создайте вторую сферу с именем Wing, сделайте ее дочерней по отношению к Enemy_0 и настройте ее компонент Transform: P:[0, 0, 0], R:[0, 0, 0], S:[5, 5, 0.5].

   Предыдущие три шага по созданию корабля Enemy_0 можно также записать, как показано ниже:

```
Enemy_0 (Empty)          P:[ -20, 10, 0 ]  R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
        Cockpit (Sphere) P:[ 0, 0, 0 ]     R:[ 0, 0, 0 ]   S:[ 2, 2, 1 ]
        Wing (Sphere)    P:[ 0, 0, 0 ]     R:[ 0, 0, 0 ]   S:[ 5, 5, 0.5 ]
```

4. Следуя этому формату, создайте четыре других вражеских корабля. По завершении вражеские корабли должны выглядеть у вас, как показано на рис. 30.5.

```
Enemy_1
    Enemy_1 (Empty)          P:[ -10, 10, 0 ]  R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
            Cockpit (Sphere) P:[ 0, 0, 0 ]     R:[ 0, 0, 0 ]   S:[ 2, 2, 1 ]
            Wing (Sphere)    P:[ 0, 0, 0 ]     R:[ 0, 0, 0 ]   S:[ 6, 4, 0.5 ]
Enemy_2
    Enemy_2 (Empty)          P:[ 0, 10, 0 ]    R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
            Cockpit (Sphere) P:[ -1.5, 0, 0 ]  R:[ 0, 0, 0 ]   S:[ 1, 3, 1 ]
            Reactor (Sphere) P:[ 2, 0, 0 ]     R:[ 0, 0, 0 ]   S:[ 2, 2, 1 ]
            Wing (Sphere)    P:[ 0, 0, 0 ]     R:[ 0, 0, 0 ]   S:[ 6, 4, 0.5 ]
Enemy_3
    Enemy_3 (Empty)          P:[ 10, 10, 0 ]   R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
            CockpitL (Sphere) P:[ -1, 0, 0 ]   R:[ 0, 0, 0 ]   S:[ 1, 3, 1 ]
            CockpitR (Sphere) P:[ 1, 0, 0 ]    R:[ 0, 0, 0 ]   S:[ 1, 3, 1 ]
            Wing (Sphere)    P:[ 0, 0.5, 0 ]   R:[ 0, 0, 0 ]   S:[ 5, 1, 0.5 ]
Enemy_4
    Enemy_4 (Empty)          P:[ 20, 10, 0 ]   R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
            Cockpit (Sphere) P:[ 0, 1, 0 ]     R:[ 0, 0, 0 ]   S:[ 1.5, 1.5, 1.5 ]
            Fuselage (Sphere) P:[ 0, 1, 0 ]    R:[ 0, 0, 0 ]   S:[ 2, 4, 1 ]
            WingL (Sphere)   P:[ -1.5, 0, 0 ]  R:[ 0, 0, -30 ] S:[ 5, 1, 0.5 ]
            WingR (Sphere)   P:[ 1.5, 0, 0 ]   R:[ 0, 0, 30 ]  S:[ 5, 1, 0.5 ]
```

5. Добавьте компонент Rigidbody во все игровые объекты вражеских кораблей (то есть Enemy_0, Enemy_1, Enemy_2, Enemy_3 и Enemy_4). Чтобы добавить Rigidbody, выполните следующие шаги:

   a. Выделите Enemy_0 в иерархии и выберите в меню пункт Component > Physics > Rigidbody (Компонент > Физика > Твердое тело).

   b. В настройках компонента Rigidbody снимите флажок Use Gravity.

   c. Установите флажок isKinematic.

   d. Щелкните на пиктограмме с треугольником рядом с именем раздела Constraints, установите флажки Freeze Position Z и Freeze Rotation X, Y и Z.

6. Скопируйте компонент Rigidbody из объекта Enemy_0 во все другие объекты вражеских кораблей. С этой целью для каждого вражеского корабля выполните следующие действия:

   a. Выделите Enemy_0 в иерархии и щелкните на кнопке с шестеренкой в правом верхнем углу, в разделе с настройками компонента Rigidbody.

   b. В открывшемся меню выберите пункт Copy Component (Копировать компонент).

   c. Выберите в иерархии объект вражеского корабля, в который хотите добавить Rigidbody (например, Enemy_1).

d. Щелкните на кнопке с шестеренкой в правом верхнем углу, в разделе с настройками компонента Transform.

e. В открывшемся меню выберите пункт Paste Component As New (Вставить компонент как новый).

В результате к игровому объекту вражеского корабля будет присоединен компонент Rigidbody с такими же настройками, как у компонента Rigidbody, скопированного из объекта Enemy_0. Обязательно проделайте эти операции для всех остальных вражеских кораблей. Если движущийся игровой объект не имеет компонента Rigidbody, его коллайдер не будет перемещаться вместе с ним. Напротив, если движущийся игровой объект имеет компонент Rigidbody, его коллайдер и коллайдеры всех вложенных дочерних объектов в каждом кадре будут изменять свое местоположение синхронно с изменением местоположения игрового объекта (именно поэтому нет необходимости добавлять компонент Rigidbody в дочерние игровые объекты, составляющие вражеские корабли).

7. Перетащите каждый вражеский корабль в папку _Prefabs в панели Project (Проект), чтобы создать из них шаблоны.

8. Удалите все экземпляры вражеских кораблей из иерархии, кроме Enemy_0.

Сценарий Enemy

Чтобы создать сценарий Enemy, выполните следующие шаги:

1. Создайте новый сценарий на C# с именем Enemy и поместите его в папку __Scripts.

2. Выберите Enemy_0 в панели Project (Проект) — не в иерархии. В инспекторе щелкните на кнопке Add Component (Добавить компонент) и в открывшемся меню выберите пункт Scripts > Enemy (Сценарии > Enemy). Если после этого щелкнуть на объекте Enemy_0 в панели Project (Проект) или Hierarchy (Иерархия), вы должны увидеть подключенный компонент Enemy (Script).

3. Откройте сценарий в MonoDevelop и введите следующий код:

```
using System.Collections;          // Необходимо для доступа к массивам и другим
                                   // коллекциям
using System.Collections.Generic; // Необходимо для доступа к спискам и словарям
using UnityEngine;                 // Необходимо для доступа к Unity

public class Enemy : MonoBehaviour {
    [Header("Set in Inspector: Enemy")]
    public float      speed = 10f;     // Скорость в м/с
    public float      fireRate = 0.3f; // Секунд между выстрелами (не используется)
    public float      health = 10;
    public int        score = 100;     // Очки за уничтожение этого корабля

    // Это свойство: метод, действующий как поле
    public Vector3 pos {                                                    // a
        get {
```

```
            return( this.transform.position );
        }
        set {
            this.transform.position = value;
        }
    }

    void Update() {
        Move();
    }

    public virtual void Move() {                                    // b
        Vector3 tempPos = pos;
        tempPos.y -= speed * Time.deltaTime;
        pos = tempPos;
    }
}
```

a. Как рассказывалось в главе 26 «Классы», *свойство* — это функция, маскирующаяся под поле. Это означает, что значение `pos` можно читать и изменять, как если бы это была переменная класса `Enemy`.

b. Метод `Move()` получает текущее местоположение данного объекта Enemy_0, перемещает его вниз, вдоль оси Y, и присваивает новые координаты свойству `pos` (устанавливает местоположение игрового объекта).

4. В Unity щелкните на кнопке Play (Играть), и экземпляр Enemy_0 в сцене должен начать движение вниз. Но в данный момент этот экземпляр выйдет за нижнюю границу экрана и продолжит существование, пока вы не остановите игру. Вы должны заставить вражеский корабль самоуничтожаться, когда он целиком выйдет за нижнюю границу экрана. Это отличный повод повторно задействовать компонент BoundsCheck.

5. Чтобы подключить сценарий BoundsCheck к экземпляру Enemy_0, выберите его в иерархии (не в панели Project (Проект)). В инспекторе щелкните на кнопке Add Component (Добавить компонент) и в открывшемся меню выберите пункт Scripts > BoundsCheck (Сценарии > BoundsCheck). В результате сценарий подключится к экземпляру Enemy_0 в иерархии, но не к шаблону Enemy_0 в панели Project (Проект). Об этом можно судить по тому, как весь текст в компоненте BoundsCheck (Script) будет выделен жирным.

6. Примените изменения в экземпляре Enemy_0 к его шаблону, щелкнув на кнопке Apply (Применить) в верхней части инспектора. Теперь выберите шаблон Enemy_0 в панели Project (Проект), чтобы убедиться, что сценарий был подключен к нему.

7. Выберите экземпляр Enemy_0 в иерархии и в разделе BoundsCheck, в инспекторе, введите в поле radius число –2.5. Обратите внимание, что это значение выделено жирным, потому что оно отличается от значения в шаблоне. Снова щелкните на кнопке Apply (Применить) в верхней части инспектора, и значение в поле radius больше не будет выделено жирным, показывая тем самым, что оно не отличается от значения в шаблоне.

8. Щелкните на кнопке Play (Играть), и вы увидите, что экземпляр Enemy_0 остановился сразу, как только оказался за нижней границей экрана. Однако нам нужно, чтобы Enemy_0 не оставался за границей экрана, на самом деле мы должны определить момент, когда он вышел из видимой области, и уничтожить его.

9. Для этого внесите в сценарий BoundsCheck следующие изменения, выделенные жирным.

```
/// <summary>
/// Предотвращает выход игрового объекта за границы экрана.
/// Важно: работает ТОЛЬКО с ортографической камерой Main Camera в [ 0, 0, 0 ].
/// </summary>
public class BoundsCheck : MonoBehaviour {
    [Header("Set in Inspector")]
    public float    radius = 1f;
    public bool     keepOnScreen = true;                                    // a

    [Header("Set Dynamically")]
    public bool     isOnScreen = true;                                      // b
    public float    camWidth;
    public float    camHeight;

    void Awake() { ... } // Напомню: многоточие означает, что метод не изменился.

    void LateUpdate () {
        Vector3 pos = transform.position;                                   // c
        isOnScreen = true;                                                  // d

        if ( pos.x > camWidth - radius ) {
            pos.x = camWidth - radius;
            isOnScreen = false;                                             // e
        }
        if ( pos.x < -camWidth + radius ) {
            pos.x = -camWidth + radius;
            isOnScreen = false;                                             // e
        }

        if ( pos.y > camHeight - radius ) {
            pos.y = camHeight - radius;
            isOnScreen = false;                                             // e
        }
        if ( pos.y < -camHeight + radius ) {
            pos.y = -camHeight + radius;
            isOnScreen = false;                                             // e
        }

        if ( keepOnScreen && !isOnScreen ) {                                // f
            transform.position = pos;                                       // g
            isOnScreen = true;
        }
    }

    ...
}
```

a. keepOnScreen помогает определить режим работы сценария BoundsCheck — когда он не позволяет игровому объекту выйти за границы экрана (true) и когда позволяет, но уведомляет вас, что объект вышел за пределы экрана (false).

b. isOnScreen получает значение false, если игровой объект вышел за границы экрана. Точнее, когда он миновал границу экрана и удалился от нее дальше, чем на величину радиуса radius. Именно поэтому radius для Enemy_0 получил значение –2.5, чтобы объект полностью скрылся с экрана, прежде чем признак isOnScreen получит значение false.

c. Напомню еще раз: многоточие в этом месте означает, что метод Awake() не изменился.

d. Переменная isOnScreen должна иметь значение true, если явно не получит значение false. В соответствии с этим правилом она вновь получит значение true, если в последнем кадре игровой объект оказался за границами экрана, но был возвращен на место в этом же кадре.

e. Если условие в любой из этих четырех инструкций if выполняется, то игровой объект оказался за пределами области, где он должен находиться. Переменной isOnScreen присваивается значение false, а свойство pos корректируется так, чтобы при необходимости легко можно было вернуть игровой объект обратно «на экран».

f. Если keepOnScreen имеет значение true, то сценарий должен заставить игровой объект оставаться в пределах экрана. Если keepOnScreen имеет значение true и isOnScreen имеет значение false, то игровой объект покинул границы экрана и его нужно вернуть назад. В этом случае в transform.position записывается обновленное значение pos, соответствующее позиции на экране, а переменной isOnScreen присваивается true, потому что игровой объект только что вернулся на экран.

   Если keepOnScreen имеет значение false, тогда значение pos не записывается в transform.position — игровому объекту позволяется покинуть экран, и переменная isOnScreen продолжает хранить false. Также возможна ситуация, когда игровой объект оставался на экране все время, и в этом случае isOnScreen сохранит значение true, присвоенное в строке с комментарием // d.

g. Обратите внимание, что теперь эта строка находится внутри инструкции if.

К счастью, все эти изменения не оказывают отрицательного влияния на поведение _Hero, и все продолжает замечательно работать. Мы создали компонент многократного использования, который смогли применить к обоим игровым объектам, _Hero и Enemy.

Удаление вражеского корабля после выхода за границы экрана

Теперь, когда сценарий BoundsCheck научился сообщать о выходе Enemy_0 за границы экрана, нам нужно правильно настроить его для этого.

1. Снимите флажок keepOnScreen в компоненте BoundsCheck (Script) шаблона Enemy_0, находящегося в папке _Prefabs в панели Project (Проект).
2. Чтобы изменения достигли экземпляра Enemy_0 в иерархии, выберите его и щелкните на кнопке с шестеренкой справа от названия компонента BoundsCheck (Script) в инспекторе. В появившемся меню выберите пункт Revert to Prefab (Вернуть из шаблона), чтобы скопировать настройки из шаблона в экземпляр в иерархии.

   После этого настройки компонента BoundsCheck (Script) в шаблоне Enemy_0 (в панели Project (Проект)) и в экземпляре Enemy_0 (в иерархии) должны выглядеть так, как показано на рис. 30.6.

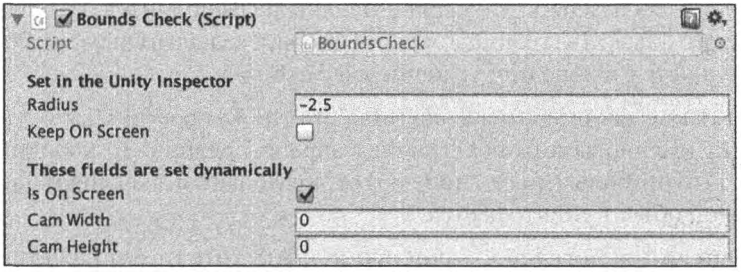


Рис. 30.6. Настройки компонента BoundsCheck (Script) в шаблоне и экземпляре Enemy_0

3. Добавьте в сценарий Enemy следующий код, выделенный жирным:

```
public class Enemy : MonoBehaviour {
    ...
    public int        score = 100; // Очки за уничтожение этого корабля

    private BoundsCheck bndCheck;                                          // a

    void Awake() {                                                         // b
        bndCheck = GetComponent<BoundsCheck>();
    }

    ...

    void Update() {
        Move();

        if ( bndCheck != null && !bndCheck.isOnScreen ) {                  // c
            // Убедиться, что корабль вышел за нижнюю границу экрана
            if ( pos.y < bndCheck.camHeight - bndCheck.radius ) {          // d
                // Корабль за нижней границей, поэтому его нужно уничтожить
                Destroy( gameObject );
            }
        }
    }

    ...
}
```

a. В этой скрытой переменной сценарий Enemy хранит ссылку на компонент BoundsCheck (Script), подключенный к тому же игровому объекту.

b. Этот метод `Awake()` получает ссылку на компонент сценария BoundsCheck, подключенного к этому игровому объекту. Если такого компонента нет, `bndCheck` получит значение `null`. Код, подобный этому, который получает ссылки на компоненты и кэширует их, часто помещается в метод `Awake()`, чтобы они были доступны сразу же после создания экземпляра игрового объекта.

c. Сначала проверяется наличие действительной ссылки в `bndCheck`. Если подключить сценарий `Enemy` к игровому объекту, не имеющему компонента сценария BoundsCheck, это первое условие не выполнится. Проверка выхода игрового объекта за границы экрана (как определено в сценарии BoundsCheck) выполняется только при условии `bndCheck != null`.

d. Если `isOnScreen` имеет значение `false`, эта строка проверит, вышел ли игровой объект именно за нижнюю границу экрана, сравнив его координату `pos.y` с достаточно большим (по модулю) отрицательным значением. Если это так, игровой объект уничтожается.

Теперь сценарий действует в точности как мы хотели, но кажется немного странным выполнять одно и то же сравнение `pos.y` с `camHeight` и `radius` здесь и в BoundsCheck.

В программировании принято писать классы на C# (или компоненты) так, чтобы все они выполняли свою работу и не имели подобных пересечений. Поэтому изменим сценарий BoundsCheck и заставим его сообщать, какую границу экрана пересек игровой объект, покидая его.

4. Измените сценарий BoundsCheck, добавив следующие строки, выделенные жирным:

```
public class BoundsCheck : MonoBehaviour {
    ...
    public float camHeight;
    [HideInInspector]
    public bool     offRight, offLeft, offUp, offDown;                    // a

    void Start() { ... }

    void LateUpdate () {
        Vector3 pos = transform.position;
        isOnScreen = true;
        offRight = offLeft = offUp = offDown = false;                    // b

        if ( pos.x > camWidth - radius ) {
            pos.x = camWidth - radius;
            offRight = true;                                             // c
        }
        if ( pos.x < -camWidth + radius ) {
            pos.x = -camWidth + radius;
```

```
            offLeft = true;                                          // c
        }

        if ( pos.y > camHeight - radius ) {
            pos.y = camHeight - radius;
            offUp = true;                                            // c
        }
        if ( pos.y < -camHeight + radius ) {
            pos.y = -camHeight + radius;
            offDown = true;                                          // c
        }

        isOnScreen = !(offRight || offLeft || offUp || offDown);     // d
        if ( keepOnScreen && !isOnScreen ) {
            transform.position = pos;
            isOnScreen = true;
            offRight = offLeft = offUp = offDown = false;            // e
        }
    }

    ...
}
```

a. Здесь объявляются четыре переменные, по одной для каждой границы экрана, которую пересек игровой объект, покинув экран. По умолчанию всем им присваивается значение `false`. Строка `[HideInInspector]`, предшествующая этому объявлению, препятствует появлению этих четырех полей в инспекторе даже при том, что они остаются общедоступными и могут читаться (и изменяться) другими классами. Атрибут `[HideInInspector]` применяется ко всем четырем переменным `off__` (то есть `offRight`, `offLeft` и т. д.), потому что все они объявляются в одной строке под ним. Если бы переменные `off__` объявлялись в четырех отдельных строках, нам пришлось бы добавить атрибут `[HideInInspector]` перед каждой из них, чтобы добиться того же эффекта.

b. В каждом вызове `LateUpdate()` сначала всем четырем переменным `off__` присваивается `false`. В этой строке сначала переменная `offDown` получит значение `false`, затем `offUp` получит значение `offDown` (то есть `false`) и так далее, пока все переменные `off__` не получат значение `false`. Эта строка добавлена взамен строки, присваивавшей `true` переменной `isOnScreen`.

c. Далее каждую инструкцию `isOnScreen = false;` мы заменили инструкцией `off__ = true;`, чтобы позднее можно было определить, какую из границ экрана пересек игровой объект, покидая экран. Может так случиться, что сразу две из этих переменных `off__` получат значение `true` — например, когда игровой объект покинет экран через правый нижний угол.

d. Здесь проверяются значения всех переменных `off__` и присваивается соответствующее значение переменной `isOnScreen`. Во-первых, в круглых скобках ко всем четырем переменным `off__` применяется логическая операция ИЛИ

(||). Если хотя бы одна из них имеет значение true, все выражение в круглых скобках вернет true. Затем к результату применяется логическая операция НЕ (!) и окончательный результат присваивается переменной isOnScreen. То есть если хотя бы одна переменная off__ имеет значение true, isOnScreen получит значение false, в противном случае isOnScreen получит значение true.

e. Если keepOnScreen имеет значение true, этот игровой объект принудительно возвращается в пределы экрана, переменной isOnScreen присваивается true, а все четыре переменные off__ получают значение false.

5. Теперь внесите в сценарий Enemy следующие изменения, выделенные жирным, чтобы воспользоваться улучшениями в сценарии BoundsCheck.

```
public class Enemy : MonoBehaviour {
    ...

    void Update() {
        Move();

        if ( bndCheck != null && bndCheck.offDown ) {                    // a
            // Корабль за нижней границей, поэтому его нужно уничтожить  // b
            Destroy( gameObject );                                       // b
        }
    }

    ...
}
```

a. Теперь достаточно проверить только bndCheck.offDown, чтобы определить, что экземпляр Enemy вышел за нижний край экрана.

b. В этих двух строках убрано по одному отступу, потому что теперь проверка выполняется одной инструкцией if вместо двух.

Реализация класса Enemy заметно упростилась — сейчас он использует возможности компонента BoundsCheck и может делать свою работу, не дублируя его функциональность.

Если вы теперь запустите игру, вы должны увидеть, как корабль Enemy_0 приближается к нижней границе экрана и уничтожается, как только зайдет за нее.

Случайное создание вражеских кораблей

Теперь, закончив работу над вражеским кораблем, можно реализовать случайное создание произвольного количества объектов Enemy_0.

1. Подключите сценарий BoundsCheck к главной камере _MainCamera и снимите в его настройках флажок keepOnScreen.

2. Создайте новый сценарий на C# с именем Main в папке __Scripts. Подключите его к _MainCamera и введите следующий код:

```
using System.Collections;          // Необходимо для доступа к массивам и другим
                                   // коллекциям
using System.Collections.Generic;  // Необходимо для доступа к спискам и словарям
using UnityEngine;                 // Необходимо для доступа к Unity
using UnityEngine.SceneManagement; // Для загрузки и перезагрузки сцен

public class Main : MonoBehaviour {
    static public Main S;                                  // Объект-одиночка Main

    [Header("Set in Inspector")]
    public GameObject[]    prefabEnemies;                  // Массив шаблонов Enemy
    public float           enemySpawnPerSecond = 0.5f;     // Вражеских кораблей
                                                           // в секунду
    public float           enemyDefaultPadding = 1.5f;     // Отступ для
                                                           // позиционирования

    private BoundsCheck    bndCheck;

    void Awake() {
        S = this;
        // Записать в bndCheck ссылку на компонент BoundsCheck этого игрового
        // объекта
        bndCheck = GetComponent<BoundsCheck>();
        // Вызывать SpawnEnemy() один раз (в 2 секунды при значениях по умолчанию)
        Invoke( "SpawnEnemy", 1f/enemySpawnPerSecond );                    // a
    }

    public void SpawnEnemy() {
        // Выбрать случайный шаблон Enemy для создания
        int ndx = Random.Range(0, prefabEnemies.Length);                   // b
        GameObject go = Instantiate<GameObject>( prefabEnemies[ ndx ] );   // c

        // Разместить вражеский корабль над экраном в случайной позиции x
        float enemyPadding = enemyDefaultPadding;                          // d
        if (go.GetComponent<BoundsCheck>() != null) {                      // e
            enemyPadding = Mathf.Abs( go.GetComponent<BoundsCheck>().radius );
        }

        // Установить начальные координаты созданного вражеского корабля   // f
        Vector3 pos = Vector3.zero;
        float xMin = -bndCheck.camWidth + enemyPadding;
        float xMax = bndCheck.camWidth - enemyPadding;
        pos.x = Random.Range( xMin, xMax );
        pos.y = bndCheck.camHeight + enemyPadding;
        go.transform.position = pos;

        // Снова вызвать SpawnEnemy()
        Invoke( "SpawnEnemy", 1f/enemySpawnPerSecond );                    // g
    }
}
```

a. Этот вызов функции `Invoke()` вызовет метод `SpawnEnemy()` через 1/0,5 секунды (то есть через 2 секунды) при значениях по умолчанию.

b. Опираясь на длину массива `prefabEnemies`, эта инструкция выберет случайное число между 0 и ближайшим целым числом меньше `prefabEnemies.Length`, то есть если в массиве `prefabEnemies` имеется четыре шаблона, инструкция вернет 0, 1, 2 или 3. Целочисленная версия `Random.Range()` никогда не вернет число, равное указанному максимальному значению (то есть второму аргументу). Версия, возвращающая число типа `float`, может вернуть максимальное значение.

c. Полученное случайное число `ndx` используется для выбора шаблона игрового объекта из `prefabEnemies`.

d. Первоначально `enemyPadding` получает значение `enemyDefaultPadding`, установленное в инспекторе.

e. Но если выбранный шаблон имеет компонент BoundsCheck, в роли отступа используется радиус этого компонента. Иногда радиус может быть отрицательным, чтобы игровой объект полностью скрылся с экрана, прежде чем его поле `isOnScreen` получит значение `false`, как в случае с Enemy_0, поэтому берется абсолютное значение радиуса.

f. Этот фрагмент кода определяет начальное местоположение нового вражеского корабля. Он использует компонент BoundsCheck объекта _MainCamera, чтобы получить `camWidth` и `camHeight`, случайно выбирает координату X с условием, чтобы вражеский корабль оказался в пределах левой и правой границ экрана, и затем определяет координату Y так, чтобы вражеский корабль оказался точно над верхней границей экрана.

g. Снова вызывается функция `Invoke()`. Она выбрана вместо `InvokeRepeating()`, чтобы иметь возможность динамически менять интервал времени между созданием вражеских кораблей. Если использовать `InvokeRepeating()`, она всегда будет вызывать указанную функцию с заданной частотой. Добавив вызов `Invoke()` в конец `SpawnEnemy()`, мы получили возможность корректировать значение `enemySpawnPerSecond` прямо в процессе игры и тем самым управлять частотой вызова `SpawnEnemy()`.

3. Сохраните сценарий, закончив ввод, вернитесь в Unity и выполните следующие шаги:

   a. Удалите экземпляр Enemy_0 из иерархии (не трогая при этом шаблон в панели Project (Проект)).

   b. Выберите _MainCamera в иерархии.

   c. Раскройте раздел prefabEnemies в компоненте Main (Script) объекта _MainCamera, щелкнув на пиктограмме с треугольником, и в разделе prefabEnemies введите 1 в поле Size.

   d. Перетащите Enemy_0 из панели Project (Проект) в поле Element 0 в массиве prefabEnemies.

   e. *Сохраните сцену!* Надеюсь, вы не забываете делать это?

Если вы не сохраняли сцену после создания всех этих вражеских кораблей, сделайте это прямо сейчас. События, протекающие в недрах компьютера и неподвластные вам, могут вызвать крах Unity, и тогда вам не удастся быстро восстановить все, что вы создали. Привычка постоянно сохранять сцену поможет вам сэкономить массу времени и оградит от неприятных переживаний.

4. Запустите сцену. Теперь каждые 2 секунды на экране должен появляться новый вражеский корабль Enemy_0. Каждый из них должен долетать до нижнего края экрана и исчезать, скрывшись за ним.

Однако в данный момент столкновение _Hero с вражеским кораблем не влечет никаких последствий. Это нужно исправить, и с этой целью обратим наше внимание на слои.

Настройка тегов, слоев и физики

Как рассказывалось в главе 28 «Прототип 1: Apple Picker», слои в Unity управляют множеством аспектов, и в том числе способны определять, какие объекты должны или не должны сталкиваться между собой. Давайте поразмышляем о прототипе Space SHMUP. В этой игре имеется несколько видов игровых объектов, которые можно разместить в разных слоях и по-разному организовать взаимодействие между ними:

- **Слой для корабля игрока (Hero):** корабль _Hero должен сталкиваться с вражескими кораблями, вражескими снарядами и кубиками-бонусами, но не должен сталкиваться со своими снарядами.
- **Слой для снарядов, выпускаемых кораблем игрока (ProjectileHero):** снаряды, выпускаемые кораблем _Hero, должны сталкиваться только с вражескими кораблями.
- **Слой для вражеских кораблей (Enemy):** вражеские корабли должны сталкиваться только с _Hero и снарядами, выпускаемыми им, но не должны сталкиваться с кубиками-бонусами.
- **Слой для снарядов, выпускаемых вражескими кораблями (ProjectileEnemy):** снаряды, выпускаемые вражескими кораблями, должны сталкиваться только с _Hero.
- **Слой для бонусов (PowerUp):** бонусы должны сталкиваться только с _Hero.

Чтобы создать эти слои, а также некоторые теги, которые пригодятся нам позже, выполните следующие шаги:

1. Откройте диспетчер тегов и слоев Tags & Layers в панели Inspector (Инспектор), выбрав в меню пункт Edit > Project Settings > Tags and Layers (Правка > Параметры проекта > Теги и слои). Теги и физические слои — разные вещи, но и те и другие настраиваются в одном месте.

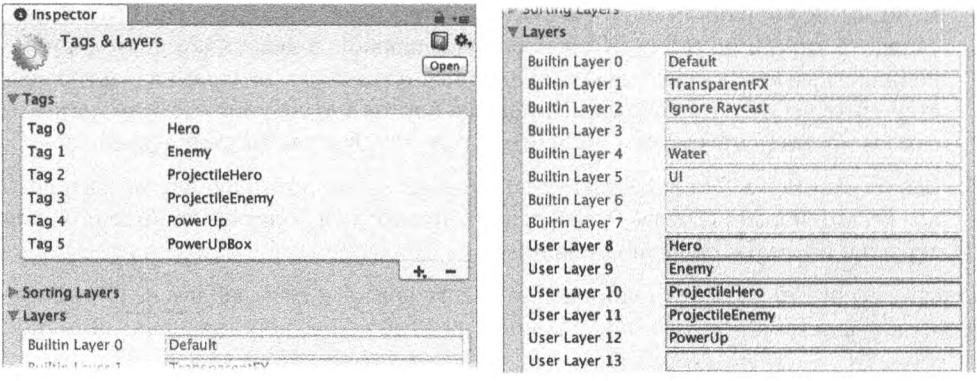


Рис. 30.7. В диспетчере тегов и слоев отображаются имена тегов и слоев для этого прототипа

2. Раскройте раздел **Tags** со списком тегов. Щелкая на значке **+** в конце списка, введите имена тегов, как показано слева на рис. 30.7.

   Если вам плохо видно, вот имена этих тегов: **Hero**, **Enemy**, **ProjectileHero**, **ProjectileEnemy**, **PowerUp** и **PowerUpBox**.

3. Раскройте раздел **Layers**. Начиная с восьмого пользовательского слоя **User Layer 8**, введите имена слоев, как показано справа на рис. 30.7. Встроенные слои **Builtin Layer 0–7** зарезервированы для нужд Unity, но вы можете определить имена для **User Layer 8–31**.

   Имена слоев: **Hero**, **Enemy**, **ProjectileHero**, **ProjectileEnemy** и **PowerUp**.

4. Откройте диспетчер физики **PhysicsManager** (**Edit > Project Settings > Physics** (Правка > Параметры проекта > Физика)) и настройте матрицу столкновений слоев **Layer Collision Matrix**, как показано на рис. 30.8.

→ В Unity есть два диспетчера физики — **Physics** и **Physics2D**. В этой главе вы должны настраивать диспетчер **Physics** (стандартная библиотека трехмерной физики PhysX), а не **Physics2D**.

Как вы узнали в главе 28, матрица внизу **PhysicsManager** определяет, какие слои будут обнаруживать столкновения друг с другом. Если флажок установлен, объекты в этих двух слоях смогут сталкиваться друг с другом, если снят — нет. Снятие флажков может увеличить скорость выполнения игры, потому что движку придется просматривать меньше объектов для определения столкновений. Матрица **Layer Collision Matrix**, показанная на рис. 30.8, обеспечивает поведение, описанное выше.

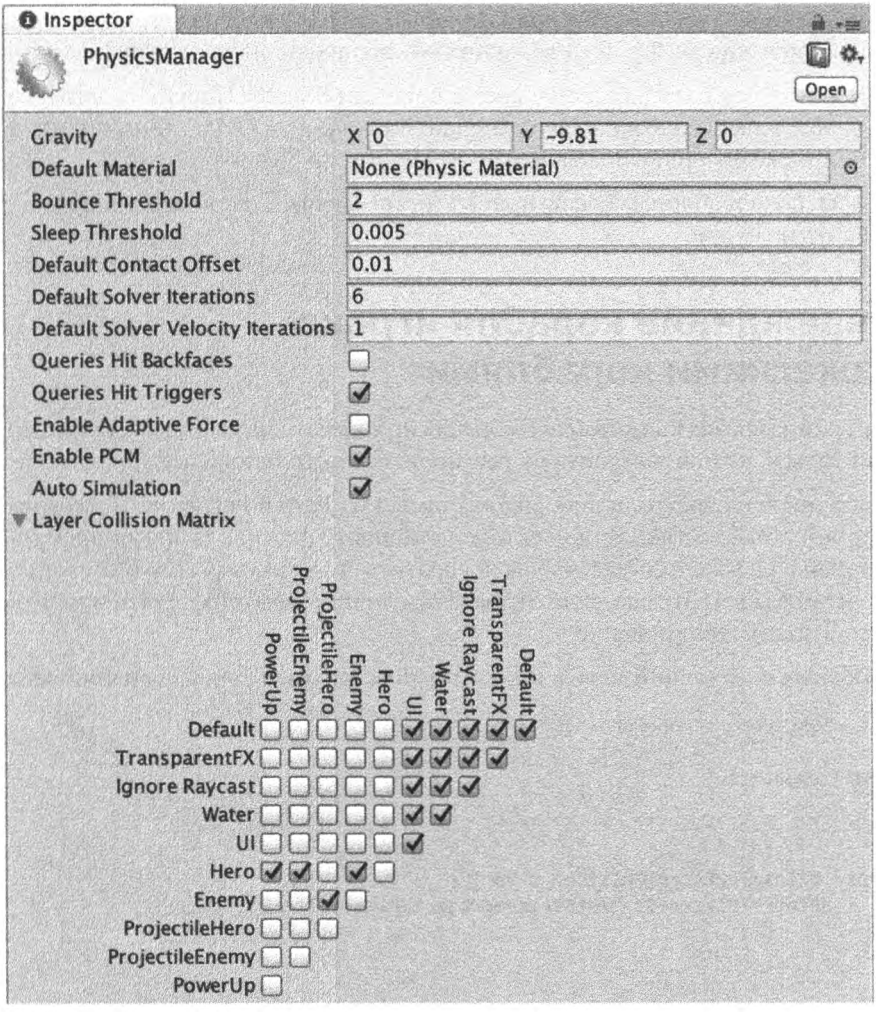


Рис. 30.8. Диспетчер PhysicsManager с настройками для этого прототипа

Распределение игровых объектов по слоям

Теперь, когда слои определены, необходимо распределить игровые объекты по соответствующим слоям:

1. Выделите _Hero в иерархии и в инспекторе выберите пункт Hero в раскрывающемся списке Layer. Когда Unity спросит, хотите ли вы поместить в этот слой также дочерние объекты, вложенные в _Hero, выберите ответ Yes (Да).

2. Там же в инспекторе с настройками объекта _Hero выберите тег Hero в раскрывающемся списке Tag. Теги дочерних объектов изменять не нужно.
3. Выделите все пять шаблонов Enemy в панели Project (Проект) и поместите их в слой Enemy. Ответьте утвердительно на вопрос о желании поместить в этот слой также дочерние объекты.
4. Также всем шаблонам Enemy присвойте тег Enemy. Теги дочерних объектов изменять не нужно.

Повреждение корабля игрока вражескими кораблями

Теперь, когда вражеские корабли и корабль игрока находятся в слоях, которые могут сталкиваться, нужно заставить их реагировать на столкновения.

1. Щелкните на треугольнике рядом с именем _Hero в иерархии и выберите дочерний объект Shield. В инспекторе установите флажок Is Trigger в настройках коллайдера Sphere Collider, чтобы превратить его в триггер. Нам не нужно, чтобы другие объекты отскакивали от Shield; мы лишь хотим определять момент, когда происходит столкновение.
2. Добавьте следующий метод, выделенный жирным, в конец сценария Hero:

```
public class Hero : MonoBehaviour {
    ...
    void Update() {
        ...
    }

    void OnTriggerEnter(Collider other) {
        print("Triggered: "+other.gameObject.name);
    }
}
```

3. Запустите сцену и попробуйте столкнуться с вражескими кораблями. Вы сможете увидеть, что в ответ на столкновения с дочерними объектами, составляющими вражеский корабль (например, Cockpit и Wing), генерируются отдельные события, но столкновение с самим вражеским кораблем не обнаруживается. Нам нужно каким-то образом получить ссылку на игровой объект Enemy_0, родительский для Cockpit и Wing, и хотелось бы иметь универсальный способ, позволяющий находить самый верхний, или *корневой*, родительский объект даже для глубоко вложенных дочерних объектов.

   К счастью, эта потребность возникает настолько часто, что для ее удовлетворения в компонент Transform было включено специальное свойство. Обращение к `transform.root` любого игрового объекта возвращает ссылку на компонент Transform корневого игрового объекта, с помощью которой легко получить ссылку на сам игровой объект.

4. В сценарии Hero замените код в методе `OnTriggerEnter()` строками, выделенными жирным в следующем листинге:

```
public class Hero : MonoBehaviour {
    ...
    void OnTriggerEnter(Collider other) {
        Transform rootT = other.gameObject.transform.root;
        GameObject go = rootT.gameObject;
        print("Triggered: "+go.name);
    }
}
```

Если теперь запустить сцену и столкнуться с вражеским кораблем, метод `OnTriggerEnter()` будет сообщать, что произошло столкновение с Enemy_0(Clone), экземпляром Enemy_0.

ИТЕРАТИВНАЯ РАЗРАБОТКА КОДА. Разрабатывая свои прототипы, вы часто будете выводить подобные сообщения в консоль для проверки работы написанного вами кода. Я считаю, что такое попутное тестирование намного лучше, чем многочасовая работа над кодом, в конце которой обнаруживается ошибка. Поэтапное тестирование значительно упрощает отладку, потому что вы знаете, какие изменения внесены после последнего тестирования, и вам будет проще найти место, где вы внесли ошибку.

Другой важный аспект — использование отладчика. В процессе работы над этой книгой каждый раз, обнаружив, что код работает немного не так, как задумывалось, я использовал отладчик, чтобы понять суть происходящего. Если вы не помните, как пользоваться отладчиком в MonoDevelop, то советую перечитать главу 25 «Отладка».

Использование отладчика часто оказывается эффективнее при решении проблем, чем многочасовое изучение страниц кода. Попробуйте добавить точку останова в только что измененный метод `OnTriggerEnter()` и посмотрите, как вызывается код и изменяются переменные.

Итеративная разработка кода обладает теми же преимуществами, что и итеративное проектирование, и является ключом к методологии гибкой разработки, обсуждавшейся в главе 14 «Гибкое мышление».

5. Измените метод `OnTriggerEnter()` класса Hero, чтобы каждое столкновение с вражеским кораблем уменьшало уровень защиты корабля игрока на 1 и уничтожало вражеский корабль. Также важно гарантировать, чтобы для одного и того же родительского игрового объекта не обнаруживалось двойных столкновений (что может случаться с быстро движущимися объектами, когда два дочерних коллайдера одного объекта в одном кадре сталкиваются с триггером Shield).

```
public class Hero : MonoBehaviour {
    ...
    public float shieldLevel = 1;

    // Эта переменная хранит ссылку на последний столкнувшийся игровой объект
    private GameObject      lastTriggerGo = null;                    // a
```

```
    ...

    void OnTriggerEnter(Collider other) {
        Transform rootT = other.gameObject.transform.root;
        GameObject go = rootT.gameObject;
        //print("Triggered: "+go.name);                                    // b

        // Гарантировать невозможность повторного столкновения с тем же объектом
        if (go == lastTriggerGo) {                                        // c
            return;
        }
        lastTriggerGo = go; // d

        if (go.tag == "Enemy") { // Если защитное поле столкнулось с вражеским
                                 // кораблем
            shieldLevel--;       // Уменьшить уровень защиты на 1
            Destroy(go);         // ... и уничтожить врага                  // e
        } else {
            print("Triggered by non-Enemy: "+go.name);                    // f
        }
    }
}
```

a. Это скрытое поле хранит ссылку на последний игровой объект, столкнувшийся с коллайдером в объекте _Hero. Первоначально хранит ссылку `null`.

b. Закомментируйте эту строку.

c. Если `lastTriggerGo` ссылается на тот же объект, что и go (текущий игровой объект, столкнувшийся с кораблем игрока), это столкновение игнорируется как повторное, и функция просто возвращает управление (то есть завершается). Такое может случиться, если два дочерних игровых объекта одного и того же родителя Enemy столкнутся с коллайдером _Hero в одном кадре.

d. Ссылка из go копируется в `lastTriggerGo` для обновления перед следующим вызовом `OnTriggerEnter()`.

e. go — игровой объект вражеского корабля — уничтожается при столкновении с защитным полем. Так как go хранит ссылку на фактический игровой объект Enemy, полученную обращением к `transform.root`, эта операция удалит весь вражеский корабль (вместе с его дочерними объектами), а не только тот дочерний объект, что столкнулся с полем.

f. Если _Hero столкнулся с каким-то другим объектом, не имеющим тега `"Enemy"`, сообщение об этом выводится в консоль, чтобы можно было узнать об этом.

6. Запустите сцену и попробуйте столкнуться с несколькими вражескими кораблями. В какой-то момент, после достаточно большого числа столкновений, можно заметить странное поведение защитного поля. Оно полностью истощается и вновь начинает набирать мощность. Как думаете, чем это вызвано? Попробуйте

выбрать _Hero в иерархии, пока сцена запущена, чтобы увидеть, что происходит с содержимым поля shieldLevel.

Сейчас нет нижнего порога для уровня защитного поля в shieldLevel, поэтому его значение перешагивает через 0 и продолжает уменьшаться в сторону отрицательных значений. Сценарий Shield транслирует этот уровень в отрицательное смещение X для Mat_Shield, а поскольку текстура материала выбирается циклически, создается ощущение, что уровень поля возвращается к полной мощности.

Чтобы исправить этот недостаток, преобразуем поле shieldLevel в свойство, защищающее и ограничивающее новое скрытое поле с именем _shieldLevel. Свойство shieldLevel будет наблюдать за значением в поле _shieldLevel, гарантировать невозможность его увеличения выше 4 и обеспечивать уничтожение корабля игрока, когда значение _shieldLevel опустится ниже 0. Защищаемое поле, такое как _shieldLevel, нужно объявить скрытым (private), потому что оно не должно быть доступно другим классам; однако в Unity скрытые поля обычно не отображаются в инспекторе. Для решения этой проблемы добавим атрибут [SerializeField] перед объявлением скрытого поля _shieldLevel, чтобы заставить Unity показать это поле в инспекторе. Свойства никогда не отображаются в инспекторе, даже общедоступные.

7. В классе Hero измените имя общедоступной переменной shieldLevel на _shieldLevel, замените спецификатор области видимости public на private и добавьте строку с атрибутом [SerializeField]:

```
public class Hero : MonoBehaviour {
    ...
    [Header("Set Dynamically")]
    [SerializeField]
    private float           _shieldLevel = 1; // Обратите внимание на символ
                                              // подчеркивания
    // Эта переменная хранит ссылку на последний столкнувшийся игровой объект
    ...
}
```

8. Добавьте свойство shieldLevel в конец класса Hero.

```
public class Hero : MonoBehaviour {
    ...
    void OnTriggerEnter(Collider other) {
        ...
    }

    public float shieldLevel {
        get {
            return( _shieldLevel );                                    // a
        }
        set {
            _shieldLevel = Mathf.Min( value, 4 );                      // b
            // Если уровень поля упал до нуля или ниже
```

```
            if (value < 0) {                                          // c
                Destroy(this.gameObject);
            }
        }
    }
}
```

a. Метод get чтения свойства просто возвращает значение _shieldLevel.

b. Mathf.Min() гарантирует, что _shieldLevel никогда не получит значение выше 4.

c. Если значение value, переданное в метод записи set, меньше 0, объект _Hero уничтожается.

Строка shieldLevel--; в методе OnTriggerEnter() вызывает оба метода, get и set, свойства shieldLevel. Сначала вызывается метод get, чтобы получить текущее значение _shieldLevel, затем из него вычитается 1 и вызывается метод set, чтобы записать обратно изменившееся значение.

Перезапуск игры

Поиграв в игру, можно заметить, что игра теряет смысл после уничтожения корабля игрока _Hero. Теперь мы изменим оба класса, Hero и Main, добавив вызов метода, который после разрушения _Hero будет ждать 2 секунды и перезапускать игру.

1. Добавьте поле gameRestartDelay в начало определения класса Hero:

```
public class Hero : MonoBehaviour {
    static public Hero      S; // Одиночка

    [Header("Set in Inspector")]
    ...
     public float           pitchMult = 30;
     public float           gameRestartDelay = 2f;

    [Header("Set Dynamically")]
    ...
}
```

2. Добавьте следующие строки в определение свойства shieldLevel в классе Hero:

```
public class Hero : MonoBehaviour {
    ...
     public float shieldLevel {
         get { ... }
         set {
            ...
             if (value < 0) {
                 Destroy(this.gameObject);
                 // Сообщить объекту Main.S о необходимости перезапустить игру
                 Main.S.DelayedRestart( gameRestartDelay );              // a
             }
```

```
        }
    }
}
```

a. Сразу после ввода вызова метода `DelayedRestart()` в MonoDevelop он подчеркивается красной волнистой линией, потому что в данный момент метод `DelayedRestart()` отсутствует в классе `Main`.

3. Добавьте следующие методы в класс `Main`, чтобы реализовать перезапуск.

```
public class Main : MonoBehaviour {

    ...

    public void SpawnEnemy() { ... }

    public void DelayedRestart( float delay ) {
        // Вызвать метод Restart() через delay секунд
        Invoke( "Restart", delay );
    }

    public void Restart() {
        // Перезагрузить _Scene_0, чтобы перезапустить игру
        SceneManager.LoadScene( "_Scene_0" );
    }
}
```

4. Щелкните на кнопке Play (Играть), чтобы протестировать игру. Теперь, после уничтожения корабля игрока, игра будет ждать пару секунд и затем перезапускаться перезагрузкой сцены.

→ Если после перезагрузки освещение в сцене выглядит не как обычно (например, корабль игрока или вражеские корабли выглядят темнее), значит, вы столкнулись с известной ошибкой в системе освещения в Unity (о которой я уже упоминал в главе 28). Надеюсь, разработчики Unity исправят эту проблему, когда вы будете читать эти строки, но если вы столкнетесь с ней, используйте следующее обходное решение, пригодное для данного проекта:

1. Выберите в меню пункт Window > Lighting > Settings (Окно > Освещение > Настройки).
2. Щелкните на кнопке Scene (Сцена) в верхней части в панели Lighting (Освещение).
3. Снимите флажок Auto Generate (Генерировать автоматически) в нижней части панели Lighting (Освещение) рядом с кнопкой Generate Lighting (Сгенерировать освещение). После этого Unity перестанет постоянно пересчитывать освещенность.
4. Щелкните на кнопке Generate Lighting (Сгенерировать освещение) в нижней части панели Lighting (Освещение), чтобы вручную пересчитать освещенность.
5. Дождитесь, когда расчеты закончатся, а затем щелкните на кнопке Play (Играть), чтобы проверить результат. Теперь после перезагрузки сцены освещенность не должна изменяться. В этой главе вам не придется вновь пересчитывать освещенность, но, если вы сами измените параметры освещения в игре, обязательно повторите процедуру ручного пересчета.

Стрельба (наконец)

Теперь, когда вражеские корабли могут наносить повреждения кораблю игрока, пришло время дать игроку возможность защищаться. В этой главе мы реализуем только один тип снарядов. Но в следующей придадим оружию дополнительные интересные особенности.

ProjectileHero, снаряды для Hero

Выполните следующие шаги, чтобы создать снаряд для стрельбы по вражеским кораблям:

1. Создайте в иерархии куб с именем ProjectileHero и следующими настройками компонента Transform:

   ```
   ProjectileHero (Cube) P:[ 10, 0, 0 ] R:[ 0, 0, 0 ] S:[ 0.25, 1, 0.5 ]
   ```

2. В раскрывающихся списках Tag и Layer для ProjectileHero выберите значение ProjectileHero.
3. Создайте новый материал с именем Mat_Projectile, поместите его в папку _Materials в панели Project (Проект), выберите для него шейдер ProtoTools > UnlitAlpha и присвойте игровому объекту ProjectileHero.
4. Добавьте в игровой объект ProjectileHero компонент Rigidbody со следующими настройками:
   - Снимите флажок Use Gravity.
   - Снимите флажок isKinematic.
   - В поле Collision Detection выберите значение Continuous.
   - В разделе Constraints установите флажки Freeze Position Z и Freeze Rotation X, Y и Z.
5. В компоненте Box Collider игрового объекта ProjectileHero установите значение Size.Z равным 10. Это гарантирует поражение снарядами объектов, которые могут находиться вне плоскости XY (то есть Z=0).
6. Создайте новый сценарий на C# с именем Projectile и подключите к ProjectileHero. Мы займемся им позже.

   По завершении настройки должны выглядеть так, как показано на рис. 30.9 (впрочем, сейчас вы не увидите компонент BoundsCheck (Script), потому что мы добавим его в шаге 8).

7. Сохраните сцену.

8. Подключите компонент сценария BoundsCheck к ProjectileHero. Снимите флажок keepOnScreen и введите –1 в поле radius. Величина radius в BoundsCheck не влияет на столкновения с другими игровыми объектами, она лишь определяет расстояние, на которое ProjectileHero должен выйти за край экрана, чтобы считаться покинувшим экран.
9. Преобразуйте ProjectileHero в шаблон, перетащив его из иерархии в папку _Prefabs в панели Project (Проект). Затем удалите экземпляр, оставшийся в иерархии.
10. Сохраните сцену — да, сохраните ее снова. Как я уже говорил, сохранять сцену нужно как можно чаще.

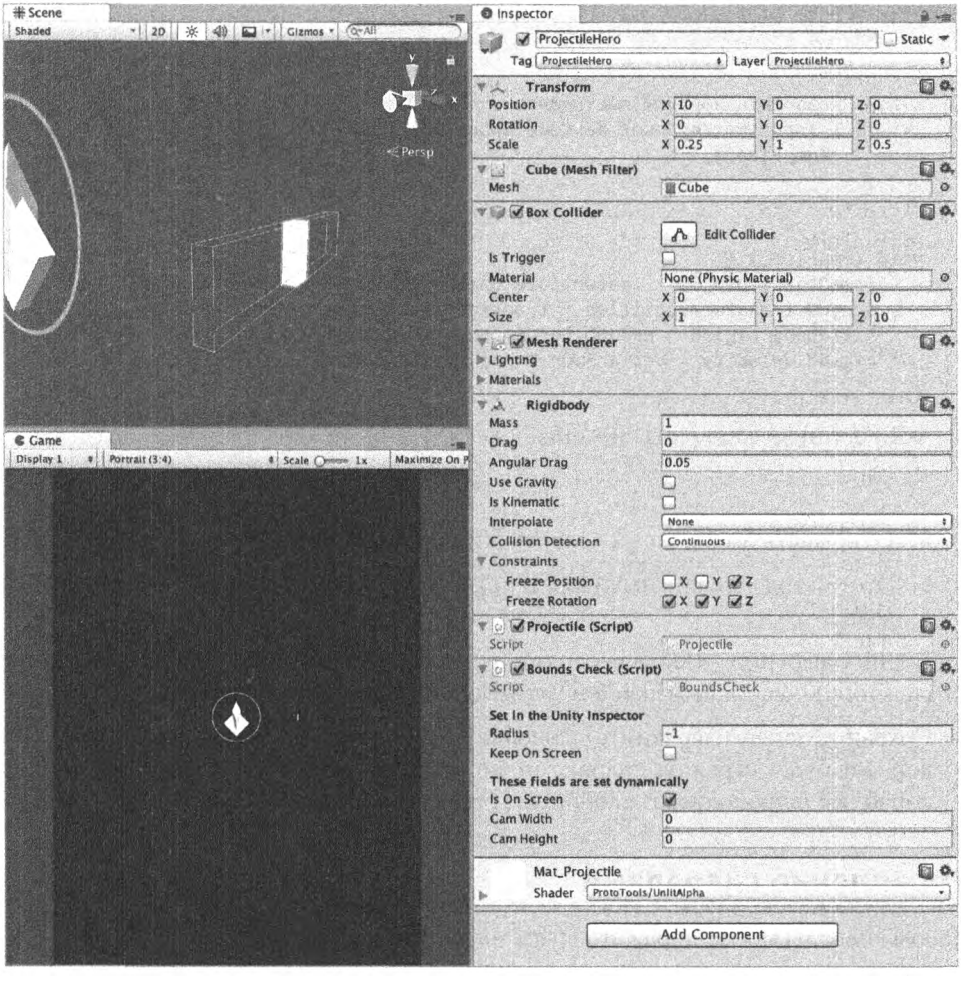


Рис. 30.9. Настройки ProjectileHero с большим размером Size.Z коллайдера Box Collider

Реализация стрельбы

Теперь дадим игроку возможность стрелять снарядами.

1. Откройте сценарий Hero и добавьте следующие строки, выделенные жирным:

```
public class Hero : MonoBehaviour {
    ...
    public float            gameRestartDelay = 2f;
    public GameObject       projectilePrefab;
    public float            projectileSpeed = 40;

    ...

    void Update () {
        ...
        transform.rotation = Quaternion.Euler(yAxis*pitchMult, xAxis*rollMult,0);

        // Позволить кораблю выстрелить
        if ( Input.GetKeyDown( KeyCode.Space ) ) {                          // a
            TempFire();
        }
    }

    void TempFire() {                                                       // b
        GameObject projGO = Instantiate<GameObject>( projectilePrefab );
        projGO.transform.position = transform.position;
        Rigidbody rigidB = projGO.GetComponent<Rigidbody>();
        rigidB.velocity = Vector3.up * projectileSpeed;
    }

    void OnTriggerEnter(Collider other) { ... }
    ...
}
```

   a. Открывать огонь, когда игрок нажимает клавишу пробела.

   b. Этот метод получил имя `TempFire()`, потому что мы изменим его в следующей главе.

2. В Unity выберите _Hero в иерархии и в поле projectilePrefab сценария Hero (в инспекторе) перетащите шаблон ProjectileHero из панели Project (Проект).

3. Сохраните сцену и щелкните на кнопке Play (Играть). Если теперь нажать пробел, корабль будет стрелять снарядами, но пока они не наносят ущерба вражеским кораблям и продолжают свой полет бесконечно, даже когда покинут экран.

Управление снарядами

Чтобы снаряды могли поражать врага, выполните следующие шаги:

1. Откройте сценарий Projectile и добавьте следующие строки, выделенные жирным. Сейчас мы реализуем только уничтожение снаряда после выхода за границы экрана. Остальное добавим в следующей главе.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Projectile : MonoBehaviour {

    private BoundsCheck     bndCheck;

    void Awake () {
        bndCheck = GetComponent<BoundsCheck>();
    }

    void Update () {
        if (bndCheck.offUp) {                                          // a
            Destroy( gameObject );
        }
    }
}
```

a. Если снаряд вышел за верхнюю границу экрана, уничтожить его.

2. И снова не забудьте сохранить.

Повреждение вражеских кораблей снарядами

Нам нужно также добавить возможность повреждения вражеских кораблей снарядами.

1. Откройте сценарий Enemy и добавьте в конец следующий метод:

```
public class Enemy : MonoBehaviour {
    ...
    public virtual void Move() { ... }

    void OnCollisionEnter( Collision coll ) {
        GameObject otherGO = coll.gameObject;                          // a
        if ( otherGO.tag == "ProjectileHero" ) {                       // b
            Destroy( otherGO );    // Уничтожить снаряд
            Destroy( gameObject ); // Уничтожить игровой объект Enemy
        } else {
            print( "Enemy hit by non-ProjectileHero: " + otherGO.name );   // c
        }
    }
}
```

2. Получите игровой объект, которому принадлежит коллайдер, столкнувшийся с вражеским кораблем.

3. Если otherGO имеет тег ProjectileHero, тогда уничтожьте его и данный экземпляр Enemy.

4. Если otherGO не имеет тега ProjectileHero, выведите его имя в консоль для отладки. Если вы захотите проверить работу этой ветки условной инструкции,

временно удалите тег ProjectileHero из шаблона ProjectileHero и выстрелите по вражескому кораблю[1].

Если теперь щелкнуть на кнопке Play (Играть), экземпляры Enemy_0 начнут движение по экрану сверху вниз, и вы сможете расстреливать их снарядами. На этом мы заканчиваем данную главу — теперь у вас есть замечательный простой прототип, но в следующей главе я продолжу работу над ним и покажу, как добавить дополнительных врагов, три вида бонусов и еще два вида пушек. Также я продемонстрирую несколько интересных приемов программирования.

Итоги

Большинство глав заканчивается разделом «Следующие шаги», где я делюсь идеями дальнейшего расширения проектов, которые вы могли бы реализовать самостоятельно. Но эта традиция не распространяется на прототип *Space SHMUP*, потому что вы продолжите работу над ним в следующей главе и познакомитесь с новыми приемами программирования. Сделайте перерыв, прежде чем двигаться дальше, и поздравьте себя с созданием очередного прототипа.

[1] Вам не удастся проверить эту ветку, столкнув свой корабль с вражеским, потому что коллайдер объекта Shield, дочернего по отношению к _Hero, является триггером, а столкновение с триггерами не приводит к вызову `OnCollisionEnter()`.

31 Прототип 3.5: SPACE SHMUP PLUS

Большинство глав заканчивается разделом «Следующие шаги», где я делюсь идеями, которые вы могли бы воплотить в игре. Эта глава как раз описывает шаги по воплощению подобных идей в игру *Space SHMUP*, которую вы создали в предыдущей главе.

В этой главе вы добавите в игру *Space SHMUP* бонусы, другие вражеские корабли и дополнительные типы оружия. Также вы узнаете больше о наследовании классов, перечислениях, функциях-делегатах и некоторых других важных аспектах. А в награду вы получите намного более увлекательную игру!

Начало: прототип 3.5

В предыдущей главе вы создали простенькую версию космического шутера. В этой главе вы сделаете его более интересным и разнообразным. Если у вас возникли какие-то проблемы с игрой, созданной в предыдущей главе, вы можете загрузить ее с веб-сайта книги.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Вместо настройки проекта согласно стандартной процедуре в этой главе у вас на выбор есть два варианта:

1. Скопировать папку проекта из предыдущей главы.
2. Загрузить законченную версию для предыдущей главы с веб-сайта книги. Для этого найдите раздел Chapter 31 на странице http://book.prototools.net.

Создав папку проекта любым из предложенных способов, откройте _Scene_0 в Unity.

Добавление других вражеских кораблей

Начнем с увеличения разнообразия вражеских кораблей, с которыми может столкнуться игрок. Позднее мы дадим игроку возможность обороняться от этих еще более опасных врагов.

1. Создайте новые сценарии на C# с именами Enemy_1, Enemy_2, Enemy_3 и Enemy_4.
2. Поместите сценарии в папку __Scripts в панели Project (Проект).
3. Свяжите каждый из этих сценариев с соответствующим шаблоном Enemy_# в папке _Prefabs в панели Project (Проект).

А теперь займемся этими сценариями по очереди.

Enemy_1

Enemy_1 будет перемещаться сверху вниз по синусоиде. Он унаследует класс Enemy, то есть унаследует все поля, функции и свойства класса Enemy (общедоступные (public) и защищенные (protected); скрытые (private) элементы не наследуются). За дополнительной информацией о классах и наследовании классов (включая переопределение методов) обращайтесь к главе 26 «Классы».

1. Откройте сценарий Enemy_1 в MonoDevelop и введите следующий код, выделенный жирным:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Enemy_1 расширяет класс Enemy
public class Enemy_1 : Enemy {                                          // a

    [Header("Set in Inspector: Enemy_1")]
    // число секунд полного цикла синусоиды
    public float    waveFrequency = 2;
    // ширина синусоиды в метрах
    public float    waveWidth = 4;
    public float    waveRotY = 45;

    private float    x0; // Начальное значение координаты X
    private float    birthTime;

    // Метод Start хорошо подходит для наших целей,
    //  потому что не используется суперклассом Enemy
    void Start() {
        // Установить начальную координату X объекта Enemy_1
        x0 = pos.x;                                                     // b

        birthTime = Time.time;
    }

    // Переопределить функцию Move суперкласса Enemy
    public override void Move() {                                       // c
        // Так как pos - это свойство, нельзя напрямую изменить pos.x
        // поэтому получим pos в виде вектора Vector3, доступного для изменения
        Vector3 tempPos = pos;
        // значение theta изменяется с течением времени
        float age = Time.time - birthTime;
        float theta = Mathf.PI * 2 * age / waveFrequency;
```

```
        float sin = Mathf.Sin(theta);
        tempPos.x = x0 + waveWidth * sin;
        pos = tempPos;

        // повернуть немного относительно оси Y
        Vector3 rot = new Vector3(0, sin*waveRotY, 0);
        this.transform.rotation = Quaternion.Euler(rot);

        // base.Move() обрабатывает движение вниз, вдоль оси Y
        base.Move();                                                    // d

        // print( bndCheck.isOnScreen );
    }
}
```

a. Расширяя класс `Enemy`, `Enemy_1` наследует общедоступные поля `speed`, `fireRate`, `health` и `score`, а также общедоступное свойство `pos` и общедоступный метод `Move()`. Но он не наследует скрытое поле `bndCheck`, о чем я подробнее расскажу ниже.

b. Установка начальной координаты X в поле `x0` этого объекта `Enemy_1` благополучно выполняется методом `Start()`, потому что к моменту его вызова позиция уже будет установлена. С другой стороны, было бы неправильным поместить эту строку в метод `Awake()`, потому что `Awake()` вызывается в момент создания экземпляра игрового объекта (то есть до того, как позиция будет установлена методом `Main:SpawnEnemy()` (в *Main.cs*)).

   Еще одна причина, почему следует избегать метода `Awake()` в `Enemy_1`, состоит в том, что он переопределил бы метод `Awake()` класса `Enemy`. `Awake()`, `Start()`, `Update()` и другие методы класса `MonoBehaviour` управляются особо, поэтому — в отличие от стандартных методов классов в языке C# — их не нужно снабжать спецификаторами `virtual` и `override` для переопределения в подклассах (см. главу 26 «Классы»).

c. Для правильного переопределения в подклассах обычные методы, такие как `Move()`, который мы добавили в класс `Enemy`, в суперклассе должны объявляться со спецификатором `virtual`, а в подклассах — со спецификатором `override`. Поскольку в суперклассе `Enemy` метод `Move()` отмечен как виртуальный (`virtual`), мы можем переопределить его здесь и заменить другим методом (также с именем `Move()`).

d. `base.Move()` вызывает метод `Move()` суперкласса `Enemy`. В данном случае метод `Move()` в подклассе `Enemy_1` отвечает за горизонтальное перемещение по синусоиде, а метод `Move()` в суперклассе `Enemy` отвечает за перемещение по вертикали.

2. Вернитесь в Unity, выберите _MainCamera в иерархии и в компоненте Main (Script) измените элемент Element 0 в массиве prefabEnemies, заменив Enemy_0 на Enemy_1 (то есть перетащив в него шаблон Enemy_1 из папки _Prefabs). Это позволит вам протестировать поведение класса `Enemy_1` вместо `Enemy`.

3. Щелкните на кнопке Play (Играть). Теперь вместо Enemy_0 появится корабль Enemy_1 и начнет движение вниз по синусоиде. Обратите внимание (в панели Scene (Сцена)), что экземпляры Enemy_1 не исчезают после выхода за нижний край экрана. Это объясняется отсутствием компонента BoundsCheck в шаблоне Enemy_1.
4. Подключите BoundsCheck к шаблону Enemy_1 и оставьте те же настройки, что были установлены в шаблоне Enemy_0. Для этого выполните следующие шаги, описывающие другой способ подключения сценария к игровому объекту:
   a. Выберите Enemy_0 в папке _Prefabs, находящийся в панели Project (Проект).
   b. В инспекторе щелкните на кнопке с шестеренкой в правом верхнем углу, в разделе с настройками компонента BoundsCheck (Script), и в открывшемся меню выберите пункт Copy Component (Копировать компонент).
   c. Выберите шаблон Enemy_1 в папке _Prefabs, в панели Project (Проект).
   d. В инспекторе щелкните на кнопке с шестеренкой в правом верхнем углу, в разделе с настройками компонента Transform, и в открывшемся меню выберите пункт Paste Component As New (Вставить компонент как новый). В результате к шаблону Enemy_1 будет подключен новый компонент BoundsCheck (Script), имеющий те же настройки, что и компонент BoundsCheck, скопированный из шаблона Enemy_0.

Изменение области видимости поля bndCheck

Малозаметной, но важной особенностью объявления поля bndCheck в классе Enemy является определение области видимости private.

```
private BoundsCheck bndCheck;
```

Это означает, что данное поле доступно только в классе Enemy и недоступно другим классам, включая Enemy_1, даже при том, что Enemy_1 является подклассом Enemy. То есть методы Awake() и Move() в классе Enemy могут обращаться к полю bndCheck, а переопределенный метод Move() в Enemy_1 ничего не знает о нем. Чтобы убедиться в этом:

1. Откройте сценарий Enemy_1 и раскомментируйте строку в конце метода Move(), выделенную жирным:

```
public override void Move() {
    ...

    base.Move();

    print( bndCheck.isOnScreen );
}
```

Так как bndCheck является скрытой переменной класса Enemy, в Enemy_1 она будет подчеркнута красной волнистой линией. Чтобы исправить проблему, нужно сделать поле bndCheck защищенным. Защищенные (private) переменные, так же

как скрытые (`private`), недоступны другим классам, но, в отличие от скрытых переменных, они доступны и наследуются дочерними подклассами:

Область видимости переменной	Доступна в подклассах	Доступна в любых классах
`private`	Нет	Нет
`protected`	Да	Нет
`public`	Да	Да

2. Откройте сценарий Enemy, замените спецификатор `private` в объявлении поля `bndCheck` на `protected` и сохраните.

```
protected BoundsCheck bndCheck;
```

Если теперь открыть сценарий Enemy_1, ссылка `bndCheck.isOnScreen` не будет подчеркнута красной волнистой линией, и код благополучно скомпилируется.

3. Вернитесь в сценарий `Enemy_1` и вновь закомментируйте строку с вызовом `print()`.

```
// print( bndCheck.isOnScreen ); // Закомментируйте эту строку.
```

4. Щелкните на кнопке Play (Играть), и вы заметите, что корабли Enemy_1 исчезают после пересечения нижней границы экрана.

СФЕРИЧЕСКИЙ КОЛЛАЙДЕР МАСШТАБИРУЕТСЯ ОДИНАКОВО ПО ВСЕМ ОСЯМ. Возможно, вы заметили, что столкновение с кораблем Enemy_1 возникает чуть раньше, чем снаряд (или корабль _Hero) коснется его крыла. Если выбрать шаблон Enemy_1 в панели Project (Проект) и перетащить его в сцену, создав экземпляр, вы увидите зеленую сферу коллайдера, окружающую Enemy_1, которая сохраняет шарообразную форму, не масштабируясь под форму уплощенного эллипса. Это не самая большая проблема, но вы должны знать о ней. Радиус сферического коллайдера выбирается по наибольшему масштабу в компоненте Transform. (В данном случае Scale.X крыла имеет значение 6, поэтому радиус сферического коллайдера выбирается равным этой величине.)

Желающие могут попробовать другие типы коллайдеров и убедиться, что некоторые из них точнее приспосабливаются под форму крыла. Коробчатый коллайдер (Box Collider) масштабируется неодинаково в разных осях. Если игровой объект сильно вытянут вдоль одной из осей, можно воспользоваться коллайдером в форме капсулы (Capsule Collider). Коллайдер в форме трехмерной сетки — Mesh Collider — точнее других принимает форму игрового объекта, но он и самый медленный. Это не является большой проблемой для современных высокопроизводительных персональных компьютеров, но для мобильных платформ, таких как iOS и Android, коллайдер Mesh Colliders работает слишком медленно.

Если вы решите выбрать для Enemy_1 коробчатый коллайдер Mesh Collider, тогда при вращении относительно оси Y края его крыльев будут выходить из плоскости XY (то есть z=0). Именно поэтому для коллайдера Box Collider в шаблоне ProjectileHero выбрано значение Size.z, равное 10 (чтобы он мог поразить крыло Enemy_1, даже если оно находится вне плоскости XY).

Подготовка других вражеских кораблей

Остальные вражеские корабли используют прием линейной интерполяции — важное понятие в разработке, которое описывается в приложении Б. Вы уже видели использование простейшей линейной интерполяции в сценарии FollowCam, в прототипе *Mission Demolition*, но здесь мы найдем ей чуть более интересное применение. Оторвитесь ненадолго и прочитайте раздел «Интерполяция» в приложении Б «Полезные идеи», прежде чем продолжить реализацию оставшихся вражеских кораблей.

Enemy_2

Enemy_2 движется, интерполируя скорость по синусоиде. Он вылетает из-за края экрана, замедляется, меняет направление полета на обратное, набирает скорость, замедляется, опять меняет направление полета и вылетает за край экрана с начальной скоростью. Для интерполяции используются всего две точки, но значение *u* сильно искажается синусоидой. Функция сглаживания (easing function) вычисления значения *u* для Enemy_2 выглядит так:

$$u = u + 0{,}6 \times \mathrm{Sin}(2\pi \times u)$$

Это одна из функций сглаживания, о которых рассказывается в разделе «Интерполяция» приложения Б.

1. Подключите сценарий BoundsCheck к шаблону Enemy_2 в папке _Prefabs в панели Project (Проект). Компонент BoundsCheck будет использоваться в Enemy_2 очень часто.
2. В инспекторе с настройками шаблона Enemy_2, в разделе BoundsCheck, введите в поле radius число 3 и снимите флажок keepOnScreen.
3. Откройте сценарий Enemy_2 и введите следующий код. Закончив, поэкспериментируйте со значением `sinEccentricity`, чтобы увидеть, как оно влияет на характер движения.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Enemy_2 : Enemy {                                          // a
    [Header("Set in Inspector: Enemy_2")]
    // Определяют, насколько ярко будет выражен синусоидальный характер движения
    public float            sinEccentricity = 0.6f;
    public float            lifeTime = 10;

    [Header("Set Dynamically: Enemy_2")]
    // Enemy_2 использует линейную интерполяцию между двумя точками,
    //   изменяя результат по синусоиде
    public Vector3          p0;
    public Vector3          p1;
    public float            birthTime;
```

```
    void Start () {
        // Выбрать случайную точку на левой границе экрана
        p0 = Vector3.zero;                                                    // b
        p0.x = -bndCheck.camWidth - bndCheck.radius;
        p0.y = Random.Range( -bndCheck.camHeight, bndCheck.camHeight );

        // Выбрать случайную точку на правой границе экрана
        p1 = Vector3.zero;
        p1.x = bndCheck.camWidth + bndCheck.radius;
        p1.y = Random.Range( -bndCheck.camHeight, bndCheck.camHeight );

        // Случайно поменять начальную и конечную точку местами
        if (Random.value > 0.5f) {
            // Изменение знака .x каждой точки
            //   переносит ее на другой край экрана
            p0.x *= -1;
            p1.x *= -1;
        }

        // Записать в birthTime текущее время
        birthTime = Time.time;                                                // c
    }

    public override void Move() {
        // Кривые Безье вычисляются на основе значения u между 0 и 1
        float u = (Time.time - birthTime) / lifeTime;

        // Если u>1, значит, корабль существует дольше, чем lifeTime
        if (u > 1) {
            // Этот экземпляр Enemy_2 завершил свой жизненный цикл
            Destroy( this.gameObject );                                       // d
            return;
        }

        // Скорректировать u добавлением значения кривой, изменяющейся по синусоиде
        u = u + sinEccentricity*(Mathf.Sin(u*Mathf.PI*2));

        // Интерполировать местоположение между двумя точками
        pos = (1-u)*p0 + u*p1;
    }
}
```

a. `Enemy_2` тоже расширяет суперкласс `Enemy`.

b. В этом фрагменте выбирается случайная точка на левой границе экрана. Первоначально выбирается позиция X, сразу же за левой границей: `-bndCheck.camWidth` — это левая граница, а `- bndCheck.radius` гарантирует, что `Enemy_2` целиком окажется за левой границей (сместив его левее на величину радиуса этого корабля).

   Затем случайно выбирается координата Y между нижним (`-bndCheck.camHeight`) и верхним (`bndCheck.camHeight`) краями экрана.

c. `birthTime` используется для интерполяции в функции `Move()`.

d. Если корабль существует дольше, чем определено в `lifeTime`, тогда `u` получит значение больше 1, и этот экземпляр класса `Enemy_2` будет уничтожен.

4. Подставьте шаблон Enemy_2 в поле Element 0 в массиве prefabEnemies компонента Main (Script) главной камеры _MainCamera и щелкните на кнопке Play (Играть).

Как видите, функция сглаживания очень гладко меняет скорость и направление движения Enemy_2 — вперед, назад и снова вперед — между двумя выбранными точками на краях экрана.

Enemy_3

Траектория движения Enemy_3 определяется кривой Безье. Он быстро вылетает из-за верхнего края экрана вниз, замедляется, меняет направление на противоположное и вылетает за верхний край. В этом примере мы используем простую версию функции вычисления кривой Безье по трем точкам. В разделе «Рекурсивные функции» приложения В вы найдете рекурсивную версию функции, способной вычислять кривую Безье по произвольному количеству точек.

1. Подключите BoundsCheck к шаблону Enemy_3 в папке _Prefabs в панели Project (Проект).
2. В инспекторе с настройками шаблона Enemy_3, в разделе BoundsCheck, введите в поле radius число 2,5 и снимите флажок keepOnScreen.
3. Откройте сценарий Enemy_3 и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Enemy_3 : Enemy { // Enemy_3 расширяет класс Enemy
    // Траектория движения Enemy_3 вычисляется путем линейной
    //   интерполяции кривой Безье по более чем двум точкам.
    [Header("Set in Inspector: Enemy_3")]
    public float            lifeTime = 5;

    [Header("Set Dynamically: Enemy_3")]
    public Vector3[]        points;
    public float            birthTime;

    // И снова метод Start хорошо подходит для наших целей,
    //  потому что не используется суперклассом Enemy
    void Start () {
        points = new Vector3[3]; // Инициализировать массив точек

        // Начальная позиция уже определена в Main.SpawnEnemy()
        points[0] = pos;

        // Установить xMin и xMax так же, как это делает Main.SpawnEnemy()
        float xMin = -bndCheck.camWidth + bndCheck.radius;
        float xMax = bndCheck.camWidth - bndCheck.radius;
```

```
        Vector3 v;
        // Случайно выбрать среднюю точку ниже нижней границы экрана
        v = Vector3.zero;
        v.x = Random.Range( xMin, xMax );
        v.y = -bndCheck.camHeight * Random.Range( 2.75f, 2 );
        points[1] = v;

        // Случайно выбрать конечную точку выше верхней границы экрана
        v = Vector3.zero;
        v.y = pos.y;
        v.x = Random.Range( xMin, xMax );
        points[2] = v;

        // Записать в birthTime текущее время
        birthTime = Time.time;
    }

    public override void Move() {
        // Кривые Безье вычисляются на основе значения u между 0 и 1
        float u = (Time.time - birthTime) / lifeTime;

        if (u > 1) {
            // Этот экземпляр Enemy_2 завершил свой жизненный цикл
            Destroy( this.gameObject );
            return;
        }

        // Интерполировать кривую Безье по трем точкам
        Vector3 p01, p12;
        p01 = (1-u)*points[0] + u*points[1];
        p12 = (1-u)*points[1] + u*points[2];
        pos = (1-u)*p01 + u*p12;
    }
}
```

4. Теперь подставьте шаблон Enemy_3 в поле Element 0 в массиве prefabEnemies главной камеры _MainCamera.

5. Щелкните на кнопке Play (Играть), чтобы увидеть, как движутся эти новые вражеские корабли. Поиграв немного, вы наверняка заметите пару особенностей, свойственных кривым Безье:

   a. Даже при том, что средняя точка находится на нижней границе экрана или ниже, корабль Enemy_3 никогда не залетает так далеко. Это объясняется тем, что только начальная и конечная точки лежат на кривой Безье, а средняя точка просто влияет на форму кривой.

   b. Enemy_3 существенно замедляется в середине кривой. Это еще одна математическая особенность кривых Безье.

6. Чтобы сделать движение вдоль кривой Безье более плавным и ослабить эффект замедления в нижней ее части, добавьте в метод `Move()` в сценарии Enemy_3

следующий код, выделенный жирным. Он *сглаживает*[1] движение Enemy_3, увеличивая его скорость в середине траектории:

```
public override void Move() {
    ...
    // Интерполировать кривую Безье по трем точкам
    Vector3 p01, p12;
    u = u - 0.2f*Mathf.Sin(u * Mathf.PI * 2);
    p01 = (1-u)*points[0] + u*points[1];
    p12 = (1-u)*points[1] + u*points[2];
    pos = (1-u)*p01 + u*p12;
}
```

Оставим Enemy_4 на потом

Прежде чем приступать к реализации поведения Enemy_4, сначала нужно кое-что изменить в снарядах и их поведении. На данный момент игрок может уничтожать любые вражеские корабли одним попаданием. В следующем разделе вы узнаете, как изменить эту ситуацию и добавить возможность оснащать корабль игрока разными видами оружия.

Снова стрельба

Способ управления стрельбой снарядами Projectile, с которым вы познакомились в предыдущей главе, прекрасно подходит для простого прототипа, но теперь, чтобы поднять игру на новый уровень, мы должны добавить новые возможности. В этом разделе вы узнаете, как создать два дополнительных вида вооружения с возможностью их расширения в будущем. Для этого создадим класс `WeaponDefinition`, который позволит определять характеристики каждого вида оружия.

Перечисление WeaponType

Как рассказывалось в главе 29 «Прототип 2: Mission Demolition», *перечисление* — это способ объединить разные варианты в переменной нового типа. В этой игре игрок будет иметь возможность переключать и совершенствовать оружие, подбирая бонусы, сбрасываемые уничтоженными вражескими кораблями. Таким же способом игрок сможет наращивать мощность своего защитного поля. Чтобы в одной переменной можно было хранить все возможные виды бонусов, определим перечисление с именем `WeaponType`.

1. Щелкните правой кнопкой мыши на папке __Scripts в панели Project (Проект) и в контекстном меню выберите пункт Create > C# Script (Создать > Сценарий C#). В результате в папке __Scripts будет создан новый сценарий NewBehaviourScript.

[1] В приложении Б «Полезные идеи» приводится подробное обсуждение сглаживания кривых Безье.

2. Переименуйте NewBehaviourScript в Weapon.
3. Откройте сценарий Weapon в MonoDevelop и введите следующий код. Объявление `public enum WeaponType` должно находиться между `using UnityEngine;` и `public class Weapon : MonoBehaviour {`.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Это перечисление всех возможных типов оружия.
/// Также включает тип "shield", чтобы дать возможность совершенствовать защиту.
/// Аббревиатурой [HP] ниже отмечены элементы, не реализованные в этой книге.
/// </summary>
public enum WeaponType {
    none,       // По умолчанию / нет оружия
    blaster,    // Простой бластер
    spread,     // Веерная пушка, стреляющая несколькими снарядами
    phaser,     // [HP] Волновой фазер
    missile,    // [HP] Самонаводящиеся ракеты
    laser,      // [HP] Наносит повреждения при долговременном воздействии
    shield      // Увеличивает shieldLevel
}

public class Weapon : MonoBehaviour {
    ... // Определение класса Weapon будет добавлено далее в этой главе.
}
```

Объявленное общедоступным и за пределами класса `Weapon` перечисление `WeaponType` может использоваться любыми сценариями в проекте. Мы часто будем использовать его в оставшейся части главы и фактические определения видов оружия посредством `WeaponType` поместим в сценарий Main, а не в Weapon.

Дополнительную информацию о перечислениях вы найдете в приложении Б «Полезные идеи», в подразделе «Перечисление», внутри раздела «Основные концепции программирования на C# и в Unity».

Сериализуемый класс WeaponDefinition

Теперь создадим класс для определения параметров разных видов оружия. В отличие от большинства других классов, созданных нами в этой книге, он не будет наследовать класс `MonoBehaviour` и его не нужно подключать к игровым объектам. Это будет простой, отдельный, общедоступный класс, который мы определим внутри сценария Weapon, так же как определили общедоступное перечисление `WeaponType`.

Еще одна важная особенность этого класса — *сериализуемость*, которая позволит просматривать и изменять его с помощью инспектора в Unity!

Откройте сценарий Weapon и введите следующий код, выделенный жирным, между определениями перечисления `WeaponType` и класса `Weapon`.

```
public enum WeaponType {
    ...
}

/// <summary>
/// Класс WeaponDefinition позволяет настраивать свойства
///   конкретного вида оружия в инспекторе. Для этого класс Main
///   будет хранить массив элементов типа WeaponDefinition.
/// </summary>
[System.Serializable]                                                    // a
public class WeaponDefinition {                                          // b
    public WeaponType   type = WeaponType.none;
    public string       letter;                 // Буква на кубике, изображающем
                                                // бонус
    public Color        color = Color.white;  // Цвет ствола оружия и кубика бонуса
    public GameObject   projectilePrefab;     // Шаблон снарядов
    public Color        projectileColor = Color.white;
    public float        damageOnHit = 0;      // Разрушительная мощность
    public float        continuousDamage = 0; // Степень разрушения в секунду
                                              // (для Laser)
    public float        delayBetweenShots = 0;
    public float        velocity = 20;        // Скорость полета снарядов
}

public class Weapon : MonoBehaviour {
    ... // Определение класса Weapon будет добавлено далее в этой главе.
}
```

a. Атрибут `[System.Serializable]` объявляет следующий за ним класс доступным для сериализации и изменения в инспекторе Unity. Некоторые классы слишком сложны и потому не поддерживают сериализацию, но `WeaponDefinition` достаточно прост для этого.

b. Каждое поле в `WeaponDefinition` управляет определенной характеристикой снарядов, выстреливаемых кораблем. Мы не будем использовать все эти характеристики здесь, что оставляет вам некоторую свободу для экспериментов, если вы решите продолжить совершенствовать эту игру.

Как отмечается в комментариях в коде, перечисление `WeaponType` объявляет все возможные типы оружия и бонусов. `WeaponDefinition` — это класс, объединяющий `WeaponType` с несколькими переменными, описывающими характеристики каждого вида оружия.

Использование WeaponDefinition и WeaponType в сценарии Main

Теперь мы должны задействовать перечисление `WeaponType` и класс `WeaponDefinition` в сценарии `Main`. Сценарий `Main` выбран по той простой причине, что именно он отвечает за создание вражеских кораблей и будет отвечать за создание бонусов.

1. Добавьте следующее объявление массива `weaponDefinitions` в класс `Main` и сохраните сценарий.

```
public class Main : MonoBehaviour {
    ...
    public float            enemySpawnPerSecond = 0.5f; // Вражеских кораблей
в секунду
    public float            enemyDefaultPadding = 1.5f; // Отступ для
позиционирования
    public WeaponDefinition[]   weaponDefinitions;

    private BoundsCheck     bndCheck;

    void Awake() {...}
    ...
}
```

2. Выберите _MainCamera в иерархии. Теперь в инспекторе вы должны увидеть массив weaponDefinitions в компоненте Main (Script).
3. Раскройте массив weaponDefinitions в инспекторе, щелкнув на пиктограмме с треугольником, и в поле Size введите число 3.
4. Настройте три элемента в weaponDefinitions, как показано на рис. 31.1. Обратите внимание, что перечисление `WeaponType` отображается в инспекторе как раскрывающийся список (при этом имена вариантов, как и имена любых других элементов, в инспекторе отображаются с пробелами между словами и каждое слово начинается с заглавной буквы). Выбор конкретных цветов не имеет принципиального значения, важно только установить значение альфа-канала равным 255, чтобы сделать цвета полностью непрозрачными, о чем можно судить по белой полосе в нижней части поля выбора цвета.

ИНОГДА ЦВЕТА ПОЛУЧАЮТ ЗНАЧЕНИЕ АЛЬФА-КАНАЛА, ДЕЛАЮЩЕЕ ИХ НЕВИДИМЫМИ. При создании сериализуемых классов, таких как `WeaponDefinition`, которые включают поля, определяющие цвет, альфа-канал в них получает значение по умолчанию 0 (ноль, или невидимый). Чтобы исправить эту проблему, убедитесь что внизу каждого поля с цветом отображается белая (не черная) полоса. Если щелкнуть на самом поле, определяющем цвет, откроются поля для ввода четырех значений (R, G, B и A). Обязательно введите число 255 (полностью непрозрачный) в поле A, иначе объекты, окрашенные в этот цвет, будут невидимы.

Если вы работаете в macOS и используете системный диалог выбора цвета вместо встроенного в Unity, установить значение A вы сможете с помощью ползунка Opacity (Непрозрачность), находящегося внизу диалога (полностью непрозрачным цветам соответствует положение ползунка 100 %).

Обобщенный словарь для определений оружия

Чтобы упростить доступ к определениям оружия, мы скопируем их во время выполнения из массива `weaponDefinitions` в защищенное поле словаря с именем `WEAP_DICT`. Словарь (`Dictionary`) — это тип обобщенной коллекции подобно списку (`List`). Однако если список является упорядоченной (линейной) коллекцией, то словари определяются типом *ключей* и типом *значений* и используют ключи для

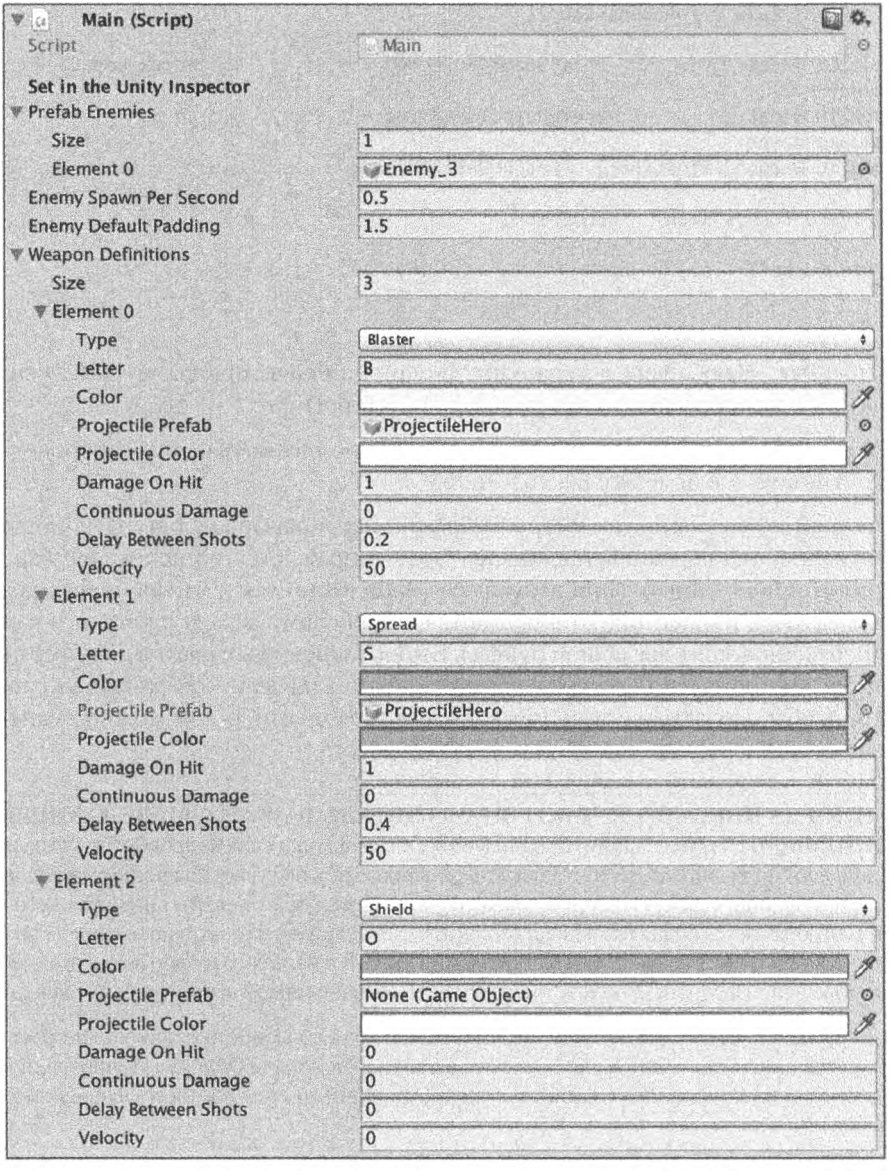


Рис. 31.1. Настройки в массиве weaponDefinitions сценария Main для blaster, spread и shield

доступа к значениям. Словари дают отличный способ хранения большого количества элементов, потому что доступ к любому из них осуществляется за постоянное время, то есть одинаково быстро независимо от того, где они находятся в структуре данных. Для сравнения: если в списке или массиве элементов типа `WeaponDefinition`

вы ищете определение оружия `blaster`, вы сразу же встретите его, тогда как для поиска определения оружия `shield` будет затрачено в три раза больше времени. За дополнительной информацией обращайтесь к главе 23 «Коллекции в C#».

Здесь ключи в словаре `WEAP_DICT` имеют тип перечисления `WeaponType`, а значения — тип `WeaponDefinition`. К сожалению, словари не отображаются в инспекторе Unity, иначе мы бы воспользовались такой возможностью с самого начала. Поэтому словарь `WEAP_DICT` определяется в методе `Awake()` класса `Main` и затем используется статической функцией `Main.GetWeaponDefinition()`.

1. Откройте сценарий `Main` в MonoDevelop и введите следующий код, выделенный жирным.

```
public class Main : MonoBehaviour {
    static public Main S; // Объект-одиночка Main
    static Dictionary<WeaponType, WeaponDefinition> WEAP_DICT;              // a
    ...

    void Awake() {
        ...
        Invoke( "SpawnEnemy", 1f/enemySpawnPerSecond );

        // Словарь с ключами типа WeaponType
        WEAP_DICT = new Dictionary<WeaponType, WeaponDefinition>();         // a
        foreach( WeaponDefinition def in weaponDefinitions ) { // b
            WEAP_DICT[def.type] = def;
        }
    }
}
```

   a. Словари объявляются и определяются с двумя типами — ключа и значения. Мы объявили словарь `WEAP_DICT` статическим и защищенным, соответственно, к нему смогут обращаться любые экземпляры и любые статические методы класса `Main`. Ниже вы увидите, какие преимущества дает это решение.

   b. Этот цикл выполняет итерации по элементам массива `weaponDefinitions` и создает для них соответствующие записи в словаре `WEAP_DICT`.

Далее мы должны создать статическую функцию, которая позволит другим классам извлекать данные из `WEAP_DICT`. Так как словарь `WEAP_DICT` также объявлен статическим, к нему смогут обращаться любые статические методы класса `Main` (`WEAP_DICT` не является общедоступным, значит, прямой доступ к `WEAP_DICT` будут иметь только экземпляры и статические методы `Main`[1]). Объявление нового метода `GetWeaponDefinition()` общедоступным и статическим позволит вызывать его как `Main.GetWeaponDefinition()` из любого места в коде проекта.

[1] Точнее, поскольку поле `WEAP_DICT` не объявлено ни общедоступным (`public`), ни скрытым (`private`), оно автоматически становится защищенным (`protected`). Защищенные элементы класса доступны самому классу и любым его подклассам (подклассы не имеют доступа к скрытым элементам родительского класса). В результате экземпляры и статические методы `Main` или его подклассов будут иметь прямой доступ к `WEAP_DICT`.

2. Добавьте следующий код, выделенный жирным, в конец сценария Main:

```
public class Main : MonoBehaviour {
    ...
    public void Restart() {
        // Перезагрузить _Scene_0, чтобы перезапустить игру
        SceneManager.LoadScene( "_Scene_0" );
    }

    /// <summary>
    /// Статическая функция, возвращающая WeaponDefinition из статического
    ///   защищенного поля WEAP_DICT класса Main.
    /// </summary>
    /// <returns>Экземпляр WeaponDefinition или, если нет такого определения
    ///   для указанного WeaponType, возвращает новый экземпляр WeaponDefinition
    ///   с типом none.</returns>
    /// <param name="wt">Тип оружия WeaponType, для которого требуется получить
    ///   WeaponDefinition</param>
    static public WeaponDefinition GetWeaponDefinition( WeaponType wt ) {  // a
        // Проверить наличие указанного ключа в словаре
        // Попытка извлечь значение по отсутствующему ключу вызовет ошибку,
        //   поэтому следующая инструкция играет важную роль.
        if (WEAP_DICT.ContainsKey(wt)) {                                   // b
            return( WEAP_DICT[wt] );
        }
        // Следующая инструкция возвращает новый экземпляр WeaponDefinition
        //   с типом оружия WeaponType.none, что означает неудачную попытку
        //   найти требуемое определение WeaponDefinition
        return( new WeaponDefinition() );                                  // c
    }
}
```

a. Документация с описанием функции в коде выше теперь включает не только раздел «summary» с кратким описанием самой функции, но также описания входного параметра и возвращаемого значения.

b. Эта инструкция `if` проверяет наличие в словаре `WEAP_DICT` элемента с ключом, переданным в параметре `wt`. Попытка извлечь из словаря элемент по отсутствующему ключу (например, `WEAP_DICT[WeaponType.phaser]`) вызовет ошибку во время выполнения.

   В ожидаемом случае, когда в словаре имеется указанный ключ, функция возвращает соответствующий ему экземпляр `WeaponDefinition`.

c. Если указанный ключ отсутствует в словаре `WEAP_DICT`, возвращается новый экземпляр `WeaponDefinition` с типом оружия `WeaponType.none`.

Использование определений WeaponDefinition в классе Projectile

Нам придется существенно изменить класс `Projectile`, чтобы задействовать в нем новые определения `WeaponDefinition`.

1. Откройте сценарий Projectile в MonoDevelop.

2. Измените его содержимое, чтобы оно в точности соответствовало следующему листингу.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Projectile : MonoBehaviour {
    private BoundsCheck     bndCheck;
    private Renderer        rend;

    [Header("Set Dynamically")]
    public Rigidbody        rigid;
    [SerializeField]                                                    // a
    private WeaponType _type;                                           // b

    // Это общедоступное свойство маскирует поле _type и обрабатывает
    //   операции присваивания ему нового значения
    public WeaponType type {                                            // c
        get {
            return( _type );
        }
        set {
            SetType( value );                                           // c
        }
    }

    void Awake () {
        bndCheck = GetComponent<BoundsCheck>();
        rend = GetComponent<Renderer>();                                // d
        rigid = GetComponent<Rigidbody>();
    }

    void Update () {
        if (bndCheck.offUp) {
            Destroy( gameObject );
        }
    }

    /// <summary>
    /// Изменяет скрытое поле _type и устанавливает цвет этого снаряда,
    ///   как определено в WeaponDefinition.
    /// </summary>
    /// <param name="eType">Тип WeaponType используемого оружия.</param>
    public void SetType( WeaponType eType ) {                           // e
        // Установить _type
        _type = eType;
        WeaponDefinition def = Main.GetWeaponDefinition( _type );
        rend.material.color = def.projectileColor;
    }
}
```

a. Атрибут `[SerializeField]` над объявлением `_type` делает это поле видимым и доступным для изменения в инспекторе Unity, хотя оно и скрытое. Но вы не должны определять значение этого поля в инспекторе.

b. Обычная практика, повсюду используемая в этой книге, давать скрытым полям, доступным через свойства, имена, начинающиеся с символа подчеркивания, за которым следует имя свойства (например, скрытое поле `_type` доступно через свойство `type`).

c. Метод чтения `get` свойства `type` действует точно так же, как у всех других свойств, которые вы видели выше, но метод записи `set` вызывает метод `SetType()` и позволяет сделать намного больше, чем просто присвоить значение полю `_type`.

d. Метод `SetType()` использует компонент `Renderer`, подключенный к этому игровому объекту, поэтому здесь мы запоминаем ссылку на него.

e. `SetType()` не только присваивает значение скрытому полю `_type`, но и устанавливает цвет снаряда в соответствии с цветом, заданным в `weaponDefinitions` в классе `Main`.

Использование функции-делегата для стрельбы

Класс `Hero` в этом прототипе игры имеет *функцию-делегат* с именем `fireDelegate`, которая вызывается, чтобы сделать выстрел из любого оружия, и каждое подключенное оружие (типа `Weapon`) добавляет свой индивидуальный целевой метод `Fire()` в `fireDelegate`.

1. Прежде чем продолжить, прочитайте раздел «Функции-делегаты» в приложении Б «Полезные идеи». Функция-делегат подобна псевдониму для одной или нескольких функций, которые все сразу можно вызвать единственным вызовом делегата.

2. Добавьте в класс `Hero` следующий код, выделенный жирным:

```
public class Hero : MonoBehaviour {
    ...
    private GameObject      lastTriggerGo = null;

    // Объявление нового делегата типа WeaponFireDelegate
    public delegate void WeaponFireDelegate();                              // a
    // Создать поле типа WeaponFireDelegate с именем fireDelegate.
    public WeaponFireDelegate fireDelegate;

    void Awake() {
        if (S == null) {
            ...
        }
        fireDelegate += TempFire;                                           // b
    }

    void Update () {
        ...
        transform.rotation = Quaternion.Euler(yAxis*pitchMult,xAxis*rollMult,0);
```

```
        // Позволить кораблю выстрелить
//      if ( Input.GetKeyDown(KeyCode.Space) ) {                    // c
//          TempFire();                                             // c
//      }                                                           // c

        // Произвести выстрел из всех видов оружия вызовом fireDelegate
        // Сначала проверить нажатие клавиши: Axis("Jump")
        // Затем убедиться, что значение fireDelegate не равно null,
        //   чтобы избежать ошибки
        if (Input.GetAxis("Jump") == 1 && fireDelegate != null) {   // d
            fireDelegate();                                         // e
        }
    }

    void TempFire() {                                               // f
        GameObject projGO = Instantiate<GameObject>( projectilePrefab );
        projGO.transform.position = transform.position;
        Rigidbody rigidB = projGO.GetComponent<Rigidbody>();
//      rigidB.velocity = Vector3.up * projectileSpeed;             // g

        Projectile proj = projGO.GetComponent<Projectile>();        // h
        proj.type = WeaponType.blaster;
        float tSpeed = Main.GetWeaponDefinition( proj.type ).velocity;
        rigidB.velocity = Vector3.up * tSpeed;
    }

    void OnTriggerEnter(Collider other) { ... }
    ...
}
```

a. Даже при том, что тип делегата `WeaponFireDelegate()` и поле `fireDelegate` объявлены общедоступными, они не отображаются в инспекторе Unity.

b. `TempFire` добавляется в `fireDelegate`, благодаря чему `TempFire` будет вызываться при каждом вызове `fireDelegate` как функции (см. `// e`).

   Обратите внимание, что при включении `TempFire` в `fireDelegate` не нужно добавлять круглые скобки после имени `TempFire`, потому что в этом случае производится добавление самого метода, а не результата его вызова (что произойдет, если добавить круглые скобки).

c. Обязательно закомментируйте (или удалите) инструкцию `if ( Input.GetKeyDown(KeyCode.Space) ) { ... }` со всем ее содержимым.

d. Вызов `Input.GetAxis("Jump")` вернет 1, если была нажата клавиша пробела на клавиатуре или кнопка на пульте, отвечающая за прыжок.

   Если вызвать делегат `fireDelegate`, к которому не подключено ни одного метода, он сгенерирует ошибку. Чтобы избежать этого, перед вызовом выполняется проверка `fireDelegate != null`.

e. Здесь производится вызов делегата `fireDelegate`, как если бы это была функция. В свою очередь, делегат вызывает все функции, подключенные к нему (на данный момент это означает вызов `TempFire()`).

f. Теперь для выстрела из стандартного бластера метод `TempFire()` вызывается делегатом `fireDelegate`. Позднее мы заменим `TempFire()`, когда создадим класс `Weapon`.

g. Закомментируйте или удалите эту строку.

h. Этот новый фрагмент извлекает информацию из `WeaponType` класса `Projectile` и использует ее для определения скорости движения игрового объекта `projGO`.

3. Щелкните на кнопке Play (Играть) в Unity и попробуйте произвести выстрел. За короткое время бластер должен выстрелить большим количеством снарядов. В следующем разделе мы добавим класс `Weapon`, который улучшит управление оружием и заменит `TempFire()` своей функцией `Fire()`.

Создание объекта Weapon для стрельбы снарядами

Начнем с создания нового игрового объекта Weapon, представляющего оружие. Благодаря такой организации мы сможем подключать к космическому кораблю _Hero любое количество разных видов оружия, каждый из которых будет добавлять себя в `fireDelegate` класса `Hero` и стрелять, когда `fireDelegate` будет вызываться как функция.

1. В иерархии создайте новый пустой игровой объект с именем Weapon и добавьте в него следующие дочерние объекты:

```
Weapon (Empty)       P:[ 0, 2, 0 ]     R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
      Barrel (Cube)  P:[ 0, 0.5, 0 ]   R:[ 0, 0, 0 ]   S:[ 0.25, 1, 0.1 ]
      Collar (Cube)  P:[ 0, 1, 0 ]     R:[ 0, 0, 0 ]   S:[ 0.375, 0.5, 0.2 ]
```

2. Удалите компоненты Collider из объектов Barrel и Collar. Для этого выберите их по отдельности, щелкните правой кнопкой мыши на компоненте Box Collider и выберите в контекстном меню пункт Remove Component (Удалить компонент). Вызвать то же самое меню можно также щелчком на кнопке с изображением шестеренки, справа от имени компонента Box Collider.

3. Создайте новый материал с именем Mat_Collar в папке _Materials в панели Project (Проект).

4. Перетащите этот материал на игровой объект Collar. В инспекторе выберите в раскрывающемся списке Shader пункт ProtoTools > UnlitAlpha (рис. 31.2).

5. Подключите сценарий Weapon на C# к игровому объекту Weapon в иерархии.

6. Перетащите игровой объект Weapon в папку _Prefabs в панели Project (Проект), чтобы превратить его в шаблон.

7. Сделайте экземпляр Weapon в иерархии дочерним по отношению к _Hero и настройте его позицию [0, 2, 0]. В результате экземпляр Weapon должен разместиться на носу корабля _Hero, как показано на рис. 31.2.

8. Сохраните сцену! Не забываете постоянно сохранять ее?

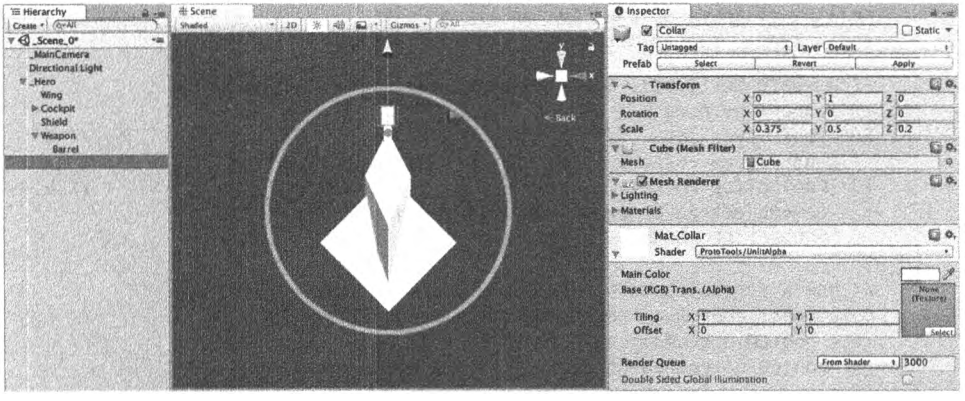


Рис. 31.2. Объект Weapon с выбранным дочерним объектом Collar и настроенными материалом и шейдером

Реализация стрельбы в сценарии Weapon

Для реализации стрельбы выполните следующие шаги:

1. Для начала запретите использование метода `TempFire()` класса `Hero` в `fireDelegate`. Откройте сценарий **Hero** в MonoDevelop и закомментируйте следующую строку, выделенную жирным:

```
public class Hero : MonoBehaviour {
    ...
    void Awake() {
        ...
//          fireDelegate += TempFire;
    }
    ...
}
```

Теперь `fireDelegate` больше не будет вызывать `TempFire()`. Но если у вас появится такое желание, можете удалить метод `TempFire()` из класса `Hero`. Если сейчас запустить игру и нажать клавишу пробела, корабль **Hero** не выстрелит.

2. Откройте сценарий **Weapon** в MonoDevelop и добавьте следующий код, выделенный жирным:

```
public class Weapon : MonoBehaviour {
    static public Transform   PROJECTILE_ANCHOR;

    [Header("Set Dynamically")]   [SerializeField]
    private WeaponType        _type = WeaponType.none;
    public WeaponDefinition   def;
    public GameObject         collar;
    public float              lastShotTime; // Время последнего выстрела
    private Renderer          collarRend;

    void Start() {
```

```
    collar = transform.Find("Collar").gameObject;
    collarRend = collar.GetComponent<Renderer>();

    // Вызвать SetType(), чтобы заменить тип оружия по умолчанию
    // WeaponType.none
    SetType( _type );                                                    // a
    // Динамически создать точку привязки для всех снарядов
    if (PROJECTILE_ANCHOR == null) {                                     // b
        GameObject go = new GameObject("_ProjectileAnchor");
        PROJECTILE_ANCHOR = go.transform;
    }
    // Найти fireDelegate в корневом игровом объекте
    GameObject rootGO = transform.root.gameObject;                       // c
    if ( rootGO.GetComponent<Hero>() != null ) {                         // d
        rootGO.GetComponent<Hero>().fireDelegate += Fire;
    }
}

public WeaponType type {
    get { return( _type ); }
    set { SetType( value ); }
}

public void SetType( WeaponType wt ) {
    _type = wt;
    if (type == WeaponType.none) {                                       // e
        this.gameObject.SetActive(false);
        return;
    } else {
        this.gameObject.SetActive(true);
    }
    def = Main.GetWeaponDefinition(_type);                               // f
    collarRend.material.color = def.color;
    lastShotTime = 0; // Сразу после установки _type можно выстрелить   // g
}

public void Fire() {
    // Если this.gameObject неактивен, выйти
    if (!gameObject.activeInHierarchy) return;                           // h
    // Если между выстрелами прошло недостаточно много времени, выйти
    if (Time.time - lastShotTime < def.delayBetweenShots) {              // i
        return;
    }
    Projectile p;
    Vector3 vel = Vector3.up * def.velocity;                             // j
    if (transform.up.y < 0) {
        vel.y = -vel.y;
    }
    switch (type) {                                                      // k
        case WeaponType.blaster:
            p = MakeProjectile();
            p.rigid.velocity = vel;
            break;

        case WeaponType.spread:                                          // l
            p = MakeProjectile(); // Снаряд, летящий прямо
```

```
                p.rigid.velocity = vel;
                p = MakeProjectile(); // Снаряд, летящий вправо
                p.transform.rotation = Quaternion.AngleAxis( 10, Vector3.back );
                p.rigid.velocity = p.transform.rotation * vel;
                p = MakeProjectile(); // Снаряд, летящий влево
                p.transform.rotation = Quaternion.AngleAxis(-10, Vector3.back );
                p.rigid.velocity = p.transform.rotation * vel;
                break;
            }
        }

        public Projectile MakeProjectile() {                                    // m
            GameObject go = Instantiate<GameObject>( def.projectilePrefab );
            if ( transform.parent.gameObject.tag == "Hero" ) {                  // n
                go.tag = "ProjectileHero";
                go.layer = LayerMask.NameToLayer("ProjectileHero");
            } else {
                go.tag = "ProjectileEnemy";
                go.layer = LayerMask.NameToLayer("ProjectileEnemy");
            }
            go.transform.position = collar.transform.position;
            go.transform.SetParent( PROJECTILE_ANCHOR, true );                  // o
            Projectile p = go.GetComponent<Projectile>();
            p.type = type;
            lastShotTime = Time.time;                                           // p
            return( p );
        }
    }
```

a. В момент создания игровой объект Weapon вызовет метод `SetType()` и передаст ему значение из поля `WeaponType _type`. Этот вызов скроет оружие Weapon (если `_type` имеет значение `WeaponType.none`) или окрасит его ствол в соответствующий цвет (если `_type` имеет значение `WeaponType.blaster` или `WeaponType.spread`).

b. `PROJECTILE_ANCHOR` — это статический компонент Transform, который должен играть роль родителя в иерархии для всех снарядов, создаваемых сценариями Weapon. Если `PROJECTILE_ANCHOR` имеет значение `null` (то есть когда этот компонент еще не создан), сценарий создает новый игровой объект с именем _ProjectileAnchor и присваивает ссылку на его компонент Transform полю `PROJECTILE_ANCHOR`.

c. Оружие всегда подключается к другому игровому объекту (такому, как _Hero). Эта строка находит корневой игровой объект, к которому подключено это оружие.

d. Если к корневому игровому объекту подключен сценарий Hero, метод `Fire()` этого экземпляра `Weapon` добавляется в делегат `fireDelegate` данного экземпляра класса `Hero`. Если у вас появится желание установить оружие на корабли врагов, можете также добавить сюда похожую инструкцию `if`, проверяющую наличие сценария Enemy. Даже если к `rootGO` будет подключен подкласс класса `Enemy` (например, `Enemy_1`, `Enemy_2` и т. д.), инструкция по-

иска Enemy вернет соответствующий компонент сценария, следуя правилам наследования классов.

e. Если type имеет значение WeaponType.none, этот игровой объект деактивируется. Когда игровой объект неактивен, не вызывается ни один из его методов, унаследованных от класса MonoBehaviour (например, Update(), LateUpdate(), FixedUpdate(), OnCollisionEnter() и т. д.), он не участвует в моделировании физики и не отображается в сцене. Однако сохраняется возможность вызывать функции и изменять значения переменных из сценариев, подключенных к неактивному игровому объекту, то есть если откуда-то из другого места вызвать SetType() или присвоить свойству type значение WeaponType.blaster или WeaponType.spread, метод SetType() активирует игровой объект, к которому подключен.

f. Метод SetType() не только активирует или деактивирует игровой объект, он также извлекает определение оружия WeaponDefinition из класса Main, устанавливает цвет Collar и сбрасывает поле lastShotTime.

g. Сброс поля lastShotTime в значение 0 позволяет тут же произвести выстрел из этого оружия (см. // i).

h. gameObject.activeInHierarchy имеет значение false, если неактивен данный экземпляр Weapon или неактивен или уничтожен игровой объект _Hero (корневой родитель для данного оружия). В любом случае, если gameObject.activeInHierarchy имеет значение false, эта функция тут же возвращает управление и выстрел не производится.

i. Если с момента последнего выстрела из этого оружия Weapon прошло меньше времени, чем значение delayBetweenShots в WeaponDefinition, выстрел не производится.

j. Задается начальная скорость полета в направлении вверх, но если transform.up.y < 0 (что возможно для вражеских кораблей, оружие которых обращено вниз), компонент y скорости vel также обращается вниз.

k. Эта инструкция switch включает варианты для каждого из двух типов оружия WeaponTypes, реализованных в этой главе. Оружие WeaponType.blaster выстреливает единственный снаряд, вызывая метод MakeProjectile() (который возвращает ссылку на экземпляр класса Projectile, подключенного к новому игровому объекту снаряда), и затем устанавливает скорость для его твердого тела Rigidbody в направлении vel.

l. Если _type имеет значение WeaponType.spread, создаются три разных снаряда. Для двух из них производится поворот направления на 10 градусов относительно оси Vector3.back (то есть оси Z, которая простирается от плоскости экрана к вам). Затем их скорости Rigidbody.velocity устанавливаются равными произведению угла поворота на vel. Когда экземпляр Quaternion умножается на Vector3, он поворачивает вектор Vector3, в результате чего снаряд полетит в сторону под заданным углом.

m. Метод `MakeProjectile()` создает экземпляры шаблона, хранящегося в `WeaponDefinition`, и возвращает ссылку на экземпляр класса `Projectile`.

n. В зависимости от того, каким кораблем произведен выстрел, _Hero или Enemy, снаряду назначается соответствующий тег и физический уровень.

o. Родителем игрового объекта Projectile назначается `PROJECTILE_ANCHOR`. В результате этого в панели Hierarchy (Иерархия) снаряд оказывается внутри _ProjectileAnchor, что обеспечивает относительный порядок в ней и отсутствие нагромождений из большого количества клонов Projectile. Аргумент `true` указывает игровому объекту go, что дочерний объект должен сохранить свои мировые координаты независимо от родителя.

p. Полю `lastShotTime` присваивается текущее время, что предотвращает возможность повторного выстрела раньше, чем через `def.delayBetweenShots` секунд.

3. Щелкните на кнопке Play (Играть), и экземпляр Weapon, подключенный к _Hero, исчезнет. Это произошло потому, что WeaponType имеет значение `WeaponType.none`.

4. Выберите в иерархии экземпляр Weapon, подключенный к _Hero, и в настройках компонента Weapon (Script) выберите в поле type значение Blaster. Щелкните на кнопке Play (Играть); если теперь удерживать нажатой клавишу пробела, бластер будет производить выстрел каждые 0,2 секунды (как определено в массиве weaponDefinitions в настройках компонента Main (Script) главной камеры _MainCamera).

5. Выберите в иерархии экземпляр Weapon, подключенный к _Hero, а в настройках компонента Weapon (Script) установите в поле type значение Spread. Щелкните на кнопке Play (Играть); ствол оружия теперь окрасится в синий цвет и, если удерживать нажатой клавишу пробела, каждые 0,4 секунды будет производиться выстрел сразу тремя снарядами.

Переделка метода OnCollisionEnter вражеского корабля

Теперь, реализовав стрельбу снарядами, которые потенциально могут причинять разрушения разной степени (хотя сейчас для них установлена одинаковая разрушительная сила), мы должны усовершенствовать метод `OnCollisionEnter()` класса `Enemy`.

1. Откройте сценарий Enemy в MonoDevelop и удалите прежний метод `OnCollisionEnter()`.

2. Замените его следующим кодом:

```
public class Enemy : MonoBehaviour {
    ...
    public virtual void Move() { ... }

    void OnCollisionEnter( Collision coll ) {                    // a
```

```
        GameObject otherGO = coll.gameObject;
        switch (otherGO.tag) {
            case "ProjectileHero":                                          // b
                Projectile p = otherGO.GetComponent<Projectile>();

                // Если вражеский корабль за границами экрана,
                //   не наносить ему повреждений.
                if ( !bndCheck.isOnScreen ) {                               // c
                    Destroy( otherGO );
                    break;
                }

                // Поразить вражеский корабль
                // Получить разрушающую силу из WEAP_DICT в классе Main.
                health -= Main.GetWeaponDefinition(p.type).damageOnHit;
                if (health <= 0) {                                          // d
                    // Уничтожить этот вражеский корабль
                    Destroy(this.gameObject);
                }
                Destroy( otherGO );                                         // e
                break;

            default:
                print( "Enemy hit by non-ProjectileHero: " + otherGO.name ); // f
                break;
        }
    }
}
```

a. Замените прежнюю версию метода `OnCollisionEnter()`целиком.

b. Если игровой объект, поразивший этот экземпляр Enemy, имеет тег ProjectileHero, вражеский корабль должен получить повреждения. Игровые объекты с другими тегами обрабатываются в варианте `default` (`// f`).

c. Если этот вражеский корабль находится за границами экрана, игровой объект Projectile, попавший в него, уничтожается и выполняется инструкция `break;`, которая производит выход из инструкции `switch`, минуя остальной код в `case "ProjectileHero"`.

d. Если уровень исправности вражеского корабля Enemy упал до 0 или ниже, он уничтожается. С уровнем исправности 10 по умолчанию и разрушающей способностью 1 бластера (`damageOnHit`) для уничтожения вражеского корабля требуется 10 попаданий.

e. Уничтожается игровой объект Projectile.

f. Если с вражеским кораблем столкнулся игровой объект с каким-то другим тегом, отличным от ProjectileHero, сообщение об этом выводится в панель Console (Консоль).

3. Прежде чем щелкнуть на кнопке Play (Играть), верните создание экземпляров обычного вражеского корабля вместо Enemy_3. Для этого выберите _MainCamera в иерархии и подставьте шаблон Enemy_0 в поле Element 0 в массиве prefabEnemies.

Теперь, запустив сцену, вы сможете уничтожать вражеские корабли, но для этого в каждый из них нужно попасть не менее 10 раз, однако пока не видно, что попадания наносят повреждения.

Отображение попаданий

Чтобы показать попадание снарядов, добавим код, который будет окрашивать вражеский корабль в красный цвет на период в пару кадров. Но для этого нам потребуется доступ ко всем материалам всех дочерних объектов всех вражеских кораблей. Эта операция может понадобиться в разных играх, поэтому реализуем ее в новом классе `Utils`, который будем постепенно наполнять кодом, пригодным для многоразового использования в играх.

Создание многоразового сценария Utils

Мы будем использовать класс `Utils` на протяжении оставшейся части книги. Класс `Utils` почти полностью будет состоять из статических функций, поэтому их легко можно вызывать откуда угодно.

1. Создайте новый сценарий на C# с именем Utils и поместите его в папку __Scripts. Откройте Utils в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Utils : MonoBehaviour {

//====================== Функции для работы с материалами ======================\\

    // Возвращает список всех материалов в данном игровом объекте
    //   и его дочерних объектах
    static public Material[] GetAllMaterials( GameObject go ) {            // a
        Renderer[] rends = go.GetComponentsInChildren<Renderer>();         // b

        List<Material> mats = new List<Material>();
        foreach (Renderer rend in rends) {                                 // c
            mats.Add( rend.material );
        }

        return( mats.ToArray() ); // d
    }
}
```

   a. Статический и общедоступный метод `GetAllMaterials()` можно вызывать из любого кода в проекте, как `Utils.GetAllMaterials()`.

   b. `GetComponentsInChildren<>()` — это метод класса `GameObject`, который выполняет обход самого игрового объекта и всех его дочерних объектов и воз-

вращает массив компонентов с типом, указанным в параметре шаблона <> (в данном примере — тип компонента `Renderer`).

c. Этот цикл `foreach` выполняет обход всех компонентов `Renderer` в массиве `rends` и извлекает значения из поля `material` каждого из них. Затем полученный материал добавляется в список `mats`.

d. В заключение список `mats` преобразуется в массив и возвращается вызывающему коду.

Использование GetAllMaterials для кратковременного окрашивания вражеского корабля в красный цвет

Теперь воспользуемся статическим методом `GetAllMaterials()` класса `Utils` в реализации вражеского корабля:

1. Добавьте в класс `Enemy` следующий код, выделенный жирным:

```
public class Enemy : MonoBehaviour {
    ...
    public int        score = 100; // Очки за уничтожение этого корабля
    public float      showDamageDuration = 0.1f; // Длительность эффекта   // a
                                                 //   попадания в секундах

    [Header("Set Dynamically: Enemy")]
    public Color[]    originalColors;
    public Material[] materials;// Все материалы игрового объекта и его потомков
    public bool       showingDamage = false;
    public float      damageDoneTime; // Время прекращения отображения эффекта
    public bool       notifiedOfDestruction = false; // Будет использовано позже

    protected BoundsCheck bndCheck;

    void Awake() {
        bndCheck = GetComponent<BoundsCheck>();
        // Получить материалы и цвет этого игрового объекта и его потомков
        materials = Utils.GetAllMaterials( gameObject );                    // b
        originalColors = new Color[materials.Length];
        for (int i=0; i<materials.Length; i++) {
            originalColors[i] = materials[i].color;
        }
    }

    ...

    void Update() {
        Move();

        if ( showingDamage && Time.time > damageDoneTime ) {                // c
            UnShowDamage();
        }

        if ( bndCheck != null && bndCheck.offDown ) {
```

```
            // Корабль за нижней границей, поэтому его нужно уничтожить
            Destroy( gameObject );
        }
    }

    ...

    void OnCollisionEnter( Collision coll ) {
        GameObject otherGO = coll.gameObject;

        switch (otherGO.tag) {
            case "ProjectileHero":
                ...
                // Поразить вражеский корабль
                ShowDamage();                                                    // d
                // Получить разрушающую силу из WEAP_DICT в классе Main.
                ...
        }
    }

    void ShowDamage() {                                                          // e
        foreach (Material m in materials) {
            m.color = Color.red;
        }
        showingDamage = true;
        damageDoneTime = Time.time + showDamageDuration;
    }

    void UnShowDamage() {                                                        // f
        for ( int i=0; i<materials.Length; i++ ) {
            materials[i].color = originalColors[i];
        }
        showingDamage = false;
    }
}
```

a. Добавьте все новые поля, выделенные жирным, в начало определения класса.

b. Вызывается новый метод `Utils.GetAllMaterials()` и заполняется массив `materials`. Затем выполняется обход всех материалов и сохраняются их исходные цвета. Несмотря на то что в настоящий момент все игровые объекты Enemy окрашены в белый цвет, этот метод позволяет окрашивать их в любые цвета по вашему выбору, подсвечивать их красным цветом при попадании и затем возвращать исходный цвет.

   Обратите внимание, что вызов `Utils.GetAllMaterials()` производится в методе `Awake()`, а результат кэшируется в `materials`. То есть для каждого экземпляра Enemy эта операция будет выполнена только один раз. `Utils.GetAllMaterials()` использует довольно медленную функцию `GetComponentsInChildren<>()`, частый вызов которой может ухудшить производительность. Поэтому намного лучше вызвать ее только один раз и сохранить результат в кэше.

c. Если в текущий момент отображается эффект попадания (то есть корабль окрашен в красный цвет) и текущее время больше `damageDoneTime`, вызывается `UnShowDamage()`.

d. Вызов `ShowDamage()` добавляется в метод `OnCollisionEnter()`, в раздел, где обрабатывается попадание снаряда.

e. `ShowDamage()` окрашивает все материалы в массиве `materials` в красный цвет, присваивает полю `showingDamage` значение `true` и устанавливает время, когда демонстрация эффекта должна прекратиться.

f. `UnShowDamage()` возвращает всем материалам в массиве `materials` их исходные цвета и сбрасывает поле `showingDamage` в значение `false`.

Теперь при попадании снаряда игрока вражеский корабль будет окрашиваться в красный цвет на `showDamageDuration` секунд изменением цвета всех его материалов. По прошествии `showDamageDuration` секунд сценарий `Enemy` вернет исходный цвет кораблю и всем его дочерним объектам.

2. Щелкните на кнопке `Play` (Играть) и протестируйте игру. Теперь намного проще заметить, когда происходит попадание снаряда во вражеский корабль, однако для полного его уничтожения все еще требуется довольно много попаданий. Давайте добавим несколько бонусов, увеличивающих количество и разрушительную силу оружия.

3. Не забыли сохранить проект? Как можно чаще сохраняйте свои проекты.

Добавление бонусов для увеличения мощности оружия

В этом разделе мы добавим три бонуса:

- **бластер [B]:** этот бонус делает бластер (blaster) текущим оружием и оснащает корабль единственной пушкой. Если до этого бластер уже был текущим оружием, этот бонус увеличит количество стволов.
- **веерная пушка [S]**: этот бонус делает веерную пушку (spread) текущим оружием и оснащает корабль единственной пушкой. Если до этого веерная пушка уже была текущим оружием, этот бонус увеличит количество стволов.
- **защита [O]**: этот бонус увеличит на 1 уровень защитного поля `shieldLevel` корабля игрока.

Подготовка ресурсов для бонусов

Бонусы отображаются в виде трехмерных букв на фоне вращающихся кубиков. (Примеры некоторых бонусов можно увидеть на рис. 30.1 в начале предыдущей главы.) Чтобы создать ресурсы для бонусов, выполните следующие шаги:

1. Создайте новый игровой объект, отображающий трехмерный текст (пункт GameObject > 3D Object > 3D Text (Игровой объект > 3D объект > 3D Текст) в главном меню) с именем PowerUp, добавьте в него дочерний куб и настройте их, как показано ниже:

```
PowerUp (3D Text)   P:[ 10, 0, 0 ]   R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
       Cube         P:[ 0, 0, 0 ]    R:[ 0, 0, 0 ]   S:[ 2, 2, 2 ]
```

2. Выберите PowerUp.
3. Настройте компонент Text Mesh объекта PowerUp, как показано на рис. 31.3.
4. Добавьте в PowerUp компонент Rigidbody (Component > Physics > Rigidbody (Компонент > Физика > Твердое тело)) и настройте его, как показано на рис. 31.3.
5. Настройте тег и физический слой в PowerUp. Ответьте Yes (Да) в ответ на предложение изменить также дочерние объекты.
6. Создайте материал для дочернего куба в PowerUp:
   a. Создайте новый материал с именем Mat_PowerUp в папке _Materials.
   b. Перетащите его на объект Cube, вложенный в PowerUp.
   c. Выберите объект Cube, вложенный в PowerUp.
   d. В разделе Mat PowerUp выберите в раскрывающемся списке Shader пункт ProtoTools > UnlitAlpha.
   e. Щелкните на кнопке Select (Выбрать) в правом нижнем углу квадратика с изображением текстуры для Mat_PowerUp и на вкладке Assets (Ресурсы) выберите текстуру с именем PowerUp. При этом вам может понадобиться щелкнуть в инспекторе на пиктограмме с треугольником в левом нижнем углу компонента Mat_PowerUp, чтобы увидеть текстуру для Mat_PowerUp.
   f. В поле Main Color компонента Mat_PowerUp выберите цвет циан (яркий сине-зеленый, со значениями RGBA: [0, 255, 255, 255]), и вы увидите, как PowerUp окрасится в этот цвет.
   g. Сделайте коллайдер Box Collider объекта Cube триггером (установите флажок Is Trigger).

Еще раз проверьте все настройки объекта PowerUp и вложенного в него объекта Cube, чтобы они соответствовали изображенным на рис. 31.3, и сохраните сцену.

Код для управления PowerUp

Далее описывается код, управляющий поведением бонусов:

1. Подключите сценарий BoundsCheck к игровому объекту PowerUp в иерархии. Введите 1 в поле radius и снимите флажок keepOnScreen.
2. Создайте новый сценарий на C# с именем PowerUp в папке __Scripts.

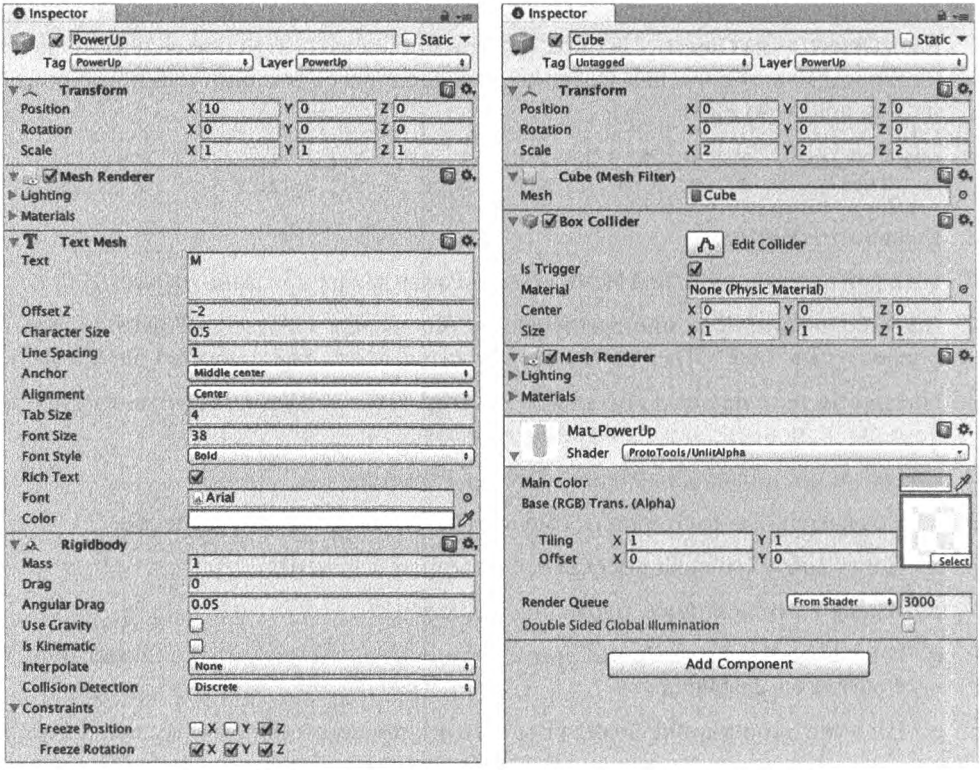


Рис. 31.3. Настройки объекта PowerUp и вложенного в него объекта Cube до подключения сценариев

3. Подключите сценарий **PowerUp** к игровому объекту **PowerUp** в иерархии.

4. Откройте сценарий **PowerUp** в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PowerUp : MonoBehaviour {
    [Header("Set in Inspector")]
    // Необычное, но удобное применение Vector2. x хранит минимальное
    //   значение, а y - максимальное значение для метода Random.Range(),
    //   который будет вызываться позже
    public Vector2       rotMinMax = new Vector2(15,90);
    public Vector2       driftMinMax = new Vector2(.25f,2);
    public float         lifeTime = 6f; // Время в секундах существования PowerUp
    public float         fadeTime = 4f; // Seconds it will then fade

    [Header("Set Dynamically")]
    public WeaponType    type;         // Тип бонуса
```

```
public GameObject    cube;         // Ссылка на вложенный куб
public TextMesh      letter;       // Ссылка на TextMesh
public Vector3       rotPerSecond; // Скорость вращения
public float         birthTime;

private Rigidbody    rigid;
private BoundsCheck  bndCheck;
private Renderer     cubeRend;

void Awake() {
    // Получить ссылку на куб
    cube = transform.Find("Cube").gameObject;
    // Получить ссылки на TextMesh и другие компоненты
    letter = GetComponent<TextMesh>();
    rigid = GetComponent<Rigidbody>();
    bndCheck = GetComponent<BoundsCheck>();
    cubeRend = cube.GetComponent<Renderer>();

    // Выбрать случайную скорость
    Vector3 vel = Random.onUnitSphere; // Получить случайную скорость XYZ
    // Random.onUnitSphere возвращает вектор, указывающий на случайную
    //   точку, находящуюся на поверхности сферы с радиусом 1 м и с центром
    //   в начале координат
    vel.z = 0;       // Отобразить vel на плоскость XY
    vel.Normalize(); // Нормализация устанавливает длину Vector3 равной 1 м
    vel *= Random.Range(driftMinMax.x, driftMinMax.y);                  // a
    rigid.velocity = vel;

    // Установить угол поворота этого игрового объекта равным R:[ 0, 0, 0 ]
    transform.rotation = Quaternion.identity;
    // Quaternion.identity равноценно отсутствию поворота.

    // Выбрать случайную скорость вращения для вложенного куба с
    //   использованием rotMinMax.x и rotMinMax.y
    rotPerSecond = new Vector3( Random.Range(rotMinMax.x,rotMinMax.y),
        Random.Range(rotMinMax.x,rotMinMax.y),
        Random.Range(rotMinMax.x,rotMinMax.y) );

    birthTime = Time.time;
}

void Update () {
    cube.transform.rotation = Quaternion.Euler( rotPerSecond*Time.time ); // b

    // Эффект растворения куба PowerUp с течением времени
    // Со значениями по умолчанию бонус существует 10 секунд
    //   а затем растворяется в течение 4 секунд.
    float u = (Time.time - (birthTime+lifeTime)) / fadeTime;
    // В течение lifeTime секунд значение u будет <= 0. Затем оно станет
    //   положительным и через fadeTime секунд станет больше 1.

    // Если u >= 1, уничтожить бонус
    if (u >= 1) {
        Destroy( this.gameObject );
```

```
            return;
        }

        // Использовать u для определения альфа-значения куба и буквы
        if (u>0) {
            Color c = cubeRend.material.color;
            c.a = 1f-u;
            cubeRend.material.color = c;
            // Буква тоже должна растворяться, но медленнее
            c = letter.color;
            c.a = 1f - (u*0.5f);
            letter.color = c;
        }

        if (!bndCheck.isOnScreen) {
            // Если бонус полностью вышел за границу экрана, уничтожить его
            Destroy( gameObject );
        }
    }

    public void SetType( WeaponType wt ) {
        // Получить WeaponDefinition из Main
        WeaponDefinition def = Main.GetWeaponDefinition( wt );
        // Установить цвет дочернего куба
        cubeRend.material.color = def.color;
        //letter.color = def.color; // Букву тоже можно окрасить в тот же цвет
        letter.text = def.letter;   // Установить отображаемую букву
        type = wt; // В заключение установить фактический тип
    }

    public void AbsorbedBy( GameObject target ) {
        // Эта функция вызывается классом Hero, когда игрок подбирает бонус
        // Можно было бы реализовать эффект поглощения бонуса, уменьшая его
        //   размеры в течение нескольких кадров, но пока просто уничтожим
        //   this.gameObject
        Destroy( this.gameObject );
    }
}
```

a. Выбор случайной скорости между значениями компонентов `x` и `y` вектора `driftMinMax`.

b. Поворот вложенного куба в каждом вызове `Update()`. Умножение `rotPerSecond` на `Time.time` обеспечивает вращение с постоянной скоростью во времени.

5. Щелкните на кнопке **Play** (Играть), и вы увидите, как бонус плавно падает вниз, вращаясь. Если вы подведете свой корабль и поймаете бонус, в консоли появится текст «Triggered by non-Enemy: PowerUp» (Столкновение с объектом, отличным от Enemy: PowerUp), сообщающий, что коллайдер-триггер в кубе **PowerUp** работает правильно.

6. Перетащите игровой объект **PowerUp** из иерархии в папку **_Prefabs** в панели **Project** (Проект), чтобы превратить его в шаблон.

Добавление возможности собирать бонусы

Далее мы должны добавить в объект корабля Hero игрока возможность собирать бонусы PowerUp. Сначала просто добавим возможность собирать бонусы, а затем реализуем усовершенствование и изменение оружия.

1. Внесите следующие изменения в сценарий Hero, чтобы добавить возможность собирать бонусы:

```
public class Hero : MonoBehaviour {
    ...
    void OnTriggerEnter(Collider other) {
        ...
        if (go.tag == "Enemy") {
            // Если защитное поле столкнулось с вражеским кораблем
            // Уменьшить уровень защиты на 1
            shieldLevel--;
            // Уничтожить врага
            Destroy(go);
        } else if (go.tag == "PowerUp") {
            // Если защитное поле столкнулось с бонусом
            AbsorbPowerUp(go);
        } else {
            print("Triggered by non-Enemy: "+go.name);
        }
    }

    public void AbsorbPowerUp( GameObject go ) {
        PowerUp pu = go.GetComponent<PowerUp>();
        switch (pu.type) {

            // Пока оставьте блок инструкции switch пустым.

        }
        pu.AbsorbedBy( this.gameObject );
    }

    public float shieldLevel { ... }
}
```

2. Если теперь щелкнуть на кнопке Play (Играть), можно заметить, что корабль игрока может сталкиваться с бонусами и подбирать их.

   Прежде чем реализовать ответную реакцию на собираемые бонусы, мы должны добавить дополнительные настройки для оружия.

3. Добавьте массив weapons в начало сценария Hero, как показано ниже.

```
public class Hero : MonoBehaviour {

    ...
    public float            projectileSpeed = 40;
    public Weapon[]         weapons;                                    // a

    [Header("Set Dynamically")]
    ...
}
```

a. В следующем разделе мы добавим в _Hero пять дочерних игровых объектов Weapon, которые будут служить точками установки пушек на корабль. Массив weapons будет хранить ссылки на каждую из них.

Расширение вариантов вооружения

Теперь, подготовив необходимый код, мы должны внести пару изменений в объект _Hero.

1. Щелкните на пиктограмме с треугольником рядом с игровым объектом _Hero в иерархии, чтобы раскрыть его.
2. Выберите дочерний объект Weapon. Нажмите комбинацию Command-D (или Ctrl+D в Windows) четыре раза, чтобы создать четыре дополнительные копии объекта Weapon[1]. Копии должны оставаться дочерними по отношению к _Hero.
3. Переименуйте объекты в Weapon_0 ... Weapon_4 и настройте их компоненты Transform, как показано ниже:

```
_Hero              P:[ 0, 0, 0 ]          R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
      Weapon_0     P:[ 0, 2, 0 ]          R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
      Weapon_1     P:[ -2, -1, 0 ]        R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
      Weapon_2     P:[ 2, -1, 0 ]         R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
      Weapon_3     P:[ -1.25, -0.25, 0 ]  R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
      Weapon_4     P:[ 1.25, -0.25, 0 ]   R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
```

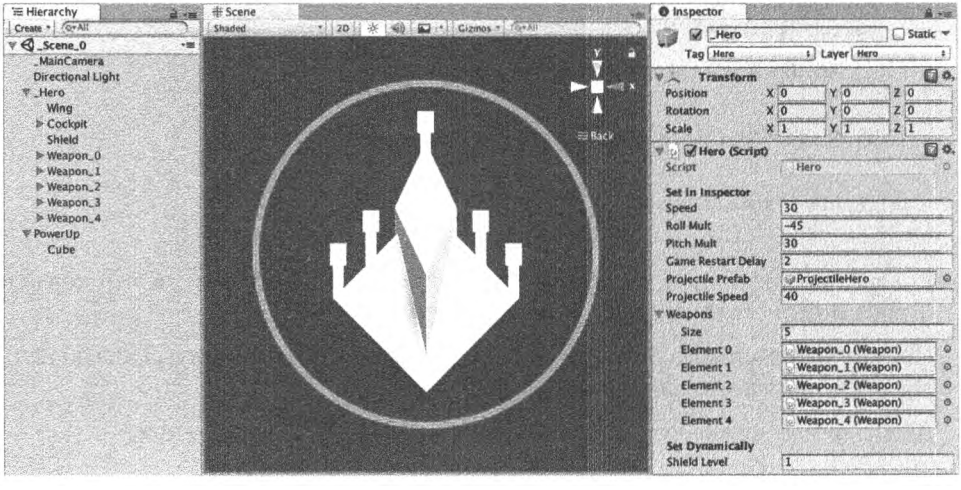


Рис. 31.4. Пять объектов Weapon, вложенных в _Hero, и настроенные слоты поля weapons

[1] Если ввод команды с клавиатуры не дал ожидаемого результата, выберите в меню пункт Edit > Duplicate (Правка > Дублировать) или щелкните правой кнопкой мыши на первоначальном объекте Weapon в иерархии и выберите в контекстном меню пункт Duplicate (Дублировать).

4. Выберите _Hero и в инспекторе раскройте раздел weapons компонента Hero (Script), щелкнув на пиктограмме с треугольником.

5. В поле Size в разделе weapons введите число 5 и поместите объекты Weapon_0 ... Weapon_4 в слоты массива по порядку (перетаскивая их мышью из иерархии или щелкая на пиктограмме с мишенью справа от каждого слота и выбирая соответствующие объекты Weapon_# на вкладке Scene (Сцена)). В результате настройки должны принять вид, как показано на рис. 31.4.

   Теперь реализуем фактическую реакцию на подбираемые бонусы. Для этого добавим некоторые изменения в сценарий Hero.

6. Откройте сценарий Hero и добавьте в конец класса `Hero` методы `GetEmptyWeaponSlot()` и `ClearWeapons()`.

```
public class Hero : MonoBehaviour {
    ...
    public float shieldLevel {
        ...
    }

    Weapon GetEmptyWeaponSlot() {
        for (int i=0; i<weapons.Length; i++) {
            if ( weapons[i].type == WeaponType.none ) {
                return( weapons[i] );
            }
        }
        return( null );
    }

    void ClearWeapons() {
        foreach (Weapon w in weapons) {
            w.SetType(WeaponType.none);
        }
    }
}
```

7. Заполните блок инструкции `switch` в методе `AbsorbPowerUp()` (который мы оставили пустым) следующим кодом.

```
public class Hero : MonoBehaviour {
    ...
    public void AbsorbPowerUp( GameObject go ) {
        PowerUp pu = go.GetComponent<PowerUp>();
        switch (pu.type) {
            case WeaponType.shield:                                          // a
                shieldLevel++;
                break;

            default:                                                         // b
                if (pu.type == weapons[0].type) { // Если оружие того же типа // c
                    Weapon w = GetEmptyWeaponSlot();
                    if (w != null) {
                        // Установить в pu.type
```

```
                    w.SetType(pu.type);
                }
            } else { // Если оружие другого типа                    // d
                ClearWeapons();
                weapons[0].SetType(pu.type);
            }
            break;
        }
        pu.AbsorbedBy( this.gameObject );
    }
    ...
}
```

a. Если бонус имеет тип `WeaponType.shield`, он увеличивает уровень защитного поля на 1.

b. Бонус любого другого типа `WeaponType` — это оружие, и обрабатывается в ветке `default`.

c. Если тип оружия бонуса совпадает с текущим типом оружия корабля, выполняется поиск пустой точки подвески и ей присваивается данный тип оружия. Если все пять точек подвески уже заняты, ничего не происходит.

d. Если тип оружия бонуса отличается от текущего типа оружия корабля, тогда очищаются все точки подвески в Weapon_0 и записывается тип оружия из подобранного бонуса.

8. Для проверки работы этого кода выберите PowerUp в иерархии и в инспекторе, внутри компонента PowerUp (Script), выберите в раскрывающемся списке type (в разделе с заголовком «Set Dynamically») пункт Spread. Обычно тип устанавливается динамически, но для нужд тестирования его можно устанавливать вручную.

9. Щелкните на кнопке Play (Играть), и с самого начала корабль будет вооружен пятью бластерами. Когда вы подберете бонус, произойдет замена бластеров одной веерной пушкой. Буква на падающем бонусе PowerUp сейчас отображается неправильно, потому что мы вручную установили тип. Таким же способом можно проверить работу бонуса с типом `shield` и убедиться, что уровень защитного поля возрастает, если подобрать бонус.

Состояние гонки

Теперь мы нарушим кое-что, чтобы уточнить один момент. Потерпите немного, это действительно важно, и в будущем вы наверняка столкнетесь с чем-то подобным.

1. Добавьте следующие строки, выделенные жирным, в метод `Awake()` сценария Hero:

```
public class Hero : MonoBehaviour {
    ...
    void Awake() {
        S = this; // Установить ссылку на объект-одиночку
//        fireDelegate += TempFire;
```

```
            // Очистить массив weapons и начать игру с 1 бластером
            ClearWeapons();
            weapons[0].SetType(WeaponType.blaster);
        }
        ...
}
```

2. Щелкните на кнопке Play (Играть), и вы увидите примерно такое сообщение об ошибке:

```
NullReferenceException: Object reference not set to an instance of an object
Weapon.SetType (WeaponType wt) (at Assets/__Scripts/Weapon.cs:82)
Hero.Awake () (at Assets/__Scripts/Hero.cs:36)
```

   В нем говорится, что предпринята попытка использовать пустую ссылку, и эта ошибка встретилась в методе `SetType()`, в файле *Weapon.cs* (в моей реализации — строка 82, но у вас номер строки может отличаться), в следующей строке:

```
collarRend.material.color = def.color;
```

   В нем также сообщается, что эта строка была достигнута в результате вызова `SetType()` из метода `Awake()` в *Hero.cs* (строка 36 у меня, но у вас может отличаться) в следующей строке:

```
weapons[0].SetType(WeaponType.blaster);
```

   Если раскручивать сообщение в обратном порядке, получается, что метод `Awake()` в сценарии Hero вызвал метод `SetType()` 0-го элемента в массиве `weapons`. Метод `SetType()`в сценарии Weapon попытался записать цвет в `collarRend`, но в переменной `collarRend` оказалось значение `null`, из-за чего возникла ошибка обращения по пустой ссылке.

   Взгляните на метод `Start()` в сценарии Weapon. Именно здесь записывается ссылка в `collarRend`. Но метод `Awake()` в Hero всегда вызывается раньше метода `Start()` в Weapon, в результате произошла попытка прочитать значение из переменной `collarRend` до того, как оно было туда записано! Чтобы исправить эту проблему, нужно предпринять дополнительные шаги.

3. В сценарии Hero измените имя метода `Awake()` на `Start()`[1].

   Если теперь щелкнуть на кнопке Play (Играть), может показаться, что это изменение устранило проблему, но на самом деле это не так. Проблема могла исчезнуть, потому что может получиться так, что `Weapon.Start()` выполнится

[1] Напомню, что метод `Awake()` вызывается в момент создания экземпляра игрового объекта, а метод `Start()` — непосредственно перед первым вызовом `Update()` этого игрового объекта. Два объекта, являющиеся частью сцены, создаются сразу после запуска игры, поэтому методы `Awake()` обоих объектов будут вызваны до метода `Start()` любого из них; кроме того, заранее неизвестно, в каком порядке произойдут вызовы методов `Start()`. Настройка порядка выполнения сценариев (которую мы выполним в шаге 4) исправляет эту неопределенность.

раньше `Hero.Start()`, однако также возможно, что первым выполнится `Hero.Start()`. Нам нужна уверенность.

4. Откройте в инспекторе диалог Script Execution Order (Порядок выполнения сценариев), выбрав в меню пункт Edit > Project Settings > Script Execution Order (Правка > Параметры проекта > Порядок выполнения сценариев).

   a. Щелкните на кнопке +, обведенной окружностью на рис. 31.5 слева, и в открывшемся списке выберите сценарий Weapon. В результате в таблицу будет добавлена строка с именем сценария Weapon и значением 100. Число 100 определяет *порядок выполнения* сценария Weapon относительно других сценариев, которые выполняются в момент времени по умолчанию Default Time. Если число выше (как сейчас), все сценарии Weapon выполнятся после других сценариев, то есть вызов метода `Weapon.Start()` произойдет *после* вызова `Hero.Start()` или в любом другом сценарии.

   b. Щелкните на кнопке Apply (Применить), чтобы зафиксировать порядок выполнения, и затем щелкните на кнопке Play (Играть).

   c. Теперь ошибка обращения по пустой ссылке будет возникать при любой попытке запустить игру. Значит, сценарий Weapon должен выполняться не последним, а первым.

   d. Снова откройте в инспекторе диалог Script Execution Order (Порядок выполнения сценариев) и, ухватив мышью пиктограмму с двумя горизонтальными полосками (находится под указателем на рис. 31.5 справа) в строке со сценарием Weapon, перетащите эту строку вверх, поместив ее *выше* раздела Default Time. Число в строке при этом изменится со 100 на –100 (как показано на рис. 31.5 справа). Теперь `Weapon.Start()` будет вызываться раньше любых других методов `Start()`.

   e. Снова щелкните на кнопке Apply (Применить). На этот раз ошибка действительно исчезнет.

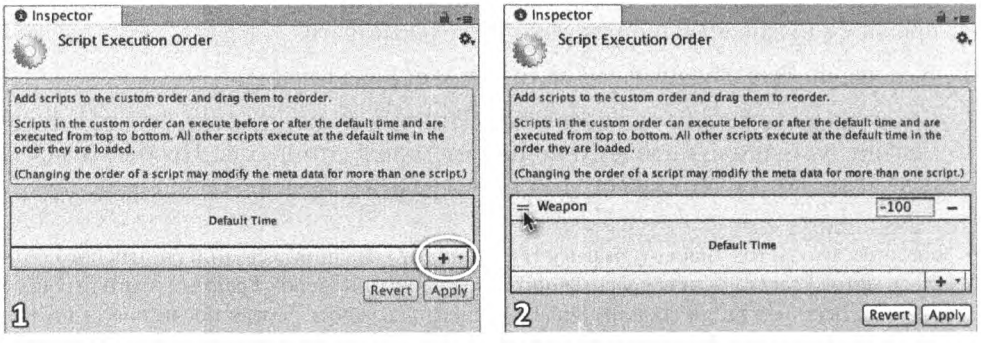


Рис. 31.5. Манипуляции со сценарием Weapon в диалоге настройки порядка выполнения сценариев в инспекторе

Состояние гонки и порядок выполнения сценариев — важные аспекты, чтобы помнить о них.

Сбрасывание бонусов с вражеских кораблей

Вернемся к нашим бонусам. Дадим вражеским кораблям возможность оставлять случайные бонусы после их разрушения. Бонусы послужат дополнительным стимулом к уничтожению врагов и дадут игроку возможность совершенствовать свой корабль.

Разрушаясь, вражеские корабли будут извещать объект-одиночку `Main`, а тот, в свою очередь, создавать новые бонусы. Такое решение может показаться окольным, но обычно лучше ограничивать количество классов, которые могут создавать новые экземпляры игровых объектов в сцене. Чем меньше сценариев отвечает за активные действия (такие, как создание новых экземпляров), тем проще эти действия поддаются отладке.

1. Начнем с создания новых экземпляров PowerUp в классе `Main`. Добавьте в сценарий Main следующий код, выделенный жирным:

```
public class Main : MonoBehaviour {
    ...
    public WeaponDefinition[]    weaponDefinitions;
    public GameObject            prefabPowerUp;                          // a
    public WeaponType[]          powerUpFrequency = new WeaponType[] {   // b
                                     WeaponType.blaster, WeaponType.blaster,
                                     WeaponType.spread,  WeaponType.shield };

    private BoundsCheck      bndCheck;

    public void ShipDestroyed( Enemy e ) {                               // c
        // Сгенерировать бонус с заданной вероятностью
        if (Random.value <= e.powerUpDropChance) {                       // d
            // Выбрать тип бонуса
            // Выбрать один из элементов в powerUpFrequency
            int ndx = Random.Range(0,powerUpFrequency.Length);           // e
            WeaponType puType = powerUpFrequency[ndx];

            // Создать экземпляр PowerUp
            GameObject go = Instantiate( prefabPowerUp ) as GameObject;
            PowerUp pu = go.GetComponent<PowerUp>();
            // Установить соответствующий тип WeaponType
            pu.SetType( puType );                                        // f

            // Поместить в место, где находился разрушенный корабль
            pu.transform.position = e.transform.position;
        }
    }

    void Awake() { ... }
    ...
}
```

a. Это поле хранит шаблон для всех бонусов.

b. Массив `powerUpFrequency` типов бонусов `WeaponType` определяет частоту создания бонусов каждого типа. По умолчанию он содержит два бластера, одну веерную пушку и один банк энергии для защитного поля, поэтому бонусы с бластерами будут появляться в два раза чаще, чем другие.

c. Метод `ShipDestroyed()` будет вызываться экземпляром вражеского корабля в момент его разрушения. Иногда он будет создавать бонус на месте разрушенного корабля.

d. Вражеские корабли всех типов будут иметь поле `powerUpDropChance`, хранящее число между 0 и 1. `Random.value` — это свойство, генерирующее случайное число типа `float` между 0 (включительно) и 1 (включительно). (Свойство `Random.value` может вернуть любое из граничных значений, и 0, и 1.) Если полученное случайное число меньше или равно `powerUpDropChance`, создается новый экземпляр `PowerUp`. Поле `powerUpDropChance` будет объявлено в классе `Enemy`, поэтому вражеские корабли разных типов могут иметь более высокую или низкую вероятность оставить после себя бонус (например, корабли типа `Enemy_0` могли бы оставлять бонусы очень редко, а корабли `Enemy_4` — всегда). Сейчас эта строка подчеркнута красной волнистой линией в редакторе MonoDevelop, потому что поле `powerUpDropChance` еще не добавлено в класс `Enemy`.

e. В этой строке используется массив `powerUpFrequency`. Когда функция `Random.Range()` вызывается с двумя целочисленными значениями, она выбирает число между первым (включительно) и вторым (исключительно): например, `Random.Range(0,4)` может вернуть целое число 0, 1, 2 или 3. Ее очень удобно использовать для выбора случайного элемента массива, как показано в этой строке.

f. После выбора типа бонуса вызывается метод `SetType()` созданного экземпляра `PowerUp`, а тот, в свою очередь, устанавливает цвет, поле `_type` и отображает соответствующую букву в TextMesh.

2. Добавьте в сценарий `Enemy` следующий код, выделенный жирным:

```
public class Enemy : MonoBehaviour {
    ...
    public float      showDamageDuration = 0.1f; // Длительность эффекта
                                                 //   попадания в секундах
    public float      powerUpDropChance = 1f;    // Вероятность сбросить бонус // a

    [Header("These fields are set dynamically")]
    ...
    void OnCollisionEnter( Collision coll ) {
        GameObject otherGO = coll.gameObject;
        switch (otherGO.tag) {
            case "ProjectileHero":
                ...
                // Поразить вражеский корабль
```

```
            ...
            if (health <= 0) {
                // Сообщить объекту-одиночке Main об уничтожении       // b
                if (!notifiedOfDestruction){
                    Main.S.ShipDestroyed( this );
                }
                notifiedOfDestruction = true;
                // Уничтожить этот вражеский корабль
                Destroy(this.gameObject);
            }
            ...
            break;
        ...
    }
  }
}
```

a. `powerUpDropChance` определяет для вражеского корабля вероятность оставить бонус после его разрушения. Со значением 0 в этом поле корабль никогда не будет оставлять бонусы, а со значением 1 бонусы будут оставляться всегда.

b. Непосредственно перед уничтожением вражеского корабля известить объект-одиночку `Main` вызовом `ShipDestroyed()`. Уведомление производится только один раз, это гарантируется логическим полем `notifiedOfDestruction`.

3. Чтобы этот код смог выполняться без ошибок, нужно выбрать _ MainCamera в иерархии и присвоить полю prefabPowerUp компонента Main (Script) шаблон PowerUp из папки _Prefabs в панели Project (Проект).

4. Выберите экземпляр PowerUp в иерархии и удалите его (он больше не нужен, потому что у нас уже есть шаблон PowerUp в панели Project (Проект)).

5. Поле powerUpFrequency компонента Main (Script) главной камеры _MainCamera должно быть уже установлено в инспекторе, тем не менее на всякий случай проверьте настройки, чтобы они соответствовали изображенным на рис. 31.6.

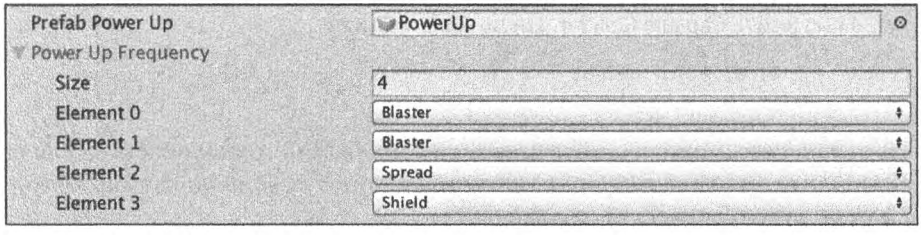


Рис. 31.6. Настройки prefabPowerUp и powerUpFrequency компонента Main (Script) главной камеры _MainCamera

6. Теперь запустите сцену и попробуйте уничтожить несколько вражеских кораблей. Они должны оставлять бонусы, улучшающие ваш корабль!

Поиграв какое-то время, вы должны заметить, что бонусы с бластером [B] появляются чаще бонусов с веерной пушкой [S] или банками энергии для защитного поля [O]. Это объясняется наличием двух элементов с типом `WeaponType.blaster` в `powerUpFrequency` и только одного элемента с типом `WeaponType.spread` и `WeaponType.shield`. Меняя количество элементов каждого типа в `powerUpFrequency`, можно определять вероятность выбора каждого из них. Аналогично можно влиять на вероятность появления вражеских кораблей разных типов, добавив в массив `prefabEnemies` типы вражеских кораблей, которые должны появляться чаще других.

Enemy_4 — самый стойкий враг

Как своеобразный босс, корабль типа Enemy_4 имеет более высокую стойкость и может разрушаться по частям (а не весь сразу). Кроме того, он всегда остается на экране, перемещаясь с места на место, пока игрок не разрушит его полностью.

Изменение коллайдера

Прежде чем перейти к программной реализации, изменим некоторые настройки коллайдера в шаблоне Enemy_4.

1. Перетащите экземпляр Enemy_4 в иерархию и поместите его в стороне от других игровых объектов в сцене (по умолчанию он помещается в позицию P:[20, 10, 0], как было определено ранее).
2. Щелкните на пиктограмме с треугольником рядом с именем Enemy_4 в иерархии и выберите вложенный объект Fuselage.
3. Удалите компонент Sphere Collider из объекта Fuselage, щелкнув в инспекторе на кнопке с шестеренкой в правом верхнем углу раздела Sphere Collider и выбрав пункт Remove Component (Удалить компонент).
4. Добавьте в Fuselage компонент Capsule Collider, выбрав в главном меню пункт Component > Physics > Capsule Collider (Компонент > Физика > Коллайдер-капсула). Настройте Capsule Collider, как показано ниже:

```
Center    [ 0, 0, 0 ]   Height      1
Radius    0.5           Direction   Y-Axis
```

   Не стесняйтесь, поэкспериментируйте с настройками, чтобы увидеть их влияние. Как видите, коллайдер в форме капсулы намного лучше аппроксимирует форму Fuselage, чем сферический коллайдер.

5. Выберите в иерархии объект WingL, вложенный в Enemy_4, и также замените Sphere Collider на Capsule Collider. В поле Direction этого коллайдера выберите значение X-Axis.

```
Center    [ 0, 0, 0 ]   Height      1
Radius    0.5           Direction   X-Axis
```

Настройка Direction коллайдера Capsule Collider определяет ось координат, вдоль которой будет вытянута капсула. Эти настройки определяются в локальной системе координат, то есть высота Height, равная 1, будет умножена на масштаб 5 вдоль оси X. Радиус 0,5 умножается на наибольший из масштабов по осям Y или Z, то есть фактический радиус закруглений на концах капсулы будет равен 0,5, потому что масштаб по оси Y равен 1. Как можно заметить, капсула не очень хорошо аппроксимирует форму крыла, но намного лучше, чем сфера.

6. Выберите объект WingR, замените его коллайдер на Capsule Collider и настройте его так же, как коллайдер объекта WingL.
7. Выберите Enemy_4 в иерархии и добавьте компонент BoundsCheck (Script), щелкнув в инспекторе на кнопке Add Component (Добавить компонент) и выбрав пункт Scripts > BoundsCheck (Сценарии > BoundsCheck).
8. В компоненте BoundsCheck (Script) введите в поле radius число 3.5 и снимите флажок keepOnScreen.
9. Щелкните на кнопке Apply (Применить) справа от слова Prefab в верхней части панели Inspector (Инспектор). В результате изменения, произведенные в этом экземпляре Enemy_4, будут перенесены в шаблон Enemy_4 в панели Project (Проект).
10. Убедитесь, что изменения попали в шаблон. Для этого перетащите второй экземпляр шаблона Enemy_4 в панель Hierarchy (Иерархия) и проверьте настройки коллайдеров — они должны в точности совпадать с настройками в первом экземпляре, в который мы вносили изменения.
11. Удалите оба экземпляра Enemy_4 из панели Hierarchy (Иерархия).
12. Сохраните сцену! Надеюсь, вам не нужно напоминать об этом?

Если захотите, можете повторить тот же трюк с компонентом Capsule Collider для Enemy_3.

Перемещение Enemy_4

Enemy_4 должен появиться, как обычно, в верхней части, случайно выбрать точку на экране и переместиться к ней за некоторое время с использованием линейной интерполяции. Добравшись до места, он должен выбрать другую точку и переместиться к ней, и т. д.

1. Откройте сценарий Enemy_4 и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Enemy_4 создается за верхней границей, выбирает случайную точку на экране
///   и перемещается к ней. Добравшись до места, выбирает другую случайную точку
```

```
///   и продолжает двигаться, пока игрок не уничтожит его.
/// </summary>
public class Enemy_4 : Enemy {

    private Vector3         p0, p1;       // Две точки для интерполяции
    private float           timeStart;    // Время создания этого корабля
    private float           duration = 4; // Продолжительность перемещения

    void Start () {
        // Начальная позиция уже выбрана в Main.SpawnEnemy(),
        //   поэтому запишем ее как начальные значения в p0 и p1
        p0 = p1 = pos;                                                    // a
        InitMovement();
    }

    void InitMovement() {                                                 // b
        p0 = p1; // Переписать p1 в p0
        // Выбрать новую точку p1 на экране
        float widMinRad = bndCheck.camWidth - bndCheck.radius;
        float hgtMinRad = bndCheck.camHeight - bndCheck.radius;
        p1.x = Random.Range( -widMinRad, widMinRad );
        p1.y = Random.Range( -hgtMinRad, hgtMinRad );

        // Сбросить время
        timeStart = Time.time;
    }

    public override void Move () { // c
        // Этот метод переопределяет Enemy.Move() и реализует
        //   линейную интерполяцию
        float u = (Time.time-timeStart)/duration;

        if (u>=1) {
            InitMovement();
            u=0;
        }

        u = 1 - Mathf.Pow( 1-u, 2 ); // Применить плавное замедление      // d
        pos = (1-u)*p0 + u*p1;       // Простая линейная интерполяция     // e
    }
}
```

a. Класс `Enemy_4` интерполирует перемещение из точки `p0` в точку `p1` (то есть реализует плавное перемещение между точками). Метод `Main.SpawnEnemy()`, создавая этот вражеский корабль, помещает его в позицию над верхним краем экрана, координаты которой здесь записываются в оба поля, `p0` и `p1`. Затем вызывается `InitMovement()`.

b. Метод `InitMovement()` сначала переписывает текущие координаты из `p1` в `p0` (потому что к моменту вызова `InitMovement()` объект `Enemy_4` должен находиться в точке `p1`). Далее выбирается новая точка `p1`, при этом используется информация из компонента `BoundsCheck`, чтобы гарантировать, что новая точка окажется в пределах экрана.

c. Этот метод `Move()` полностью переопределяет унаследованный метод `Enemy.Move()`. Он интерполирует перемещение из точки `p0` в точку `p1` за `duration` секунд (по умолчанию 4 секунды). В течение этого времени значение `u` увеличивается от 0 до 1 и, когда `u` оказывается >= 1, вызывается `InitMovement()`, чтобы начать новый цикл интерполяции.

d. Эта строка применяет сглаживание к значению `u`, вследствие которого корабль движется с непостоянной скоростью. В данном случае корабль начинает движение с высокой скоростью и замедляется по мере приближения к точке `p1`.

e. Эта строка выполняет простую линейную интерполяцию между `p0` и `p1`.

Узнать больше об интерполяции и функциях сглаживания вы сможете в разделе «Интерполяция» в приложении Б «Полезные идеи».

2. Выберите _MainCamera в иерархии. Перетащите шаблон Enemy_4 из папки _Prefabs в панели Project (Проект) в элемент Element 0 массива prefabEnemies компонента Main (Script) в инспекторе.

3. Щелкните на кнопке Play (Играть). Теперь экземпляры Enemy_4 должны оставаться на экране до их уничтожения игроком. Однако прямо сейчас они уничтожаются так же легко, как другие вражеские корабли.

Деление Enemy_4 на несколько частей

Теперь разделим корабль Enemy_4 на четыре разные части и защитим центральный кокпит Cockpit другими частями.

1. Откройте сценарий Enemy_4 и добавьте новый сериализуемый класс с именем Part в начало файла *Enemy_4.cs*. Также добавьте в класс `Enemy_4` массив типа `Part[]` с именем `parts` и выделенный жирным код в конец метода `Start()` класса `Enemy_4`.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Part -- еще один сериализуемый класс подобно WeaponDefinition,
///   предназначенный для хранения данных
/// </summary>
[System.Serializable]
public class Part {
    // Значения этих трех полей должны определяться в инспекторе
    public string        name;         // Имя этой части
    public float         health;       // Степень стойкости этой части
    public string[]      protectedBy; // Другие части, защищающие эту

    // Эти два поля инициализируются автоматически в Start().
    // Кэширование, как здесь, ускоряет получение необходимых данных
    [HideInInspector] // Не позволяет следующему полю появиться в инспекторе
    public GameObject    go; // Игровой объект этой части
```

```
    [HideInInspector]
    public Material      mat; // Материал для отображения повреждений
}

...

public class Enemy_4 : Enemy {
    [Header("Set in Inspector: Enemy_4")]                                    // a
    public Part[]           parts; // Массив частей, составляющих корабль

    private Vector3         p0, p1;       // Две точки для интерполяции
    private float           timeStart;    // Время создания этого корабля
    private float           duration = 4; // Продолжительность перемещения

    void Start () {
        // Начальная позиция уже выбрана в Main.SpawnEnemy(),
        //   поэтому запишем ее как начальные значения в p0 и p1
        p0 = p1 = pos;                                                       // a

        InitMovement();

        // Записать в кэш игровой объект и материал каждой части в parts
        Transform t;
        foreach (Part prt in parts) {
            t = transform.Find(prt.name);
            if (t != null) {
                prt.go = t.gameObject;
                prt.mat = prt.go.GetComponent<Renderer>().material;
            }
        }
    }
    ...
}
```

a. В инспекторе все общедоступные поля класса `Enemy` перечисляются выше полей класса `Enemy_4`. Добавление «: Enemy_4» в конец заголовка делает это деление более отчетливым и позволяет с первого взгляда определить, какое поле какому классу принадлежит (см. рис. 31.7).

Сериализуемый[1] класс `Part` хранит информацию о каждой из четырех частей, составляющих корабль Enemy_4: Cockpit, Fuselage, WingL и WingR.

2. Вернитесь в Unity и выполните следующие шаги:

   a. Выберите шаблон Enemy_4 в панели Project (Проект).

   b. В инспекторе раскройте массив parts в компоненте Enemy_4 (Script), щелкнув на пиктограмме с треугольником.

   c. Измените настройки, как показано на рис. 31.7. Будьте внимательны, постарайтесь не допускать опечаток в именах.

[1] Напомню, что объявление класса *сериализуемым* делает его поля доступными для просмотра и изменения в инспекторе Unity. Простые классы почти наверняка будут отображаться в инспекторе. Но если класс слишком сложный, инспектор Unity не сможет показать его.

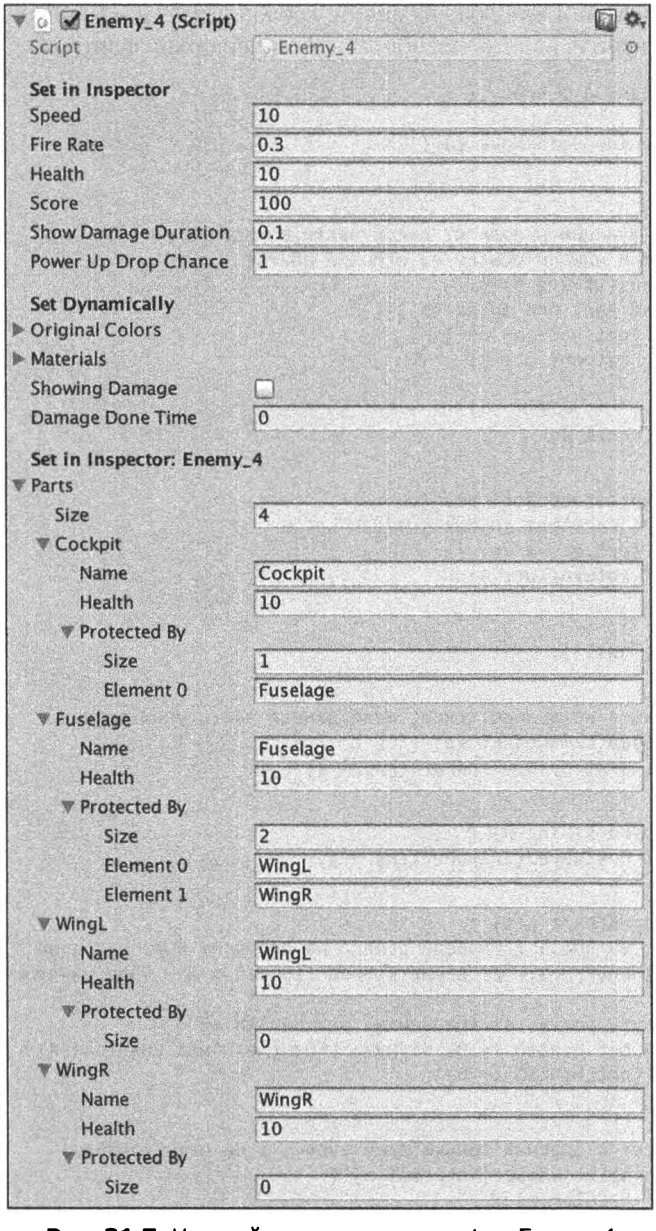


Рис. 31.7. Настройки для массива parts в Enemy_4

Как видно на рис. 31.7, каждая часть имеет стойкость (Health) 10, и все они образуют иерархическое дерево защиты. Кокпит (Cockpit) защищен фюзеляжем (Fuselage), а фюзеляж защищен обоими крыльями, левым (WingL) и правым (WingR). Обязательно сохраните сцену!

3. Вернитесь в MonoDevelop и добавьте следующие методы в конец класса Enemy_4, чтобы обеспечить работоспособность этой иерархии защиты:

```
public class Enemy_4 : Enemy {
    ...
    public override void Move () {
        ...
    }

    // Эти две функции выполняют поиск части в массиве parts п
    //   по имени или по ссылке на игровой объект
    Part FindPart(string n) {                                          // a
        foreach( Part prt in parts ) {
            if (prt.name == n) {
                return( prt );
            }
        }
        return( null );
    }

    Part FindPart(GameObject go) {                                     // b
        foreach( Part prt in parts ) {
            if (prt.go == go) {
                return( prt );
            }
        }
        return( null );
    }

    // Эти функции возвращают true, если данная часть уничтожена
    bool Destroyed(GameObject go) { // c
        return( Destroyed( FindPart(go) ) );
    }

    bool Destroyed(string n) {
        return( Destroyed( FindPart(n) ) );
    }

    bool Destroyed(Part prt) {
        if (prt == null) { // Если ссылка на часть не была передана
            return(true);  // Вернуть true (то есть: да, была уничтожена)
        }
        // Вернуть результат сравнения: prt.health <= 0
        // Если prt.health <= 0, вернуть true (да, была уничтожена)
        return (prt.health <= 0);
    }

    // Окрашивает в красный только одну часть, а не весь корабль.
    void ShowLocalizedDamage(Material m) {                             // d
        m.color = Color.red;
        damageDoneTime = Time.time + showDamageDuration;
        showingDamage = true;
    }

    // Переопределяет метод OnCollisionEnter из сценария Enemy.cs.
    void OnCollisionEnter( Collision coll ) {                          // e
        GameObject other = coll.gameObject;
        switch (other.tag) {
```

```
        case "ProjectileHero":
            Projectile p = other.GetComponent<Projectile>();
            // Если корабль за границами экрана, не повреждать его.
            if ( !bndCheck.isOnScreen ) {
                Destroy( other );
                break;
            }

            // Поразить вражеский корабль
            GameObject goHit = coll.contacts[0].thisCollider.gameObject; // f
            Part prtHit = FindPart(goHit);
            if (prtHit == null) { // If prtHit wasn't found...          // g
                goHit = coll.contacts[0].otherCollider.gameObject;
                prtHit = FindPart(goHit);
            }
            // Проверить, защищена ли еще эта часть корабля
            if (prtHit.protectedBy != null) {                          // h
                foreach( string s in prtHit.protectedBy ) {
                    // Если хотя бы одна из защищающих частей еще
                    // не разрушена...
                    if (!Destroyed(s)) {
                        // ...не наносить повреждений этой части
                        Destroy(other); // Уничтожить снаряд ProjectileHero
                        return;         // выйти, не повреждая Enemy_4
                    }
                }
            }

            // Эта часть не защищена, нанести ей повреждение
            // Получить разрушающую силу из Projectile.type и Main.WEAP_DICT
            prtHit.health -= Main.GetWeaponDefinition( p.type ).damageOnHit;
            // Показать эффект попадания в часть
            ShowLocalizedDamage(prtHit.mat);
            if (prtHit.health <= 0) {                                  // i
                // Вместо разрушения всего корабля
                //   деактивировать уничтоженную часть
                prtHit.go.SetActive(false);
            }
            // Проверить, был ли корабль полностью разрушен
            bool allDestroyed = true; // Предположить, что разрушен
            foreach( Part prt in parts ) {
                if (!Destroyed(prt)) { // Если какая-то часть еще существует...
                    allDestroyed = false; // ...записать false в allDestroyed
                    break;                // и прервать цикл foreach
                }
            }
            if (allDestroyed) { // Если корабль разрушен полностью...  // j
                // ...уведомить объект-одиночку Main, что этот корабль разрушен
                Main.S.ShipDestroyed( this );
                // Уничтожить этот объект Enemy
                Destroy(this.gameObject);
            }
            Destroy(other); // Уничтожить снаряд ProjectileHero
            break;
    }
  }
}
```

a. Методы `FindPart()` в строках `// a` и `// b` — это перегруженные версии с одинаковыми именами, но с разными параметрами (один принимает строку, а другой — ссылку на игровой объект `GameObject`). Исходя из типа аргумента, производится вызов соответствующей перегруженной версии функции `FindPart()`. Обе версии `FindPart()` выполняют поиск в массиве `parts` части корабля по строковому имени или по ссылке на игровой объект.

b. Перегруженная версия `FindPart()`, принимающая ссылку на игровой объект `GameObject`. Раньше вы уже пользовались перегруженной функцией `Random.range()`, которая действует по-разному, получая аргументы типа `float` или `int`.

c. Три перегруженные версии метода `Destroyed()` проверяют, была ли разрушена указанная часть или она еще сохранила какую-то долю стойкости.

d. `ShowLocalizedDamage()` — более специализированная версия унаследованного метода `Enemy.ShowDamage()`. Она окрашивает в красный цвет не весь корабль, а только одну его часть.

e. Этот метод `OnCollisionEnter()` полностью переопределяет унаследованный метод `Enemy.OnCollisionEnter()`. Из-за особенностей объявления обычных для Unity функций в классе `MonoBehaviour`, таких как `OnCollisionEnter()`, ключевое слово `override` в данном случае не требуется.

f. Эта функция пытается найти игровой объект, в который попал снаряд. Описатель столкновения `coll` типа `Collision` включает поле `contacts[]` — массив точек контакта `ContactPoint`. Поскольку речь в данном случае идет о столкновении, у нас гарантированно будет иметься хотя бы одна точка контакта `ContactPoint` (то есть `contacts[0]`). Каждая точка `ContactPoint` имеет поле с именем `thisCollider`, представляющее коллайдер части корабля `Enemy_4`, которая была поражена.

g. Если искомая часть корабля `prtHit` не найдена (и, соответственно, `prtHit == null`), значит, `thisCollider` в `contacts[0]` ссылается на снаряд `ProjectileHero` (что случается очень редко), попавший в корабль, а не на пораженную часть корабля. В этом случае проверяется `contacts[0].otherCollider`.

h. Если эта часть корабля все еще защищена другой частью, не разрушенной до конца, применить разрушение к защищающей части.

i. Если степень стойкости части достигла 0, она деактивируется, благодаря чему перестает отображаться на экране и взаимодействовать с другими игровыми объектами.

j. Если корабль разрушен полностью, уведомить `Main.S.ShipDestroyed()`, как это сделал бы сценарий `Enemy` (если бы мы не переопределили метод `OnCollisionEnter()`).

4. Запустите сцену. В конце концов на экране появится множество кораблей `Enemy_4`, каждый из которых имеет два крыла, защищающих фюзеляж, кото-

рый, в свою очередь, защищает кокпит. Если захотите получить больше шансов противостоять им, уменьшите значение поля enemySpawnPerSecond компонента Main (Script) в _MainCamera, чтобы увеличить интервал времени между созданием новых кораблей Enemy_4 (правда, при этом увеличится время ожидания появления первого такого корабля).

5. Мы еще больше приблизились к игре, в которую можно играть! Теперь нам нужно настроить массив prefabEnemies компонента Main (Script) в _MainCamera, чтобы организовать создание разных кораблей с разумной частотой.
   a. Выберите _MainCamera в иерархии.
   b. В инспекторе введите число 10 в поле Size массива prefabEnemies в компоненте Main (Script).
   c. В поля Elements 0, 1 и 2 перетащите шаблон Enemy_0 (из папки _Prefabs в панели Project (Проект)).
   d. В поля Elements 3 и 4 перетащите шаблон Enemy_1.
   e. В поля Elements 5 и 6 перетащите шаблон Enemy_2.
   f. В поля Elements 7 и 8 перетащите шаблон Enemy_3.
   g. В поле Element 9 перетащите шаблон Enemy_4.

   Согласно таким настройкам корабли Enemy_0 будут появляться чаще, а корабли Enemy_4 — реже.
6. Настройте параметр powerUpDropChance для каждого типа вражеского корабля.
   a. Выберите шаблон Enemy_0 в папке _Prefabs в панели Project (Проект) и в инспекторе введите в поле powerUpDropChance компонента Enemy (Script) число 0,25 (чтобы в 25 % случаев корабли типа Enemy_0 оставляли бонусы после их разрушения).
   b. Для шаблона Enemy_1 введите в поле powerUpDropChance число 0,5.
   c. Для шаблона Enemy_2 введите в поле powerUpDropChance число 0,5.
   d. Для шаблона Enemy_3 введите в поле powerUpDropChance число 0,75.
   e. Для шаблона Enemy_4 введите в поле powerUpDropChance число 1.
7. Сохраните сцену и щелкните на кнопке Play (Играть), чтобы опробовать игру!

Скроллинг фона с изображением звездного поля

Закончив реализацию основной логики игры, можно сделать еще шаг и немного улучшить визуальное восприятие игры: создать двуслойный фон с изображением звездного поля, чтобы увеличить сходство внешнего вида сцены с межзвездным пространством.

1. Создайте в иерархии новый квадрат (GameObject > 3D Object > Quad (Игровой объект > 3D объект > Квадрат)) с именем StarfieldBG.

   ```
   StarfieldBG (Quad) P:[ 0, 0, 10 ] R:[ 0, 0, 0 ] S:[ 80, 80, 1 ]
   ```

   С этими настройкам квадрат StarfieldBG окажется в центре поля зрения камеры и заполнит его целиком.

2. Создайте новый материал с именем Mat_Starfield и выберите для него шейдер ProtoTools > UnlitAlpha. Выберите для материала Mat_Starfield двумерную текстуру Space, которая находится в папке _Materials и была импортирована вами в начале этого учебного примера.

3. Перетащите Mat_Starfield на StarfieldBG, и за кораблем _Hero должно появиться звездное поле.

4. Выберите Mat_Starfield в панели Project (Проект) и создайте его копию (Command-D в Mac или Ctrl+D на PC). Дайте новому материалу имя Mat_Starfield_Transparent. В качестве текстуры для этого материала выберите Space_Transparent (в папке _Materials).

5. Выберите объект StarfieldBG в иерархии и создайте его копию с именем StarfieldFG_0. Перетащите материал Mat_Starfield_Transparent на StarfieldFG_0 и настройте компонент Transform, как показано ниже:

   ```
   StarfieldFG_0 P:[ 0, 0, 5 ] R:[ 0, 0, 0 ] S:[ 160, 160, 1 ]
   ```

   Если попробовать подвигать StarfieldFG_0 в сцене, можно заметить, как некоторые звезды на переднем плане перемещаются на фоне звезд на заднем плане, создавая интересный эффект параллакса.

6. Создайте копию Starfield_FG_0 с именем Starfield_FG_1. Нам нужны две копии переднего плана для трюка с прокруткой, который мы собираемся использовать.

7. Создайте новый сценарий на C# с именем Parallax и отредактируйте его в MonoDevelop.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Parallax : MonoBehaviour {
    [Header("Set in Inspector")]
    public GameObject       poi; // Корабль игрока
    public GameObject[]     panels; // Прокручиваемые панели переднего плана
    public float            scrollSpeed = -30f;
    // motionMult определяет степень реакции панелей на перемещение корабля игрока
    public float            motionMult = 0.25f;

    private float           panelHt; // Высота каждой панели
    private float           depth;   // Глубина панелей (то есть pos.z)

    void Start () {
        panelHt = panels[0].transform.localScale.y;
```

```
        depth = panels[0].transform.position.z;

        // Установить панели в начальные позиции
        panels[0].transform.position = new Vector3(0,0,depth);
        panels[1].transform.position = new Vector3(0,panelHt,depth);
    }

    void Update () {
        float tY, tX=0;
        tY = Time.time * scrollSpeed % panelHt + (panelHt*0.5f);

        if (poi != null) {
            tX = -poi.transform.position.x * motionMult;
        }

        // Сместить панель panels[0]
        panels[0].transform.position = new Vector3(tX, tY, depth);
        // Сместить панель panels[1], чтобы создать эффект непрерывности
        //   звездного поля
        if (tY >= 0) {
            panels[1].transform.position = new Vector3(tX, tY-panelHt, depth);
        } else {
            panels[1].transform.position = new Vector3(tX, tY+panelHt, depth);
        }
    }
}
```

8. Сохраните сценарий, вернитесь в Unity, подключите сценарий Parallax к объекту StarfieldBG. Выберите StarfieldBG в иерархии и найдите компонент Parallax (Script) в инспекторе. Затем перетащите объект _Hero из иерархии в поле poi и добавьте StarfieldFG_0 и StarfieldFG_1 в массив panels.
9. Щелкните на кнопке Play (Играть), и вы увидите, как прокручивается звездное поле в ответ на действия игрока.
10. Разумеется, не забудьте сохранить сцену.

Итоги

Это была длинная глава, но она показала множество важных идей, которые, я надеюсь, помогут вам в проектировании ваших игр. На протяжении многих лет я широко использовал линейную интерполяцию и кривые Безье в моих играх и других проектах, чтобы создать эффект гладкого и нелинейного движения объектов. Простая функция сглаживания способна придать объектам интересный эффект энергичного или заторможенного движения и добавить в игру особый шарм.

В следующей главе мы займемся созданием игры совершенно иного вида: карточного пасьянса (фактически пасьянс — моя любимая карточная игра). В ней нам придется научиться читать информацию из XML-файла, чтобы сконструировать целую колоду карт из небольшого количества графических ресурсов, а также использовать формат XML для организации самого игрового поля. В конце вы получите забавную карточную игру.

Следующие шаги

Из опыта, полученного при реализации предыдущих учебных примеров, вы уже знаете, как реализовать многое из того, что будет перечислено в этом разделе. Вот лишь несколько рекомендаций для желающих продолжить работу над этим прототипом.

Настройка переменных

Как вы знаете, настройка чисел имеет огромную важность для бумажных и цифровых игр и оказывает большое влияние на их восприятие. Далее перечислены переменные, изменение значений которых может повлиять на восприятие этой игры:

- **Корабль игрока:** характер движения.
  - Скорректируйте скорость.
  - Измените настройки Sensitivity и Gravity по осям Horizontal и Vertical в диспетчере ввода InputManager.
- **Оружие:** увеличение различий.
  - **Веерная пушка:** могла бы выстреливать сразу пять снарядов вместо трех, но с большей задержкой между выстрелами delayBetweenShots.
  - **Бластер:** мог бы быть более скорострельным (меньшее значение delayBetweenShots), но наносить меньше повреждений каждым снарядом (уменьшить damageOnHit).

Дополнительные элементы

В этом прототипе реализованы пять видов врагов и два вида оружия, но ваши возможности этим не ограничиваются:

- **Оружие:** добавьте дополнительные виды оружия.
  - **Фазер:** выстреливает два снаряда, летящие по синусоиде (напоминает движение Enemy_1).
  - **Лазер:** вместо однократного поражающего эффекта лазер уменьшает стойкость врага в зависимости от времени воздействия на него.
  - **Ракеты:** ракеты могут быть самонаводящимися и иметь очень низкую скорострельность, зато способными следить за вражескими кораблями и поражать их с абсолютной точностью. Также ракеты можно реализовать как отдельный вид оружия с ограниченным боезапасом и запускать их отдельной кнопкой (то есть не клавишей пробела).
  - **Пушка на поворотной турели:** своим действием напоминает бластер и всегда стреляет в сторону ближайшего врага, но каждый снаряд наносит очень небольшие повреждения.
- **Враги:** добавьте дополнительные виды вражеских кораблей. Для этой игры можно создать бесчисленное многообразие врагов.

- Добавьте врагам дополнительные возможности.
  - Добавьте некоторым типам врагов возможность стрелять.
  - Некоторые враги могли бы определять местоположение корабля игрока и следовать в его направлении, действуя подобно ракетам с самонаведением.
- Добавьте уровни сложности.
  - Говоря более конкретно, реализуйте запуск вражеских кораблей не случайно и вразнобой, как в этом прототипе, а волнами. Для этого можно использовать класс `[System.Serializable] Wave`, показанный ниже:

```
[System.Serializable]
public class Wave {
    float        delayBeforeWave=1; // задержка в секундах между волнами
    GameObject[] ships;             // массив кораблей в этой волне
    // Задержать следующую волну до полного уничтожения текущей?
    bool         delayNextWaveUntilThisWaveIsDead=false;
}
```

- Добавьте класс `Level`, содержащий массив `Wave[]`:

```
[System.Serializable]
public class Level {
    Wave[]       waves; // Хранилище для волн
    float        timeLimit=-1; // -1 означает, что время не ограничено
    string       name = ""; // Имя уровня
}
```

Однако у такой организации есть проблема: даже при том, что класс `Level` объявлен сериализуемым, массив `Wave[]` не будет отображаться в инспекторе, потому что Unity не поддерживает отображение вложенных сериализуемых классов. А значит, для определения уровней и волн вам, скорее всего, придется использовать некоторый XML-документ, который потом программно можно прочитать и на его основе создать экземпляры классов `Level` и `Wave`. О работе с XML-документами рассказывается в разделе «XML» приложения Б «Полезные идеи», а практический пример вы найдете в следующем прототипе, в главе 32 «Прототип 4: Prospector Solitaire».

- Добавьте больше структурных элементов игры и графический интерфейс пользователя:
  - Дайте игроку определенное число жизней и начисляйте очки (об этом рассказывалось в обсуждении прототипа *Mission Demolition*).
  - Добавьте возможность настраивать сложность.
  - Запоминайте наивысшие достижения (как рассказывалось при обсуждении прототипов *Apple Picker* и *Mission Demolition*).
  - Создайте экран-заставку, который будет приглашать игроков сыграть и давать возможность выбрать уровень сложности. Здесь также можно выводить таблицу высших достижений.

32 Прототип 4: PROSPECTOR SOLITAIRE

В этой главе вы создадите свою первую карточную игру — цифровую версию популярного пасьянса *Tri-Peaks* (Три вершины). К концу этой главы у вас будет не только действующая игра, но и великолепная основа для будущих карточных игр, которые вы, возможно, захотите создать.

Эта глава рассматривает несколько новых приемов, включая использование конфигурационных XML-файлов и проектирование для мобильных устройств, а также знакомит с инструментами Unity для работы с двумерными спрайтами.

Начало: прототип 4

Описание этого прототипа, так же как описание прототипа 3, начинается с предложения загрузить и импортировать пакет с ресурсами для этой игры. Графические ресурсы, которые мы будем использовать, созданы на основе общедоступного комплекта карт *Vectorized Playing Cards 1.3* Криса Агиляра[1].

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. Чтобы освежить в памяти стандартную процедуру настройки проекта, обращайтесь к приложению А «Стандартная процедура настройки проекта». При создании проекта вам будет предложено выбрать настройки по умолчанию для дву- или трехмерной игры. Выберите настройки для двумерной версии.

- **Имя проекта:** Prospector Solitaire.
- **Загрузите и импортируйте пакет:** Найдите раздел Chapter 32 на сайте http://book.prototools.net. Загрузите этот пакет с настройками для сцены и несколькими папками.

[1] Изображения карт для рисунков и цифровых карточных игр, представленных в этой книге, основаны на наборе Vectorized Playing Cards 1.3, Copyright 2011, Chris Aguilar. Набор доступен на условиях лицензии LGPL 3 (http://www.gnu.org/copyleft/lesser.html), по адресу http://sourceforge.net/projects/vector-cards/.

- **Имя сцены:** (Сцена __Prospector_Scene_0 будет импортирована с пакетом, поэтому вам не придется создавать ее.)
- **Папки проекта:** нет (папки *__Scripts*, *_Prefabs*, *_Sprites* и *Resources* будут созданы в процессе импортирования пакета.)
- **Имена сценариев на C#:** (пока нет)
- **Переименовать:** измените имя главной камеры Main Camera на _MainCamera.

Откройте сцену __Prospector_Scene_0 и проверьте настройки _MainCamera.

```
_MainCamera (Camera)     P:[ 0, 0, -40 ]   R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
        Projection: Orthographic
        Size: 10
```

Обратите внимание, что импортированный пакет содержит версию сценария Utils с дополнительными функциями кроме тех, что мы написали в предыдущей главе.

Настройка сборки

Это первый проект, предусматривающий возможность компиляции для мобильных устройств. Например, я буду использовать настройки для Apple iPad, но вы можете настроить сборку для Android, WebGL или даже автономной игры. Автономная сборка автоматически поддерживается Unity, и вы можете использовать мастер установки Unity, чтобы добавить возможность компиляции для iOS, Android или WebGL. Этот проект рассчитан на соотношение 4:3 сторон экрана устройства iPad в книжной ориентации. Это то же самое соотношение, что отображается в панели Game (Игра) как пункт Standalone (1024x768) (Автономное (1024 × 768)) в меню выбора отношения сторон экрана. Сейчас просто выберите в этом меню пункт 4:3.

Даже при том, что этот проект предусматривает возможность компиляции для мобильного устройства, мы не будем рассматривать в этой книге фактическую процедуру сборки (она сильно отличается в зависимости от вида устройства), на веб-сайте Unity вы сможете найти достаточно информации о сборке для разных платформ. Вот список страниц с информацией о разных платформах для начинающих[1]:

- **Android** – https://docs.unity3d.com/Manual/android-GettingStarted.html
- **iOS** – https://docs.unity3d.com/Manual/iphone-GettingStarted.html
- **WebGL** – https://docs.unity3d.com/Manual/webgl-gettingstarted.html

Теперь приступим к разработке. Если вы собираетесь попробовать вести разработку для мобильной платформы:

[1] Все три ссылки были доступны в июне 2018 года.

1. Щелкните дважды на сцене __Prospector_Scene_0 в панели Project (Проект), чтобы открыть ее.
2. В главном меню выберите пункт File > Build Settings (Файл > Параметры сборки), после чего откроется окно, изображенное на рис. 32.1.

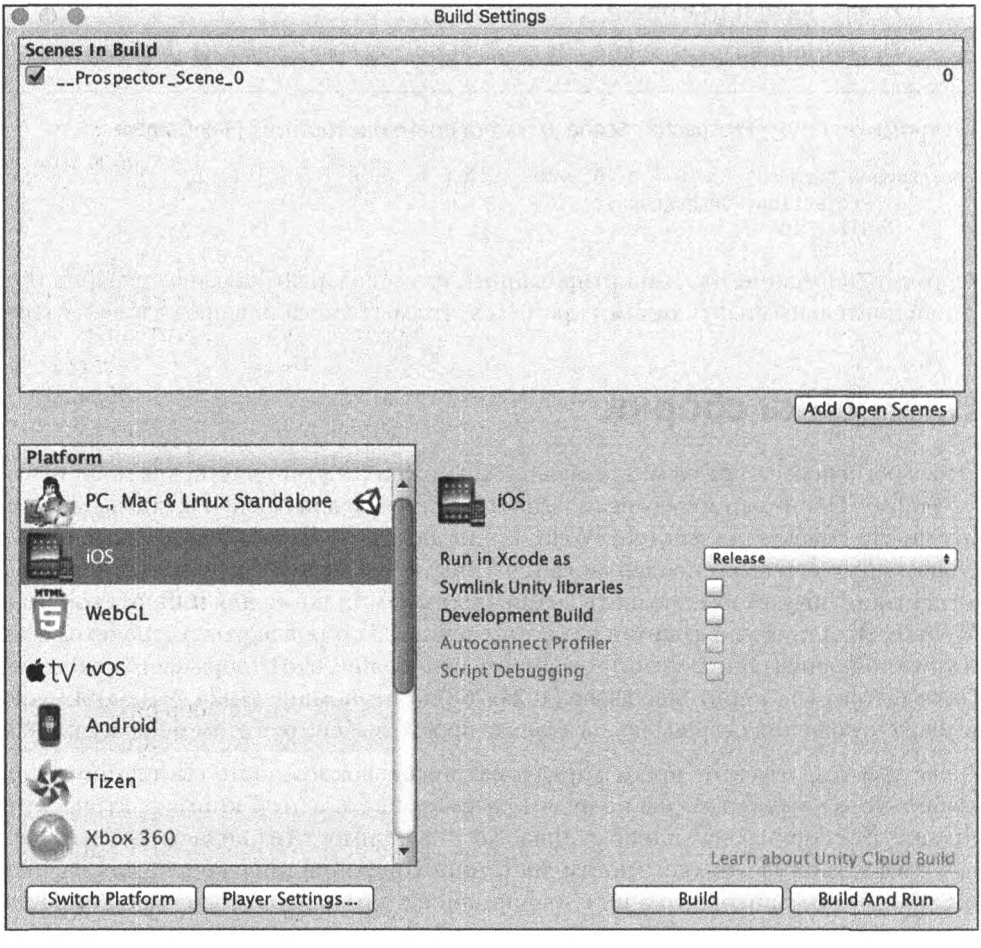


Рис. 32.1. Окно Build Settings (Параметры сборки)

3. Щелкните на кнопке Add Open Scenes (Добавить открытые сцены), чтобы добавить __Prospector_Scene_0 в список сцен для этой сборки.
4. Выберите iOS (или другую платформу по своему желанию) из списка платформ и щелкните на кнопке Switch Platform (Выбрать платформу). После этого Unity повторно импортирует все изображения, чтобы привести их в соответствие с настройками по умолчанию для iOS, а сама кнопка Switch Platform (Выбрать

платформу) примет неактивный вид. Определив настройки сборки, как показано на рис. 32.1, можете закрыть окно. (Не щелкайте пока на кнопке Build (Собрать); это произойдет, когда вы закончите работу над программным кодом игры.)

Импортирование изображений в виде спрайтов

Далее нужно правильно импортировать изображения, чтобы их можно было использовать как *спрайты*. Спрайт — это двумерное изображение, которое может перемещаться по экрану, масштабироваться и поворачиваться. Они часто используются в двумерных играх:

1. Откройте папку _Sprites в панели Project (Проект) и выберите все изображения в ней. (Щелкните на первом изображении в списке, нажмите клавишу Shift и, удерживая ее, щелкните на последнем изображении.) Взгляните на область Preview (Предварительный просмотр) внизу в панели Inspector (Инспектор), в ней вы должны увидеть все импортированные в данный момент изображения с непривычными соотношениями сторон и без прозрачности. Давайте изменим это положение дел и превратим изображения в полезные спрайты.

2. В разделе 21 Texture 2Ds Import Settings (Настройки импортирования 21 двумерной текстуры), в панели Inspector (Инспектор), выберите в раскрывающемся списке Texture Type (Тип текстуры) пункт Sprite (2D and UI) (Спрайт (2D и ПИ)). Щелкните на кнопке Apply (Применить), и Unity повторно импортирует все изображения с правильными соотношениями сторон. На рис. 32.2 показаны окончательные настройки импорта.

   Взглянув на панель Project (Проект), можно увидеть, что теперь рядом с каждым изображением появилась пиктограмма с треугольником. Если щелкнуть на ней, вы увидите, что внутри каждого изображения имеется спрайт с тем же именем.

3. Выберите в панели Project (Проект) изображение с именем Letters. В большинстве случаев наиболее подходящим является режим импортирования изображений в одиночный спрайт (значение Single в раскрывающемся списке Sprite Mode на рис. 32.2), когда каждое изображение преобразуется в единственный спрайт. Однако изображение Letters в действительности является *атласом спрайтов* (последовательностью спрайтов в единственном изображении), поэтому для него необходимо использовать немного иные настройки импортирования.

4. В разделе Letters Import Settings (Настройки импортирования Letters), в панели Inspector (Инспектор), выберите в раскрывающемся списке Sprite Mode (Режим спрайта) пункт Multiple (Множественный) и щелкните на кнопке Apply (Применить). В результате ниже поля Extrude Edges (Выдавить края) появится кнопка Sprite Editor (Редактор спрайтов).

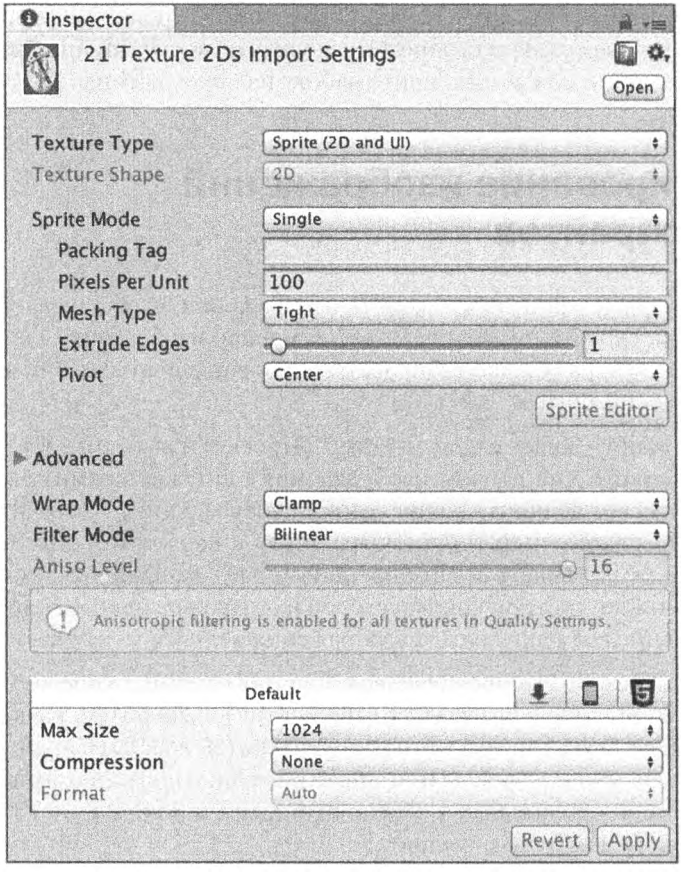


Рис. 32.2. Настройки импортирования двумерных текстур в спрайты

5. Щелкните на кнопке Sprite Editor (Редактор спрайтов), чтобы открыть редактор. Вы увидите изображение Letters в редакторе, окруженное синей рамкой, определяющей границы спрайта Letters.
6. Щелкните в редакторе спрайтов на маленькой пиктограмме с изображением радуги или буквы A (в красной окружности на рис. 32.3), чтобы переключиться между просмотром фактического изображения и его альфа-канала. Поскольку Letters содержит изображения белых букв на прозрачном фоне, вам проще будет видеть происходящее, глядя на альфа-канал.
7. Щелкните на раскрывающемся списке Slice (Нарезать) в левом верхнем углу редактора спрайтов и:
   a. В раскрывающемся списке Type (Тип) замените значение Automatic (Автоматически) на Grid by Cell Size (По сетке с размером ячейки), как показано на рис. 32.3.

b. В поле Pixel size (Размер в пикселах) установите значения X:32 и Y:32.

c. Щелкните на кнопке Slice (Нарезать). В результате изображение Letters будет разделено по горизонтали на 16 спрайтов с размерами 32 × 32 пиксела.

d. Щелкните на кнопке Apply (Применить) в правом верхнем углу редактора спрайтов, чтобы сгенерировать спрайты в панели Project (Проект). Теперь в панели Project (Проект) вместо одного спрайта Letters внутри текстуры Letters появятся 16 спрайтов с именами от Letters_0 до Letters_15. Спрайты Letters_1 — Letters_13 будут использоваться в этой игре для изображения достоинства карт (от туза до короля). Теперь все спрайты на месте и готовы к использованию.

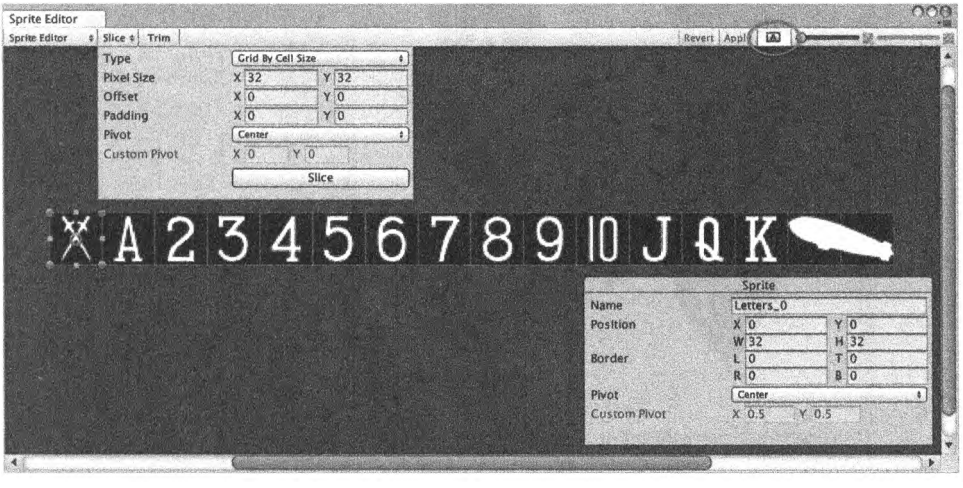


Рис. 32.3. Настройки сетки для правильной нарезки Letters в редакторе спрайтов. Кнопка в красной окружности справа вверху переключает режим просмотра между отображением каналов цвета и альфа-канала

8. Сохраните сцену. В действительности мы еще ничего не изменили в сцене, но сохранять ее постоянно — хорошая привычка. Возьмите за правило сохранять сцену после любого изменения.

Конструирование карт из спрайтов

Один из самых важных аспектов этого проекта — мы сконструируем всю колоду карт программно, используя 21 импортированное изображение. Это поможет уменьшить окончательный размер сборки и увидеть, как работать с файлами XML.

Изображение на рис. 32.4 демонстрирует пример, как это будет делаться. Десятка пик сконструирована из следующих спрайтов: Card_Front, 12 копий Spade и 2 копии Letters_10.

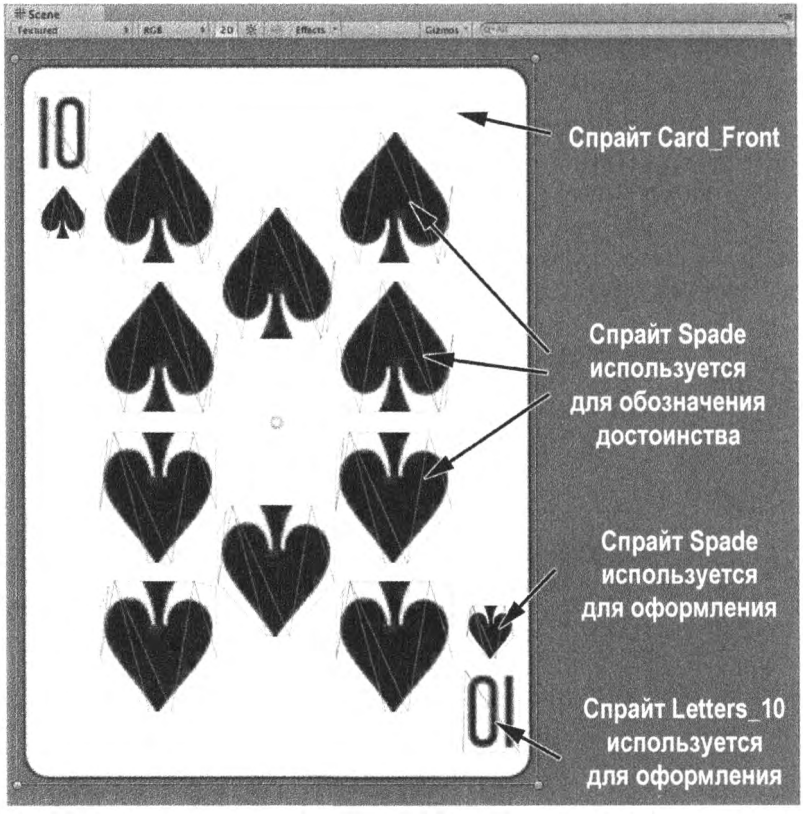


Рис. 32.4. На изображении десятки пик видны границы вокруг спрайтов, из которых она состоит, сгенерированных автоматически. Видимая часть карты составлена из 15 разных спрайтов (12 спрайтов Spade, 2 спрайта Letter_10 и 1 спрайт Card_Front)

Это оформление карты определяется в XML-файле. Прочитайте прямо сейчас раздел «XML» в приложении Б «Полезные идеи», чтобы узнать больше о формате XML и о том, как читать файлы в этом формате с помощью сценария PT_XMLReader, который является частью импортированного пакета. В этом разделе (в приложении Б) также описывается структура файла *DeckXML.xml*, используемого в этом проекте.

Использование XML в программном коде

1. Для первой части этого проекта создайте три сценария на C# с именами Card, Deck и Prospector. Поместите их в папку __Scripts.

   - **Card:** класс, представляющий каждую отдельную карту в колоде. Сценарий Card содержит также классы `CardDefinition` (который хранит информацию

о позициях всех спрайтов на карте каждого достоинства) и `Decorator` (который хранит информацию обо всех значках, описанных в XML-документе, определяющих достоинство и используемых для оформления, — разница между ними показана на рис. 32.4).

- **Deck:** класс, интерпретирующий информацию из файла *DeckXML.xml* и на ее основе создающий колоду карт.
- **Prospector:** класс, управляющий всей игрой. Если класс `Deck` обрабатывает создание карт, то класс `Prospector` включает эти карты в игру. `Prospector` собирает карты в разные стопки (например, стопка свободных карт и стопка для сброшенных карт) и управляет логикой игры.

2. Откройте сценарий **Card** и введите следующий код. Эти маленькие классы в *Card.cs* предназначены для хранения информации, которую класс `Deck` получает при чтении XML-файла.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Card : MonoBehaviour {
    // Будет определен позже
}

[System.Serializable] // Сериализуемый класс доступен для правки в инспекторе
public class Decorator {
    // Этот класс хранит информацию из DeckXML о каждом значке на карте
    public string    type; // Значок, определяющий достоинство карты, имеет
                           // type = "pip"
    public Vector3   loc;  // Местоположение спрайта на карте
    public bool      flip = false; // Признак переворота спрайта по вертикали
    public float     scale = 1f;   // Масштаб спрайта
}

[System.Serializable]
public class CardDefinition {
    // Этот класс хранит информацию о достоинстве карты
    public string face; // Спрайт, изображающий лицевую сторону карты
    public int rank;    // Достоинство карты (1-13)
    public List<Decorator> pips = new List<Decorator>(); // Значки          // a
}
```

   a. `Pips` — это список экземпляров `Decorator`, отображаемых на лицевой стороне карты, например, десять больших значков масти пик на десятке пик, как показано на рис. 32.4. Значки `Decorator`, что отображаются в углах карты (например, значки пик рядом с числом 10 на карте в углу как на рис. 32.4), не нужно хранить в `CardDefinition`, потому что они находятся в одной и той же позиции на любой карте в колоде.

3. Откройте сценарий **Deck** в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Deck : MonoBehaviour {

    [Header("Set Dynamically")]
    public PT_XMLReader             xmlr;

    // InitDeck вызывается экземпляром Prospector, когда будет готов
    public void InitDeck(string deckXMLText) {
        ReadDeck(deckXMLText);
    }

    // ReadDeck читает указанный XML-файл и создает массив экземпляров
CardDefinition
    public void ReadDeck(string deckXMLText) {
        xmlr = new PT_XMLReader(); // Создать новый экземпляр PT_XMLReader
        xmlr.Parse(deckXMLText);   // Использовать его для чтения DeckXML

        // Вывод проверочной строки, чтобы показать, как использовать xmlr.
        // За дополнительной информацией о чтении файлов XML обращайтесь
        //   к приложению "Полезные идеи"
        string s = "xml[0] decorator[0] ";
        s += "type="+xmlr.xml["xml"][0]["decorator"][0].att("type");
        s += " x="+xmlr.xml["xml"][0]["decorator"][0].att("x");
        s += " y="+xmlr.xml["xml"][0]["decorator"][0].att("y");
        s += " scale="+xmlr.xml["xml"][0]["decorator"][0].att("scale");
        print(s);
    }
}
```

4. Теперь откройте сценарий Prospector и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement; // Будет использоваться позже
using UnityEngine.UI;              // Будет использоваться позже

public class Prospector : MonoBehaviour {
    static public Prospector    S;

    [Header("Set in Inspector")]
    public TextAsset            deckXML;

    [Header("Set Dynamically")]
    public Deck                 deck;

    void Awake() {
        S = this; // Подготовка объекта-одиночки Prospector
    }

    void Start () {
```

```
        deck = GetComponent<Deck>(); // Получить компонент Deck
        deck.InitDeck(deckXML.text); // Передать ему DeckXML
    }
}
```

5. Обязательно сохраните все эти сценарии перед возвратом в Unity. В меню MonoDevelop выберите пункт File > Save All (Файл > Сохранить все). Если пункт Save All (Сохранить все) неактивен, значит, вы уже сохранили их.
6. Теперь вернитесь в Unity и подключите оба сценария — Prospector и Deck — к _MainCamera. (Перетащите их по очереди из панели Project (Проект) на объект _MainCamera в панели Hierarchy (Иерархия).) Выберите _MainCamera в иерархии. Вы должны увидеть, что оба сценария подключены как компоненты Script.
7. Перетащите DeckXML из папки Resources в панели Project (Проект) на поле deckXML TextAsset компонента Prospector (Script) в инспекторе.
8. Сохраните сцену и щелкните на кнопке Play (Играть). В консоли должна появиться строка:

```
xml[0] decorator[0] type=letter x=-1.05 y=1.42 scale=1.25
```

Она выводится тестовым кодом в `Deck:ReadDeck()` и показывает, что `ReadDeck()` правильно читает атрибуты `type`, `x`, `y` и `scale` из описания 0-го элемента `decorator` 0-го элемента `xml` в XML-файле, которые в *DeckXML.xml* определяются в показанных ниже строках. (Полное содержимое файла *DeckXML.xml* можно увидеть в разделе «XML» приложения B или открыв *DeckXML.xml* в MonoDevelop.)

```
<xml>
    <!-- элементы decorator отображаются в углах каждой карты
         и представляют их масть и достоинство. -->
    <decorator type="letter" x="-1.05" y="1.42" z="0" flip="0" scale="1.25"/>
    ...
</xml>
```

Анализ информации из DeckXML

Теперь приступим к практическому использованию этой информации.

1. Внесите следующие изменения в класс `Deck`:

```
public class Deck : MonoBehaviour {

    [Header("Set Dynamically")]
    public PT_XMLReader              xmlr;
    public List<string>              cardNames;
    public List<Card>                cards;
    public List<Decorator>           decorators;
    public List<CardDefinition>      cardDefs;
    public Transform                 deckAnchor;
    public Dictionary<string,Sprite> dictSuits;
```

```
// InitDeck вызывается экземпляром Prospector, когда будет готов
public void InitDeck(string deckXMLText) {
    ReadDeck(deckXMLText);
}

// ReadDeck читает указанный XML-файл и создает массив экземпляров
// CardDefinition
public void ReadDeck(string deckXMLText) {
    xmlr = new PT_XMLReader(); // Создать новый экземпляр PT_XMLReader
    xmlr.Parse(deckXMLText);   // Использовать его для чтения DeckXML

    // Вывод проверочной строки, чтобы показать, как использовать xmlr.
    // За дополнительной информацией о чтении файлов XML обращайтесь
    //   к приложению "Полезные идеи"
    string s = "xml[0] decorator[0] ";
    s += "type="+xmlr.xml["xml"][0]["decorator"][0].att("type");
    s += " x="+xmlr.xml["xml"][0]["decorator"][0].att("x");
    s += " y="+xmlr.xml["xml"][0]["decorator"][0].att("y");
    s += " scale="+xmlr.xml["xml"][0]["decorator"][0].att("scale");
    //print(s); // Закомментируйте эту строку,
                //   так как мы уже закончили тестирование

    // Прочитать элементы <decorator> для всех карт
    decorators = new List<Decorator>(); // Инициализировать список
                                        //   экземпляров Decorator
    // Извлечь список PT_XMLHashList всех элементов <decorator> из XML-файла
    PT_XMLHashList xDecos = xmlr.xml["xml"][0]["decorator"];
    Decorator deco;
    for (int i=0; i<xDecos.Count; i++) {
        // Для каждого элемента <decorator> в XML
        deco = new Decorator(); // Создать экземпляр Decorator
        // Скопировать атрибуты из <decorator> в Decorator
        deco.type = xDecos[i].att("type");
        // deco.flip получит значение true, если атрибут flip содержит
        // текст "1"
        deco.flip = ( xDecos[i].att ("flip") == "1" );                  // a
        // Получить значения float из строковых атрибутов
        deco.scale = float.Parse( xDecos[i].att ("scale") );
        // Vector3 loc инициализируется как [0,0,0],
        //   поэтому нам остается только изменить его
        deco.loc.x = float.Parse( xDecos[i].att ("x") );
        deco.loc.y = float.Parse( xDecos[i].att ("y") );
        deco.loc.z = float.Parse( xDecos[i].att ("z") );
        // Добавить deco в список decorators
        decorators.Add (deco);
    }

    // Прочитать координаты для значков, определяющих достоинство карты
    cardDefs = new List<CardDefinition>(); // Инициализировать список карт
    // Извлечь список PT_XMLHashList всех элементов <card> из XML-файла
    PT_XMLHashList xCardDefs = xmlr.xml["xml"][0]["card"];
    for (int i=0; i<xCardDefs.Count; i++) {
        // Для каждого элемента <card>
        // Создать экземпляр CardDefinition
```

```
            CardDefinition cDef = new CardDefinition();
            // Получить значения атрибута и добавить их в cDef
            cDef.rank = int.Parse( xCardDefs[i].att ("rank") );
            // Извлечь список PT_XMLHashList всех элементов <pip>
            //   внутри этого элемента <card>
            PT_XMLHashList xPips = xCardDefs[i]["pip"];
            if (xPips != null) {
                for (int j=0; j<xPips.Count; j++) {
                    // Обойти все элементы <pip>
                    deco = new Decorator();
                    // Элементы <pip> в <card> обрабатываются классом Decorator
                    deco.type = "pip";
                    deco.flip = ( xPips[j].att ("flip") == "1" );
                    deco.loc.x = float.Parse( xPips[j].att ("x") );
                    deco.loc.y = float.Parse( xPips[j].att ("y") );
                    deco.loc.z = float.Parse( xPips[j].att ("z") );
                    if ( xPips[j].HasAtt("scale") ) {
                        deco.scale = float.Parse( xPips[j].att ("scale") );
                    }
                    cDef.pips.Add(deco);
                }
            }
            // Карты с картинками (Валет, Дама и Король) имеют атрибут face
            if (xCardDefs[i].HasAtt("face")) {
                cDef.face = xCardDefs[i].att ("face");                    // b
            }
            cardDefs.Add(cDef);
        }
    }
}
```

a. Необычное, но вполне допустимое применение оператора сравнения ==. Он вернет значение `true` или `false`, которое будет присвоено логическому полю `deco.flip`.

b. `cDef.face` — *базовое имя* спрайта с картинкой для карты. Например, FaceCard_11 — это базовое имя спрайта с изображением валета, у спрайта с изображением трефового валета (Jack of Clubs) имя FaceCard_11C, у спрайта с изображением червового валета (Jack of Hearts) имя FaceCard_11H, и т. д.

Теперь у нас есть метод `ReadDeck()`, который читает содержимое XML-файла и преобразует в списки экземпляров `Decorator` (масть и достоинство в углах карты) и экземпляров `CardDefinition` (с информацией о достоинстве каждой карты (от туза до короля)).

2. Вернитесь в Unity и щелкните на кнопке Play (Играть). Выберите _MainCamera и рассмотрите содержимое компонента Deck (Script) в инспекторе. Так как оба класса — `Decorator` и `CardDefinition` — снабжены атрибутом `[System.Serializable]`, инспектор правильно отображает содержимое списков decorators и cardDefs, как показано на рис. 32.5.

3. Остановите игру и сохраните сцену.

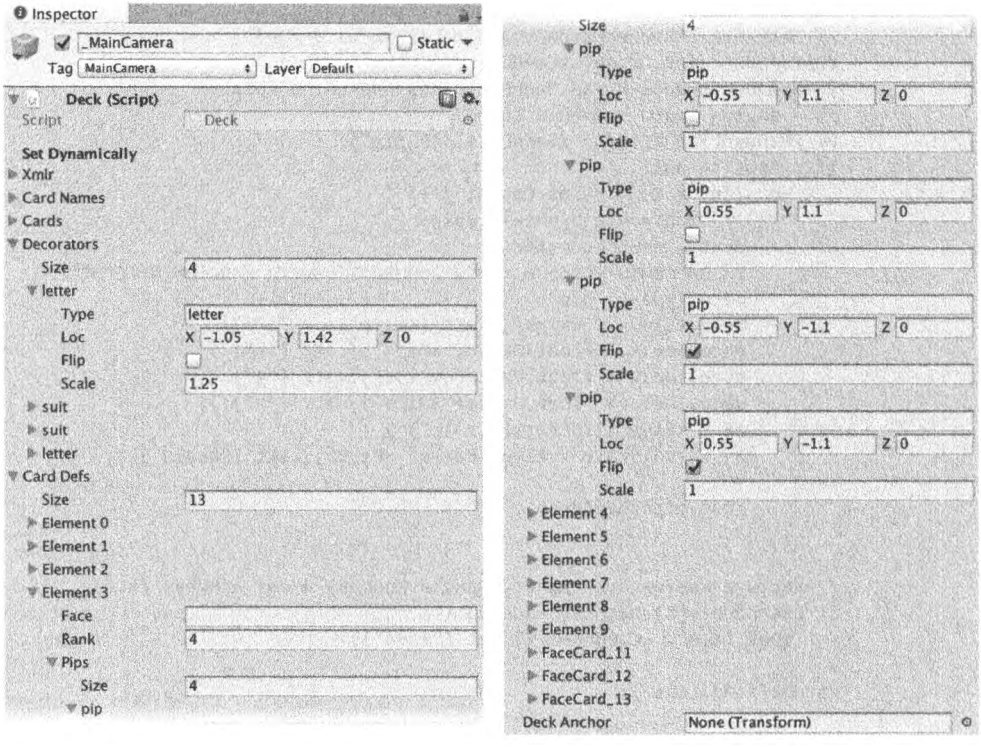


Рис. 32.5. Компонент Deck (Script) главной камеры _MainCamera в инспекторе со списками Decorators и CardDefs, содержимое которых прочитано из файла DeckXML.xml

Связывание спрайтов, составляющих карты

Теперь, когда содержимое XML-файла благополучно читается в списке, можно приступать к конструированию карт. Для начала получим ссылки на все спрайты, созданные выше в этой главе:

1. Добавьте следующие поля в начало класса Deck для хранения ссылок на спрайты:

```
public class Deck : MonoBehaviour {

    [Header("Set in Inspector")]
    // Масти
    public Sprite                suitClub;
    public Sprite                suitDiamond;
    public Sprite                suitHeart;
    public Sprite                suitSpade;

    public Sprite[]              faceSprites;
    public Sprite[]              rankSprites;

    public Sprite                cardBack;
```

```
    public Sprite                cardBackGold;
    public Sprite                cardFront;
    public Sprite                cardFrontGold;

    // Шаблоны
    public GameObject            prefabCard;
    public GameObject            prefabSprite;

    [Header("Set Dynamically")]
    ...
}
```

Сохранив сценарий и вернувшись в Unity, вы увидите, что в инспекторе, в компоненте Deck (Sprite) объекта _MainCamera, появилось множество общедоступных переменных, которые требуется определить.

2. Перетащите текстуры Club, Diamond, Heart и Spade из папки _Sprites в панели Project (Проект) в соответствующие поля компонента Deck (suitClub, suitDiamond, suitHeart и suitSpade). Unity автоматически запишет переменные ссылки на спрайты (а не на текстуры `Texture2D`).

3. Следующий шаг чуть сложнее. Заблокируйте инспектор для _MainCamera, выбрав _MainCamera в иерархии и затем щелкнув на маленьком значке с изображением *замка* вверху на панели Inspector (Инспектор) — в красной рамке на рис. 32.6. Блокировка панели Inspector (Инспектор) гарантирует, что ее содержимое не изменится при выборе чего-то еще.

4. Свяжите все спрайты, начиная с FaceCard_, с элементами массива faceSprites компонента Deck (Script) в инспекторе:

   a. Выберите FaceCard_11C в папке _Sprites панели Project (Проект), нажмите клавишу Shift и, не отпуская ее, щелкните на FaceCard_13S. В результате должны быть выделены все 12 спрайтов FaceCard_.

   b. Перетащите эту группу из панели Project (Проект) на имя массива faceSprites компонента Deck (Script) в инспекторе. Когда указатель мыши окажется над именем массива faceSprites, рядом с ним должен появиться значок + и слово <multiple> (на PC может появиться только значок +).

   c. Отпустите кнопку мыши, и если все было сделано правильно, размер массива faceSprites должен увеличиться до 12, а его элементы — заполниться ссылками на спрайты FaceCard_. Если что-то не получилось, добавьте спрайты по одному. Порядок перетаскивания не имеет значения, главное, чтобы конечный результат получился таким, как показано на рис. 32.6.

5. Щелкните на пиктограмме с изображением треугольника рядом со списком текстур Letters в папке _Sprites в панели Project (Проект). Повторите процедуру, описанную на предыдущем шаге, чтобы выбрать текстуры с Letters_0 по Letters_15. Теперь у вас в разделе Letters должно быть выделено 16 спрайтов. Перетащите эту группу на имя поля rankSprites в компоненте Deck (Script). Если все было сделано правильно, список rankSprites должен заполниться 16 спрайтами Letters_ с име-

нами от Letters_0 по Letters_15. Проверьте порядок следования спрайтов: Letters_0 должен быть в Element 0, а Letters_15 — в Element 15; если что-то не получилось, добавьте спрайты по одному.

6. Перетащите спрайты Card_Back, Card_Back_Gold, Card_Front и Card_Front_Gold из панели Project (Проект) в соответствующие им переменные компонента Deck (Script) в инспекторе.

   Настройки Deck (Script) в инспекторе теперь должны выглядеть так, как показано на рис. 32.6.

7. Разблокируйте панель Inspector (Инспектор), снова щелкнув на значке с изображением замка (в красной рамке на рис. 32.6). Сохраните сцену! Будет обидно, если всю эту работу придется повторить.

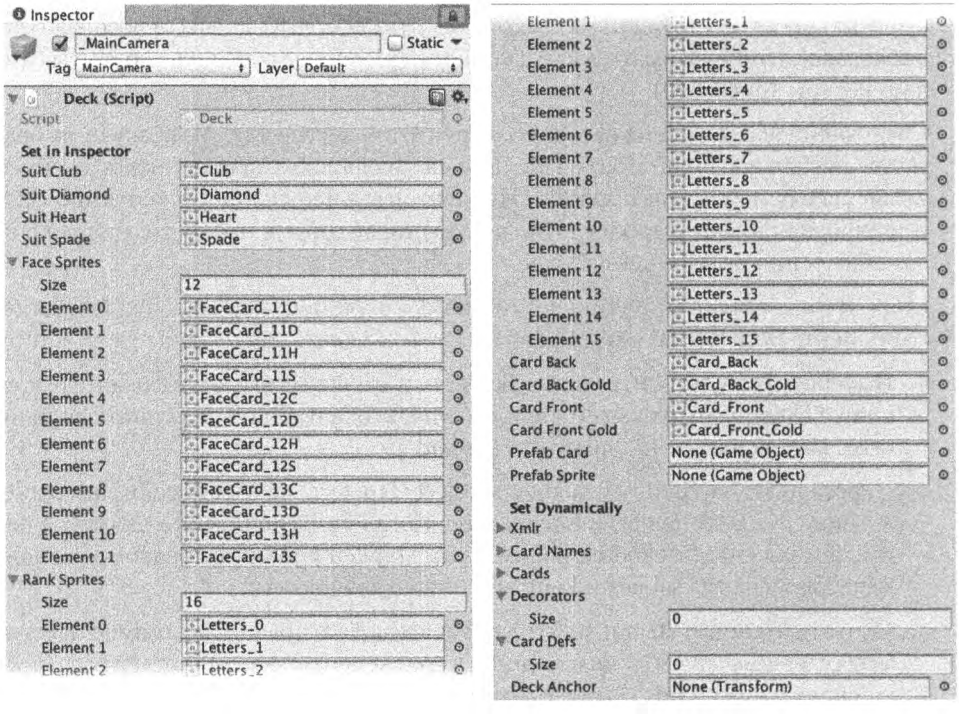


Рис. 32.6. Компонент Deck (Script) главной камеры _MainCamera в инспекторе с общедоступными переменными, связанными с соответствующими спрайтами

Создание шаблонов игровых объектов для отображения спрайтов и карт

Как и все остальное отображаемое на экране, спрайты должны быть заключены в игровые объекты. Для этого проекта нам понадобятся два шаблона: универсальный

PrefabSprite, вмещающий все значки и числа (импортированные раньше из пакета с ресурсами), и PrefabCard, составляющий основу для всех карт в колоде.

Чтобы создать игровой объект PrefabCard, выполните следующие шаги:

1. В главном меню выберите пункт GameObject > 2D Object > Sprite (Игровой объект > 2D объект > Спрайт). Дайте новому игровому объекту имя PrefabCard.
2. Перетащите Card_Front из панели Project (Проект) на переменную Sprite в компоненте Sprite Renderer объекта PrefabCard в инспекторе. После этого спрайт Card_Front должен появиться в панели Scene (Сцена).
3. Перетащите сценарий Card из панели Project (Проект) на объект PrefabCard в иерархии. В результате сценарий Card подключится к объекту PrefabCard (и в инспекторе с настройками для PrefabCard появится компонент Card (Script)).
4. В инспекторе с настройками для PrefabCard щелкните на кнопке Add Component (Добавить компонент). В открывшемся меню выберите пункт Physics > Box Collider (Физика > Коробчатый коллайдер). (То же самое можно проделать, выбрав в главном меню пункт Component > Physics > Box Collider (Компонент > Физика > Коробчатый коллайдер).) Поле Size компонента Box Collider должно автоматически получить настройки [2.56, 3.56, 0.2], но если этого не произошло, введите эти значения вручную.
5. Перетащите PrefabCard из иерархии в папку _Prefabs, чтобы создать шаблон.
6. Удалите экземпляр PrefabCard, оставшийся в иерархии, и сохраните сцену.

   Теперь нужно перетащить шаблоны PrefabCard и PrefabSprite на соответствующие общедоступные переменные компонента Deck (Script) главной камеры _MainCamera в инспекторе.
7. Выберите _MainCamera в иерархии и перетащите PrefabCard и PrefabSprite из панели Project (Проект) в соответствующие переменные компонента Deck (Script) в инспекторе.
8. Сохраните сцену.

Конструирование карт программным способом

Прежде чем добавлять в класс Deck методы, конструирующие карты, мы должны добавить переменные в класс Card, как показано ниже (объем кода получился большим, но результат вас восхитит!):

1. Замените комментарий // Будет определен позже в классе Card следующим кодом.

```
public class Card : MonoBehaviour {
    [Header("Set Dynamically")]
    public string           suit; // Масть карты (C,D,H или S)
    public int              rank; // Достоинство карты (1-14)
    public Color            color = Color.black; // Цвет значков
    public string           colS = "Black";      // или "Red". Имя цвета
```

```
    // Этот список хранит все игровые объекты Decorator
    public List<GameObject> decoGOs = new List<GameObject>();
    // Этот список хранит все игровые объекты Pip
    public List<GameObject> pipGOs = new List<GameObject>();

    public GameObject       back; // Игровой объект рубашки карты

    public CardDefinition   def;  // Извлекается из DeckXML.xml
}
```

2. Теперь добавьте код в класс Deck:

```
public class Deck : MonoBehaviour {
    ...
    // InitDeck вызывается экземпляром Prospector, когда будет готов
    public void InitDeck(string deckXMLText) {
        // Создать точку привязки для всех игровых объектов Card в иерархии
        if (GameObject.Find("_Deck") == null) {
            GameObject anchorGO = new GameObject("_Deck");
            deckAnchor = anchorGO.transform;
        }

        // Инициализировать словарь со спрайтами значков мастей
        dictSuits = new Dictionary<string, Sprite>() {
            { "C", suitClub },
            { "D", suitDiamond },
            { "H", suitHeart },
            { "S", suitSpade }
        };

        ReadDeck(deckXMLText); // Эта строка существовала в предыдущей версии
                               // сценария

        MakeCards();
    }

    // ReadDeck читает указанный XML-файл и создает массив экземпляров
    // CardDefinition
    public void ReadDeck(string deckXMLText) { ... }

    // Получает CardDefinition на основе значения достоинства
    //   (от 1 до 14 - от туза до короля)
    public CardDefinition GetCardDefinitionByRank(int rnk) {
        // Поиск во всех определениях CardDefinition
        foreach (CardDefinition cd in cardDefs) {
            // Если достоинство совпадает, вернуть это определение
            if (cd.rank == rnk) {
                return( cd );
            }
        }
        return( null );
    }

    // Создает игровые объекты карт
    public void MakeCards() {
```

```
    // cardNames будет содержать имена сконструированных карт
    // Каждая масть имеет 14 значений достоинства
    //   (например для треф (Clubs): от C1 до C14)
    cardNames = new List<string>();
    string[] letters = new string[] {"C","D","H","S"};
    foreach (string s in letters) {
        for (int i=0; i<13; i++) {
            cardNames.Add(s+(i+1));
        }
    }

    // Создать список со всеми картами
    cards = new List<Card>();

    // Обойти все только что созданные имена карт
    for (int i=0; i<cardNames.Count; i++) {
        // Создать карту и добавить ее в колоду
        cards.Add ( MakeCard(i) );
    }
}

private Card MakeCard(int cNum) {                                    // a
    // Создать новый игровой объект с картой
    GameObject cgo = Instantiate(prefabCard) as GameObject;
    // Настроить transform.parent новой карты в соответствии с точкой привязки.
    cgo.transform.parent = deckAnchor;
    Card card = cgo.GetComponent<Card>(); // Получить компонент Card

    // Эта строка выкладывает карты в аккуратный ряд
    cgo.transform.localPosition = new Vector3( (cNum%13)*3, cNum/13*4, 0 );

    // Настроить основные параметры карты
    card.name = cardNames[cNum];
    card.suit = card.name[0].ToString();
    card.rank = int.Parse( card.name.Substring(1) );
    if (card.suit == "D" || card.suit == "H") {
        card.colS = "Red";
        card.color = Color.red;
    }
    // Получить CardDefinition для этой карты
    card.def = GetCardDefinitionByRank(card.rank);

    AddDecorators(card);

    return card;
}

// Следующие скрытые переменные используются вспомогательными методами
private Sprite          _tSp = null;
private GameObject      _tGO = null;
private SpriteRenderer  _tSR = null;

private void AddDecorators(Card card) {                              // a
    // Добавить оформление
    foreach( Decorator deco in decorators ) {
```

```
            if (deco.type == "suit") {
                // Создать экземпляр игрового объекта спрайта
                tGO = Instantiate( prefabSprite ) as GameObject;
                // Получить ссылку на компонент SpriteRenderer
                tSR = _tGO.GetComponent<SpriteRenderer>();
                // Установить спрайт масти
                tSR.sprite = dictSuits[card.suit];
            } else {
                tGO = Instantiate( prefabSprite ) as GameObject;
                tSR = _tGO.GetComponent<SpriteRenderer>();
                // Получить спрайт для отображения достоинства
                tSp = rankSprites[ card.rank ];
                // Установить спрайт достоинства в SpriteRenderer
                tSR.sprite = _tSp;
                // Установить цвет, соответствующий масти
                tSR.color = card.color;
            }
            // Поместить спрайты над картой
            tSR.sortingOrder = 1;
            // Сделать спрайт дочерним по отношению к карте
            tGO.transform.SetParent( card.transform );
            // Установить localPosition, как определено в DeckXML
            tGO.transform.localPosition = deco.loc;
            // Перевернуть значок, если необходимо
            if (deco.flip) {
                // Эйлеров поворот на 180° относительно оси Z-axis
                tGO.transform.rotation = Quaternion.Euler(0,0,180);
            }
            // Установить масштаб, чтобы уменьшить размер спрайта
            if (deco.scale != 1) {
                tGO.transform.localScale = Vector3.one * deco.scale;
            }
            // Дать имя этому игровому объекту для наглядности
            tGO.name = deco.type;
            // Добавить этот игровой объект с оформлением в список card.decoGOs
            card.decoGOs.Add(_tGO);
        }
    }
}
```

a. `MakeCard()` и `AddDecorator()` — это скрытые вспомогательные методы, используемые в `MakeCards()`. Такой прием помогает сделать метод `MakeCards()` короче, и если вы работаете в группе, каждый из этих трех методов мог бы написать другой программист, главное, чтобы они делали то, что от них требуется. Лично я предпочитаю писать как можно более короткие функции, подобные этим, в чем вы убедитесь в главе 35 «Прототип 7: Dungeon Delver».

3. Сохраните все сценарии, вернитесь в Unity и щелкните на кнопке **Play** (Играть). Вы должны увидеть 52 карты, выложенные в ряд. На них пока не отображаются значки, определяющие достоинство, зато оформление в углах уже имеется.

4. Теперь организуем отображение значков достоинства и картинок на картах, добавив в класс `Deck` еще три вспомогательных метода:

```
public class Deck : MonoBehaviour {
    ...
    private Card MakeCard(int cNum) {
        ...
        card.def = GetCardDefinitionByRank(card.rank);

        AddDecorators(card);
        AddPips(card);
        AddFace(card);

        return card;
    }

    private void AddDecorators(Card card) { ... }

    private void AddPips(Card card) {
        // Для каждого значка в определении...
        foreach( Decorator pip in card.def.pips ) {
            // ...Создать игровой объект спрайта
            _tGO = Instantiate( prefabSprite ) as GameObject;
            // Назначить родителем игровой объект карты
            _tGO.transform.SetParent( card.transform );
            // Установить localPosition, как определено в XML-файле
            _tGO.transform.localPosition = pip.loc;
            // Перевернуть, если необходимо
            if (pip.flip) {
                _tGO.transform.rotation = Quaternion.Euler(0,0,180);
            }
            // Масштабировать, если необходимо (только для туза)
            if (pip.scale != 1) {
                _tGO.transform.localScale = Vector3.one * pip.scale;
            }
            // Дать имя игровому объекту
            _tGO.name = "pip";
            // Получить ссылку на компонент SpriteRenderer
            _tSR = _tGO.GetComponent<SpriteRenderer>();
            // Установить спрайт масти
            _tSR.sprite = dictSuits[card.suit];
            // Установить sortingOrder, чтобы значок отображался над Card_Front
            _tSR.sortingOrder = 1;
            // Добавить этот игровой объект в список значков
            card.pipGOs.Add(_tGO);
        }
    }

    private void AddFace(Card card) {
        if (card.def.face == "") {
            return; // Выйти, если это не карта с картинкой
        }

        _tGO = Instantiate( prefabSprite ) as GameObject;
        _tSR = _tGO.GetComponent<SpriteRenderer>();
        // Сгенерировать имя и передать его в GetFace()
        _tSp = GetFace( card.def.face+card.suit );
        _tSR.sprite = _tSp;     // Установить этот спрайт в _tSR
        _tSR.sortingOrder = 1; // Установить sortingOrder
        _tGO.transform.SetParent( card.transform );
```

```
        _tGO.transform.localPosition = Vector3.zero;
        _tGO.name = "face";
    }

    // Находит спрайт с картинкой для карты
    private Sprite GetFace(string faceS) {
        foreach (Sprite _tSP in faceSprites) {
            // Если найден спрайт с требуемым именем...
            if (_tSP.name == faceS) {
                // ...вернуть его
                return( _tSP );
            }
        }
        // Если ничего не найдено, вернуть null
        return( null );
    }
}
```

5. Щелкните на кнопке Play (Играть). Вы должны увидеть 52 карты, выложенные в ряд, и на картах с лицевой стороны должны отображаться правильные картинки.

   Далее добавим на карты рубашки. На самом деле карты не будут переворачиваться, вместо этого на карту будет накладываться спрайт с большим значением `sortingOrder`, чем у стальных спрайтов, в результате будет видна только рубашка карты, а все остальное окажется под ней.

6. Для переключения режима отображения добавьте свойство `faceUp` в конец класса `Card`, как показано ниже. Будучи свойством, `faceUp` фактически состоит из двух функций (`get` и `set`), маскирующихся под единое поле:

```
public class Card : MonoBehaviour {
    ...
    public GameObject       back; // Игровой объект рубашки карты

    public CardDefinition   def;  // Извлекается из DeckXML.xml

    public bool faceUp {
        get {
            return( !back.activeSelf );
        }
        set {
            back.SetActive(!value);
        }
    }
}
```

7. Теперь добавим рубашку карты в класс `Deck`. Добавьте в класс `Deck` следующее поле и вспомогательный метод:

```
public class Deck : MonoBehaviour {
    [Header("Set in Inspector")]
    public bool                 startFaceUp = false;
    // Масти
    public Sprite               suitClub;
    ...
```

```
    private Card MakeCard(int cNum) {
        ...
        AddPips(card);
        AddFace(card);
        AddBack(card);

        return card;
    }
    ...

    // Находит спрайт с картинкой для карты
    private Sprite GetFace(string faceS) { ... }

    private void AddBack(Card card) {
        // Добавить рубашку
        // Card_Back будет покрывать все остальное на карте
        _tGO = Instantiate( prefabSprite ) as GameObject;
        _tSR = _tGO.GetComponent<SpriteRenderer>();
        _tSR.sprite = cardBack;
        _tGO.transform.SetParent( card.transform );
        _tGO.transform.localPosition = Vector3.zero;
        // Большее значение sortingOrder, чем у других спрайтов
        _tSR.sortingOrder = 2;
        _tGO.name = "back";
        card.back = _tGO;

        // По умолчанию картинкой вверх
        card.faceUp = startFaceUp; // Использовать свойство faceUp карты
    }
}
```

8. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и щелкните на кнопке Play (Играть). Теперь все карты появятся перевернутыми лицевой стороной вниз.
9. Остановите игру, установите флажок startFaceUp компонента Deck (Script) главной камеры _MainCamera в инспекторе и запустите игру снова. На этот раз все карты появятся лицевой стороной вверх.
10. Сохраните сцену. Всегда сохраняйте сцены.

Перемешивание карт

Теперь, когда карты готовы и отображаются на экране, добавим в класс `Deck` возможность перемешивания колоды:

1. Добавьте в конец класса `Deck` следующий общедоступный статический метод `Shuffle()`:

```
public class Deck : MonoBehaviour {
    ...
    private void AddBack(Card card) { ... }

    // Перемешивает карты в Deck.cards
```

```
    static public void Shuffle(ref List<Card> oCards) {                    // a
        // Создать временный список для хранения карт в перемешанном порядке
        List<Card> tCards = new List<Card>();

        int ndx; // Будет хранить индекс перемещаемой карты
        tCards = new List<Card>(); // Инициализировать временный список
        // Повторять, пока не будут перемещены все карты в исходном списке
        while (oCards.Count > 0) {
            // Выбрать случайный индекс карты
            ndx = Random.Range(0,oCards.Count);
            // Добавить эту карту во временный список
            tCards.Add (oCards[ndx]);
            // и удалить карту из исходного списка
            oCards.RemoveAt(ndx);
        }
        // Заменить исходный список временным
        oCards = tCards;
        // Так как oCards - это параметр-ссылка (ref), оригинальный аргумент,
        //   переданный в метод, тоже изменится.
    }
}
```

a. Ключевое слово `ref` сообщает, что в параметре `List<Card> oCards` передается ссылка на переменную типа `List<Card>`, а не ее копия. То есть любые изменения в параметре `oCards` в действительности будут производиться с переменной, переданной в метод. Проще говоря, если передать методу список `Deck.cards` по ссылке, он перемешает карты в самом этом списке и вызывающему коду не потребуется использовать возвращаемое значение.

2. Добавьте следующие строки в метод `Prospector.Start()` для проверки работы этого метода:

```
public class Prospector : MonoBehaviour {
    ...
    void Start () {
        deck = GetComponent<Deck>();  // Получить компонент Deck
        deck.InitDeck(deckXML.text);  // Передать ему DeckXML
        Deck.Shuffle(ref deck.cards); // Перемешать колоду, передав ее по ссылке // a

        Card c;
        for (int cNum=0; cNum<deck.cards.Count; cNum++) {                  // b
            c = deck.cards[cNum];
            c.transform.localPosition = new Vector3( (cNum%13)*3, cNum/13*4, 0 );
        }
    }
}
```

a. Ключевое слово `ref` также требуется использовать при вызове метода с параметром-ссылкой.

b. Этот цикл `for` выводит карты в новом, перемешанном порядке.

3. Если теперь сохранить сценарии и запустить сцену, вы сможете выбрать `_MainCamera` в иерархии и посмотреть содержимое переменной `Deck.cards`, чтобы увидеть перемешанный массив карт.

Теперь, когда класс `Deck` может перемешивать любые списки карт, у нас есть все основные инструменты для создания любых карточных игр. Игра, прототип которой мы создаем в этой главе, называется *Prospector*[1].

Игра Prospector

К данному моменту мы написали код, реализующий основные инструменты для создания любой карточной игры. Теперь поговорим о конкретной игре, которую мы собираемся создать[2].

Пасьянс *Prospector* основан на классическом карточном пасьянсе *Tri-Peaks*. Правила обеих игр совпадают, кроме двух аспектов:

- По сюжету *Prospector* игрок добывает золото, тогда как в *Tri-Peaks* он пытается покорить три вершины.
- Цель игрока в *Tri-Peaks* — собрать все карты в одну колоду. Цель игрока в *Prospector* — зарабатывать очки, собирая как можно более длинные цепочки карт, и каждая золотая карта в цепочке удваивает количество очков всей цепочки.

Правила игры в Prospector

Для пробы возьмите обычную колоду игральных карт (то есть колоду настоящих карт, не виртуальных, которые мы только что создали). Уберите из колоды джокеры и перетасуйте оставшиеся 52 карты:

1. Разложите 28 карт, как показано на рис. 32.7. Карты в трех нижних рядах положите рубашкой вверх, а карты в верхнем ряду — лицевой стороной вверх. Карты не должны перекрывать друг друга с боков, но карты в верхних рядах должны перекрывать карты в нижних рядах. Это начальная раскладка, изображающая «шахту», которую должен выкопать старатель.
2. Остальные карты из колоды образуют стопку свободных карт. Положите эту стопку над верхним рядом рубашкой вверх.
3. Снимите верхнюю карту со стопки свободных карт и положите ее в центре над верхним рядом. Это — целевая карта. На рис. 32.8 показано, как теперь должна выглядеть полная раскладка.
4. Вы можете перенести на нее любую карту из раскладки шахты, которая на ступень старше или младше целевой, и таким образом объявить новую целевую карту. Для тузов и королей действует правило циклического перехода старшинства, то есть туз можно положить на короля и наоборот.

---

[1] Старатель. — *Примеч. пер.*

[2] Пасьянс *Prospector* был разработан Джереми Гибсоном Бондом, Итаном Барроу (Ethan Burrow) и Майком Вабшаллом (Mike Wabschall) в 2001 году для нашей компании Digital Mercenaries, Inc.

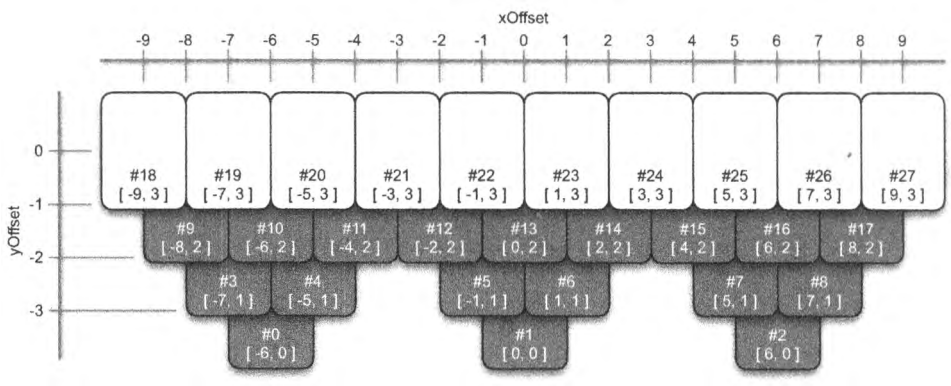


Рис. 32.7. Начальная раскладка карт в пасьянсе Prospector

5. Если карта, повернутая рубашкой вверх, не перекрывается картами из верхнего ряда, ее можно перевернуть лицевой стороной вверх.
6. Если ни одну из карт в раскладке, лежащих лицевой стороной вверх, нельзя положить на целевую карту, снимите новую карту со стопки свободных карт.
7. Если вам удалось переложить все карты из раскладки до исчерпания стопки свободных карт, вы победили! (Обсуждение правил подсчета очков и появления золотых карт в цифровой версии игры я отложу на потом.)

Пример игры

На рис. 32.8 показан пример начальной раскладки в пасьянсе *Prospector*. В данной ситуации на восьмерку червей (8H — 8 of Hearts) игрок может положить либо девятку треф (9C — 9 of Clubs), либо семерку пик (7S — 7 of Spades).

Оранжевые и зеленые цифры показывают две возможные последовательности продолжения игры. В оранжевой последовательности сначала играет девятка треф (9C — 9 of Clubs) и становится новой целевой картой. Это позволит сделать следующий ход восьмеркой пик (8S — 8 of Spades), восьмеркой бубен (8D — 8 of Diamond) или восьмеркой треф (8C — 8 of Clubs). Игрок выбрал восьмерку пик (8S — 8 of Spades), потому что потом он сможет открыть карту, перекрытую девяткой треф (9C — 9 of Clubs) и восьмеркой пик (8S — 8 of Spades). Затем в оранжевой последовательности перекладывается семерка пик (7S — 7 of Spades) и, наконец, восьмерка треф (8C — 8 of Clubs). Получившаяся в результате раскладка показана на рис. 32.9.

Теперь, так как ни одну из карт в раскладке, лежащих лицевой стороной вверх, нельзя положить на целевую карту, игрок должен снять верхнюю карту из свободной стопки и объявить ее новой целевой картой.

И снова я советую взять колоду настоящих карт и попробовать сыграть в игру несколько раз, чтобы почувствовать ее суть.

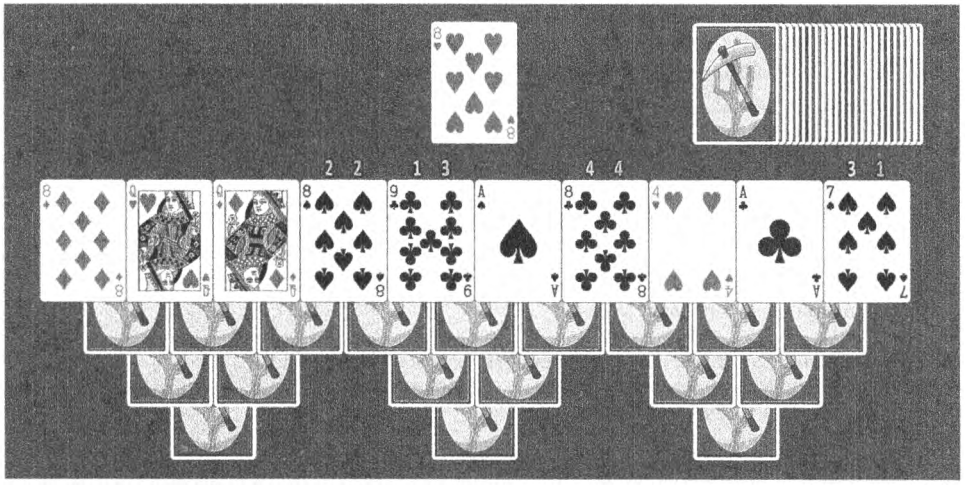


Рис. 32.8. Пример начальной раскладки в пасьянсе Prospector

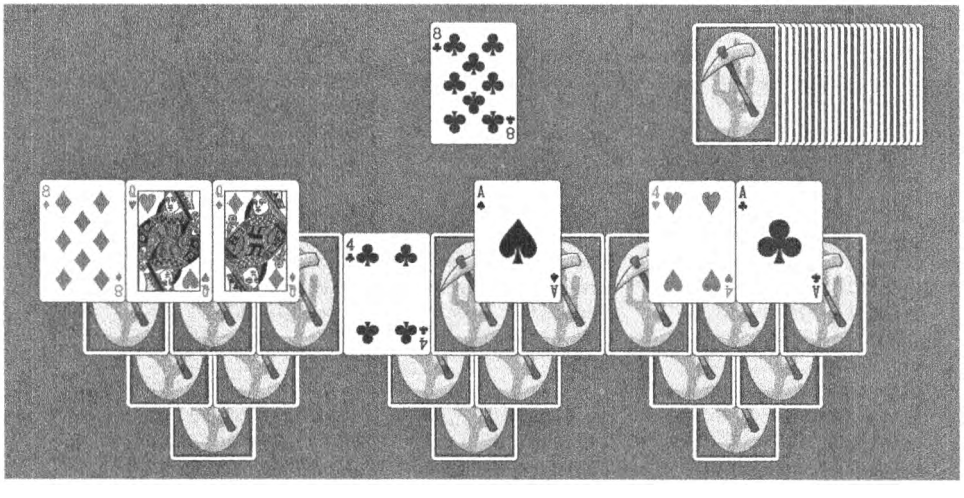


Рис. 32.9. Состояние раскладки в пасьянсе Prospector после первого раунда

Программная реализация Prospector

Как можно заметить из предыдущего описания, *Prospector* — очень простая игра, но достаточно увлекательная. Позднее вы сможете добавить к этой увлекательности еще и визуальные эффекты, а также подсчет очков, но пока давайте реализуем простейший рабочий вариант игры.

Раскладка карт

В цифровой версии *Prospector* нам нужно реализовать ту же раскладку карт, имитирующую шахту, которую мы видели выше, когда пробовали сыграть с настоящими картами. Для этого воспользуемся XML-файлом, описывающим раскладку, изображенную на рис. 32.7.

1. В Unity откройте файл *LayoutXML.xml*, находящийся в папке Resources, чтобы увидеть информацию о раскладке. Обратите внимание, что комментарии в разметке XML заключены в <!-- и --> (по аналогии с /* и */ в коде на C#).

```
<xml>
    <!-- Этот файл хранит информацию о раскладке карт
    в карточной игре Prospector. -->

    <!-- Элемент multiplier имеет атрибуты x и y с множителями. -->
    <!-- Множители определяют, насколько свободной или плотной будет раскладка. -->
    <multiplier x="1.25" y="1.5" />

    <!-- В разметке XML ниже атрибут id определяет номер карты -->
    <!-- x и y определяют позицию в раскладке -->
    <!-- если faceup имеет значение 1, карта повернута лицом вверх -->
    <!-- layer определяет номер слоя, необходимого для правильного перекрытия -->
    <!-- hiddenby – номер карты, перекрывающей эту -->

    <!-- Layer0, самый нижний ряд карт. -->
    <slot id="0" x="-6" y="-5" faceup="0" layer="0" hiddenby="3,4" />
    <slot id="1" x="0" y="-5" faceup="0" layer="0" hiddenby="5,6" />
    <slot id="2" x="6" y="-5" faceup="0" layer="0" hiddenby="7,8" />

    <!-- Layer1, второй ряд снизу. -->
    <slot id="3" x="-7" y="-4" faceup="0" layer="1" hiddenby="9,10" />
    <slot id="4" x="-5" y="-4" faceup="0" layer="1" hiddenby="10,11" />
    <slot id="5" x="-1" y="-4" faceup="0" layer="1" hiddenby="12,13" />
    <slot id="6" x="1" y="-4" faceup="0" layer="1" hiddenby="13,14" />
    <slot id="7" x="5" y="-4" faceup="0" layer="1" hiddenby="15,16" />
    <slot id="8" x="7" y="-4" faceup="0" layer="1" hiddenby="16,17" />

    <!-- Layer2, третий ряд снизу. -->
    <slot id="9" x="-8" y="-3" faceup="0" layer="2" hiddenby="18,19" />
    <slot id="10" x="-6" y="-3" faceup="0" layer="2" hiddenby="19,20" />
    <slot id="11" x="-4" y="-3" faceup="0" layer="2" hiddenby="20,21" />
    <slot id="12" x="-2" y="-3" faceup="0" layer="2" hiddenby="21,22" />
    <slot id="13" x="0" y="-3" faceup="0" layer="2" hiddenby="22,23" />
    <slot id="14" x="2" y="-3" faceup="0" layer="2" hiddenby="23,24" />
    <slot id="15" x="4" y="-3" faceup="0" layer="2" hiddenby="24,25" />
    <slot id="16" x="6" y="-3" faceup="0" layer="2" hiddenby="25,26" />
    <slot id="17" x="8" y="-3" faceup="0" layer="2" hiddenby="26,27" />

    <!-- Layer3, верхний ряд. -->
    <slot id="18" x="-9" y="-2" faceup="1" layer="3" />
    <slot id="19" x="-7" y="-2" faceup="1" layer="3" />
    <slot id="20" x="-5" y="-2" faceup="1" layer="3" />
    <slot id="21" x="-3" y="-2" faceup="1" layer="3" />
```

```
    <slot id="22" x="-1" y="-2" faceup="1" layer="3" />
    <slot id="23" x="1" y="-2" faceup="1" layer="3" />
    <slot id="24" x="3" y="-2" faceup="1" layer="3" />
    <slot id="25" x="5" y="-2" faceup="1" layer="3" />
    <slot id="26" x="7" y="-2" faceup="1" layer="3" />
    <slot id="27" x="9" y="-2" faceup="1" layer="3" />

    <!-- Позиция стопки свободных карт и их смещение
         друг относительно друга -->
    <slot type="drawpile" x="6" y="4" xstagger="0.15" layer="4"/>

    <!-- Позиция стопки сброшенных карт и целевой карты -->
    <slot type="discardpile" x="0" y="1" layer="5"/>
</xml>
```

Как видите, здесь описывается каждая карта в раскладке (элементы `<slot>` без атрибута `type`), а также два специальных слота (элементы `<slot>` с атрибутом `type`) для стопки свободных и сброшенных карт.

2. Реализуем анализ этого файла LayoutXML и сохранение информации для дальнейшего использования. Создайте в папке __Scripts новый сценарий на C# с именем Layout и введите в него следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Класс SlotDef не наследует MonoBehaviour, поэтому для него не требуется
//   создавать отдельный файл на C#.
[System.Serializable] // Сделает экземпляры SlotDef видимыми в инспекторе Unity
public class SlotDef {
    public float         x;
    public float         y;
    public bool          faceUp = false;
    public string        layerName = "Default";
    public int           layerID = 0;
    public int           id;
    public List<int>     hiddenBy = new List<int>();
    public string        type = "slot";
    public Vector2       stagger;
}

public class Layout : MonoBehaviour {
    public PT_XMLReader      xmlr; // Так же, как Deck, имеет PT_XMLReader
    public PT_XMLHashtable   xml;  // Используется для ускорения доступа к xml
    public Vector2           multiplier; // Смещение от центра раскладки
    // Ссылки SlotDef
    public List<SlotDef>     slotDefs; // Все экземпляры SlotDef для рядов 0-3
    public SlotDef           drawPile;
    public SlotDef           discardPile;
    // Хранит имена всех рядов
    public string[]          sortingLayerNames = new string[] { "Row0", "Row1",
                                "Row2", "Row3", "Discard", "Draw" };

    // Эта функция вызывается для чтения файла LayoutXML.xml
```

```
    public void ReadLayout(string xmlText) {
        xmlr = new PT_XMLReader();
        xmlr.Parse(xmlText);      // Загрузить XML
        xml = xmlr.xml["xml"][0]; // И определяется xml для ускорения доступа к XML

        // Прочитать множители, определяющие расстояние между картами
        multiplier.x = float.Parse(xml["multiplier"][0].att("x"));
        multiplier.y = float.Parse(xml["multiplier"][0].att("y"));

        // Прочитать слоты
        SlotDef tSD;
        // slotsX используется для ускорения доступа к элементам <slot>
        PT_XMLHashList slotsX = xml["slot"];

        for (int i=0; i<slotsX.Count; i++) {
            tSD = new SlotDef(); // Создать новый экземпляр SlotDef
            if (slotsX[i].HasAtt("type")) {
                // Если <slot> имеет атрибут type, прочитать его
                tSD.type = slotsX[i].att("type");
            } else {
                // Иначе определить тип как "slot"; это отдельная карта в ряду
                tSD.type = "slot";
            }
            // Преобразовать некоторые атрибуты в числовые значения
            tSD.x = float.Parse( slotsX[i].att("x") );
            tSD.y = float.Parse( slotsX[i].att("y") );
            tSD.layerID = int.Parse( slotsX[i].att("layer") );
            // Преобразовать номер ряда layerID в текст layerName
            tSD.layerName = sortingLayerNames[ tSD.layerID ];              // a

            switch (tSD.type) {
                // прочитать дополнительные атрибуты, опираясь на тип слота
                case "slot":
                    tSD.faceUp = (slotsX[i].att("faceup") == "1");
                    tSD.id = int.Parse( slotsX[i].att("id") );
                    if (slotsX[i].HasAtt("hiddenby")) {
                        string[] hiding = slotsX[i].att("hiddenby").Split(',');
                        foreach( string s in hiding ) {
                            tSD.hiddenBy.Add ( int.Parse(s) );
                        }
                    }
                    slotDefs.Add(tSD);
                    break;

                case "drawpile":
                    tSD.stagger.x = float.Parse( slotsX[i].att("xstagger") );
                    drawPile = tSD;
                    break;
                case "discardpile":
                    discardPile = tSD;
                    break;
            }
        }
    }
}
```

a. Поле `layerName` экземпляра `SlotDef` обеспечивает правильное перекрытие карт. В двумерных проектах Unity все ресурсы фактически имеют одну и ту же координату Z, поэтому необходима дополнительная информация о слоях, чтобы определить, как должны перекрываться спрайты.

На данный момент большая часть кода должна выглядеть для вас знакомой. Класс `SlotDef` предназначен для сохранения информации из XML-элемента `<slot>` в более удобной форме. Далее определяется класс `Layout` и его метод `ReadLayout()`, который получает строку с разметкой XML и преобразует ее в последовательность экземпляров `SlotDef`.

3. Откройте сценарий **Prospector** и добавьте следующие строки, выделенные жирным:

```
public class Prospector : MonoBehaviour {
    static public Prospector    S;

    [Header("Set in Inspector")]
    public TextAsset            deckXML;
    public TextAsset            layoutXML;

    [Header("Set Dynamically")]
    public Deck                 deck;
    public Layout               layout;

    void Awake() {
        S = this; // Подготовка объекта-одиночки Prospector
    }

    void Start () {
        deck = GetComponent<Deck>();  // Получить компонент Deck
        deck.InitDeck(deckXML.text);  // Передать ему DeckXML
        Deck.Shuffle(ref deck.cards); // Перемешать колоду, передав ее по ссылке

// Этот фрагмент нужно закомментировать; сейчас мы создаем фактическую раскладку
//        Card c;
//        for (int cNum=0; cNum<deck.cards.Count; cNum++) {
//            c = deck.cards[cNum];
//            c.transform.localPosition = new Vector3((cNum%13)*3,cNum/13*4,0);
//        }

        layout = GetComponent<Layout>();   // Получить компонент Layout
        layout.ReadLayout(layoutXML.text); // Передать ему содержимое LayoutXML
    }
}
```

4. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

5. В Unity выберите **_MainCamera** в иерархии. В главном меню выберите пункт **Component > Scripts > Layout** (Компонент > Сценарии > Layout), чтобы подключить сценарий **Layout** к **_MainCamera** (это еще один способ подключения сценария к игровому объекту). После этого в панели **Inspector** (Инспектор), внизу, должен появиться компонент **Layout (Script)**.

6. Найдите компонент Prospector (Script) главной камеры _MainCamera. Вы увидите, что в нем появились новые общедоступные поля layout и layoutXML. Щелкните на пиктограмме с изображением мишени рядом с полем layoutXML и выберите LayoutXML на вкладке Assets. (Возможно, вам придется щелкнуть на вкладке Assets в верхней части диалога Select TextAsset (Выбор текстового ресурса).)
7. Сохраните сцену.
8. Щелкните на кнопке Play (Играть). Если после этого выбрать _MainCamera в иерархии и прокрутить содержимое в инспекторе вниз до компонента Layout (Script), вы сможете щелкнуть на пиктограмме с треугольником рядом с полем slotDefs и увидеть все определения слотов, полученные из XML-файла.

CardProspector — наследник класса Card

Прежде чем поместить карты в раскладку, нужно расширить класс `Card`, подготовив его для использования в игре *Prospector*. Так как классы `Card` и `Deck` проектировались с возможностью повторного использования в других карточных играх, вместо изменения самого класса `Card` создадим класс `CardProspector`, наследующий `Card`.

1. Создайте в папке _Scripts новый сценарий на C# с именем CardProspector и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Перечисление, определяющее тип переменной, которая может принимать
//   несколько предопределенных значений                                    // a
public enum eCardState {
     drawpile,
     tableau,
     target,
     discard
}

public class CardProspector : Card { // CardProspector должен расширять Card
    [Header("Set Dynamically: CardProspector")]
     // Так используется перечисление eCardState
     public eCardState            state = eCardState.drawpile;
     // hiddenBy - список других карт, не позволяющих перевернуть эту лицом вверх
    public List<CardProspector>   hiddenBy = new List<CardProspector>();
    // layoutID определяет для этой карты ряд в раскладке
    public int                    layoutID;
    // Класс SlotDef хранит информацию из элемента <slot> в LayoutXML
    public SlotDef                slotDef;
}
```

a. Это перечисление определяет тип переменных, которые могут хранить лишь несколько именованных значений. Переменная `state` типа `eCardState` может хранить одно из четырех значений: `drawpile`, `tableau`, `target` и `discard`. Она

поможет экземплярам CardProspector определять свое место в игре. Я предпочитаю давать перечислениям имена, начинающиеся со строчной буквы e.

Новый класс CardProspector, расширяющий Card, реализует поддержку таких аспектов, как признак местоположения карты в игре (стопка для свободных карт drawpile, основная раскладка tableau (одна из первых 28 карт, определяющих шахту), стопка для сброшенных карт discard и целевая карта target (активная карта на вершине стопки сброшенных карт)), информация о раскладке (slotDef) и то, как должна отображаться карта — лицевой стороной вверх или вниз (hiddenBy и layoutID).

Теперь, определив подкласс, нужно изменить тип карт в колоде с Cards на CardProspectors.

2. Для этого добавьте следующий код в класс Prospector:

```
public class Prospector : MonoBehaviour {
    ...
    [Header("Set Dynamically")]
    public Deck                 deck;
    public Layout               layout;
    public List<CardProspector> drawPile;

    void Awake() { ... }

    void Start () {
        ...
        layout = GetComponent<Layout>();   // Получить компонент Layout
        layout.ReadLayout(layoutXML.text); // Передать ему содержимое LayoutXML
        drawPile = ConvertListCardsToListCardProspectors( deck.cards );
    }

    List<CardProspector> ConvertListCardsToListCardProspectors(List<Card> lCD) {
        List<CardProspector> lCP = new List<CardProspector>();
        CardProspector tCP;
        foreach( Card tCD in lCD ) {
            tCP = tCD as CardProspector;                                    // a
            lCP.Add( tCP );
        }
        return( lCP );
    }
}
```

a. Ключевое слово as пытается преобразовать тип с Card на CardProspector.

3. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

4. Запустите игру и посмотрите на поле drawPile компонента Prospector (Script) главной камеры _MainCamera в панели Inspector (Инспектор).

Вы заметите, что все карты в drawPile имеют значение null. (Увидеть, как это произошло, можно, поместив точку останова в строке с комментарием // a в предыдущем коде и запустив отладчик.) При попытке интерпретировать Card

`tCD` как экземпляр `CardProspector`, ключевое слово `as` возвращает `null` вместо преобразованного экземпляра `Card`. Это одна из особенностей работы объектно-ориентированного кода на языке C# (см. врезку «Суперклассы и подклассы»).

СУПЕРКЛАССЫ И ПОДКЛАССЫ

В главе 26 «Классы» вы познакомились с суперклассами и подклассами. Однáко многие из вас наверняка обеспокоены, почему попытка привести суперкласс к подклассу не увенчалась успехом.

В игре *Prospector* класс `Card` является суперклассом, а `CardProspector` — его подклассом. Такое отношение легко представить на примере суперкласса `Animal` (Животное) и подкласса `Scorpion` (Скорпион). Все скорпионы — животные, но не все животные — скорпионы. На скорпиона можно указать и сказать: «Это животное», — но нельзя указать на любое животное и сказать: «Это скорпион». Кроме того, класс `Scorpion` может иметь функцию `Sting()` (Жалить), а класс `Cow` (Корова) — нет. Именно поэтому нельзя считать любое животное скорпионом — попытка вызвать метод `Sting()` экземпляра другого животного может привести к ошибке.

В *Prospector* мы собираемся использовать карты `Card`, созданные сценарием `Deck`, как если бы они были экземплярами `CardProspector`. Это похоже на ситуацию с группой животных, обращаться с которыми хотелось бы как со скорпионами (но мы уже решили, что это невозможно). Однако мы всегда можем сослаться на скорпиона как на животное. Вот как решается эта проблема: если сразу создать скорпионов и в некоторых функциях обращаться с ними как с животными (что допустимо, потому что скорпион является подклассом животного), тогда выражение `Scorpion s = Animal as Scorpion;` благополучно будет возвращать нужный результат, потому что втайне каждое животное остается скорпионом.

Чтобы проделать то же самое в *Prospector*, вместо компонента Card (Script) к шаблону PrefabCard можно подключить компонент CardProspector (Script). Тогда все функции в классе `Deck` будут ссылаться на экземпляры `CardProspector` как на экземпляры `Card`, и мы сможем обращаться к ним как к экземплярам `CardProspector`, когда это потребуется.

Как было объяснено во врезке, решение проблемы заключается в том, чтобы экземпляры `CardProspector` были экземплярами `CardProspector` и просто маскировались под экземпляры `Card` для кода в классе `Deck`.

5. Чтобы провернуть этот трюк, выберите шаблон **PrefabCard** в панели **Project** (Проект), после чего он появится в инспекторе с компонентом **Card (Script)**.
6. Щелкните на кнопке **Add Component** (Добавить компонент) и в открывшемся меню выберите пункт **Scripts > CardProspector** (Сценарии > CardProspector). Это действие добавит компонент **CardProspector (Script)** в шаблон **PrefabCard**.
7. Чтобы удалить ненужный компонент **Card (Script)**, щелкните на кнопке с изображением шестеренки в правом верхнем углу в разделе **Card (Script)** и в открывшемся меню выберите пункт **Remove Component** (Удалить компонент).

8. Выберите _MainCamera в иерархии и запустите сцену; вы увидите, что теперь вместо null все элементы в drawPile хранят ссылки на экземпляры CardProspector.

   Когда сценарий Deck создает экземпляр шаблона PrefabCard и получает его компонент Card, он не замечает подмены, потому что на экземпляр CardProspector всегда можно сослаться как на экземпляр Card. Далее, когда функция ConvertListCardsToListCardProspectors() попытается выполнить выражение tCP = tCD as CardProspector;, она получит требуемый результат.

9. Сохраните сцену. Вы знаете правила.

Позиционирование карт в раскладке

Теперь у нас есть все что нужно, и мы можем добавить в класс Prospector код, который фактически создает раскладку карт в игре:

```
public class Prospector : MonoBehaviour {
    static public Prospector    S;

    [Header("Set in Inspector")]
    public TextAsset                deckXML;
    public TextAsset                layoutXML;
    public float                    xOffset = 3;
    public float                    yOffset = -2.5f;
    public Vector3                  layoutCenter;

    [Header("Set Dynamically")]
    public Deck                     deck;
    public Layout                   layout;
    public List<CardProspector>     drawPile;
    public Transform                layoutAnchor;
    public CardProspector           target;
    public List<CardProspector>     tableau;
    public List<CardProspector>     discardPile;

    void Awake() { ... }

    void Start () {
        ...
        drawPile = ConvertListCardsToListCardProspectors( deck.cards );
        LayoutGame();
    }

    List<CardProspector> ConvertListCardsToListCardProspectors(List<Card> lCD) {
        ...
    }

    // Функция Draw снимает одну карту с вершины drawPile и возвращает ее
    CardProspector Draw() {
        CardProspector cd = drawPile[0]; // Снять 0-ю карту CardProspector
        drawPile.RemoveAt(0);            // Удалить из List<> drawPile
        return(cd);                      // И вернуть ее
    }
```

```
    // LayoutGame() размещает карты в начальной раскладке - "шахте"
    void LayoutGame() {
        // Создать пустой игровой объект, который будет служить центром раскладки // а
        if (layoutAnchor == null) {
            GameObject tGO = new GameObject("_LayoutAnchor");
            // ^ Создать пустой игровой объект с именем _LayoutAnchor в иерархии
            layoutAnchor = tGO.transform; // Получить его компонент Transform
            layoutAnchor.transform.position = layoutCenter; // Поместить в центр
        }

        CardProspector cp;
        // Разложить карты
        foreach (SlotDef tSD in layout.slotDefs) {
            // ^ Выполнить обход всех определений SlotDef в layout.slotDefs
            cp = Draw(); // Выбрать первую карту (сверху) из стопки drawPile
            cp.faceUp = tSD.faceUp; // Установить ее признак faceUp
                                    //   в соответствии с определением в SlotDef
            cp.transform.parent = layoutAnchor; // Назначить layoutAnchor ее
родителем
            // Эта операция заменит предыдущего родителя: deck.deckAnchor, который
            //   после запуска игры отображается в иерархии с именем _Deck.
            cp.transform.localPosition = new Vector3(
                layout.multiplier.x * tSD.x,
                layout.multiplier.y * tSD.y,
                -tSD.layerID );
            // ^ Установить localPosition карты в соответствии
            //   с определением в SlotDef
            cp.layoutID = tSD.id;
            cp.slotDef = tSD;
            // Карты CardProspector в основной раскладке имеют состояние
            // CardState.tableau
            cp.state = eCardState.tableau;
            tableau.Add(cp); // Добавить карту в список tableau
        }
    }
}
```

Сохраните сценарий и вернитесь в Unity. Запустите игру, и вы увидите, что карты действительно разместились в раскладке, представляющей шахту, которая описана в файле *LayoutXML.xml*, при этом карты в верхнем ряду правильно отображаются лицевой стороной вверх, а остальные — лицевой стороной вниз. Но есть серьезная проблема с наложением (как показано на рис. 32.10).

Нажмите клавишу Option/Alt и, удерживая ее, пощелкайте левой кнопкой мыши на картах в панели Scene (Сцена). Как видите, при использовании двумерных инструментов расстояние от двумерных объектов до камеры не имеет ничего общего с сортировкой объектов по глубине (то есть с правильным отображением одних объектов поверх других). Раньше, когда мы создавали карты, нам просто повезло, потому что мы начинали их конструирование от дальних элементов к ближним, поэтому все эти значки и числа отображались поверх лицевой стороны карты. Однако здесь мы должны внимательнее отнестись к размещению карт, чтобы избежать проблемы, изображенной на рис. 32.10.

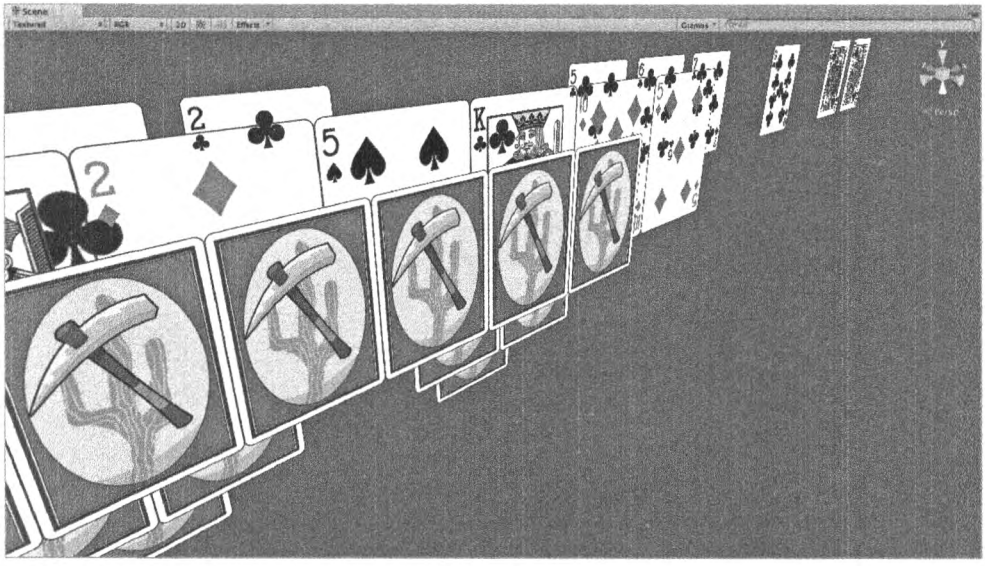


Рис. 32.10. Карты выложены правильно, но наблюдаются проблемы с наложением (и остались карты из первоначальной раскладки)

Unity поддерживает два механизма сортировки двумерных объектов по глубине:

- **Сортировка по слоям:** сортировка по слоям используется для группировки двумерных объектов. Все объекты в нижнем слое отображаются позади объектов в слоях выше. Каждый компонент SpriteRenderer имеет строковую переменную `sortingLayerName`, куда можно записать имя слоя.
- **Сортировка внутри слоя:** каждый компонент SpriteRenderer имеет также переменную `sortingOrder`. Она используется для позиционирования объектов друг относительно друга внутри слоя.

Если сортировка по слоям и внутри слоя не определена, спрайты часто отображаются от нижних к верхним в порядке создания, но это правило соблюдается не всегда. Остановите игру, прежде чем продолжить.

Настройка сортировки по слоям

Чтобы настроить сортировку по слоям, выполните следующие шаги:

1. В главном меню выберите пункт Edit > Project Settings > Tags and Layers (Правка > Параметры проекта > Теги и слои). Вы уже использовали этот инструмент для определения физических слоев и тегов, но мы еще не касались слоев сортировки.
2. Щелкните на пиктограмме с треугольником рядом с именем Sorting Layers (Слои сортировки), чтобы раскрыть этот список, и введите настройки слоев, как показано на рис. 32.11. Чтобы добавить каждый новый слой сортировки, вам при-

дется щелкнуть на кнопке + внизу справа в списке. В данном случае слой Draw, последний в списке, будет отображаться над всеми другими слоями.

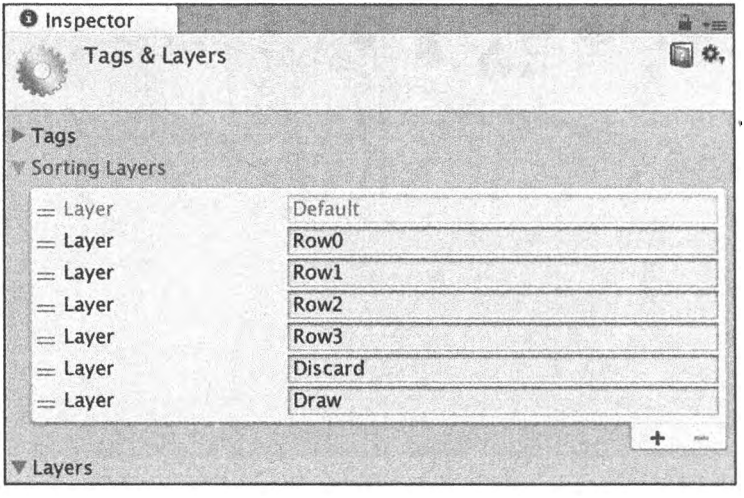


Рис. 32.11. Слои сортировки, необходимые для игры *Prospector*

Компоненты SpriteRenderer и сортировка по глубине потребуются в любой карточной игре, основанной на этой кодовой базе, поэтому добавим все необходимое для сортировки по глубине в класс `Card` (не в подкласс `CardProspector`, который используется только в этой игре).

3. Откройте сценарий Card и добавьте следующий код:

```
public class Card : MonoBehaviour {
    ...
    public CardDefinition   def;  // Извлекается из DeckXML.xml

    // Список компонентов SpriteRenderer этого и вложенных в него игровых объектов
    public SpriteRenderer[]        spriteRenderers;

    void Start() {
        SetSortOrder(0); // Обеспечит правильную сортировку карт
    }

    // Если spriteRenderers не определен, эта функция определит его
    public void PopulateSpriteRenderers() {
        // Если spriteRenderers содержит null или пустой список
        if (spriteRenderers == null || spriteRenderers.Length == 0) {
            // Получить компоненты SpriteRenderer этого игрового объекта
            //   и вложенных в него игровых объектов
            spriteRenderers = GetComponentsInChildren<SpriteRenderer>();
        }
    }
```

```
    // Инициализирует поле sortingLayerName во всех компонентах SpriteRenderer
    public void SetSortingLayerName(string tSLN) {
        PopulateSpriteRenderers();

        foreach (SpriteRenderer tSR in spriteRenderers) {
            tSR.sortingLayerName = tSLN;
        }
    }

    // Инициализирует поле sortingOrder всех компонентов SpriteRenderer
    public void SetSortOrder(int sOrd) {                                    // a
        PopulateSpriteRenderers();

        // Выполнить обход всех элементов в списке spriteRenderers
        foreach (SpriteRenderer tSR in spriteRenderers) {
            if (tSR.gameObject == this.gameObject) {
                // Если компонент принадлежит текущему игровому объекту, это фон
                tSR.sortingOrder = sOrd; // Установить порядковый номер
                                         //   для сортировки в sOrd
                continue; // И перейти к следующей итерации цикла
            }

            // Каждый дочерний игровой объект имеет имя
            // Установить порядковый номер для сортировки, в зависимости от имени
            switch (tSR.gameObject.name) {
                case "back": // если имя "back"
                    // Установить наибольший порядковый номер
                    //   для отображения поверх других спрайтов
                    tSR.sortingOrder = sOrd+2;
                    break;

                case "face": // если имя "face"
                default:     // или же другое
                    // Установить промежуточный порядковый номер
                    //   для отображения поверх фона
                    tSR.sortingOrder = sOrd+1;
                    break;
            }
        }
    }

    public bool faceUp { ... }
}
```

a. Белый фон карты отображается самым нижним (`sOrd`).

Поверх фона отображается все остальное — значки, числа, картинки и т. д. (`sOrd+1`).

Рубашка, когда она видима, должна покрывать все спрайты, образующие карту (`sOrd+2`).

4. В конец метода `LayoutGame()` класса `Prospector` нужно добавить одну строку, чтобы обеспечить распределение карт в основной раскладке по слоям сортировки:

```
public class Prospector : MonoBehaviour {
    ...
    // LayoutGame() размещает карты в начальной раскладке - "шахте"
    void LayoutGame() {
        ...
        foreach (SlotDef tSD in layout.slotDefs) {
            ...
            // Карты CardProspector в основной раскладке имеют состояние
            // CardState.tableau
            cp.state = eCardState.tableau;
            cp.SetSortingLayerName(tSD.layerName); // Назначить слой сортировки

            tableau.Add(cp); // Добавить карту в список tableau
        }
    }
}
```

5. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и запустите сцену.

Теперь карты будут правильно перекрывать друг друга в основной раскладке. Мы еще не собрали остальные карты в стопку свободных карт, но вскоре займемся этим.

Реализация логики игры

Прежде чем начать двигать карты, определим, что может происходить в игре:

A. Если целевая карта замещается любой другой картой, прежняя целевая карта перемещается в стопку сброшенных карт.

B. Верхнюю карту в стопке свободных карт можно открыть и сделать целевой.

C. Открытую карту в основной раскладке, которая на единицу старше или младше целевой, можно сделать новой целевой картой.

D. Если карта, лежащая лицевой стороной вниз, не перекрывается другими картами, она поворачивается лицевой стороной вверх.

E. Игра завершается, когда убирается последняя карта из основной раскладки (победа) или опустошается стопка свободных карт и нет возможных ходов (проигрыш).

Пункты B и C в этом списке связаны с активными действиями — перемещением карт игроком. А пункты A, D и E определяют пассивные события, происходящие в результате действий в пунктах B и C.

Добавление в карты реакции на щелчок мыши

Поскольку все активные действия инициируются щелчком мыши на карте, первым делом добавим в карты возможность реагировать на щелчок.

1. Реакция карт на щелчок мыши необходима в любой карточной игре, поэтому добавим следующий метод в конец класса `Card`:

```
public class Card : MonoBehaviour {
    ...
    public bool faceUp {
        get { ... }
        set { ... }
    }

    // Виртуальные методы могут переопределяться в подклассах
    //   определением методов с теми же именами
    virtual public void OnMouseUpAsButton() {
        print (name); // По щелчку эта строка выведет имя карты
    }
}
```

Если теперь запустить сцену и щелкнуть на какой-нибудь карте, в консоли появится ее имя.

2. Однако в игре *Prospector* щелчок на карте должен выполнять нечто большее, поэтому добавим в конец класса `CardProspector` следующий метод:

```
public class CardProspector : Card { // CardProspector должен расширять Card
    ...
    // Класс SlotDef хранит информацию из элемента <slot> в LayoutXML
    public SlotDef                slotDef;

    // Определяет реакцию карт на щелчок мыши
    override public void OnMouseUpAsButton() {
        // Вызвать метод CardClicked объекта-одиночки Prospector
        Prospector.S.CardClicked(this);
        // а также версию этого метода в базовом классе (Card.cs)
        base.OnMouseUpAsButton();                                    // a
    }
}
```

a. Эта строка вызывает версию `OnMouseUpAsButton()` из базового класса (`Card`), поэтому, кроме вызова нового метода `Prospector.S.CardClicked()` (см. следующий шаг), в ответ на щелчок мыши карты `CardProspectors` все так же будут выводить свои имена в панель Console (Консоль).

3. Дополнительно мы должны добавить в сценарий Prospector метод `CardClicked` (именно из-за его отсутствия только что введенный код подчеркнут красной волнистой линией), но сначала напишем несколько вспомогательных функций. Добавьте в конец класса `Prospector` методы `MoveToDiscard()`, `MoveToTarget()` и `UpdateDrawPile()`.

```
public class Prospector : MonoBehaviour {
    ...
    void LayoutGame() { ... }

    // Перемещает текущую целевую карту в стопку сброшенных карт
    void MoveToDiscard(CardProspector cd) {
```

```
        // Установить состояние карты как discard (сброшена)
        cd.state = eCardState.discard;
        discardPile.Add(cd); // Добавить ее в список discardPile
        cd.transform.parent = layoutAnchor; // Обновить значение transform.parent

        // Переместить эту карту в позицию стопки сброшенных карт
        cd.transform.localPosition = new Vector3(
            layout.multiplier.x * layout.discardPile.x,
            layout.multiplier.y * layout.discardPile.y,
            -layout.discardPile.layerID+0.5f );
        cd.faceUp = true;
        // Поместить поверх стопки для сортировки по глубине
        cd.SetSortingLayerName(layout.discardPile.layerName);
        cd.SetSortOrder(-100+discardPile.Count);
    }

    // Делает карту cd новой целевой картой
    void MoveToTarget(CardProspector cd) {
        // Если целевая карта существует, переместить ее в стопку сброшенных карт
        if (target != null) MoveToDiscard(target);
        target = cd; // cd - новая целевая карта
        cd.state = eCardState.target;
        cd.transform.parent = layoutAnchor;

        // Переместить на место для целевой карты
        cd.transform.localPosition = new Vector3(
            layout.multiplier.x * layout.discardPile.x,
            layout.multiplier.y * layout.discardPile.y,
            -layout.discardPile.layerID );
        cd.faceUp = true; // Повернуть лицевой стороной вверх
        // Настроить сортировку по глубине
        cd.SetSortingLayerName(layout.discardPile.layerName);
        cd.SetSortOrder(0);
    }

    // Раскладывает стопку свободных карт, чтобы было видно, сколько карт осталось
    void UpdateDrawPile() {
        CardProspector cd;
        // Выполнить обход всех карт в drawPile
        for (int i=0; i<drawPile.Count; i++) {
            cd = drawPile[i];
            cd.transform.parent = layoutAnchor;

            // Расположить с учетом смещения layout.drawPile.stagger
            Vector2 dpStagger = layout.drawPile.stagger;
            cd.transform.localPosition = new Vector3(
                layout.multiplier.x * (layout.drawPile.x + i*dpStagger.x),
                layout.multiplier.y * (layout.drawPile.y + i*dpStagger.y),
                -layout.drawPile.layerID+0.1f*i );
            cd.faceUp = false; // повернуть лицевой стороной вниз
            cd.state = eCardState.drawpile;
            // Настроить сортировку по глубине
            cd.SetSortingLayerName(layout.drawPile.layerName);
            cd.SetSortOrder(-10*i);
        }
    }
}
```

4. Добавьте следующий код в конец `Prospector.LayoutGame()`, чтобы нарисовать начальную целевую карту и разложить стопку свободных карт. В этом листинге, в конце класса `Prospector`, также определена начальная версия метода `CardClicked()` для обработки щелчков мыши на всех картах `CardProspector`. Сейчас он лишь перемещает карту из стопки свободных карт на место целевой карты (пункт B из списка выше), но вскоре мы дополним его.

```
public class Prospector : MonoBehaviour {
    ...
    // LayoutGame() размещает карты в начальной раскладке - "шахте"
    void LayoutGame() {
        ...
        foreach (SlotDef tSD in layout.slotDefs) {
            ...
            tableau.Add(cp); // Добавить карту в список tableau
        }

        // Выбрать начальную целевую карту
        MoveToTarget(Draw ());

        // Разложить стопку свободных карт
        UpdateDrawPile();
    }

    // Перемещает текущую целевую карту в стопку сброшенных карт
    void MoveToDiscard(CardProspector cd) { ... }

    void MoveToTarget(CardProspector cd) { ... }
    ...
    void UpdateDrawPile() { ... }

    // CardClicked вызывается в ответ на щелчок на любой карте
    public void CardClicked(CardProspector cd) {
        // Реакция определяется состоянием карты
        switch (cd.state) {
            case eCardState.target:
                // Щелчок на целевой карте игнорируется
                break;

            case eCardState.drawpile:
                // Щелчок на любой карте в стопке свободных карт приводит
                //   к смене целевой карты
                MoveToDiscard(target); // Переместить целевую карту в discardPile
                MoveToTarget(Draw());  // Переместить верхнюю свободную карту
                                       //   на место целевой
                UpdateDrawPile();      // Повторно разложить стопку свободных карт
                break;

            case eCardState.tableau:
                // Для карты в основной раскладке проверяется возможность
                //   ее перемещения на место целевой
                break;
        }
    }
}
```

5. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и запустите сцену.

Теперь вы сможете щелкнуть на стопке свободных карт (в правом верхнем углу экрана), чтобы вытянуть новую целевую карту. Сейчас мы уже близки к завершению игры!

Проверка карт из основной раскладки

Чтобы можно было переместить выбранную карту из основной раскладки, мы должны убедиться, что она на одну ступень старше или младше целевой (конечно же, с учетом циклического переноса старшинства туз/король).

1. Добавьте в метод `CardClicked()` класса `Prospector` следующие строки, выделенные жирным:

```
public class Prospector : MonoBehaviour {
    ...
    // CardClicked вызывается в ответ на щелчок на любой карте
    public void CardClicked(CardProspector cd) {
        // Реакция определяется состоянием карты
        switch (cd.state) {
            ...
            case eCardState.tableau:
                // Для карты в основной раскладке проверяется возможность
                //   ее перемещения на место целевой
                bool validMatch = true;
                if (!cd.faceUp) {
                    // Карта, повернутая лицевой стороной вниз, не может
                    // перемещаться
                    validMatch = false;
                }
                if (!AdjacentRank(cd, target)) {
                    // Если правило старшинства не соблюдается,
                    //   карта не может перемещаться
                    validMatch = false;
                }
                if (!validMatch) return; // Выйти, если карта не может перемещаться

                // Мы оказались здесь: Ура! Карту можно переместить.
                tableau.Remove(cd); // Удалить из списка tableau
                MoveToTarget(cd);   // Сделать эту карту целевой
                break;
        }
    }

    // Возвращает true, если две карты соответствуют правилу старшинства
    //   (с учетом циклического переноса старшинства между тузом и королем)
    public bool AdjacentRank(CardProspector c0, CardProspector c1) {
        // Если любая из карт повернута лицевой стороной вниз,
        //   правило старшинства не соблюдается.
        if (!c0.faceUp || !c1.faceUp) return(false);

        // Если достоинства карт отличаются на 1, правило старшинства соблюдается
```

```
        if (Mathf.Abs(c0.rank - c1.rank) == 1) {
            return(true);
        }
        // Если одна карта - туз, а другая - король, правило старшинства
        // соблюдается
        if (c0.rank == 1 && c1.rank == 13) return(true);
        if (c0.rank == 13 && c1.rank == 1) return(true);

        // Иначе вернуть false
        return(false);
    }
}
```

2. Сохраните сценарий в MonoDevelop и вернитесь в Unity.

Теперь можно играть в игру и даже перемещать карты из верхнего ряда в основной раскладке! Но, поиграв немного, вы заметите, что карты, лежащие лицевой стороной вниз, никогда не переворачиваются. Именно для этой цели мы предусмотрели поле `List<CardProspector> CardProspector.hiddenBy`. В списке `List<int> SlotDef.hiddenBy` хранятся номера слотов других карт, мешающих перевернуть данную, поэтому нам нужно преобразовать эти номера в фактические экземпляры `CardProspector`.

3. Для этого добавьте в класс `Prospector` следующий код:

```
public class Prospector : MonoBehaviour {

    ...
    // LayoutGame() размещает карты в начальной раскладке - "шахте"
    void LayoutGame() {
        ...
        CardProspector cp;
        // Разложить карты
        foreach (SlotDef tSD in layout.slotDefs) {
            ...
            tableau.Add(cp); // Добавить карту в список tableau
        }

        // Настроить списки карт, мешающих перевернуть данную
        foreach (CardProspector tCP in tableau) {
            foreach( int hid in tCP.slotDef.hiddenBy ) {
                cp = FindCardByLayoutID(hid);
                tCP.hiddenBy.Add(cp);
            }
        }

        // Выбрать начальную целевую карту
        MoveToTarget(Draw ());

        // Разложить стопку свободных карт
        UpdateDrawPile();
    }

    // Преобразует номер слота layoutID в экземпляр CardProspector с этим номером
    CardProspector FindCardByLayoutID(int layoutID) {
```

```
        foreach (CardProspector tCP in tableau) {
            // Поиск по всем картам в списке tableau
            if (tCP.layoutID == layoutID) {
                // Если номер слота карты совпадает с искомым, вернуть ее
                return( tCP );
            }
        }
        // Если ничего не найдено, вернуть null
        return( null );
    }

    // Поворачивает карты в основной раскладке лицевой стороной вверх или вниз
    void SetTableauFaces() {
        foreach( CardProspector cd in tableau ) {
            bool faceUp = true; // Предположить, что карта должна быть
                                //   повернута лицевой стороной вверх
            foreach( CardProspector cover in cd.hiddenBy ) {
                // Если любая из карт, перекрывающих текущую, присутствует
                //   в основной раскладке
                if (cover.state == eCardState.tableau) {
                    faceUp = false; // повернуть лицевой стороной вниз
                }
            }
            cd.faceUp = faceUp; // Повернуть карту так или иначе
        }
    }

    // Перемещает текущую целевую карту в стопку сброшенных карт
    void MoveToDiscard(CardProspector cd) { ... }

    // Делает карту cd новой целевой картой
    void MoveToTarget(CardProspector cd) { ... }

    // Раскладывает стопку свободных карт, чтобы было видно, сколько карт осталось
    void UpdateDrawPile() { ... }

    // CardClicked вызывается в ответ на щелчок на любой карте
    public void CardClicked(CardProspector cd) {
        // Реакция определяется состоянием карты
        switch (cd.state) {
            ...
            case eCardState.tableau:
                ...
                // Мы оказались здесь: Ура! Карту можно переместить.
                tableau.Remove(cd); // Удалить из списка tableau
                MoveToTarget(cd);   // Сделать эту карту целевой
                SetTableauFaces();  // Повернуть карты в основной раскладке
                                    //   лицевой стороной вниз или вверх
                break;
        }
    }

    // Возвращает true, если две карты соответствуют правилу старшинства
    //   (с учетом циклического переноса старшинства между тузом и королем)
    public bool AdjacentRank(CardProspector c0, CardProspector c1) { ... }
}
```

Теперь, сохранив сценарии и вернувшись в Unity, вы сможете сыграть полноценный раунд!

4. Следующий наш шаг — проверка завершения игры. Она должна происходить, только когда игрок щелкнет на карте, то есть в конце метода `Prospector.CardClicked()`. Добавьте следующие строки в класс `Prospector`:

```
public class Prospector : MonoBehaviour {
    ...
    // CardClicked вызывается в ответ на щелчок на любой карте
    public void CardClicked(CardProspector cd) {
        // Реакция определяется состоянием карты
        switch (cd.state) {
            ...
                SetTableauFaces();  // Повернуть карты в основной раскладке
                                    //   лицевой стороной вниз или вверх
                break;
        }

        // Проверить завершение игры
        CheckForGameOver();
    }

    // Проверяет завершение игры
    void CheckForGameOver() {
        // Если основная раскладка опустела, игра завершена
        if (tableau.Count==0) {
            // Вызвать GameOver() с признаком победы
            GameOver(true);
            return;
        }

        // Если еще есть свободные карты, игра не завершилась
        if (drawPile.Count>0) {
            return;
        }

        // Проверить наличие допустимых ходов
        foreach ( CardProspector cd in tableau ) {
            if (AdjacentRank(cd, target)) {
                // Если есть допустимый ход, игра не завершилась
                return;
            }
        }

        // Так как допустимых ходов нет, игра завершилась
        // Вызвать GameOver с признаком проигрыша
        GameOver (false);
    }

    // Вызывается, когда игра завершилась. Пока не делает ничего особенного,
    //   но потом мы расширим этот метод
    void GameOver(bool won) {
        if (won) {
            print ("Game Over. You won! :)");
```

```
        } else {
            print ("Game Over. You Lost. :(");
        }
        // Перезагрузить сцену и сбросить игру в исходное состояние
        SceneManager.LoadScene("__Prospector_Scene_0");
    }

    // Возвращает true, если две карты соответствуют правилу старшинства
    //   (с учетом циклического переноса старшинства между тузом и королем)
    public bool AdjacentRank(CardProspector c0, CardProspector c1) { ... }
}
```

5. Сохраните *ВСЕ* сценарии в MonoDevelop, вернитесь в Unity и протестируйте игру.

Теперь в игру можно сыграть, узнать, победили вы или нет, и повторить раунд. Чтобы проверить проигрыш, запустите игру и щелкайте на стопке свободных карт до ее опустошения; после этого сцена должна перезагрузиться, и начнется новый раунд. Для проверки возможности выигрыша может понадобиться несколько попыток. ;-)

А теперь пришло время добавить подсчет очков.

Подсчет очков в Prospector

Оригинальная карточная игра *Prospector* (или *Tri-Peaks*, на которой она основана) не имеет механизма подсчета очков — определяется только победа или проигрыш. Но во всякой цифровой игре полезно подсчитывать очки и сохранять высшие достижения, чтобы у игроков был стимул продолжать играть (с целью побить чужой рекорд).

Правила начисления очков в игре

Мы реализуем несколько способов заработать очки в *Prospector*:

A. Перемещение карты из основной раскладки в позицию целевой карты дает 1 очко.

B. За каждую следующую карту, убранную из основной раскладки без розыгрыша свободной карты, будет даваться на 1 очко больше, чем за предыдущую. То есть если игрок уберет из основной раскладки пять карт подряд, не обращаясь к стопке свободных карт, он заработает в этом *ходе* 1, 2, 3, 4 и 5 очков, всего 15 очков (1 + 2 + 3 + 4 + 5 = 15).

C. Если игрок выиграл раунд, заработанные им очки переносятся в следующий. В случае проигрыша заработанные очки суммируются и сопоставляются со списком высших достижений.

D. Количество очков, заработанных в течение хода (то есть между обращениями к стопке свободных карт), удваивается за каждую золотую карту. Если в при-

мере хода с пятью картами в пункте B игроку встретятся две золотые карты, он заработает 60 очков (15 × 2 × 2 = 60).

Управлять подсчетом очков будет класс `Prospector`, потому что только он знает обо всех условиях, влияющих на подсчет очков. Мы также создадим сценарий Scoreboard, который будет управлять всеми визуальными элементами для отображения очков.

В этой главе мы реализуем пункты A, B и C, а реализацию пункта D я оставляю вам как самостоятельное упражнение.

Создание цепочки подсчета очков

Для подсчета очков в этой игре мы создадим сценарий ScoreManager и подключим его к _MainCamera. Так как правила предусматривают постепенное увеличение очков за каждую следующую карту и даже их удвоение, имеет смысл подсчитывать очки в течение одного хода отдельно и затем добавлять их к общей сумме за раунд, когда ход завершится (обращением к стопке со свободными картами).

1. Создайте в папке __Scripts новый сценарий на C# с именем ScoreManager.
2. Подключите его к _MainCamera.
3. Откройте ScoreManager в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Перечисление со всеми возможными событиями начисления очков
public enum eScoreEvent {
    draw,
    mine,
    mineGold,
    gameWin,
    gameLoss
}

// ScoreManager управляет подсчетом очков
public class ScoreManager : MonoBehaviour {                                    // a
    static private ScoreManager  S;                                            // b

    static public int     SCORE_FROM_PREV_ROUND = 0;
    static public int     HIGH_SCORE = 0;

    [Header("Set Dynamically")]
    // Поля для хранения информации о заработанных очках
    public int            chain = 0;
    public int            scoreRun = 0;
    public int            score = 0;

    void Awake() {
        if (S == null) {                                                       // c
            S = this; // Подготовка скрытого объекта-одиночки
```

```
    } else {
        Debug.LogError("ERROR: ScoreManager.Awake(): S is already set!");
    }

    // Проверить рекорд в PlayerPrefs
    if (PlayerPrefs.HasKey ("ProspectorHighScore")) {
        HIGH_SCORE = PlayerPrefs.GetInt("ProspectorHighScore");
    }
    // Добавить очки, заработанные в последнем раунде
    //  которые должны быть >0, если раунд завершился победой
    score += SCORE_FROM_PREV_ROUND;
    // И сбросить SCORE_FROM_PREV_ROUND
    SCORE_FROM_PREV_ROUND = 0;
}

static public void EVENT(eScoreEvent evt) {                          // d
    try { // try-catch не позволит ошибке аварийно завершить программу
        S.Event(evt);
    } catch (System.NullReferenceException nre) {
        Debug.LogError( "ScoreManager:EVENT() called while S=null.\n"+nre );
    }
}

void Event(eScoreEvent evt) {
    switch (evt) {
        // В случае победы, проигрыша и завершения хода
        //   выполняются одни и те же действия
        case eScoreEvent.draw:     // Выбор свободной карты
        case eScoreEvent.gameWin:  // Победа в раунде
        case eScoreEvent.gameLoss: // Проигрыш в раунде
            chain = 0;             // сбросить цепочку подсчета очков
            score += scoreRun;     // добавить scoreRun к общему числу очков
            scoreRun = 0;          // сбросить scoreRun
            break;

        case eScoreEvent.mine:     // Удаление карты из основной раскладки
            chain++;               // увеличить количество очков в цепочке
            scoreRun += chain;     // добавить очки за карту
            break;
    }

    // Эта вторая инструкция switch обрабатывает победу и проигрыш в раунде
    switch (evt) {
        case eScoreEvent.gameWin:
            // В случае победы перенести очки в следующий раунд
            // статические поля НЕ сбрасываются вызовом
            // SceneManager.LoadScene()
            SCORE_FROM_PREV_ROUND = score;
            print ("You won this round! Round score: "+score);
            break;

        case eScoreEvent.gameLoss:
            // В случае проигрыша сравнить с рекордом
            if (HIGH_SCORE <= score) {
```

```
                print("You got the high score! High score: "+score);
                HIGH_SCORE = score;
                PlayerPrefs.SetInt("ProspectorHighScore", score);
            } else {
                print ("Your final score for the game was: "+score);
            }
            break;

        default:
            print ("score: "+score+" scoreRun:"+scoreRun+" chain:"+chain);
            break;
    }
}

static public int CHAIN { get { return S.chain; } }                    // e
static public int SCORE { get { return S.score; } }
static public int SCORE_RUN { get { return S.scoreRun; } }
}
```

a. В первом издании книги не было отдельного класса `ScoreManager`, а его методы входили в состав класса `Prospector`, но с тех пор я стал строже следовать принципам компонентно-ориентированного подхода к разработке программного обеспечения. Согласно этим принципам, разработчики должны стараться создавать маленькие независимые классы, допускающие возможность многократного использования. Организовав подсчет очков в отдельном классе `ScoreManager`, я получил компонент, который смогу повторно использовать в будущем, и сохранил свой код простым. Узнать больше о принципах компонентно-ориентированного программирования вы сможете в разделе «Шаблоны проектирования программного обеспечения» приложения Б «Полезные идеи», а их широкое применение на практике вы увидите в главе 35 «Прототип 6: Dungeon Delver».

b. Поле `static private ScoreManager S;` — это скрытая версия реализации шаблона проектирования «Одиночка». Большинство объектов-одиночек в этой книге объявляется общедоступными, но этот я сделал скрытым, чтобы доступ к нему имел только класс `ScoreManager`.

c. Эта более сложная процедура присваивания ссылки на объект-одиночку возбуждает ошибку, если два разных экземпляра `ScoreManager` попробуют объявить себя одиночками `S`.

d. Эта статическая общедоступная версия метода `EVENT()` позволяет другим классам (таким, как `Prospector`) посылать события `eScoreEvent` классу `ScoreManager`. Метод `EVENT()`, в свою очередь, вызывает общедоступный нестатический метод `Event()` скрытого объекта-одиночки `ScoreManager S`. Инструкция `try-catch` предупредит, если вызов `EVENT()` произойдет тогда, когда `S` все еще хранит значение `null`.

e. Эти статические свойства открывают доступ только для чтения к общедоступным полям скрытого объекта-одиночки `ScoreManager S`.

4. Добавьте в методы `CardClicked()` и `GameOver()` класса `Prospector` следующие четыре строки, выделенные жирным, чтобы задействовать `ScoreManager`:

```
public class Prospector : MonoBehaviour {
    ...
    // CardClicked вызывается в ответ на щелчок на любой карте
    public void CardClicked(CardProspector cd) {
        // Реакция определяется состоянием карты
        switch (cd.state) {
            ...
            case eCardState.drawpile:
                // Щелчок на любой карте в стопке свободных карт приводит
                //   к смене целевой карты
                MoveToDiscard(target); // Переместить целевую карту в discardPile
                MoveToTarget(Draw());  // Переместить верхнюю свободную карту
                                       //   на место целевой
                UpdateDrawPile();      // Повторно разложить стопку свободных карт
                ScoreManager.EVENT(eScoreEvent.draw);
                break;

            case eCardState.tableau:
                ...
                // Мы оказались здесь: Ура! Карту можно переместить.
                tableau.Remove(cd); // Удалить из списка tableau
                MoveToTarget(cd);   // Сделать эту карту целевой
                SetTableauFaces();  // Повернуть карты в основной раскладке
                                    //   лицевой стороной вниз или вверх
                ScoreManager.EVENT(eScoreEvent.mine);
                break;
        }

        // Проверить завершение игры
        CheckForGameOver();
    }

    // Проверяет завершение игры
    void CheckForGameOver() { ... }

    // Вызывается, когда игра завершилась. Пока не делает ничего особенного,
    //   но потом мы расширим этот метод
    void GameOver(bool won) {
        if (won) {
            // print ("Game Over. You won! :)"); // Закомментируйте эту строку
            ScoreManager.EVENT(eScoreEvent.gameWin);
        } else {
            // print ("Game Over. You Lost. :("); // Закомментируйте эту строку
            ScoreManager.EVENT(eScoreEvent.gameLoss);
        }
        // Перезагрузить сцену и сбросить игру в исходное состояние
        SceneManager.LoadScene("__Prospector_Scene_0");
    }
    ...
}
```

5. Сохраните сценарии в MonoDevelop, вернитесь в Unity и щелкните на кнопке Play (Играть).

Теперь в процессе игры в панель Console (Консоль) будет выводиться информация, сообщающая о количестве заработанных очков. Кроме того, выбрав _MainCamera в иерархии, вы сможете увидеть в разделе ScoreManager (Script), в инспекторе, что в случае победы заработанные вами очки переносятся в следующий раунд. Такой способ информирования хорош для нас, разработчиков, но давайте организуем вывод этой же информации для игроков.

Отображение достижений игроков

Для этой игры мы создадим еще пару компонентов, отображающих заработанные очки. Один из них, класс `Scoreboard`, будет управлять отображением очков на экране, а другой, `FloatingScore`, — представлять число, способное перемещаться по экрану. Также мы воспользуемся функцией `SendMessage()` из библиотеки Unity, которая может вызывать методы по именам и передавать им любые игровые объекты в параметре:

1. Создайте в папке __Scripts новый сценарий на C# с именем FloatingScore и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

// Перечисление со всеми возможными состояниями FloatingScore
public enum eFSState {
    idle,
    pre,
    active,
    post
}

// FloatingScore может перемещаться на экране по траектории,
//   которая определяется кривой Безье
public class FloatingScore : MonoBehaviour {
    [Header("Set Dynamically")]
    public eFSState         state = eFSState.idle;

    [SerializeField]
    protected int           _score = 0;
    public string           scoreString;

    // Свойство score устанавливает два поля, _score и scoreString
    public int score {
        get {
            return(_score);
        }
        set {
            _score = value;
            scoreString = _score.ToString("N0"); // аргумент "N0" требует
                                                 // добавить точки в число
```

```
        // Поищите в интернете по фразе "C# Строки стандартных числовых
        // форматов"
        //   чтобы найти описание форматов, поддерживаемых методом ToString
        GetComponent<Text>().text = scoreString;
    }
}

public List<Vector2>    bezierPts; // Точки, определяющие кривую Безье
public List<float>      fontSizes; // Точки кривой Безье для масштабирования
                                   // шрифта
public float            timeStart = -1f;
public float            timeDuration = 1f;
public string           easingCurve = Easing.InOut; // Функция сглаживания
                                                    // из Utils.cs

// Игровой объект, для которого будет вызван метод SendMessage, когда этот
//   экземпляр FloatingScore закончит движение
public GameObject       reportFinishTo = null;

private RectTransform   rectTrans;
private Text            txt;

// Настроить FloatingScore и параметры движения
// Обратите внимание, что для параметров eTimeS и eTimeD определены
//   значения по умолчанию
public void Init(List<Vector2> ePts, float eTimeS = 0, float eTimeD = 1) {
    rectTrans = GetComponent<RectTransform>();
    rectTrans.anchoredPosition = Vector2.zero;

    txt = GetComponent<Text>();

    bezierPts = new List<Vector2>(ePts);

    if (ePts.Count == 1) { // Если задана только одна точка
        // ...просто переместиться в нее.
        transform.position = ePts[0];
        return;
    }

    // Если eTimeS имеет значение по умолчанию,
    // запустить отсчет от текущего времени
    if (eTimeS == 0) eTimeS = Time.time;
    timeStart = eTimeS;
    timeDuration = eTimeD;

    state = eFSState.pre; // Установить состояние pre - готовность
                          // начать движение
}

public void FSCallback(FloatingScore fs) {
    // Когда SendMessage вызовет эту функцию,
    // она должна добавить очки из вызвавшего экземпляра FloatingScore
    score += fs.score;
}
```

```
    // Update вызывается в каждом кадре
    void Update () {
        // Если этот объект никуда не перемещается, просто выйти
        if (state == eFSState.idle) return;

        // Вычислить u на основе текущего времени и продолжительности движения
        // u изменяется от 0 до 1 (обычно)
        float u = (Time.time - timeStart)/timeDuration;
        // Использовать класс Easing из Utils для корректировки значения u
        float uC = Easing.Ease (u, easingCurve);
        if (u<0) { // Если u<0, объект не должен двигаться.
            state = eFSState.pre;
            txt.enabled = false; // Изначально скрыть число
        } else {
            if (u>=1) { // Если u>=1, выполняется движение
                uC = 1; // Установить uC=1, чтобы не выйти за крайнюю точку
                state = eFSState.post;
                if (reportFinishTo != null) { // Если игровой объект указан
                    // использовать SendMessage для вызова метода FSCallback
                    //   и передачи ему текущего экземпляра в параметре.
                    reportFinishTo.SendMessage("FSCallback", this);
                    // После отправки сообщения уничтожить gameObject
                    Destroy (gameObject);
                } else { // Если не указан
                    // ...не уничтожать текущий экземпляр. Просто оставить его
                    // в покое.
                    state = eFSState.idle;
                }
            } else {
                // Если 0<=u<1, значит, текущий экземпляр активен и движется
                state = eFSState.active;
                txt.enabled = true; // Показать число очков
            }

            // Использовать кривую Безье для перемещения к заданной точке
            Vector2 pos = Utils.Bezier(uC, bezierPts);
            // Опорные точки RectTransform можно использовать для позиционирования
            //   объектов пользовательского интерфейса относительно общего
            //   размера экрана
            rectTrans.anchorMin = rectTrans.anchorMax = pos;
            if (fontSizes != null && fontSizes.Count>0) {
                // Если список fontSizes содержит значения
                // ...скорректировать fontSize этого объекта GUIText
                int size = Mathf.RoundToInt( Utils.Bezier(uC, fontSizes) );
                GetComponent<Text>().fontSize = size;
            }
        }
    }
}
```

2. Создайте в папке __Scripts новый сценарий на C# с именем Scoreboard и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;
using UnityEngine.UI;

// Класс Scoreboard управляет отображением очков игрока
public class Scoreboard : MonoBehaviour {
    public static Scoreboard S; // Это объект-одиночка Scoreboard

    [Header("Set in Inspector")]
    public GameObject     prefabFloatingScore;

    [Header("Set Dynamically")]
    [SerializeField] private int    _score = 0;
    [SerializeField] private string _scoreString;

    private Transform     canvasTrans;

    // Свойство score также устанавливает scoreString
    public int score {
        get {
            return(_score);
        }
        set {
            _score = value;
            scoreString = _score.ToString("N0");
        }
    }

    // Свойство scoreString также устанавливает Text.text
    public string scoreString {
        get {
            return(_scoreString);
        }
        set {
            _scoreString = value;
            GetComponent<Text>().text = _scoreString;
        }
    }

    void Awake() {
        if (S == null) {
            S = this; // Подготовка скрытого объекта-одиночки
        } else {
            Debug.LogError("ERROR: Scoreboard.Awake(): S is already set!");
        }
        canvasTrans = transform.parent;
    }

    // Когда вызывается методом SendMessage, прибавляет fs.score к this.score
    public void FSCallback(FloatingScore fs) {
        score += fs.score;
    }

    // Создает и инициализирует новый игровой объект FloatingScore.
    // Возвращает указатель на созданный экземпляр FloatingScore, чтобы
```

```
    //   вызывающая функция могла выполнить с ним дополнительные операции
    //   (например, определить список fontSizes и т. д.)
    public FloatingScore CreateFloatingScore(int amt, List<Vector2> pts) {
        GameObject go = Instantiate <GameObject> (prefabFloatingScore);
        go.transform.SetParent( canvasTrans );
        FloatingScore fs = go.GetComponent<FloatingScore>();
        fs.score = amt;
        fs.reportFinishTo = this.gameObject; // Настроить обратный вызов
        fs.Init(pts);
        return(fs);
    }
}
```

3. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

Теперь нужно создать игровые объекты для обоих сценариев, Scoreboard и FloatingScore.

Создание шаблона игрового объекта FloatingScore

Чтобы создать шаблон игрового объекта для FloatingScore:

1. В главном меню Unity выберите пункт GameObject > UI > Text (Игровой объект > ПИ > Текст). Переименуйте созданный игровой объект Text в PrefabFloatingScore.
2. Прежде чем изменить любые настройки в PrefabFloatingScore, проверьте, чтобы в раскрывающемся списке соотношения сторон в панели Game (Игра) было выбрано значение Standalone (1024x768) или iPad Wide (1024x768). Это обеспечит соответствие ваших и моих настроек.
3. Настройте PrefabFloatingScore, как показано на рис. 32.12. После этого в середине панели Game (Игра) должно появиться число «ноль».
4. Подключите сценарий FloatingScore к игровому объекту PrefabFloatingScore (перетащив сценарий на объект PrefabFloatingScore в иерархии).
5. Преобразуйте PrefabFloatingScore в шаблон, перетащив объект из иерархии в папку _Prefabs, в панели Project (Проект).
6. Удалите экземпляр PrefabFloatingScore, оставшийся в панели Hierarchy (Иерархия).

Создание игрового объекта Scoreboard

Чтобы создать игровой объект Scoreboard:

1. Создайте в сцене еще один игровой объект Text (GameObject > UI > Text (Игровой объект > ПИ > Текст)).
2. Переименуйте его в Scoreboard.
3. Подключите к объекту Scoreboard сценарий Scoreboard и настройте его, как показано на рис. 32.13. В процессе настройки вы должны перетащить шаблон

PrefabFloatingScore из папки _Prefabs в общедоступное поле prefabFloatingScore компонента Scoreboard (Script).

4. Сохраните сцену.

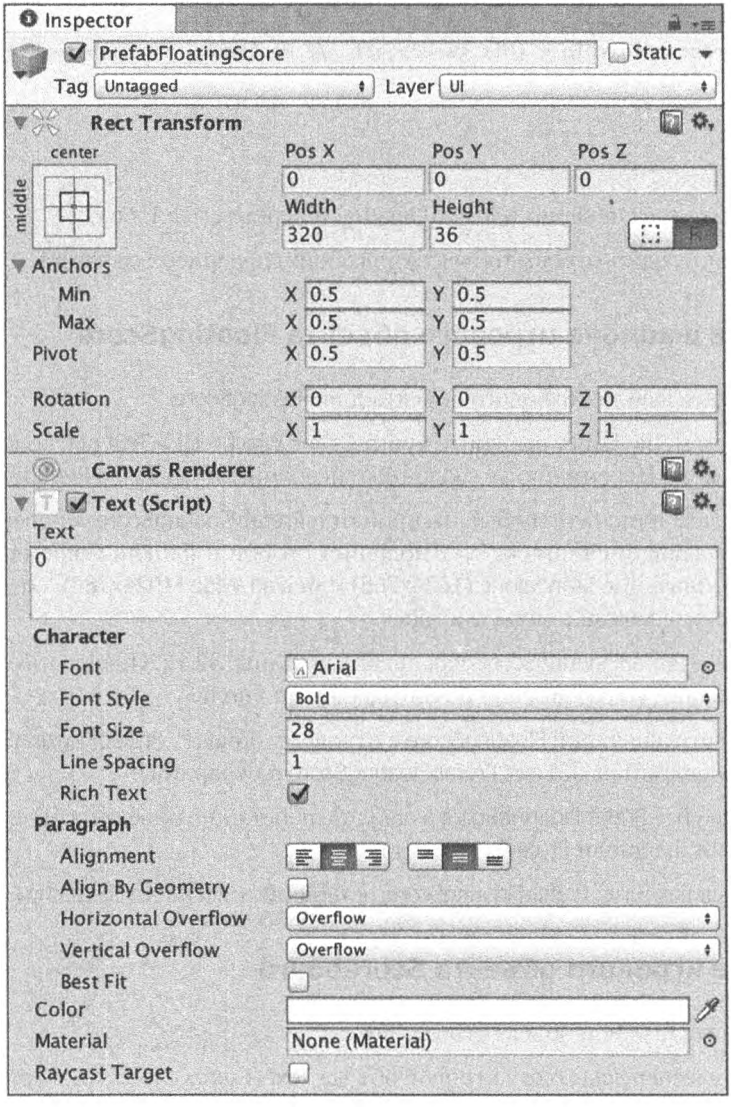


Рис. 32.12. Настройки игрового объекта PrefabFloatingScore

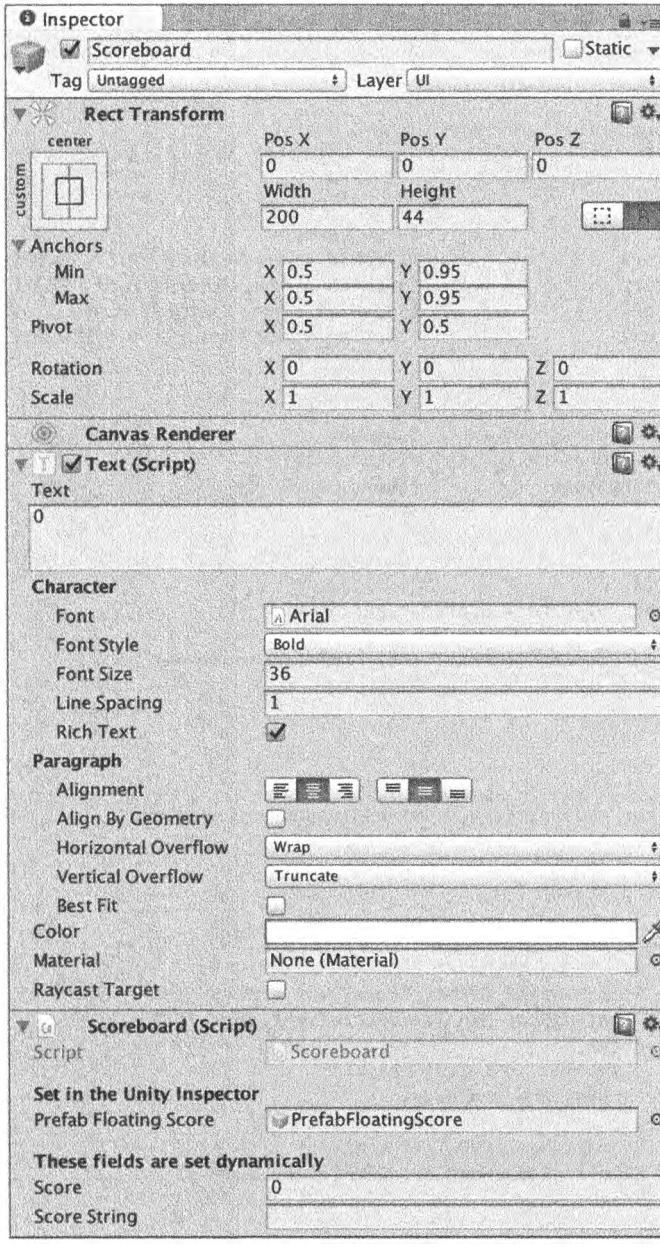


Рис. 32.13. Настройки игрового объекта Scoreboard

5. Теперь осталось только внести изменения в класс `Prospector`, чтобы задействовать новый код и игровые объекты. Добавьте в класс `Prospector` следующие строки, выделенные жирным:

```
public class Prospector : MonoBehaviour {
    ...
    [Header("Set in Inspector")]
    ...
    public Vector3                 layoutCenter;
    public Vector2                 fsPosMid = new Vector2( 0.5f, 0.90f );
    public Vector2                 fsPosRun = new Vector2( 0.5f, 0.75f );
    public Vector2                 fsPosMid2 = new Vector2( 0.4f, 1.0f );
    public Vector2                 fsPosEnd = new Vector2( 0.5f, 0.95f );

    [Header("Set Dynamically")]
    ...
    public List<CardProspector>    tableau;
    public List<CardProspector>    discardPile;
    public FloatingScore           fsRun;

    void Awake() { ... }

    void Start () {
        Scoreboard.S.score = ScoreManager.SCORE;

        deck = GetComponent<Deck>(); // Получить компонент Deck
        ...
    }

    ...

    // CardClicked вызывается в ответ на щелчок на любой карте
    public void CardClicked(CardProspector cd) {
        // Реакция определяется состоянием карты
        switch (cd.state) {
            ...
            case eCardState.drawpile:
                ...
                ScoreManager.EVENT(eScoreEvent.draw);
                FloatingScoreHandler(eScoreEvent.draw);
                break;

            case eCardState.tableau:
                ...
                ScoreManager.EVENT(eScoreEvent.mine);
                FloatingScoreHandler(eScoreEvent.mine);
                break;
        }
        ...
    }

    // Проверяет завершение игры
    void CheckForGameOver() { ... }

    // Вызывается, когда игра завершилась. Пока не делает ничего особенного,
```

```
//   но потом мы расширим этот метод
void GameOver(bool won) {
    if (won) {
        // print ("Game Over. You won! :)"); // Закомментируйте эту строку
        ScoreManager.EVENT(eScoreEvent.gameWin);
        FloatingScoreHandler(eScoreEvent.gameWin);
    } else {
        // print ("Game Over. You Lost. :("); // Закомментируйте эту строку
        ScoreManager.EVENT(eScoreEvent.gameLoss);
        FloatingScoreHandler(eScoreEvent.gameLoss);
    }
    // Перезагрузить сцену и сбросить игру в исходное состояние
    SceneManager.LoadScene("__Prospector_Scene_0");
}

...

// Возвращает true, если две карты соответствуют правилу старшинства
//   (с учетом циклического переноса старшинства между тузом и королем)
public bool AdjacentRank(CardProspector c0, CardProspector c1) { ... }

// Обрабатывает движение FloatingScore
void FloatingScoreHandler(eScoreEvent evt) {
    List<Vector2> fsPts;
    switch (evt) {
        // В случае победы, проигрыша и завершения хода
        //   выполняются одни и те же действия
        case eScoreEvent.draw:     // Выбор свободной карты
        case eScoreEvent.gameWin:  // Победа в раунде
        case eScoreEvent.gameLoss: // Проигрыш в раунде
            // Добавить fsRun в Scoreboard
            if (fsRun != null) {
                // Создать точки для кривой Безье
                fsPts = new List<Vector2>();
                fsPts.Add( fsPosRun );
                fsPts.Add( fsPosMid2 );
                fsPts.Add( fsPosEnd );
                fsRun.reportFinishTo = Scoreboard.S.gameObject;
                fsRun.Init(fsPts, 0, 1);
                // Также скорректировать fontSize
                fsRun.fontSizes = new List<float>(new float[] {28,36,4});
                fsRun = null; // Очистить fsRun, чтобы создать заново
            }
            break;

        case eScoreEvent.mine: // Удаление карты из основной раскладки
            // Создать FloatingScore для отображения этого количества очков
            FloatingScore fs;
            // Переместить из позиции указателя мыши mousePosition в fsPosRun
            Vector2 p0 = Input.mousePosition;
            p0.x /= Screen.width;
            p0.y /= Screen.height;
            fsPts = new List<Vector2>();
            fsPts.Add( p0 );
            fsPts.Add( fsPosMid );
```

```
                fsPts.Add( fsPosRun );
                fs = Scoreboard.S.CreateFloatingScore(ScoreManager.CHAIN, fsPts);
                fs.fontSizes = new List<float>(new float[] {4,50,28});
                if (fsRun == null) {
                    fsRun = fs;
                    fsRun.reportFinishTo = null;
                } else {
                    fs.reportFinishTo = fsRun.gameObject;
                }
                break;
        }
    }
}
```

6. Сохраните сценарии в MonoDevelop и попробуйте сыграть в игру в Unity.

Теперь, играя в игру, вы должны увидеть, как число очков пролетает по экрану. Это очень важно, потому что помогает игроку понять, за какие действия начисляются очки, и осознать механику игры (даже не читая инструкций).

Улучшение оформления игры

Давайте намного улучшим оформление игры, добавив фон. В папке **Materials**, импортированной в проект в начале главы, имеется файл PNG с именем **ProspectorBackground** и материал **ProspectorBackground Mat**. Они уже настроены для вас, потому что теперь, после чтения предыдущих глав, вы знаете, как это делается.

1. В Unity добавьте в сцену квадрат (**GameObject > 3D Object > Quad** (Игровой объект > 3D объект > Квадрат)).
2. Перетащите **ProspectorBackground Mat** из папки **Materials** на этот квадрат.
3. Переименуйте квадрат в **ProspectorBackground** и настройте его компонент **Transform** как показано ниже:

   ```
   ProspectorBackground (Quad) P:[ 0, 0, 0 ] R:[ 0, 0, 0 ] S:[ 26.667, 20, 1 ]
   ```

   Размер 10 поля зрения ортографической камеры **_MainCamera** означает, что расстояние от центра до ближайшего края экрана составляет 10 единиц (в данном случае ближайшими являются верхний и нижний края), а общая высота видимого экрана составляет 20 единиц. Поэтому квадрат **ProspectorBackground** должен иметь высоту (**Scale Y**) 20 единиц. Кроме того, учитывая соотношение сторон экрана 4 : 3, ширину (**Scale X**) следует установить равной 20 / 3 × 4 = 26,667.
4. Сохраните сцену.

   Если теперь запустить игру, она должна выглядеть, как показано на рис. 32.14[1].

[1] Это оригинальное оформление, включая изображение персонажа, фон и рисунок рубашки карт, было создано в 2001 году для моей компании, тогда называвшейся Digital Mercenaries, художником Джимми Товаром (Jimmy Tovar, http://jimmytovar.com).

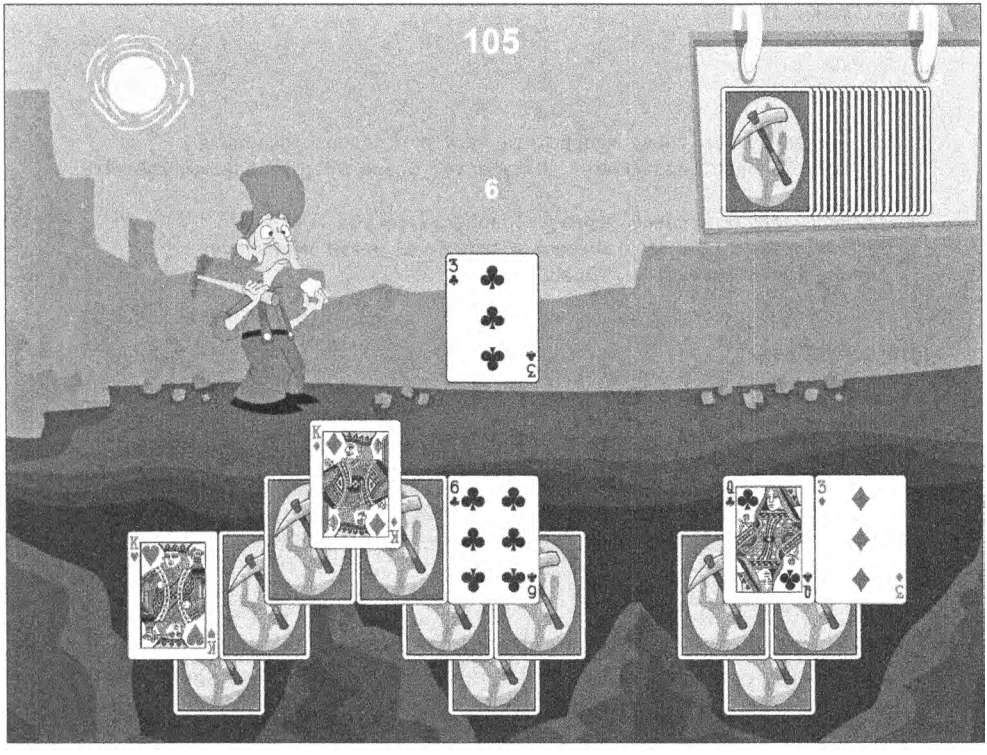


Рис. 32.14. Игра *Prospector* с фоновым рисунком

Объявление о начале и конце раунда

Я уверен, вы заметили, что раунды игры заканчиваются совершенно неожиданно. Давайте сделаем что-нибудь, чтобы исправить ситуацию. Для начала добавим задержку перед перезагрузкой уровня с помощью функции `Invoke()`. Добавьте в класс `Prospector` следующие строки, выделенные жирным:

```
public class Prospector : MonoBehaviour {
    ...
    [Header("Set in Inspector")]
    ...
    public Vector2           fsPosEnd = new Vector2( 0.5f, 0.95f );
    public float             reloadDelay = 2f; // Задержка между раундами 2 секунды

    [Header("Set Dynamically")]

    ...

    // Вызывается, когда игра завершилась. Пока не делает ничего особенного,
    //   но потом мы расширим этот метод
    void GameOver(bool won) {
```

```
        if (won) {
            ...
        } else {
            ...
        }
        // Перезагрузить сцену и сбросить игру в исходное состояние
        // SceneManager.LoadScene("__Prospector_Scene_0"); // Закомментируйте!

        // Перезагрузить сцену через reloadDelay секунд
        // Это позволит числу с очками долететь до места назначения
        Invoke ("ReloadLevel", reloadDelay);                          // a
    }

    void ReloadLevel() {
        // Перезагрузить сцену и сбросить игру в исходное состояние
        SceneManager.LoadScene("__Prospector_Scene_0");
    }

    // Возвращает true, если две карты соответствуют правилу старшинства
    //   (с учетом циклического переноса старшинства между тузом и королем)
    public bool AdjacentRank(CardProspector c0, CardProspector c1) { ... }
    ...
}
```

a. Функция `Invoke()` вызовет функцию с именем `"ReloadLevel"` через `reloadDelay` секунд. Своим поведением она напоминает `SendMessage()`, но выполняет вызов с указанной задержкой. Теперь игра будет приостанавливаться на 2 секунды перед перезагрузкой.

Вывод информации о заработанных очках

В конце каждого раунда мы должны также сообщить игроку, сколько очков он заработал.

1. Добавьте в сцену новый объект Text: выберите Canvas в иерархии, а в главном меню выберите пункт GameObject > UI > Text (Игровой объект > ПИ > Текст).
2. Переименуйте игровой объект Text в GameOver и настройте его, как показано слева на рис. 32.15.
3. Добавьте в сцену еще один объект Text: щелкните правой кнопкой мыши на объекте GameOver в иерархии и в контекстном меню выберите пункт Duplicate (Дублировать).
4. Переименуйте объект GameOver (1) в RoundResult и настройте его, как показано справа на рис. 32.15.
5. Добавьте третий объект Text, как дочерний по отношению к Canvas, и дайте ему имя HighScore.
6. Настройте HighScore, как показано на рис. 32.16.

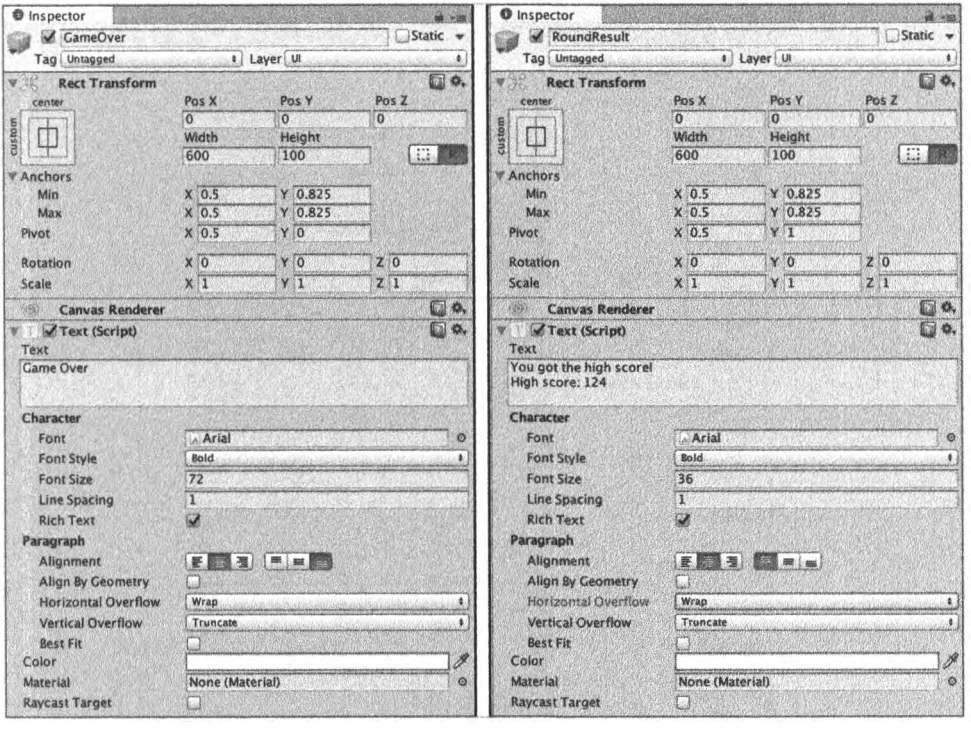


Рис. 32.15. Настройки объектов GameOver и RoundResult

Числовые значения для этих настроек были подобраны методом проб и ошибок. Вы тоже можете поэкспериментировать с ними и подобрать другие значения, которые покажутся вам оптимальными. Согласно этим настройкам, число с рекордом отображается прямо над вывеской справа.

7. Сохраните сцену.
8. Чтобы задействовать все эти надписи, добавьте в класс `Prospector` следующие строки, выделенные жирным:

```
public class Prospector : MonoBehaviour {
    ...
    [Header("Set in Inspector")]
    ...
    public float               reloadDelay = 1f;// Задержка между раундами
    public Text                gameOverText, roundResultText, highScoreText;

    [Header("Set Dynamically")]
    ...
    void Awake() {
        S = this;
        SetUpUITexts();
    }
```

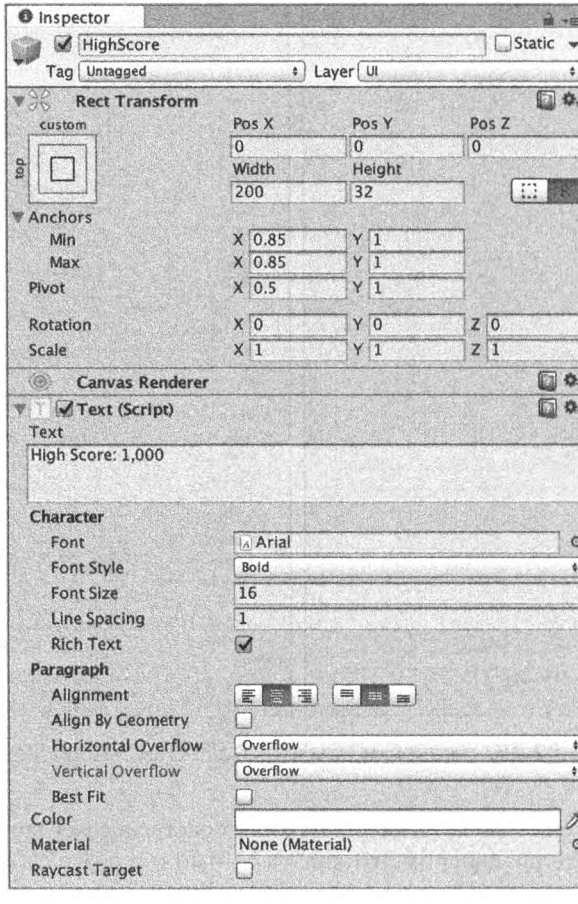


Рис. 32.16. Настройки объекта HighScore

```
void SetUpUITexts() {
    // Настроить объект HighScore
    GameObject go = GameObject.Find("HighScore");
    if (go != null) {
        highScoreText = go.GetComponent<Text>();
    }
    int highScore = ScoreManager.HIGH_SCORE;
    string hScore = "High Score: "+Utils.AddCommasToNumber(highScore);
    go.GetComponent<Text>().text = hScore;

    // Настроить надписи, отображаемые в конце раунда
    go = GameObject.Find ("GameOver");
    if (go != null) {
        gameOverText = go.GetComponent<Text>();
    }

    go = GameObject.Find ("RoundResult");
```

```
        if (go != null) {
            roundResultText = go.GetComponent<Text>();
        }

        // Скрыть надписи
        ShowResultsUI( false );
    }

    void ShowResultsUI(bool show) {
        gameOverText.gameObject.SetActive(show);
        roundResultText.gameObject.SetActive(show);
    }

    ...

    // Вызывается, когда игра завершилась. Пока не делает ничего особенного,
    //   но потом мы расширим этот метод
    void GameOver(bool won) {
        int score = ScoreManager.SCORE;
        if (fsRun != null) score += fsRun.score;
        if (won) {
            gameOverText.text = "Round Over";
            roundResultText.text ="You won this round!\nRound Score: "+score;
            ShowResultsUI( true );
            // print ("Game Over. You won! :)"); // Закомментируйте эту строку
            ScoreManager.EVENT(eScoreEvent.gameWin);
            FloatingScoreHandler(eScoreEvent.gameWin);
        } else {
            gameOverText.text = "Game Over";
            if (ScoreManager.HIGH_SCORE <= score) {
                string str = "You got the high score!\nHigh score: "+score;
                roundResultText.text = str;
            } else {
                roundResultText.text = "Your final score was: "+score;
            }
            ShowResultsUI( true );
            // print ("Game Over. You Lost. :("); // Закомментируйте эту строку
            ScoreManager.EVENT(eScoreEvent.gameLoss);
            FloatingScoreHandler(eScoreEvent.gameLoss);
        }
        // SceneManager.LoadScene("__Prospector_Scene_0"); // Закомментируйте!

        // Перезагрузить сцену через reloadDelay секунд
        // Это позволит числу с очками долететь до места назначения
        Invoke ("ReloadLevel", reloadDelay);
    }

    ...

}
```

9. Сохраните сценарии в MonoDevelop и снова попробуйте сыграть в игру в Unity.

Теперь, по завершении раунда или игры, должны появляться сообщения, как показано на рис. 32.17.

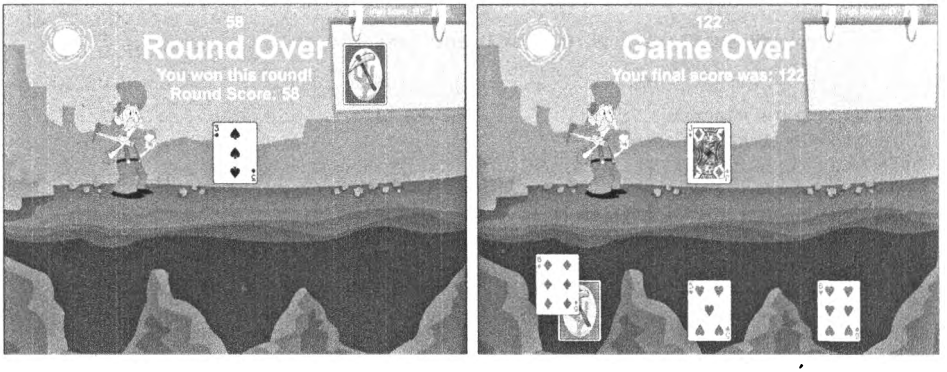


Рис. 32.17. Примеры сообщений в конце раунда и игры

Итоги

В этой главе мы создали законченную карточную игру, которая конструируется на основе XML-файлов, поддерживает подсчет очков, а также имеет красочное оформление фона. Одна из целей учебных примеров в этой книге состоит в том, чтобы дать вам основу для создания других игр. В следующей главе этой стороне уделяется особое внимание. В ней я проведу вас через процесс создания игры *Bartok*, о которой рассказывалось в главе 1, и мы будем использовать код, разработанный в этой главе.

Следующие шаги

Далее перечислены некоторые возможные направления дальнейшего развития этой игры.

Золотые карты

В пункте *D* списка в разделе «Правила начисления очков в игре» я упоминал о золотых картах, но мы не реализовали их в этой главе. В импортированном пакете имеются изображения для создания золотых карт (Card_Back_Gold и Card_Front_Gold). Золотые карты должны удваивать количество очков, заработанных в течение хода. Золотые карты могут встречаться только в основной раскладке, и золотой может быть любая карта в основной раскладке с 10 % вероятностью. Попробуйте реализовать золотые карты самостоятельно.

Компиляция этой игры для мобильного устройства

Настройки проекта этой игры предусматривают компиляцию для iPad, однако обучение сборке игр для конкретных мобильных устройств выходит далеко за рамки этой книги. С другой стороны, в онлайн-руководстве Unity есть несколько

страниц, документирующих этот процесс, и я рекомендую прочитать инструкцию для вашего устройства. А чтобы получить самую свежую информацию по этому вопросу, я советую поискать в интернете по фразе *Unity сборка для*, в конце которой нужно указать имя вашей платформы (например, *Unity сборка для iOS*). В данный момент это могут быть *iOS* или *Android* для мобильных платформ, или *WebGL* для встраивания игры в веб-страницу. Документация к Unity включает страницы «первых шагов» для всех этих платформ.

На личном опыте я убедился, что процесс компиляция для Android выполняется проще и за меньшее время, необходимое для установки и настройки дополнительного программного обеспечения. Чтобы скомпилировать эту игру для iOS, мне потребовалось около двух часов (из которых большая часть была потрачена на настройку учетной записи разработчика Apple iOS и заполнение профиля), а чтобы скомпилировать ее для Android, понадобилось всего 20 минут.

Я также настоятельно рекомендую ознакомиться с инструментами, которые могут помочь в разработке для мобильных устройств. Служба *Test Flight*, которую компания Apple приобрела несколько лет назад, поможет вам в распространении тестовых сборок вашей игры для iOS через интернет (https://developer.apple.com/testflight/). Этой службой пользуются почти все разработчики для iOS. Желающим заниматься кросс-платформенной разработкой можно посоветовать службу *TestFairy* (http://testfairy.com), которая поддерживает также Android (но менее удобна для iOS).

Кроме того, очень советую обратить внимание на *Unity Cloud Build*, прежде независимую компанию *Tsugi* (упоминалась в первом издании книги). *Unity Cloud Build* проверит наличие изменений в вашем коде в репозитории *Unity Collaborate* (или другом) и автоматически скомпилирует новые версии, если обнаружит, что что-то изменилось. Если вы занимаетесь кроссплатформенной разработкой для мобильных устройств или WebGL, *Unity Cloud Build* сэкономит вам массу времени, перенося выполнение тяжелых задач, связанных с компиляцией, на высокопроизводительный сервер и избавляя от них ваш персональный компьютер.

33 Прототип 5: BARTOK

Эта глава отличается от предыдущих учебных примеров, потому что вместо создания нового проекта с нуля в ней демонстрируется, как можно сконструировать новую игру на основе прототипов, разработанных выше в этой книге.

Прежде чем начать работу над этим проектом, дочитайте до конца главу 32 «Прототип 4: Prospector Solitaire» и разберитесь в особенностях работы фреймворка карточных игр, разработанного в ней.

Bartok — это первая игра, с которой вы встретились в этой книге в главе 1 «Думать как дизайнер». Теперь вы сконструируете ее своими руками.

Начало: прототип 5

На этот раз вместо загрузки пакета с ресурсами для Unity, как в предыдущей главе, просто скопируйте целиком папку проекта игры *Prospector* из предыдущей главы (можете также загрузить ее с сайта книги http://book.prototools.net, из раздела Chapter 33). Мы снова будем использовать графические ресурсы, созданные на основе общедоступного комплекта карт *Vectorized Playing Cards 1.3* Криса Агиляра[1].

Описание игры Bartok

Подробное описание игры *Bartok* и правил игры в нее вы найдете в главе 1, где я использовал эту игру в качестве примера. Вкратце: *Bartok* очень напоминает коммерческую игру *Uno*, только в традиционной карточной игре *Bartok* используется полная колода карт и победитель каждого раунда может добавить в игру новое правило. В пример из главы 1 я также включил три варианта правил, но мы не будем создавать их в этой главе, я оставляю это вам как самостоятельное упражнение.

---

[1] Vectorized Playing Cards 1.3 (http://code.google.com/p/vectorized-playing-cards/), ©2011, Chris Aguilar.

Чтобы сыграть в онлайн-версию игры *Bartok*, зайдите на сайт http://book.prototools.net и найдите игру в разделе Chapter 1.

Создание новой сцены

Как и многое другое в этом проекте, сцена основана на сцене, созданной в проекте *Prospector*.

1. Выберите __Prospector_Scene_0 в панели Project (Проект) и в главном меню выберите пункт Edit > Duplicate (Правка > Дублировать). В результате будет создана новая сцена с именем __Prospector_Scene_1.
2. Переименуйте новую сцену в __Bartok_Scene_0 и откройте ее двойным щелчком. Открытие сцены должно подтвердить появление имени сцены __Bartok_Scene_0 в заголовке окна Unity и в начале списка в панели Hierarchy (Иерархия).

   Теперь избавимся от всего, что не понадобится в игре *Bartok*.
3. Выберите _Scoreboard и HighScore внутри объекта Canvas в панели Hierarchy (Иерархия) и удалите их (пункт меню Edit > Delete (Правка > Удалить)). В этой игре не ведется подсчет очков, соответственно, эти компоненты нам не понадобятся.
4. Также удалите из сцены объекты GameOver и RoundResult, вложенные в Canvas. Мы используем их позднее и, когда понадобится, скопируем из __Prospector_Scene_0.
5. Выберите _MainCamera и удалите компоненты Prospector (Script), ScoreManager (Script) и Layout (Script), щелкнув правой кнопкой на каждом (или на кнопке с шестеренкой правее имени), и в контекстном меню выберите пункт Remove Component (Удалить компонент). У вас должен остаться только объект _MainCamera, который уже имеет все необходимые настройки для Transform и Camera, с подключенным к нему компонентом Deck (Script).
6. Измените фон. Выберите игровой объект ProspectorBackground в панели Hierarchy (Иерархия) — не Texture2D в панели Project (Проект) — и переименуйте его в BartokBackground.
7. Создайте в папке Materials новый материал (пункт меню Assets > Create > Material (Ресурсы > Создать > Материал)) с именем BartokBackground Mat. Перетащите новый материал на объект BartokBackground в иерархии. Обратите внимание, что после этого изображение в панели Game (Игра) должно потемнеть. (Это объясняется тем, что новый материал имеет шейдер Standard, тогда как для предыдущего материала использовался шейдер Unlit.)
8. Чтобы исправить этот недостаток, добавьте в сцену источник направленного света (GameObject > Light > Directional Light (Игровой объект > Освещение > Направленное освещение)). Настройте компоненты Transform и Directional Light объекта BartokBackground, как показано ниже:

```
BartokBackground (Quad)  P:[ 0, 0, 1 ]       R:[ 0, 0, 0 ]     S:[ 26.667, 20, 1 ]
Directional Light        P:[ -100, -100, 0 ] R:[ 50, -30, 0 ]  S:[ 1, 1, 1 ]
```

На этом настройку сцены можно считать завершенной. Имейте в виду, что местоположение источника направленного света Directional Light в сцене не имеет никакого значения (значение имеет только угол поворота), но он обеспечивает необходимую освещенность в панели Scene (Сцена). Сохраните сцену.

Важность добавления анимации карт

Эта игра предназначена для одного игрока, но сама игра *Bartok* лучше всего работает, когда в ней принимают участие четверо, поэтому роль трех других игроков возьмет на себя искусственный интеллект (ИИ). Так как *Bartok* имеет очень простые правила, в ней не нужен особенно сложный ИИ. Работая над пошаговыми играми с несколькими игроками, особенно когда роль оппонентов играет ИИ, — вы должны ясно показать игроку, чей черед ходить и как ходят другие игроки. Для этого в данной игре мы будем перемещать карты с места на место, используя анимационный эффект. В игре *Prospector* в этом не было необходимости, потому что все действия выполнял сам игрок и результат был для него очевиден. Однако в *Bartok* будут принимать участие еще три игрока со своими картами, повернутыми лицевой стороной вниз, поэтому анимационный эффект послужит важным способом сообщения действий, выполняемых ИИ.

Наибольшую сложность в проектировании этого учебного примера представляет реализация хороших анимационных эффектов и ожидание их завершения перед следующим ходом. Поэтому в данном проекте вы столкнетесь с использованием функций `SendMessage()` и `Invoke()`, а также с применением более специализированных объектов обратных вызовов, чем позволяет `SendMessage()`. Мы будем передавать экземпляр класса C# объекту и затем вызывать *функцию обратного вызова* экземпляра, когда объект завершит перемещение. Это менее гибкое решение, чем `SendMessage()`, но более быстрое и специализированное, что также позволяет использовать классы C#, не наследующие класс `MonoBehaviour`.

Настройка сборки

В отличие от предыдущего проекта, который создавался как мобильное приложение, этот проект предусматривает сборку в виде онлайн-игры WebGL или автономного приложения для Mac или PC, поэтому нужно изменить параметры сборки.

1. В главном меню выберите пункт File > Build Settings (Файл > Параметры сборки), после чего откроется диалог, изображенный на рис. 33.1.

   Вы увидите, что сцена __Prospector_Scene_0 присутствует в списке Scenes in Build (Сцены для сборки), а __Bartok_Scene_0 — нет.

2. Щелкните на кнопке Add Open Scenes (Добавить открытые сцены), чтобы добавить __Bartok_Scene_0 в список сцен для этой сборки.

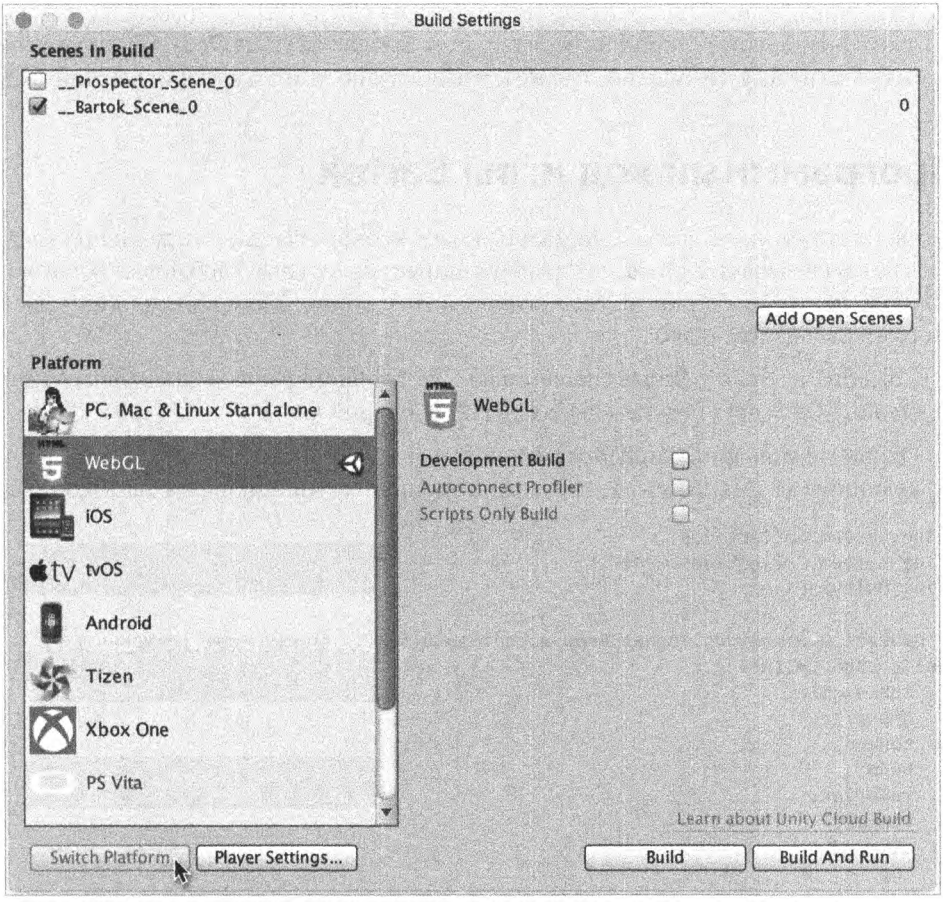


Рис. 33.1. Диалог Build Settings (Параметры сборки)

3. Снимите флажок рядом со сценой __Prospector_Scene_0, чтобы удалить ее из списка сцен. После этого список Scenes in Build (Сцены для сборки) должен выглядеть, как показано на рис. 33.1.

4. Если вы уже установили поддержку инструментов WebGL с помощью мастера установки Unity, выберите WebGL в списке платформ; если нет — выберите PC, Mac & Linux Standalone. Затем щелкните на кнопке Switch Platform (Выбрать платформу).

   Когда переключение на выбранную платформу завершится, кнопка Switch Platform (Выбрать платформу) приобретет неактивный вид. На это может уйти одна-две секунды. Все остальные настройки оставьте как есть.

   Выполнив настройки, как показано на рис. 33.1, закройте диалог. (Не щелкайте пока на кнопке Build (Собрать), вы сделаете это потом, когда закончите создание игры.)

5. Теперь взгляните на заголовок панели Game (Игра). В раскрывающемся списке соотношений сторон выберите пункт Standalone (1024x768). Благодаря этому окно игры у вас будет выглядеть, как показано на иллюстрациях в этой главе.

Программный код игры Bartok

В предыдущем примере мы написали класс `Prospector` для управления игрой и класс `CardProspector:Card`, который расширяет класс `Card` и добавляет некоторые отличительные особенности. Аналогично, в этом примере нам понадобятся классы `Bartok` и `CardBartok:Card`.

1. Создайте в папке _Scripts сценарии на C# с именами Bartok и CardBartok (Assets > Create > C# Script (Ресурсы > Создать > Сценарий C#)).
2. Откройте сценарий CardBartok в MonoDevelop двойным щелчком и введите следующий код. (Часть этого кода можно скопировать из сценария CardProspector.)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// CBState включает состояния игры и состояния to..., описывающие движения // a
public enum CBState {
    toDrawpile,
    drawpile,
    toHand,
    hand,
    toTarget,
    target,
    discard,
    to,
    idle
}

public class CardBartok : Card {                                          // b
    // Статические переменные совместно используются всеми экземплярами CardBartok
    static public float       MOVE_DURATION = 0.5f;
    static public string      MOVE_EASING = Easing.InOut;
    static public float       CARD_HEIGHT = 3.5f;
    static public float       CARD_WIDTH = 2f;

    [Header("Set Dynamically: CardBartok")]
    public CBState            state = CBState.drawpile;

    // Поля с информацией, необходимой для перемещения и поворачивания карты
    public List<Vector3>      bezierPts;
    public List<Quaternion>   bezierRots;
    public float              timeStart, timeDuration;

    // По завершении перемещения карты будет вызываться
    // reportFinishTo.SendMessage()
```

```
public GameObject         reportFinishTo = null;

// MoveTo запускает перемещение карты в новое местоположение с заданным
// поворотом
public void MoveTo(Vector3 ePos, Quaternion eRot) {
    // Создать новые списки для интерполяции.
    // Траектории перемещения и поворота определяются двумя точками каждая.
    bezierPts = new List<Vector3>();
    bezierPts.Add ( transform.localPosition ); // Текущее местоположение
    bezierPts.Add ( ePos );                    // Новое местоположение

    bezierRots = new List<Quaternion>();
    bezierRots.Add ( transform.rotation );     // Текущий угол поворота
    bezierRots.Add ( eRot );                   // Новый угол поворота

    if (timeStart == 0) {                                                  // c
        timeStart = Time.time;
    }
    // timeDuration всегда получает одно и то же значение, но потом
    //   это можно исправить
    timeDuration = MOVE_DURATION;
    state = CBState.to;                                                    // d
}

public void MoveTo(Vector3 ePos) {                                         // e
    MoveTo(ePos, Quaternion.identity);
}

void Update() {
    switch (state) {
        case CBState.toHand:                                               // f
        case CBState.toTarget:
        case CBState.toDrawpile:
        case CBState.to:
            float u = (Time.time - timeStart)/timeDuration;                // g
            float uC = Easing.Ease (u, MOVE_EASING);
            if (u<0) {                                                     // h
                transform.localPosition = bezierPts[0];
                transform.rotation = bezierRots[0];
                return;
            } else if (u>=1) {                                             // i
                uC = 1;
                // Перевести из состояния to... в соответствующее
                //   следующее состояние
                if (state == CBState.toHand) state = CBState.hand;
                if (state == CBState.toTarget) state = CBState.target;
                if (state == CBState.toDrawpile) state = CBState.drawpile;
                if (state == CBState.to) state = CBState.idle;

                // Переместить в конечное местоположение
                transform.localPosition = bezierPts[bezierPts.Count-1];
                transform.rotation = bezierRots[bezierPts.Count-1];

                // Сбросить timeStart в 0, чтобы в следующий раз
```

```
                //   можно было установить текущее время
                timeStart = 0;

                if (reportFinishTo != null) {                          // j
                    reportFinishTo.SendMessage("CBCallback", this);
                    reportFinishTo = null;
                } else { // Если ничего вызывать не надо
                    // Оставить все как есть.
                }
            } else { // Нормальный режим интерполяции (0 <= u < 1)     // k
                Vector3 pos = Utils.Bezier(uC, bezierPts);
                transform.localPosition = pos;
                Quaternion rotQ = Utils.Bezier(uC, bezierRots);
                transform.rotation = rotQ;
            }
            break;
        }

    }
}
```

a. Перечисление `CBState` включает возможные состояния карты `CardBartok` и состояния `to...`, представляющие разные этапы в течение воспроизведения анимационного эффекта.

b. `CardBartok` расширяет `Card` так же, как класс `CardProspector`.

c. Если переменная `timeStart` хранит 0, записать в нее текущее время (чтобы немедленно начать перемещение), иначе перемещение начнется в момент `timeStart`. То есть если прежде в переменную `timeStart` было записано значение, отличное от 0, это значение не затирается. Это позволит нам синхронизировать разные анимационные эффекты.

d. Первоначально устанавливается состояние `CBState.to`. Позднее вызывающий метод определит фактическое состояние, `CBState.toHand` или `CBState.toTarget`.

e. Перегруженная версия `MoveTo()`, которая не требует передачи угла поворота.

f. Так как инструкция `switch` допускает возможность «проваливания» между вариантами, если они не разделяются никаким дополнительным кодом, все состояния `to...` (то есть `toHand`, `toTarget` и т. д.), когда карта перемещается из одного местоположения в другое, обрабатываются одним и тем же блоком кода.

g. Значение переменной `float u` интерполируется в диапазоне от 0 до 1 в процессе перемещения этой карты `CardBartok`. Оно вычисляется как отношение времени, прошедшего с момента `timeStart`, к желаемой продолжительности перемещения (например, если `timeStart` = 5, `timeDuration` = 10 и `Time.time` = 11, тогда `u` = (11-5) / 10 = 0.6). Затем полученное значение `u` передается

в метод `Easing.Ease()`, в сценарии *Utils.cs*, для вычисления значения `uC`, чтобы обеспечить более естественный эффект перемещения карты. Дополнительную информацию вы найдете в разделе «Сглаживание для линейной интерполяции» приложения Б.

h. Обычно `u` принимает значения в диапазоне от 0 до 1. Здесь обрабатывается ситуация, когда `u < 0`. В этом случае карта должна оставаться без движения в своей текущий позиции. Случай `u < 0` возможен, когда переменной `timeStart` присвоено время в будущем, чтобы начать перемещение с задержкой.

i. Если `u >= 1`, значение `u` нужно усечь до 1, чтобы карта не переместилась дальше заданной точки. Также это значение соответствует моменту, когда движение должно остановиться переключением в другое состояние `CBState`.

j. Если задан игровой объект для обратного вызова, тогда с помощью `SendMessage()` необходимо вызвать метод `CBCallback` с параметром `this`. После вызова `SendMessage()` в `reportFinishTo` нужно записать `null`, чтобы в будущем эта карта `CardBartok` не посылала сообщения о своем перемещении этому же игровому объекту.

k. Когда `0 <= u < 1`, просто интерполировать перемещение из текущей точки в следующую. Для вычисления новых координат используется функция интерполяции вдоль кривой Безье. Вычисление нового местоположения и нового угла поворота выполняется отдельно разными перегруженными версиями метода `Utils.Bezier()`. Дополнительную информацию вы найдете в разделе «Кривые Безье» приложения Б.

Большая часть этого кода является адаптацией и расширением версии из предыдущей главы, использовавшейся для перемещения экземпляров класса `FloatingScore`. Версия в `CardBartok` также интерполирует экземпляры `Quaternion` (класса, описывающего поворот), что важно для игры *Bartok*, где карты размещаются на игровом поле, как если бы их держали в руках четыре игрока.

3. Откройте сценарий **Bartok** и введите следующий код. Первым делом убедимся, что класс `Deck` без ошибок создает все 52 карты:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class Bartok : MonoBehaviour {
    static public Bartok S;

    [Header("Set in Inspector")]
    public TextAsset            deckXML;
    public TextAsset            layoutXML;
    public Vector3              layoutCenter = Vector3.zero;
```

```
    [Header("Set Dynamically")]
    public Deck                 deck;
    public List<CardBartok>     drawPile;
    public List<CardBartok>     discardPile;

    void Awake() {
        S = this;
    }

    void Start () {
        deck = GetComponent<Deck>();  // Получить компонент Deck
        deck.InitDeck(deckXML.text);  // Передать ему DeckXML
        Deck.Shuffle(ref deck.cards); // Перетасовать колоду              // a
    }
}
```

4. Ключевое слово `ref` сообщает, что в параметре передается ссылка на `deck.cards`; это позволит методу `Deck.Shuffle()` перетасовать карты непосредственно в `deck.cards`.

Как видите, по большей части это тот же код, что мы видели в *Prospector*, только вместо класса `CardProspector` отдельные карты представляет класс `CardBartok`.

Настройка PrefabCard в инспекторе

На этот раз нам нужно настроить в инспекторе еще несколько параметров шаблона PrefabCard.

1. Выберите PrefabCard в папке _Prefabs, в панели Project (Проект).
2. В компоненте Box Collider установите флажок Is Trigger.
3. В поле Size.z компонента Box Collider введите значение 0.1.
4. Добавьте в шаблон PrefabCard компонент Rigidbody (Component > Physics > Rigidbody (Компонент > Физика > Твердое тело)).
5. Сбросьте флажок Use Gravity в компоненте Rigidbody.
6. Установите флажок Is Kinematic в компоненте Rigidbody.

   После этого настройки компонентов Box Collider и Rigidbody в шаблоне PrefabCard должны выглядеть так, как показано на рис. 33.2.
7. Замените существующий компонент CardProspector (Script) новым компонентом CardBartok (Script).
   a. Щелкните на кнопке с шестеренкой справа от имени компонента CardProspector (Script) и в открывшемся меню выберите пункт Remove Component (Удалить компонент).
   b. Подключите к PrefabCard сценарий CardBartok.

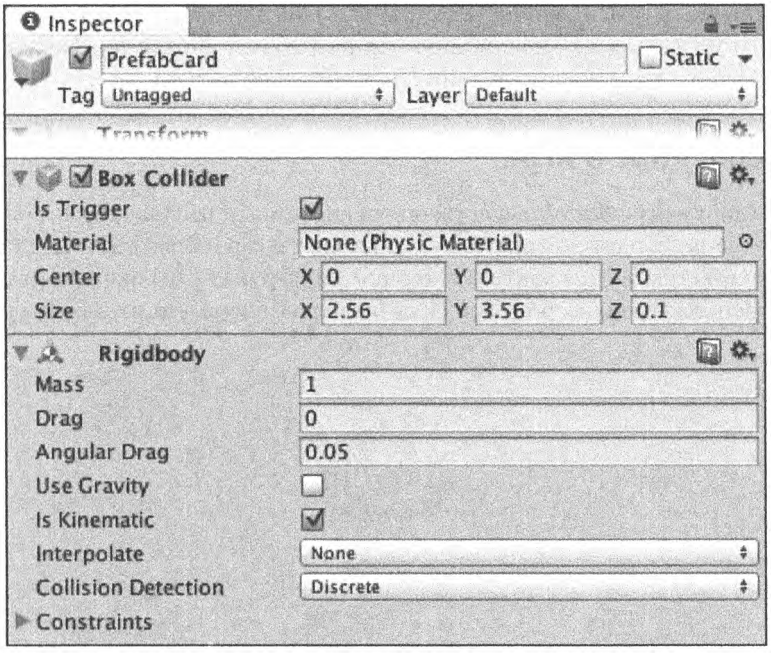


Рис. 33.2. Настройки компонентов Box Collider b Rigidbody в шаблоне PrefabCard

Настройка _MainCamera в инспекторе

Выполните следующие шаги, чтобы настроить _MainCamera в инспекторе:

1. Подключите сценарий Bartok к объекту _MainCamera в иерархии (выберите способ, который вам больше нравится; сейчас вы уже должны знать, как это делается).
2. В панели Hierarchy (Иерархия) выберите _MainCamera. Внизу в инспекторе должен появиться вновь подключенный компонент Bartok (Script). (При желании его можно переместить вверх, для чего щелкните на кнопке с шестеренкой рядом с именем компонента и в открывшемся меню выберите пункт Move Up (Переместить вверх).)
3. Перетащите файл DeckXML из папки Resources в панели Project (Проект) на поле DeckXML компонента Bartok (Script). (Так как состав колоды не изменился (в ней все так же 13 карт каждой из 4 мастей), это тот же файл, что использовался в игре *Prospector*.)
4. Установите флажок startFaceUp в компоненте Deck (Script). Благодаря этому после щелчка на кнопке Play (Играть) все карты появятся лицевой стороной вверх.

Если теперь щелкнуть на кнопке Play (Играть), на экране должна появиться сетка из карт, которую вы уже видели, начиная разработку *Prospector*. Перевернув несколько страниц, мы смогли продвинуться довольно далеко.

Раскладка карт в игре

Раскладка карт в игре *Bartok* существенно отличается от раскладки карт в *Prospector*. В центре экрана находятся стопки свободных карт и карт сброса, а вверху, слева, внизу и справа располагаются карты, находящиеся на руках у игроков. Карты на руках должны располагаться веером, как если бы игроки держали их в руке (рис. 33.3).

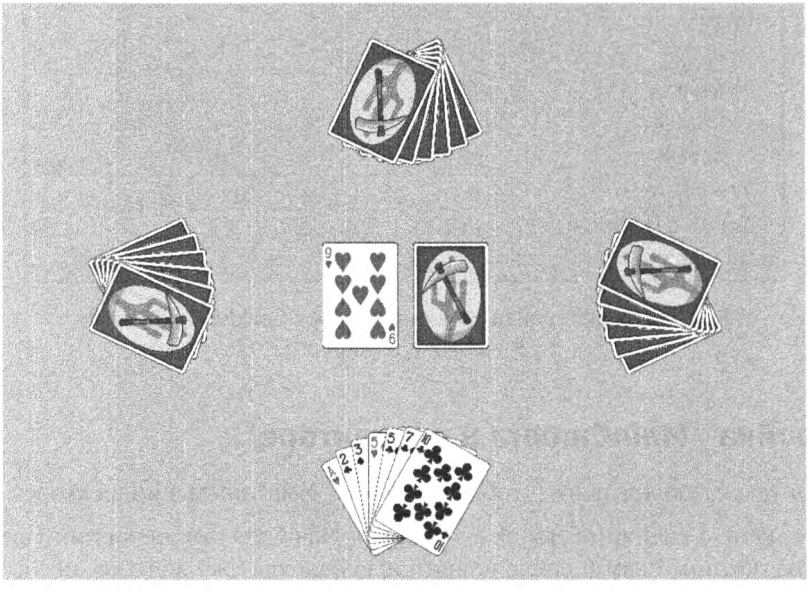


Рис. 33.3. Раскладка карт в игре *Bartok*

Для этого потребуется другой документ XML, описывающий раскладку, отличный от использовавшегося в игре *Prospector*.

1. Выберите LayoutXML в папке Resources, в панели Project (Проект) и создайте копию (Edit > Duplicate (Правка > Дублировать)).
2. Дайте копии имя BartokLayoutXML и введите в этот файл следующий текст. Жирным выделены строки, отличающиеся от строк в исходном файле LayoutXML. Обязательно удалите строки, отсутствующие в этом листинге.

```
<xml>
    <!-- Этот файл хранит информацию о раскладке карт в карточной игре Bartok. -->

    <!-- Элемент multiplier имеет атрибуты x и y с множителями. -->
```

```
    <!-- Множители определяют, насколько свободной или плотной будет раскладка. -->
    <multiplier x="1" y="1" />

    <!-- Позиция стопки свободных карт и смещение относительно
         друг друга -->
    <slot type="drawpile" x="1.5" y="0" xstagger="0.05" layer="1"/>

    <!-- Позиция стопки сброшенных карт и целевой карты -->
    <slot type="discardpile" x="-1.5" y="0" layer="2"/>

    <!-- Позиция целевой карты -->
    <slot type="target" x="-1.5" y="0" layer="4"/>

    <!-- Следующие слоты определяют позиции карт в руках каждого из
         четырех игроков -->
    <slot type="hand" x="0" y="-8" rot="0" player="1" layer="3"/>
    <slot type="hand" x="-10" y="0" rot="270" player="2" layer="3"/>
    <slot type="hand" x="0" y="8" rot="180" player="3" layer="3"/>
    <slot type="hand" x="10" y="0" rot="90" player="4" layer="3"/>
</xml>
```

Сценарий BartokLayout

Мы также должны переписать класс, раскладывающий карты и использующий анимационные эффекты для перемещения карт.

1. Создайте в папке Scripts новый сценарий на C# с именем BartokLayout и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[System.Serializable]                                                    // a
public class SlotDef {                                                   // b
    public float        x;
    public float        y;
    public bool         faceUp = false;
    public string       layerName = "Default";
    public int          layerID = 0;
    public int          id;
    public List<int>    hiddenBy = new List<int>(); // Не используется в Bartok
    public float        rot;        // поворот в зависимости от игрока
    public string       type = "slot";
    public Vector2      stagger;
    public int          player;     // порядковый номер игрока
    public Vector3      pos;        // вычисляется на основе x, y и multiplier
}

public class BartokLayout : MonoBehaviour {
    // Пока оставьте определение класса пустым
}
```

a. `[System.Serializable]` делает `SlotDef` видимым в инспекторе Unity.

b. Класс `SlotDef` не наследует `MonoBehaviour`, поэтому его не требуется определять в отдельном файле.

2. Сохраните сценарий и вернитесь в Unity.

Вы должны заметить, что в консоли появилось сообщение об ошибке:

```
error CS0101: The namespace 'global::' already contains a definition for 'SlotDef'.[1]
```

Это объясняется конфликтом общедоступного класса `SlotDef` из сценария Layout (из игры *Prospector*) с общедоступным классом `SlotDef` в новом сценарии BartokLayout.

3. Удалите сценарий Layout полностью или откройте его в MonoDevelop и закомментируйте определение класса `SlotDef`.

a. Чтобы закомментировать большой фрагмент кода, просто добавьте `/*` перед фрагментом и `*/` после него. Закомментировать большой фрагмент кода можно, выделив его в MonoDevelop и выбрав в меню пункт Edit > Format > Toggle Line Comment(s) (Правка > Форматирование > Закомментировать/раскомментировать), после чего в начало каждой выделенной строки будет добавлена пара символов, начинающих однострочный комментарий (`//`).

b. Независимо от метода, каким вы закомментировали определение `SlotDef` в сценарии Layout, закомментируйте также строку `[System.Serializable]`, предшествующую определению `SlotDef`.

c. Устранив определение класса `SlotDef` из сценария Layout, сохраните его.

4. Вернитесь в сценарий BartokLayout, добавьте следующие строки, выделенные жирным:

```
public class BartokLayout : MonoBehaviour {
    [Header("Set Dynamically")]
    public PT_XMLReader     xmlr; // Так же, как Deck, имеет PT_XMLReader
    public PT_XMLHashtable  xml;  // Используется для ускорения доступа к xml
    public Vector2          multiplier; // Смещение в раскладке
    // Ссылки на SlotDef
    public List<SlotDef>    slotDefs; // Список SlotDef для игроков
    public SlotDef          drawPile;
    public SlotDef          discardPile;
    public SlotDef          target;

    // Этот метод вызывается для чтения файла BartokLayoutXML.xml
    public void ReadLayout(string xmlText) {
        xmlr = new PT_XMLReader();
        xmlr.Parse(xmlText);       // Загрузить XML
        xml = xmlr.xml["xml"][0]; // И определить xml для ускорения доступа к XML

        // Прочитать множители, определяющие расстояние между картами
```

[1] «Пространство имен 'global::' уже содержит определение 'SlotDef'». — *Примеч. пер.*

```
        multiplier.x = float.Parse(xml["multiplier"][0].att("x"));
        multiplier.y = float.Parse(xml["multiplier"][0].att("y"));

        // Прочитать слоты
        SlotDef tSD;
        // slotsX используется для ускорения доступа к элементам <slot>
        PT_XMLHashList slotsX = xml["slot"];

        for (int i=0; i<slotsX.Count; i++) {
            tSD = new SlotDef(); // Создать новый экземпляр SlotDef
            if (slotsX[i].HasAtt("type")) {
                // Если <slot> имеет атрибут type, прочитать его
                tSD.type = slotsX[i].att("type");
            } else {
                // Иначе определить тип как "slot"; это отдельная карта в ряду
                tSD.type = "slot";
            }
            // Преобразовать некоторые атрибуты в числовые значения
            tSD.x = float.Parse( slotsX[i].att("x") );
            tSD.y = float.Parse( slotsX[i].att("y") );
            tSD.pos = new Vector3( tSD.x*multiplier.x, tSD.y*multiplier.y, 0 );

            // Слои сортировки
            tSD.layerID = int.Parse( slotsX[i].att("layer") );              // a
            tSD.layerName = tSD.layerID.ToString();                         // b

            // Прочитать дополнительные атрибуты, опираясь на тип слота
            switch (tSD.type) {
                case "slot":
                    // игнорировать слоты с типом "slot"
                    break;

                case "drawpile":                                            // c
                    tSD.stagger.x = float.Parse( slotsX[i].att("xstagger") );
                    drawPile = tSD;
                    break;

                case "discardpile":
                    discardPile = tSD;
                    break;

                case "target":
                    target = tSD;
                    break;

                case "hand":                                                // d
                    tSD.player = int.Parse( slotsX[i].att("player") );
                    tSD.rot = float.Parse( slotsX[i].att("rot") );
                    slotDefs.Add (tSD);
                    break;
            }
        }
    }
}
```

a. В этой игре слоям сортировки присваиваются имена 1, 2, 3, ... 10. Слои гарантируют правильное перекрытие одних карт другими. В проектах Unity двумерных игр все ресурсы получают одну и ту же координату Z, поэтому для их упорядочения по глубине используются слои сортировки.

b. Преобразование числа `layerID` в текст с записью в `layerName`.

c. Атрибут `xstagger` для стопки свободных карт все еще извлекается из XML-файла, но его значение не используется в *Bartok*, потому что игрокам не требуется знать количество оставшихся свободных карт.

d. Этот фрагмент извлекает сведения о картах в руках каждого игрока в отдельности, включая угол поворота и порядковый номер игрока.

5. Подключите сценарий **BartokLayout** к **_MainCamera**. (Перетащите сценарий **BartokLayout** из панели **Project** (Проект) на объект **_MainCamera** в панели **Hierarchy** (Иерархия).)

6. Перетащите файл **BartokLayoutXML** из папки **Resources** в панели **Project** (Проект) на поле **layoutXML** компонента **Bartok (Script)** главной камеры **_MainCamera**.

7. Откройте сценарий **Bartok** и добавьте следующие строки, выделенные жирным, чтобы задействовать **BartokLayout**:

```
public class Bartok : MonoBehaviour {
    static public Bartok S;

    ...
    public List<CardBartok>     discardPile;

    private BartokLayout        layout;
    private Transform           layoutAnchor;

    void Awake() { ... }

    void Start () {
        deck = GetComponent<Deck>();  // Получить компонент Deck
        deck.InitDeck(deckXML.text);  // Передать ему DeckXML
        Deck.Shuffle(ref deck.cards); // Перетасовать колоду

        layout = GetComponent<BartokLayout>(); // Получить ссылку на компонент
                                               // Layout
        layout.ReadLayout(layoutXML.text);     // Передать ему LayoutXML

        drawPile = UpgradeCardsList( deck.cards );
    }

    List<CardBartok> UpgradeCardsList(List<Card> lCD) {                    // a
        List<CardBartok> lCB = new List<CardBartok>();
        foreach( Card tCD in lCD ) {
            lCB.Add ( tCD as CardBartok );
        }
        return( lCB );
    }

}
```

a. Этот метод приводит все карты в списке `List<Card> lCD` к типу `CardBartoks` и сохраняет их в новом списке `List<CardBartok>`. Он действует точно так же, как аналогичный метод в классе `Prospector`, то есть сами карты изначально имеют тип `CardBartok`, но таким способом мы явно сообщаем об этом движку Unity.

8. Вернитесь в Unity и запустите проект.

После запуска проекта выберите _MainCamera в панели Hierarchy (Иерархия) и раскройте компонент BartokLayout (Script), чтобы убедиться, что его переменные заполнились верными значениями из BartokLayoutXML. Также рассмотрите содержимое поля drawPile компонента Bartok (Script) — в нем должны находиться все 52 экземпляра `CardBartok` в перемешанном порядке.

Класс Player

Игра *Bartok* рассчитана на четырех игроков, поэтому я решил создать класс, представляющий игрока, который способен оценить карты, находящиеся у него в руках, и, используя несложный алгоритм, решить, какой картой пойти. Единственное, что отличает класс `Player` от других, которые вы уже видели, — он не наследует `MonoBehaviour` (и никакие другие классы), но все равно определен в отдельном файле сценария на C#. Так как `Player` не наследует `MonoBehaviour`, он не получает вызовы `Awake()`, `Start()` или `Update()`, сам не может вызывать некоторые функции, такие как `print()`, и его нельзя подключить ни к какому игровому объекту как компонент. Впрочем, ничего из этого и не требуется, поэтому в действительности даже проще иметь класс `Player`, не наследующий `MonoBehaviour`.

1. Создайте в папке __Scripts новый сценарий на C# с именем Player и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq; // Подключает механизм запросов LINQ, о котором рассказывается
ниже

// Игрок может быть человеком или ИИ
public enum PlayerType {
    human,
    ai
}

[System.Serializable]                                                    // a
public class Player {                                                    // b
    public PlayerType        type = PlayerType.ai;
    public int               playerNum;
    public SlotDef           handSlotDef;
    public List<CardBartok>  hand; // Карты в руках игрока

    // Добавляет карту в руки
```

```
    public CardBartok AddCard(CardBartok eCB) {
        if (hand == null) hand = new List<CardBartok>();

        // Добавить карту
        hand.Add (eCB);
        return( eCB );
    }

    // Удаляет карту из рук
    public CardBartok RemoveCard(CardBartok cb) {
        // Если список hand пуст или не содержит карты cb, вернуть null
        if ( hand == null || !hand.Contains(cb) ) return null;
        hand.Remove(cb);
        return(cb);
    }
}
```

a. Атрибут `[System.Serializable]` подсказывает Unity, что `Player` — сериализуемый класс и доступен для отображения и правки в инспекторе.

b. Класс `Player` хранит информацию об отдельном игроке. Как упоминалось выше, он не наследует ни `MonoBehaviour`, ни любые другие классы, поэтому вы должны удалить фрагмент строки «`:MonoBehaviour`».

2. Добавьте следующие строки в сценарий **Bartok**, чтобы задействовать класс `Player`:

```
public class Bartok : MonoBehaviour {
    ...
    [Header("Set in Inspector")]
    ...
    public Vector3              layoutCenter = Vector3.zero;
    public float                handFanDegrees = 10f;                    // a

    [Header("Set Dynamically")]
    ...
    public List<CardBartok>     discardPile;
    public List<Player>         players;                                 // b
    public CardBartok           targetCard;

    private BartokLayout        layout;
    private Transform           layoutAnchor;

    void Awake() { ... }

    void Start () {
        ...
        drawPile = UpgradeCardsList( deck.cards );
        LayoutGame();
    }

    List<CardBartok> UpgradeCardsList(List<Card> lCD) { ... }

    // Позиционирует все карты в drawPile
    public void ArrangeDrawPile() {
        CardBartok tCB;
```

```
    for (int i=0; i<drawPile.Count; i++) {
        tCB = drawPile[i];
        tCB.transform.SetParent( layoutAnchor );
        tCB.transform.localPosition = layout.drawPile.pos;
        // Угол поворота начинается с 0
        tCB.faceUp = false;
        tCB.SetSortingLayerName(layout.drawPile.layerName);
        tCB.SetSortOrder(-i*4); // Упорядочить от первых к последним
        tCB.state = CBState.drawpile;
    }
}

// Выполняет первоначальную раздачу карт в игре
void LayoutGame() {
    // Создать пустой GameObject - точку привязки для раскладки        // c
    if (layoutAnchor == null) {
        GameObject tGO = new GameObject("_LayoutAnchor");
        layoutAnchor = tGO.transform;
        layoutAnchor.transform.position = layoutCenter;
    }

    // Позиционировать свободные карты
    ArrangeDrawPile();

    // Настроить игроков
    Player pl;
    players = new List<Player>();
    foreach (SlotDef tSD in layout.slotDefs) {
        pl = new Player();
        pl.handSlotDef = tSD;
        players.Add(pl);
        pl.playerNum = tSD.player;
    }
    players[0].type = PlayerType.human; // 0-й игрок - человек
}

// Функция Draw снимает верхнюю карту со стопки свободных карт
//  и возвращает ее
public CardBartok Draw() {
    CardBartok cd = drawPile[0]; // Извлечь 0-ю карту
    drawPile.RemoveAt(0);        // Удалить ее из списка drawPile
    return(cd);                  // и вернуть
}

// Метод Update() временно используется для проверки
//   добавления карты в руки игрока
void Update() {                                                        // d
    if (Input.GetKeyDown(KeyCode.Alpha1)) {
        players[0].AddCard(Draw ());
    }
    if (Input.GetKeyDown(KeyCode.Alpha2)) {
        players[1].AddCard(Draw ());
    }
    if (Input.GetKeyDown(KeyCode.Alpha3)) {
        players[2].AddCard(Draw ());
```

```
        }
        if (Input.GetKeyDown(KeyCode.Alpha4)) {
            players[3].AddCard(Draw ());
        }
    }
}
```

a. `handFanDegrees` определяет угол поворота каждой карты относительно предыдущей в одних руках.

b. Список `List<Player> players` хранит ссылки на экземпляры класса `Player` с информацией об игроках. Так как класс `Player` отмечен атрибутом `[System.Serializable]`, этот список можно исследовать в инспекторе Unity.

c. `layoutAnchor` — это экземпляр `Transform`, цель которого — служить родителем всех карт в раскладке в панели **Hierarchy** (Иерархия). Здесь сначала создается пустой игровой объект с именем `_LayoutAnchor`. Затем ссылка на компонент `Transform` этого игрового объекта присваивается полю `layoutAnchor`. Наконец, `layoutAnchor` перемещается в позицию, определяемую полем `layoutCenter`.

d. Эта функция `Update()` будет использоваться для проверки кода, добавляющего карты в руки игрока. Это временная реализация, и далее в главе мы изменим ее. Значения `Keycode.Alpha1` ... `Keycode.Alpha4` соответствуют цифровым клавишам **1** ... **4** на основной клавиатуре. Нажатие одной из этих клавиш добавит карту в руки соответствующего игрока.

3. Сохраните сценарии, вернитесь в Unity и запустите игру.

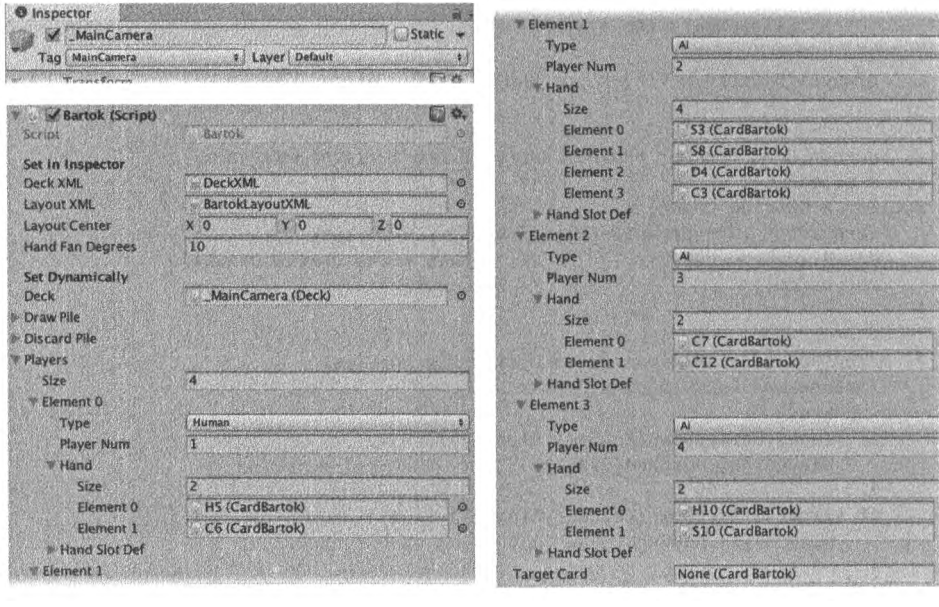


Рис. 33.4. Компонент Bartok (Script) с информацией об игроках и картах в их руках

4. Выберите _MainCamera в иерархии и найдите поле players в компоненте Bartok (Script). Раскройте его, щелкнув на пиктограмме с треугольником рядом с именем players, и вы увидите четыре элемента, по одному для каждого игрока. Раскройте каждый элемент, щелкнув на пиктограмме с треугольником, а затем также распахните поля hand в элементах. Если теперь щелкнуть на панели Game (Игра), чтобы передать ей фокус ввода и позволить реагировать на нажатия клавиш, вы сможете нажимать клавиши 1 ... 4 на клавиатуре (в верхнем ряду, не на дополнительной цифровой клавиатуре) и добавлять карты в руки игроков. В инспекторе вы должны увидеть, как в полях hand появляются новые карты (рис. 33.4).

Конечно, этот метод `Update()` не будет использоваться в окончательной версии игры, однако часто бывает полезно писать подобные короткие функции для предварительной проверки некоторых особенностей. В данном случае нам нужно было проверить правильную работу метода `Player.AddCard()`, и такой подход позволил нам быстро сделать это.

Размещение карт веером в руках

Теперь, когда карты из `drawPile` раздаются в руки игроков, настал момент расположить их на экране.

1. Для этого добавьте следующий код в класс `Player`:

```
public class Player {
    ...

    public CardBartok AddCard(CardBartok eCB) {
        if (hand == null) hand = new List<CardBartok>();

        // Добавить карту
        hand.Add (eCB);
        FanHand();
        return( eCB );
    }

    // Удаляет карту из рук
    public CardBartok RemoveCard(CardBartok cb) {
        // Если список hand пуст или не содержит карты cb, вернуть null
        if ( hand == null || !hand.Contains(cb) ) return null;
        hand.Remove(cb);
        FanHand();
        return(cb);
    }

    public void FanHand() {                                          // a
        // startRot - угол поворота первой карты относительно оси Z    // b
        float startRot = 0;
        startRot = handSlotDef.rot;
        if (hand.Count > 1) {
            startRot += Bartok.S.handFanDegrees * (hand.Count-1) / 2;
```

```
        }

        // Переместить все карты в новые позиции
        Vector3 pos;
        float rot;
        Quaternion rotQ;
        for (int i=0; i<hand.Count; i++) {
            rot = startRot - Bartok.S.handFanDegrees*i;
            rotQ = Quaternion.Euler( 0, 0, rot );                          // c

            pos = Vector3.up * CardBartok.CARD_HEIGHT / 2f;                // d

            pos = rotQ * pos;                                              // e

            // Прибавить координаты позиции руки игрока
            //   (внизу в центре веера карт)
            pos += handSlotDef.pos;                                        // f
            pos.z = -0.5f*i;                                               // g

            // Установить локальную позицию и поворот i-й карты в руках
            hand[i].transform.localPosition = pos;                         // h
            hand[i].transform.rotation = rotQ;
            hand[i].state = CBState.hand;
            hand[i].faceUp = (type == PlayerType.human);                   // i

            // Установить SortOrder карт, чтобы обеспечить правильное перекрытие
            hand[i].SetSortOrder(i*4);                                     // j
        }
    }
}
```

a. `FanHand()` поворачивает карты так, чтобы на экране они располагались веером, как показано на рис. 33.1.

b. `startRot` — угол поворота первой карты относительно оси Z (наибольший угол поворота против часовой стрелки). Первоначально получает значение, указанное в BartokLayoutXML, а затем добавляется поворот против часовой стрелки так, чтобы карты легли веером относительно центра в позиции руки. После вычисления значения `startRot` каждая последующая карта поворачивается относительно предыдущей на угол `Bartok.S.handFanDegrees` по часовой стрелке.

c. `rotQ` хранит экземпляр `Quaternion`, представляющий угол поворота относительно оси Z.

d. Далее вычисляется `pos` — вектор `Vector3` с координатами точки, находящейся на половину высоты карты над центром (то есть `localPosition` = [0, 0, 0]) для текущего игрока, соответственно, первоначально `pos` принимает значение [0, 1.75, 0].

e. Затем `rotQ` умножается на `pos`. Умножение экземпляра `Quaternion` на экземпляр `Vector3` выполняет поворот вектора, то есть после умножения `pos` окажется повернутым на `rotQ` градусов относительно оси Z.

f. К вектору pos добавляются координаты руки.

g. Координаты pos.z разных карт в руке задаются так, чтобы они размещались ступенями. На экране этого не видно (потому что мы имеем дело с двумерными спрайтами), но такое решение предотвращает перекрытие трехмерных коллайдеров Box Collider, используемых в игре.

h. Вычисленные значения pos и rotQ применяются к i-й карте в руках.

i. Карты игрока-человека поворачиваются лицевой стороной вверх.

j. Настройка порядка сортировки для каждой карты обеспечивает правильное перекрытие внутри слоя сортировки.

2. Сохраните сценарий Player, вернитесь в Unity и щелкните на кнопке Play (Играть).

Попробуйте понажимать цифровые клавиши 1, 2, 3 и 4 в верхнем ряду на клавиатуре; вы должны увидеть, как карты добавляются в руки игроков и располагаются веером. Однако вы, возможно, заметили, что карты в руках игрока не сортируются по достоинству, что выглядит несколько небрежно. К счастью, мы легко можем решить эту проблему[1].

Краткое введение в LINQ

LINQ, сокращенно от «Language INtegrated Query» (язык интегрированных запросов), — это удивительное расширение языка C#, которому посвящено великое множество книг. В потрясающей книге «C# 5.0 Pocket Reference»[2] Джозефа Албахари (Joseph Albahari) и Бена Албахари (Ben Albahari) расширению LINQ посвящено целых 24 страницы (тогда как для массивов отведено только четыре страницы). Подробное описание LINQ выходит далеко за рамки этой книги, но я надеюсь, что это краткое введение поможет вам взглянуть на LINQ как на возможное решение проблем, с которыми вы наверняка столкнетесь в своих будущих проектах.

LINQ позволяет выполнять запросы, напоминающие запросы к базе данных, в коде на C#, давая возможность выбирать конкретные элементы из массивов и упорядочивать их. Вот как с помощью этого расширения можно отсортировать карты в руках игрока-человека.

1. Добавьте в Player.AddCard() следующие строки, выделенные жирным:

```
public class Player {
    ...

    // Добавляет карту в руки
```

[1] Возможно, вы также заметили, что после раздачи всех карт очередное нажатие на цифровую клавишу генерирует исключение с сообщением «Argument Out Of Range» («Аргумент вне диапазона»). Не волнуйтесь, мы исправим эту ошибку ниже.

[2] Албахари Джозеф, Албахари Бен. C# 5.0. Карманный справочник. М.: Вильямс, 2013. — *Примеч. пер.*

```
    public CardBartok AddCard(CardBartok eCB) {
        if (hand == null) hand = new List<CardBartok>();

        // Добавить карту
        hand.Add (eCB);

        // Если это человек, отсортировать карты по достоинству с помощью LINQ
        if (type == PlayerType.human) {
            CardBartok[] cards = hand.ToArray();                    // a

            // Это вызов LINQ
            cards = cards.OrderBy( cd => cd.rank ).ToArray();       // b

            hand = new List<CardBartok>(cards);                     // c

            // Примечание: LINQ выполняет операции довольно медленно
            //   (затрачивая по несколько миллисекунд), но так как
            //   мы делаем это один раз за раунд, это не проблема.
        }

        FanHand();
        return( eCB );
    }

    ...
}
```

a. LINQ работает с массивами значений, поэтому мы создаем массив `CardBartok[]` карт из списка `List<CardBartok> hand`.

b. Эта строка — вызов LINQ, который обрабатывает массив карт. Он выполняет обход элементов массива подобно циклу `foreach(CardBartok cd in cards)` и сортирует их по значению `rank` (на что указывает аргумент `cd => cd.rank`). Отсортированный массив присваивается переменной `cards`, затирая старый, несортированный массив. Синтаксис LINQ отличается от обычного синтаксиса языка C#, из-за чего первое время он может казаться вам странным.

   Обратите внимание, что операции LINQ выполняются довольно медленно — на выполнение одного вызова может потребоваться несколько миллисекунд, но так как мы вызываем LINQ только один раз в каждом ходе, это не является проблемой.

c. После сортировки массива `cards` из него создается новый список карт и записывается в `hand` взамен старого, несортированного списка.

Как видите, мы смогли отсортировать список всего одной строкой кода на LINQ. LINQ обладает огромными возможностями. Я не буду рассказывать о них в этой книге, но настоятельно советую познакомиться с ними на тот случай, если вам понадобится выполнять сортировку или другие операции с массивами (например, с помощью LINQ легко можно отыскать в массиве всех людей в возрасте от 18 до 25 с именами, начинающимися на «J»).

2. Сохраните сценарий `Player`, вернитесь в Unity и запустите сцену. Теперь карты в руках игрока-человека всегда будут отсортированы по достоинству.

Чтобы игра была понятна игроку, карты должны перемещаться с места на место с воспроизведением анимационного эффекта, и сейчас самое время им заняться.

Перемещаем карты!

Мы подошли к самой увлекательной части — к реализации плавного перемещения карт с места на место с одновременным поворотом методом интерполяции. Анимационный эффект придаст игре реалистичность и, как вы убедитесь сами, благодаря ему игроку будет проще понять происходящее в игре.

Реализация анимационного эффекта в этой игре во многом похожа на аналогичную реализацию эффекта перемещения FloatingScore в *Prospector*. Так же как в случае с FloatingScore, интерполяцию будет выполнять сама карта, и, завершив перемещение, она уведомит игру об этом, выполнив обратный вызов.

Начнем с плавного перемещения карт в руки игроков при раздаче. Класс `CardBartok` уже содержит большую часть кода, обрабатывающего перемещение, поэтому просто воспользуемся им.

1. Измените следующие строки в методе `Player.FanHand()`, выделенные жирным:

```
public class Player {
    ...
    public void FanHand() {
        ...
        for (int i=0; i<hand.Count; i++) {
            ...
            pos.z = -0.5f*i;

            // Установить локальную позицию и поворот i-й карты в руках
            hand[i].MoveTo(pos, rotQ); // Сообщить карте, что она должна начать
                                       // интерполяцию
            hand[i].state = CBState.toHand;
            // Закончив перемещение, карта запишет в поле state значение
            // CBState.hand

            /* <= Это начало многострочного комментария                    // a
            hand[i].transform.localPosition = pos;
            and[i].transform.rotation = rotQ;
            hand[i].state = CBState.hand;
            Это конец многострочного комментария => */                   // b

            hand[i].faceUp = (type == PlayerType.human);

            ...
        }
    }
}
```

a. Пара символов /* начинает многострочный комментарий, то есть все строки кода между этой парой и парой */ считаются закомментированными (и игнорируются компилятором C#). Именно таким способом вы могли закомментировать класс SlotDef в сценарии Layout в начале этой главы.

b. Пара символов */ завершает многострочный комментарий.

2. Сохраните сценарий Player, вернитесь в Unity и запустите сцену.

Теперь, нажимая цифровые клавиши (1, 2, 3, 4), можно увидеть, как карты перемещаются из колоды в руки игроков! Основную работу взял на себя класс CardBartok, поэтому нам удалось реализовать этот эффект с минимумом кода. Это одно из самых больших преимуществ объектно-ориентированного программирования. Класс CardBartok знает, как перемещать свои экземпляры, поэтому нам остается только вызвать метод MoveTo() и передать ему точку назначения и поворот, а все остальное сделает CardBartok.

Управление раздачей карт в начале игры

В начале раунда в игре *Bartok* каждому игроку сдается по семь карт, верхняя свободная карта снимается и объявляется целевой.

1. Добавьте следующий код в сценарий Bartok, чтобы выполнить раздачу в начале игры:

```
public class Bartok : MonoBehaviour {
    ...
    [Header("Set in Inspector")]
    ...
    public float                handFanDegrees = 10f;
    public int                  numStartingCards = 7;
    public float                drawTimeStagger = 0.1f;

    ...
    void LayoutGame() {
        ...
        players[0].type = PlayerType.human; // 0-й игрок - человек

        CardBartok tCB;
        // Раздать игрокам по семь карт
        for (int i=0; i<numStartingCards; i++) {
            for (int j=0; j<4; j++) {                                    // a
                tCB = Draw (); // Снять карту
                // Немного отложить начало перемещения карты.
                tCB.timeStart = Time.time + drawTimeStagger * ( i*4 + j );    // b

                players[ (j+1)%4 ].AddCard(tCB);                         // c
            }
        }

        Invoke("DrawFirstTarget", drawTimeStagger * (numStartingCards*4+4) ); // d
    }
```

```
    public void DrawFirstTarget() {
        // Перевернуть первую целевую карту лицевой стороной вверх
        CardBartok tCB = MoveToTarget( Draw () );
    }

    // Делает указанную карту целевой
    public CardBartok MoveToTarget(CardBartok tCB) {
        tCB.timeStart = 0;
        tCB.MoveTo(layout.discardPile.pos+Vector3.back);
        tCB.state = CBState.toTarget;
        tCB.faceUp = true;

        targetCard = tCB;

        return(tCB);
    }

    // Функция Draw снимает верхнюю карту со стопки свободных карт
    //  и возвращает ее
    public CardBartok Draw() { ... }
    ...
}
```

a. Значение переменной `j` изменяется в диапазоне от 0 до 3, потому что в игре участвуют четыре игрока. Если бы в игре могло участвовать разное количество игроков, вместо повсеместного использования в коде целочисленного литерала `4` мы должны были бы определить переменную, изменяющуюся динамически. Это отличный повод использовать `const`, но тогда получились бы слишком длинные строки кода, не умещающиеся по ширине книжной страницы.

b. Смещение времени `timeStart` начала перемещения каждой карты позволяет организовать эффект их последовательной раздачи друг за другом. Сначала определяется смещение времени начала перемещения карты `drawTimeStagger * ( i*4 + j )`, а затем к нему добавляется текущее время `Time.time`. В результате все карты после 0-й начнут перемещение немного позже, что создаст приятный для глаз анимационный эффект.

c. Добавляет карту в руки игрока. Выражение `(j+1)%4` обеспечивает последовательное индексирование игроков в списке `players` в порядке 1, 2, 3, 0. В результате раздача карт игрокам происходит по кругу, начиная с `players[1]` (первый игрок, если идти по часовой стрелке от игрока-человека `players[0]`).

d. После начальной раздачи карт вызывается `DrawFirstTarget()`.

2. Сохраните сценарий **Bartok**, вернитесь в Unity и запустите сцену.

Сразу после запуска вы увидите, как произойдет последовательная раздача карт игрокам и откроется первая целевая карта, однако карты игрока-человека перекрывают друг друга не так, как должны бы. Так же как в *Prospector*, мы должны уделить особое внимание настройкам `sortingLayerName` и `sortingOrder` каждой карты.

Управление сортировкой по глубине в двумерной игре

Теперь, кроме стандартной проблемы сортировки двумерных объектов по глубине, нам придется учитывать тот факт, что карты перемещаются, и организовать одну сортировку в начале перемещения и другую в конце. Для этого добавим в класс `CardBartok` дополнительные поля, `eventualSortLayer` и `eventualSortOrder`, определяющие конечные порядок и слой сортировки. Благодаря этому, начав перемещение, карта сможет на полпути установить параметры сортировки, заданные в `eventualSortLayer` и `eventualSortOrder`.

1. Сначала переименуйте все слои сортировки. Откройте диспетчер тегов и слоев **Tags & Layers**, выбрав в меню пункт **Edit > Project Settings > Tags & Layers** (Правка > Параметры проекта > Теги и слои).
2. Назначьте слоям сортировки с 1 по 10 имена от **1** до **10**, как показано на рис. 33.5. Добавьте дополнительные слои сортировки, если потребуется.

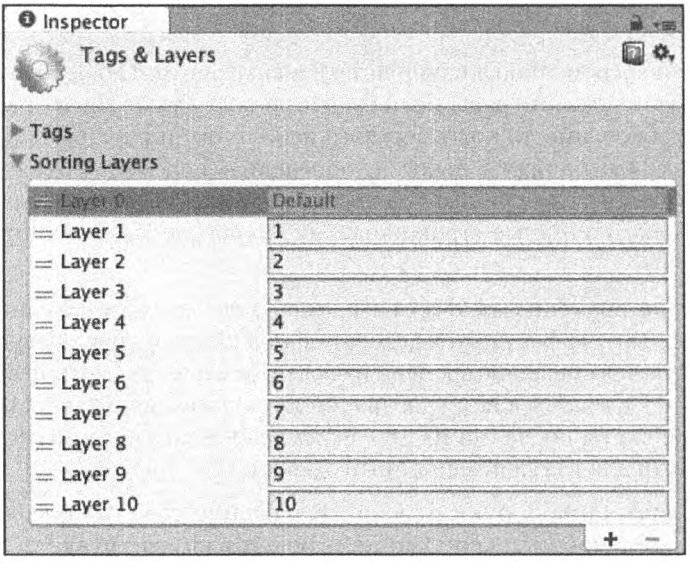


Рис. 33.5. Слои сортировки с простыми именами для игры Bartok

3. Добавьте в сценарий **CardBartok** следующие строки, выделенные жирным:

```
public class CardBartok : Card {
    ...
    [Header("Set Dynamically")]
    ...
    public float              timeStart, timeDuration;
    public int                eventualSortOrder;
```

```
    public string             eventualSortLayer;

    ...

    void Update() {
        switch (state) {
            case CBState.toHand:
            case CBState.toTarget:
            case CBState.to:
                ...
                } else {
                    Vector3 pos = Utils.Bezier(uC, bezierPts);
                    transform.localPosition = pos;
                    Quaternion rotQ = Utils.Bezier(uC, bezierRots);
                    transform.rotation = rotQ;

                    if (u>0.5f) {                                          // a
                        SpriteRenderer sRend = spriteRenderers[0];
                        if (sRend.sortingOrder != eventualSortOrder) {
                            // Установить конечный порядок сортировки
                            SetSortOrder(eventualSortOrder);
                        }
                        if (sRend.sortingLayerName != eventualSortLayer) {
                            // Установить конечный слой сортировки
                            SetSortingLayerName(eventualSortLayer);
                        }
                    }

                }
                break;
        }
    }
}
```

a. Когда карта преодолеет половину пути (то есть `u>0.5f`), произойдет изменение порядка сортировки и смена слоя сортировки на `eventualSortOrder` и `eventualSortLayer` соответственно.

Теперь, добавив поля `eventualSortOrder` и `eventualSortLayer`, мы должны задействовать их в уже написанном коде.

4. Внедрим их в метод `MoveToTarget()` класса `Bartok`, а также добавим функцию `MoveToDiscard()`, перемещающую целевую карту в стопку сброса `discardPile`:

```
public class Bartok : MonoBehaviour {
    ...

    public CardBartok MoveToTarget(CardBartok tCB) {
        tCB.timeStart = 0;
        tCB.MoveTo(layout.discardPile.pos+Vector3.back);
        tCB.state = CBState.toTarget;
        tCB.faceUp = true;

        tCB.SetSortingLayerName("10");
        tCB.eventualSortLayer = layout.target.layerName;
        if (targetCard != null) {
```

```
            MoveToDiscard(targetCard);
        }

        targetCard = tCB;

        return(tCB);
    }

    public CardBartok MoveToDiscard(CardBartok tCB) {
        tCB.state = CBState.discard;
        discardPile.Add ( tCB );
        tCB.SetSortingLayerName(layout.discardPile.layerName);
        tCB.SetSortOrder( discardPile.Count*4 );
        tCB.transform.localPosition = layout.discardPile.pos + Vector3.back/2;

        return(tCB);
    }

    // Функция Draw снимает верхнюю карту со стопки свободных карт
    //  и возвращает ее
    public CardBartok Draw() { ... }
    ...
}
```

5. Также нужно внести некоторые изменения в методы `AddCard()` и `FanHand()` класса `Player`:

```
public class Player {
    ...
    public CardBartok AddCard(CardBartok eCB) {
        ...
        // Если это человек, отсортировать карты по достоинству с помощью LINQ
        if (type == PlayerType.human) {
            ...
        }

        eCB.SetSortingLayerName("10"); // Перенести перемещаемую карту в верхний
                                       // слой // a
        eCB.eventualSortLayer = handSlotDef.layerName;

        FanHand();
        return( eCB );
    }

    // Удаляет карту из рук
    public CardBartok RemoveCard(CardBartok cb) { ... }

    public void FanHand() {
        ...
            hand[i].faceUp = (type == PlayerType.human);

            // Установить SortOrder карт, чтобы обеспечить правильное перекрытие
            hand[i].eventualSortOrder = i*4;                                  // b
            //hand[i].SetSortOrder(i*4);
        }
    }
}
```

a. Благодаря переносу в слой сортировки "10", перемещаемая карта оказывается выше всех других карт, пока продолжает движение. Когда карта преодолеет половину пути, код, добавленный в сценарий CardBartok на шаге 3 в этом разделе, перенесет ее в слой eventualSortLayer.

b. Закомментируйте строку, которая находилась здесь (сейчас находится строкой ниже) и замените ее этой.

6. Обязательно сохраните изменения во всех сценариях, вернитесь в Unity и щелкните на кнопке Play (Играть). Теперь, благодаря изменениям в сортировке карт по слоям, их перемещение должно выглядеть более реалистично.

Выполнение ходов

В этой игре игроки должны ходить по очереди. Для начала добавим в сценарий Bartok код, определяющий, чья очередь сделать ход.

1. Откройте сценарий Bartok и добавьте следующие строки, выделенные жирным:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// Это перечисление определяет разные этапы в течение одного игрового хода
public enum TurnPhase {
    idle,
    pre,
    waiting,
    post,
    gameOver
}

public class Bartok : MonoBehaviour {
    static public Bartok S;
    static public Player CURRENT_PLAYER;                                    // a

    ...

     [Header("Set Dynamically")]
    ...
    public CardBartok targetCard;
    public TurnPhase phase = TurnPhase.idle;

    private BartokLayout layout;

    ...

    public void DrawFirstTarget() {
        // Перевернуть первую целевую карту лицевой стороной вверх
        CardBartok tCB = MoveToTarget( Draw () );
        // Вызвать метод CBCallback сценария Bartok, когда карта закончит
        // перемещение
        tCB.reportFinishTo = this.gameObject;                               // b
```

```
    }

    // Этот обратный вызов используется последней розданной картой в начале игры
    public void CBCallback(CardBartok cb) {                                    // c
        // Иногда желательно сообщить о вызове метода, как здесь
        Utils.tr("Bartok:CBCallback()",cb.name);                               // d
        StartGame(); // Начать игру
    }

    public void StartGame() {
        // Право первого хода принадлежит игроку слева от человека.
        PassTurn(1);                                                           // e
    }

    public void PassTurn(int num=-1) { // f
        // Если порядковый номер игрока не указан, выбрать следующего по кругу
        if (num == -1) {
            int ndx = players.IndexOf(CURRENT_PLAYER);
            num = (ndx+1)%4;
        }
        int lastPlayerNum = -1;
        if (CURRENT_PLAYER != null) {
            lastPlayerNum = CURRENT_PLAYER.playerNum;
        }
        CURRENT_PLAYER = players[num];
        phase = TurnPhase.pre;

//        CURRENT_PLAYER.TakeTurn();                                           // g

        // Сообщить о передаче хода
        Utils.tr("Bartok:PassTurn()", "Old: "+lastPlayerNum,                   // h
                ➥"New: "+CURRENT_PLAYER.playerNum);                           // h
    }

    // ValidPlay проверяет возможность сыграть выбранной картой
    public bool ValidPlay(CardBartok cb) {
        // Картой можно сыграть, если она имеет такое же достоинство,
        //   как целевая карта
        if (cb.rank == targetCard.rank) return(true);

        // Картой можно сыграть, если ее масть совпадает с мастью целевой карты
        if (cb.suit == targetCard.suit) {
            return(true);
        }

        // Иначе вернуть false
        return(false);
    }

    // Делает указанную карту целевой
    public CardBartok MoveToTarget(CardBartok tCB) { ... }

    ...

    /* Теперь можно закомментировать тестовый код                              // i
```

```
    // Метод Update() временно используется для проверки
    //   добавления карты в руки игрока
    void Update() {
        if (Input.GetKeyDown(KeyCode.Alpha1)) {
            players[0].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha2)) {
            players[1].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha3)) {
            players[2].AddCard(Draw ());
        }
        if (Input.GetKeyDown(KeyCode.Alpha4)) {
            players[3].AddCard(Draw ());
        }
    }
    */                                                          // i
}
```

a. `CURRENT_PLAYER` объявлена статической и общедоступной по двум причинам: текущим в игре может быть только один игрок, а объявление этого поля статическим упростит обращение к нему из сценария TurnLight, который мы добавим в следующем разделе.

b. `reportFinishTo` — это поле типа `GameObject`, уже имеющееся в классе `CardBartok`. Через него экземпляр `CardBartok` получает ссылку на игровой объект (в данном случае _MainCamera), к которому подключен этот экземпляр сценария Bartok. Существующий код `CardBartok` уже включает вызов метода `SendMessage("CBCallback",this)` игрового объекта, ссылка на который хранится в `reportFinishTo`, если она не равна `null`.

c. Метод `CDCallback()` вызывается, когда первая целевая карта завершает перемещение (как только что было описано).

d. Вызов `Utils.tr()` в этой строке выведет в консоль сообщение о вызове метода `CBCallback()`. Это первый случай использования статического общедоступного метода `Utils.tr()` (имя `tr` является сокращением от англ. «trace» — «трассировка», или «диагностическое сообщение»). Этот метод принимает произвольное количество аргументов (в именованном параметре `params`), объединяет их, вставляя символ табуляции, и выводит в панель Console (Консоль). Это один из элементов, добавленных в класс `Utils` из пакета, импортированного в проект *Prospector*.

   В вызов `tr()` здесь передается строковый литерал с именем метода (`"Bartok:CBCallback()"`) и имя игрового объекта, вызвавшего метод `CBCallback()`.

e. Право первого хода всегда принадлежит игроку слева от человека. Так как человек — это `players[0]`, соответственно, право хода передается игроку с индексом 1, то есть `players[1]`.

f. Метод `PassTurn()` имеет необязательный параметр, с помощью которого можно явно указать игрока, которому принадлежит право хода. Если метод вызывается без аргумента, параметр `num` получает значение по умолчанию −1, и в четырех строках ниже ему присваивается номер игрока, следующего за текущим по часовой стрелке.

g. Сейчас эта строка закомментирована, потому что в классе `Player` пока нет метода `TakeTurn()`. Мы раскомментируем ее в следующем разделе.

h. Эти две строки в действительности представляют одну строку кода, которая получилась слишком длинной и не умещается по ширине книжной страницы. Вы можете ввести ее в одну строку или в две. Так как первая из этих двух строк не завершается точкой с запятой (`;`), Unity прочитает обе строки как одну инструкцию. Обратите внимание, что комментарий `// h` в конце первой строки не препятствует интерпретации этих двух строк как единой инструкции. В таких случаях я использую символ ➥ продолжения кода в начале следующей строки, вводить этот символ **не нужно**.

i. Первоначально этот метод `Update()` использовался для тестирования, но теперь надобность в этом отпала. Закомментируйте его целиком, добавив `/*` до и `*/` после определения метода.

2. Сохраните сценарий **Bartok**, вернитесь в Unity и щелкните на кнопке **Play** (Играть). Вы должны увидеть, как произойдет начальная раздача карт, а затем в консоли появится сообщение:

```
Bartok:PassTurn()    Old: -1    New: 1
```

Дополнительное прояснение ситуации в игре

Сообщение с текстом `Bartok:PassTurn()` в консоли помогает понять, кому принадлежит право хода, но консоль будет недоступна нашим игрокам. Поэтому нужен какой-то другой способ показать, кому принадлежит право хода. Для этого организуем подсветку фона в позиции текущего игрока.

1. В меню Unity выберите пункт **GameObject > Light > Point Light** (Игровой объект > Свет > Точечный источник света), чтобы создать новый точечный источник света **Point Light**.

2. Дайте новому источнику света имя **TurnLight** и настройте его компонент **Transform**:

```
TurnLight (Point Light)   P:[ 0, 0, -3 ]   R:[ 0, 0, 0 ]   S:[ 1, 1, 1 ]
```

   Как видите, этот источник создает приятную подсветку. Теперь добавьте код, показывающий, кто является текущим игроком.

3. Создайте в папке **__Scripts** новый сценарий на C# с именем **TurnLight**.

4. Подключите его к игровому объекту **TurnLight** в иерархии.

5. Откройте сценарий **TurnLight** и добавьте следующий код.

```
using UnityEngine;
using System.Collections;

public class TurnLight : MonoBehaviour {

    void Update () {
        transform.position = Vector3.back*3;                          // a
        if (Bartok.CURRENT_PLAYER == null) {                          // b
            return;
        }

        transform.position += Bartok.CURRENT_PLAYER.handSlotDef.pos;  // c
    }
}
```

a. Перемещает источник света в позицию по умолчанию над центром игрового поля ([0, 0, -3]).

b. Если `Bartok.CURRENT_PLAYER` содержит `null`, на этом работа метода закончена.

c. Если `Bartok.CURRENT_PLAYER` содержит действительную ссылку, тогда просто добавить позицию текущего игрока, чтобы расположить источник света над его картами.

В предыдущем издании книги код сценария **TurnLight** был частью класса `Bartok`, но в этом издании я выбрал более компонентно-ориентированный путь, основная идея которого заключается в разделении кода на небольшие фрагменты, решающие простые задачи. Для сценария **Bartok** нет никакой необходимости знать о существовании подсветки, поэтому во втором издании я решил создать отдельный сценарий, управляющий подсветкой.

6. Сохраните сценарий **TurnLight**, вернитесь в Unity и щелкните на кнопке **Play** (Играть).

Теперь, после начальной раздачи карт, вы должны увидеть, как подсветка переместилась в позицию игрока слева от человека, подсказывая, что право хода принадлежит ему.

Простой искусственный интеллект в игре Bartok

Теперь реализуем ИИ, который будет выполнять ходы за игроков.

1. Откройте сценарий **Bartok** и взгляните на строку, отмеченную комментарием `// g`, в листинге из раздела «Выполнение ходов» несколькими страницами выше. Уберите символы слеша в начале этой строки. Теперь эта строка должна выглядеть так:

```
CURRENT_PLAYER.TakeTurn();                                    // g
```

2. Сохраните сценарий **Bartok**.
3. Откройте сценарий **Player** и добавьте следующие строки, выделенные жирным:

```
public class Player {
    ...

    public void FanHand() {
        ...
        Quaternion rotQ;
        for (int i=0; i<hand.Count; i++) {
            ...
            pos += handSlotDef.pos;
            pos.z = -0.5f*i;

            // Если это не начальная раздача, начать перемещение карты немедленно.
            if (Bartok.S.phase != TurnPhase.idle) {                          // a
                hand[i].timeStart = 0;
            }

            // Установить локальную позицию и поворот i-й карты в руках
            hand[i].MoveTo(pos, rotQ); // Сообщить карте, что она должна начать
                                       // интерполяцию
            ...
        }
    }

    // Функция TakeTurn() реализует ИИ для игроков, управляемых компьютером
    public void TakeTurn() {
        Utils.tr ("Player.TakeTurn");

        // Ничего не делать для игрока-человека.
        if (type == PlayerType.human) return;

        Bartok.S.phase = TurnPhase.waiting;

        CardBartok cb;

        // Если этим игроком управляет компьютер, нужно выбрать карту для хода
        // Найти допустимые ходы
        List<CardBartok> validCards = new List<CardBartok>();               // b
        foreach (CardBartok tCB in hand) {
            if (Bartok.S.ValidPlay(tCB)) {
                validCards.Add ( tCB );
            }
        }
        // Если допустимых ходов нет
        if (validCards.Count == 0) {                                        // c
            // ...взять карту
            cb = AddCard( Bartok.S.Draw () );
            cb.callbackPlayer = this;                                       // e
            return;
        }

        // Итак, у нас есть одна или несколько карт, которыми можно сыграть
        //   теперь нужно выбрать одну из них
        cb = validCards[ Random.Range (0,validCards.Count) ];               // d
        RemoveCard(cb);
        Bartok.S.MoveToTarget(cb);
```

```
        cb.callbackPlayer = this;                                    // e
    }

    public void CBCallback(CardBartok tCB) {
        Utils.tr ("Player.CBCallback()",tCB.name,"Player "+playerNum);
        // Карта завершила перемещение, передать право хода
        Bartok.S.PassTurn();
    }
}
```

a. Задержка перед началом перемещения карты необходима только в момент начальной раздачи, в начале игры, в дальнейшем никаких задержек не должно быть, поэтому начинаем перемещение сразу же.

b. Здесь ИИ пытается найти допустимые карты для выполнения хода. Он вызывает `ValidPlay()` для каждой карты в руках, и если картой можно выполнить ход, она добавляется в список `validCards`.

c. Если в списке `validCards` оказалось 0 карт (то есть у игрока нет ни одной карты, с которой он мог бы пойти), тогда ИИ берет еще одну карту и выходит.

d. Если в списке `validCards` имеются карты, ИИ выбирает случайную и делает ее новой целевой картой (то есть перекладывает карту в стопку сброшенных карт).

e. Имя `callbackPlayer` окрашено в красный цвет в этих двух строках, потому что поле с этим именем еще не добавлено в класс `CardBartok`.

4. Сохраните сценарий **Player**.

   В конец сценария **Player** мы добавили функцию `CBCallback()`, которую должен вызвать класс `CardBartok`, завершив перемещение карты; однако из-за того, что `Player` не наследует `MonoBehaviour`, мы не можем использовать `SendMessage()` для вызова `CBCallback()`. Поэтому мы передаем экземпляру `CardBartok` ссылку на экземпляр `Player`, благодаря чему `CardBartok` получает возможность напрямую вызвать метод `CBCallback()` экземпляра `Player`. Эта ссылка сохраняется в поле `callbackPlayer` класса `CardBartok`.

5. Откройте сценарий **CardBartok** и добавьте следующий код:

```
public class CardBartok : Card {
    ...
    [Header("Set Dynamically")]
    ...
    public GameObject         reportFinishTo = null;
    [System.NonSerialized]                                           // a
    public Player             callbackPlayer = null;                 // b

    // MoveTo запускает перемещение карты в новое местоположение с заданным
    // поворотом
    public void MoveTo(Vector3 ePos, Quaternion eRot) { ... }
    ...

    void Update() {
```

```
        switch (state) {
            case CBState.toHand:
            case CBState.toTarget:
            case CBState.to:
                ...
                if (u<0) {
                    ...
                } else if (u>=1) {
                    ...
                    if (reportFinishTo != null) {
                        reportFinishTo.SendMessage("CBCallback", this);
                        reportFinishTo = null;
                    } else if (callbackPlayer != null) {                    // c
                        // Если имеется ссылка на экземпляр Player
                        // Вызвать метод CBCallback этого экземпляра
                        callbackPlayer.CBCallback(this);
                        callbackPlayer = null;
                    } else { // Если ничего вызывать не надо,
                        // Оставить все как есть.
                    }
                } else {
                    ...
                }
                break;
        }
    }
}
```

a. Подобно `[System.Serialized]`, атрибут `[System.NonSerialized]` применяется к строке, следующей за ним. В данном случае мы указываем, что поле `callbackPlayer` не должно сериализоваться, то есть оно не будет отображаться в инспекторе и вы не сможете присвоить ему значение в инспекторе. Второе особенно важно по причине, которая раскрывается в пункте **c** ниже.

b. После объявления поля `callbackPlayer` его имя не будет окрашиваться в красный цвет в сценарии `Player`.

c. Здесь мы просто вызываем `callbackPlayer.CBCallback()`, если поле `callbackPlayer` содержит ссылку, отличную от `null`. Именно поэтому потребовалось объявить поле `callbackPlayer` несериализуемым. Если бы мы позволили инспектору сериализовать это поле, он смог бы создать новый экземпляр `Player` для сохранения ссылки в `callbackPlayer`, чтобы показать ее в панели Inspector (Инспектор). Иначе говоря, если разрешить сериализацию поля `callbackPlayer`, оно могло бы получить значение, отличное от `null` еще до начала игры. Чтобы этого не происходило, мы снабдили поле `callbackPlayer` атрибутом `NonSerialized`. Для проверки попробуйте закомментировать строку `[System.NonSerialized]` и запустить игру. Вы тут же получите сообщение об ошибке, потому что `CardBartok` попытается вызвать `CBCallback()` недопустимого экземпляра `Player`, созданного инспектором.

6. Сохраните сценарий CardBartok и вернитесь в Unity.

Теперь, когда вы запустите сцену, три игрока, управляемые компьютером, сделают ход по очереди.

Включение в игру игрока-человека

Пришло время дать человеку возможность сделать свой ход. Для этого мы должны добавить в карты реакцию на щелчок мыши.

1. Добавьте в конец класса `CardBartok` следующие строки, выделенные жирным:

```
public class CardBartok : Card {

    ...

    void Update() { ... }

    // Этот метод определяет реакцию карты на щелчок мышью
    override public void OnMouseUpAsButton() {
        // Вызвать метод CardClicked объекта-одиночки Bartok
        Bartok.S.CardClicked(this);                                    // a
        // Также вызвать версию этого метода в базовом класса (Card.cs)
        base.OnMouseUpAsButton();
    }
}
```

   a. Имя `CardClicked` окрашено в красный цвет, потому что мы еще не добавили метод `CardClicked()` в класс `Bartok`.

2. Сохраните сценарий CardBartok.

3. Добавьте метод `CardClicked()` в конец сценария Bartok:

```
public class Bartok : MonoBehaviour {
    ...
    public CardBartok Draw() { ... }

    public void CardClicked(CardBartok tCB) {
        if (CURRENT_PLAYER.type != PlayerType.human) return;           // a
        if (phase == TurnPhase.waiting) return;                        // b

        switch (tCB.state) {                                           // c
            case CBState.drawpile:                                     // d
                // Взять верхнюю карту, не обязательно ту,
                //   по которой выполнен щелчок.
                CardBartok cb = CURRENT_PLAYER.AddCard( Draw() );
                cb.callbackPlayer = CURRENT_PLAYER;
                Utils.tr ("Bartok:CardClicked()","Draw",cb.name);
                phase = TurnPhase.waiting;
                break;

            case CBState.hand:                                         // e
                // Проверить допустимость выбранной карты
                if (ValidPlay(tCB)) {
                    CURRENT_PLAYER.RemoveCard(tCB);
                    MoveToTarget(tCB);
```

```
                tCB.callbackPlayer = CURRENT_PLAYER;
                Utils.tr("Bartok:CardClicked()","Play",tCB.name,
                    ➥targetCard.name+" is target");                    // f
                phase = TurnPhase.waiting;
            } else {
                // Игнорировать выбор недопустимой карты,
                //   но сообщить о попытке игрока
                Utils.tr("Bartok:CardClicked()","Attempted to Play",
                    ➥tCB.name,targetCard.name+" is target");           // f
            }
            break;
        }
    }
}
```

a. Если право хода не принадлежит человеку, вообще никак не реагировать на щелчок; просто выйти.

b. Если игра ждет, пока карта завершит перемещение, не реагировать на щелчок. Это заставит игрока-человека ждать, пока до него дойдет черед.

c. Эта инструкция описывает действия по щелчку на карте в руках игрока и в стопке свободных карт `drawPile`.

d. Если карта находится в стопке свободных карт, снимается верхняя карта. Поскольку спрайты в `drawPile` вообще никак не сортируются, снимаемая карта фактически может не соответствовать карте, по которой выполнен щелчок мышью.

e. Если карта находится в руках игрока, проверяется возможность выполнить ход этой картой. Если выбранной картой можно сыграть, она становится целевой картой, иначе щелчок игнорируется, но в консоль выводится сообщение о попытке.

f. Напомню, что вы не должны вводить символ ➥ продолжения строки.

4. Сохраните сценарий **Bartok**, вернитесь в Unity и щелкните на кнопке **Play** (Играть).

Теперь вы тоже сможете принять участие в игре! Но в данный момент отсутствует логика обработки окончания игры. Добавим несколько штрихов, и наш прототип будет закончен.

Обработка исчерпания свободных карт

Теперь, когда вы и ИИ получили возможность играть, стопка свободных карт может опустеть, и это обстоятельство вызовет аварийное завершение игры. Давайте добавим в **Bartok** код, обрабатывающий эту ситуацию:

```
public class Bartok : MonoBehaviour {
    ...
    // Функция Draw снимает верхнюю карту со стопки свободных карт
    //  и возвращает ее
```

```
    public CardBartok Draw() {
        CardBartok cd = drawPile[0]; // Извлечь 0-ю карту

        if (drawPile.Count == 0) {   // Если список drawPile опустел
            // нужно перетасовать сброшенные карты
            //   и переложить их в стопку свободных карт
            int ndx;
            while (discardPile.Count > 0) {
                // Вынуть случайную карту из стопки сброшенных карт
                ndx = Random.Range(0, discardPile.Count);                    // a
                drawPile.Add( discardPile[ndx] );
                discardPile.RemoveAt( ndx );
            }
            ArrangeDrawPile();
            // Показать перемещение карт в стопку свободных карт
            float t = Time.time;
            foreach (CardBartok tCB in drawPile) {
                tCB.transform.localPosition = layout.discardPile.pos;
                tCB.callbackPlayer = null;
                tCB.MoveTo(layout.drawPile.pos);
                tCB.timeStart = t;
                t += 0.02f;
                tCB.state = CBState.toDrawpile;
                tCB.eventualSortLayer = "0";
            }
        }

        drawPile.RemoveAt(0);        // Удалить ее из списка drawPile
        return(cd);                  // И вернуть
    }
    ...
}
```

a. Проще воспользоваться циклом `while` и в каждой итерации вынимать случайную карту из `discardPile`, чем преобразовывать `discardPile` из типа `List<CardBartok>` в тип `List<Card>`, чтобы получить возможность вызвать `Deck.Shuffle()`.

Добавление пользовательского интерфейса в игру

Так же как в *Prospector*, мы должны сообщить игроку о завершении игры. Для этого нам потребуется создать несколько текстовых полей.

1. В главном меню выберите пункт **GameObject > UI > Text** (Игровой объект > ПИ > Текст), чтобы добавить новый объект **Text** в объект **Canvas** в иерархии.
2. Переименуйте только что созданный объект **Text** в **GameOver** и настройте его, как показано на рис. 33.6 слева.
3. Создайте копию объекта **GameOver**, выделив его и выбрав в меню пункт **Edit > Duplicate** (Правка > Дублировать).

4. Переименуйте вновь созданный объект **GameOver (1)** в **RoundResult** и настройте его, как показано на рис. 33.6 справа. Часто при изменении значений **Min** и **Max** в разделе **Anchors** Unity изменяет также **Pos X** и **Pos Y**. Это можно предотвратить, щелкнув на кнопке [R] в разделе **RectTransform**, как это сделал я, но даже после этого Unity иногда может ошибаться.

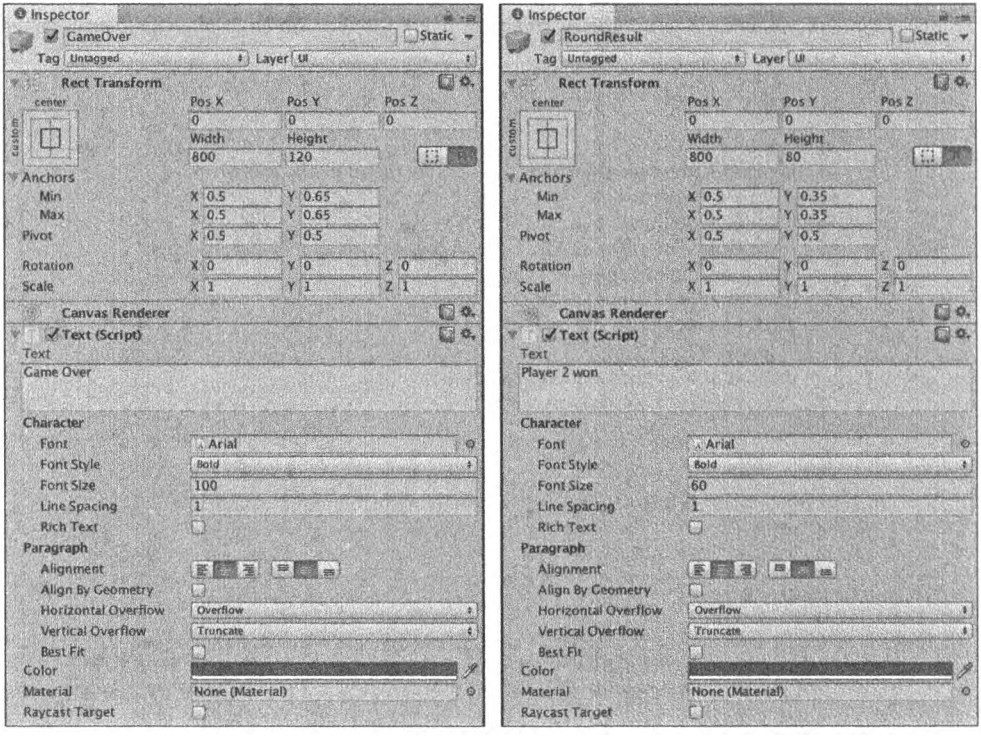


Рис. 33.6. Настройки объектов GameOver и RoundResult

По аналогии с источником света **TurnLight** можно для каждого поля **Text** написать свой сценарий, чтобы класс **Bartok** вообще не подозревал об их существовании.

5. Создайте в папке **__Scripts** новый сценарий на C# с именем **GameOverUI**, подключите его к объекту **GameOver** в иерархии и добавьте в него следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;                    // Необходимо для классов ПИ, таких как Text

public class GameOverUI : MonoBehaviour {
    private Text    txt;

    void Awake() {
```

```
        txt = GetComponent<Text>();
        txt.text = "";
    }

    void Update () {
        if (Bartok.S.phase != TurnPhase.gameOver) {
            txt.text = "";
            return;
        }
        // В эту точку мы попадаем, только когда игра завершилась
        if (Bartok.CURRENT_PLAYER == null) return;                          // a
        if (Bartok.CURRENT_PLAYER.type == PlayerType.human) {
            txt.text = "You won!";
        } else {
            txt.text = "Game Over";
        }
    }
}
```

a. В начале игры поле `Bartok.CURRENT_PLAYER` содержит `null`, и мы должны учесть этот случай.

6. Сохраните сценарий GameOverUI.

7. Создайте в папке __Scripts новый сценарий на C# с именем RoundResultUI, подключите его к объекту RoundResult в иерархии и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;                  // Необходимо для классов ПИ, таких как Text

public class RoundResultUI : MonoBehaviour {
    private Text    txt;

    void Awake() {
        txt = GetComponent<Text>();
        txt.text = "";
    }

    void Update () {
        if (Bartok.S.phase != TurnPhase.gameOver) {
            txt.text = "";
            return;
        }
        // В эту точку мы попадаем, только когда игра завершилась
        Player cP = Bartok.CURRENT_PLAYER;
        if (cP == null || cP.type == PlayerType.human) {                    // a
            txt.text = "";
        } else {
            txt.text = "Player "+(cP.playerNum)+" won";
        }
    }
}
```

a. Напомню, что оператор || (логическое ИЛИ) выполняется по короткой схеме, то есть если cP содержит null, эта строка никогда не обратится к полю cP.type и не столкнется с ошибкой обращения по пустой ссылке.

8. Сохраните сценарий RoundResultUI.

Логика завершения игры

Теперь, имея пользовательский интерфейс для отображения сообщений о завершении игры, добавим логику, которая поможет игре определить, что она завершилась.

1. Откройте сценарий Bartok и для управления завершением игры добавьте следующие строки, выделенные жирным.

```
public class Bartok : MonoBehaviour {
    ...

    public void PassTurn(int num=-1) {
        ...
        if (CURRENT_PLAYER != null) {
            lastPlayerNum = CURRENT_PLAYER.playerNum;
            // Проверить завершение игры и необходимость перетасовать
            //   стопку сброшенных карт
            if ( CheckGameOver() ) {
                return;                                                    // a
            }
        }
        ...
    }

    public bool CheckGameOver() {
        // Проверить, нужно ли перетасовать стопку сброшенных карт и
        //   перенести ее в стопку свободных карт
        if (drawPile.Count == 0) {
            List<Card> cards = new List<Card>();
            foreach (CardBartok cb in discardPile) {
                cards.Add (cb);
            }
            discardPile.Clear();
            Deck.Shuffle( ref cards );
            drawPile = UpgradeCardsList(cards);
            ArrangeDrawPile();
        }

        // Проверить победу текущего игрока
        if (CURRENT_PLAYER.hand.Count == 0) {
            // Игрок, только что сделавший ход, победил!
            phase = TurnPhase.gameOver;
            Invoke("RestartGame", 1);                                      // b
            return(true);
        }

        return(false);
    }
```

```
    public void RestartGame() {
        CURRENT_PLAYER = null;
        SceneManager.LoadScene("__Bartok_Scene_0");
    }

    // ValidPlay проверяет возможность сыграть выбранной картой
    public bool ValidPlay(CardBartok cb) { ... }
    ...
}
```

a. Если игра завершилась — выйти до передачи хода. В результате в CURRENT_PLAYER останется ссылка на победившего игрока, что позволит сценариям GameOverUI и RoundResultUI прочитать и вывести информацию о нем.

b. Вызов RestartGame() с задержкой в 1 секунду, чтобы перезапуск игры произошел через секунду после вывода результатов.

2. Сохраните сценарий Bartok, вернитесь в Unity и щелкните на кнопке Play (Играть).

Теперь игра не только правильно играется, но и завершается в нужный момент и автоматически перезапускается.

Сборка для WebGL

Теперь, когда у нас есть готовая игра, создадим ее версию, пригодную к распространению. Следующие инструкции описывают сборку для WebGL, но создание автономного приложения производится точно так же. Процесс сборки для iOS или Android включает больше шагов.

1. В главном меню Unity выберите пункт File > Build Settings (Файл > Параметры сборки). Откроется диалог, который мы использовали в начале главы.

2. В диалоге Build Settings (Параметры сборки) щелкните на кнопке Player Settings (Параметры игрока). В результате в панели Inspector (Инспектор) откроется диспетчер управления настройками игрока PlayerSettings. Если вы решили выполнить сборку игры для WebGL, настройки в PlayerSettings должны выглядеть так, как показано на рис. 33.7.

3. Щелкните на вкладке Resolution and Presentation (Разрешение и представление) в PlayerSettings и введите в поле Default Screen Width (Ширина экрана по умолчанию) число 1024, а в поле Default Screen Height (Высота экрана по умолчанию) число 768, как показано на рис. 33.7.

4. Поля Company Name (Название компании) и Product Name (Название продукта) можете заполнить по своему усмотрению. Все другие настройки в PlayerSettings можно оставить как есть. Сохраните сцену.

5. Вернитесь в диалог Build Settings (Параметры сборки), если вы закрыли его; выберите в главном меню пункт File > Build Settings (Файл > Параметры сборки) и щелкните на кнопке Build (Собрать).

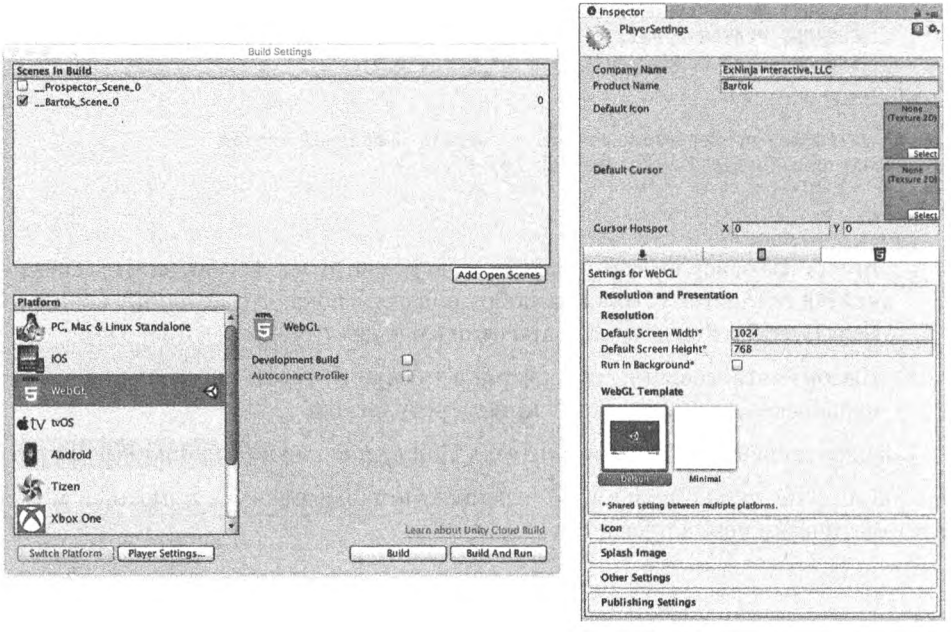


Рис. 33.7. Диалог Build Settings и диспетчер PlayerSettings в панели Inspector

6. Откроется стандартный диалог сохранения файла с предложением выбрать место для сохранения папки со сборкой для WebGL. Я советую сохранить ее на рабочий стол[1], чтобы потом проще было отыскать ее (по умолчанию предлагается сохранить папку со сборкой в каталог проекта, но мне такой выбор не кажется идеальным).

7. Введите имя папки в поле **Save As:** (Сохранить как:). Создавая сборки для WebGL, важно **не использовать пробелы в именах папок**; в некоторых системах попытка запустить файл внутри папки с именем, содержащим пробелы, вызывает аварийное завершение JavaScript (WebGL) (это еще одна причина, почему я не рекомендую сохранять сборку в папку проекта: она или любая другая папка уровнем выше может содержать пробелы в имени). Попробуйте, к примеру, дать имя *Bartok_WebGL*.

8. Щелкните на кнопке **Save** (Сохранить) и приготовьтесь ждать какое-то время. Иногда сборка для WebGL может выполняться в течение нескольких минут, а в редких случаях ожидание может затянутся даже до 30 минут или больше. Однако если в течение часа не наблюдается никакого прогресса, отмените сборку

[1] В русифицированной версии Windows рабочему столу соответствует папка *Рабочий стол* (содержит пробел в имени), из-за чего этот совет не подходит пользователям Windows в нашей стране (см. п. 7). Поэтому предпочтительнее выглядит вариант с сохранением папки со сборкой в корень файловой системы на диске *C:*. — *Примеч. пер.*

(иногда процесс сборки для WebGL завершается аварийно). Сборка игры *Bartok* выполняется примерно 5 минут на моем i7 MacBook Pro.

9. Найдите папку *Bartok_WebGL* на рабочем столе. Откройте ее и выполните двойной щелчок на файле *index.html*.

Вы можете увидеть такое сообщение об ошибке:

> It seems your browser does not support running Unity WebGL content from file:// urls. Please upload it to an http server, or try a different browser.[1]

Это объясняется стремлением некоторых браузеров, например Google Chrome, к повышенной безопасности, из-за чего они отказываются запускать код, такой как Unity WebGL, находящийся на локальном жестком диске. Поэкспериментировав, я обнаружил, что локальные файлы *index.html*, генерируемые средой Unity, прекрасно открывает браузер Firefox (по состоянию на июль 2017 года).

После того как вы выберете браузер, запускающий файлы с локального диска, или выгрузите сборку на HTTP-сервер, у вас должна появиться возможность сыграть в *Bartok* в веб-браузере.

Итоги

Целью этой главы была демонстрация использования прототипов из этой книги в качестве основы для создания своих игр. Закончив читать все главы с учебными примерами, вы получите основу для создания классических аркадных игр (*Apple Catcher*), казуальных игр с поддержкой физики (*Mission Demolition*), космических шутеров (*Space SHMUP*), карточных игр (*Prospector* и *Bartok*), игр со словами (в следующей главе) и приключенческих игр сверху вниз (*Dungeon Delver*). Будучи прототипом, ни один из перечисленных проектов не является законченной игрой, но любой может служить основой для других игр.

Следующие шаги

Классическая бумажная версия карточной игры *Bartok* дает победителям раундов возможность добавлять в игру свои правила. Конечно, добавление произвольных правил в цифровую игру не представляется возможным, но включать и выключать в коде действие необязательных и предопределенных правил вполне возможно, как это было показано на примере версии в главе 1.

На сайте книги http://book.prototools.net, в разделе Chapter 33, вы найдете проект Unity с расширенной версией *Bartok*, включающий все дополнительные правила, которые описывались в главе 1. Он послужит вам прекрасной отправной точкой для добавления в игру своих дополнительных правил.

---

[1] Ваш браузер не поддерживает запуск содержимого Unity WebGL из URL file://. Выгрузите приложение на http-сервер или попробуйте другой браузер. — *Примеч. пер.*

34 Прототип 6: Word Game

В этой главе вы узнаете, как создать простую игру в слова. В этой игре используются некоторые уже знакомые вам идеи, а также вводится новое для вас понятие *сопрограмм* — методов, которые могут прерывать свое выполнение, чтобы дать процессору возможность выполнить другие методы.

К концу этой главы вы получите забавную игру в слова, которую сможете продолжить расширять самостоятельно.

Начало: прототип 6

Как обычно, перед тем как начать читать эту главу, импортируйте пакет для Unity. Этот пакет содержит несколько графических ресурсов и сценариев на C#, созданных в предыдущих главах.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. Чтобы вспомнить, как это делается, обращайтесь к приложению A «Стандартная процедура настройки проекта». При создании проекта вам будет предложено выбрать настройки по умолчанию для дву- или трехмерной игры. Выберите настройки для трехмерной версии.

Главная сцена для этого проекта импортируется из пакета для Unity, поэтому вам не придется настраивать главную камеру _MainCamera.

- **Имя проекта:** Word Game.
- **Загрузите и импортируйте пакет:** находится в разделе Chapter 34 на сайте http://book.prototools.net.
- **Имя сцены:** _WordGame_Scene_0 (импортируется из пакета).
- **Папки проекта:** *_Scripts*, *_Prefabs*, *Materials & Textures*, *Resources*.
- **Имена сценариев на C#:** только импортированные сценарии в папке *ProtoTools*.

Откройте сцену __WordGame_Scene_0, и вы увидите, что главная камера _MainCamera уже настроена на ортографическую проекцию. Также некоторые сценарии многократного пользования, которые вы видели в предыдущих главах, теперь помещены в папку *__Scripts/ProtoTools*, чтобы отделить их от новых сценариев, которые будут созданы в этом проекте. Я считаю это очень удобным, потому что для добавления готовой функциональности достаточно просто скопировать папку *ProtoTools* в папку *__Scripts* нового проекта.

В настройках сборки Build Settings я советую выбрать платформу PC, Mac & Linux Standalone. В панели Game (Игра) настройте соотношение сторон, выбрав пункт Standalone (1024 x 768). Вы можете предусмотреть сборку этой игры для WebGL или мобильной платформы, но я не буду касаться этого вопроса в данной главе.

Об игре Word Game

Это классическая форма игры в слова. В числе коммерческих версий этой игры можно назвать *Word Whomp* студии Pogo.com, *Jumbline 2* студии Branium, *Pressed for Words* студии Words and Maps и многие другие. Игроку предлагается перемешанный набор букв, из которых можно составить не менее одного слова определенной длины (обычно шесть букв), и он должен найти все слова, которые можно создать перестановкой букв. Версия игры в этой главе включает некоторые простые анимационные эффекты (с использованием интерполяции вдоль кривых Безье) и модель подсчета очков, побуждающая игрока искать как можно более длинные слова. На рис. 34.1 показан скриншот игры, которую нам предстоит создать.

На этом скриншоте можно видеть, что каждое слово составлено из отдельных плиток с буквами, и плитки могут быть двух размеров: большого — для букв внизу экрана и маленького — для всех слов выше. Следуя принципам объектно-ориентированного программирования, мы определим класс `Letter` для работы с отдельными буквами и класс `Word`, в котором будут собираться буквы, составляющие слова. Также мы определим класс `WordList` для чтения большого словаря возможных слов и превратим его в удобный источник данных для игры. Управление игрой будет осуществлять класс `WordGame`. Для отображения набранных очков будут использоваться классы `Scoreboard` и `FloatingScore` из предыдущих прототипов. Кроме того, мы используем класс `Utils` для создания анимационных эффектов. Вместе с пакетом в проект импортирован класс `PT_XMLReader`, но здесь он не используется. Я оставил этот сценарий в импортируемом пакете, чтобы побудить вас начать собирать свою коллекцию полезных сценариев, которые вы сможете импортировать в любые проекты и использовать (как папку *ProtoTools* для проектов в этой книге). Не стесняйтесь добавлять любые полезные сценарии в эту коллекцию и вспоминайте о возможности импортировать ее всякий раз, когда начинаете создавать прототип новой игры.

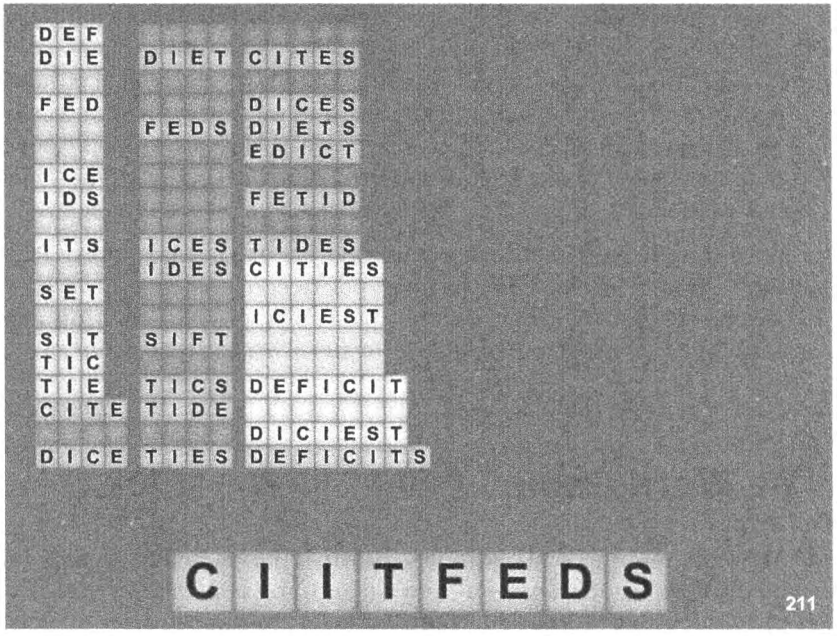


Рис. 34.1. Скриншот игры, которая будет создана в этой главе, использующей в качестве основы восьмисимвольное слово

Парсинг списка слов

Эта игра использует видоизмененный список слов *2of12inf*, созданный Аланом Билом[1]. Я удалил оскорбительные слова и постарался исправить некоторые из оставшихся. Вы свободно можете использовать этот список в будущем, при соблюдении условий авторских прав Алана Била и Кевина Аткинсона, как определено в сноске. Я также изменил список, преобразовав все буквы в верхний регистр и заменив заключительные символы строк `\r\n` (возврат каретки и перенос строки, которые стандартно используются в текстовых файлах в операционной системе Windows) на `\n` (единственный символ перевода строки, стандартно использу-

[1] Алан Бил (Alan Beale) выпустил все свои списки слов в общий доступ, кроме элементов списка *2of12inf*, основанных на списке слов AGID, авторское право на который в 2000 году зарегистрировано за Кевином Аткинсоном (Kevin Atkinson). Право использовать, копировать, изменять, распространять и продавать эту [AGID] базу данных, связанные с ней сценарии, вывод, генерируемый сценариями, и документацию к ним для любых целей предоставляется на безвозмездной основе, при условии включения уведомлений об авторских правах и этого примечания о разрешении во все копии и в сопроводительную документацию. Кевин Аткинсон не делает никаких заявлений о пригодности этого массива для любых целей. Он предоставляется «как есть», без явных или подразумеваемых гарантий.

емый в текстовых файлах в macOS). Я сделал это, чтобы упростить выделение отдельных слов из файла по символу перевода строки и для нужд этой главы. Получившийся результат с успехом можно использовать не только в macOS, но также в Windows.

Я попытался удалить оскорбительные слова из списка из-за особенностей игры. В таких играх, как *Scrabble* или *Letterpress*, игрок получает набор плиток с буквами и может выбирать, какие слова записать, используя эти буквы. Но в этой игре в зачет игроку идут только слова, присутствующие в списке. Это означает, что игра может вынудить игрока запечатлеть слова, оскорбительные для него. Решение о допустимости слов в этой игре принимает компьютер, а не игрок, и я посчитал неправильным заставлять игроков составлять слова, потенциально оскорбительные для них. Однако, учитывая, что в списке находится более 75 000 слов, я мог что-то пропустить, поэтому если вы найдете слова, которые, по вашему мнению, следовало бы исключить (или добавить), дайте мне знать, отправив сообщение на веб-сайте книги http://book.prototools.net. Заранее вам благодарен.

Чтобы прочитать файл со списком слов, нужно извлечь все содержимое файла в одну большую строку и затем разбить ее (по символу `\n`) на массив строк с отдельными словами. После этого нужно проанализировать каждое слово и решить, стоит ли добавлять его в словарь игры (исходя из его длины). На такой последовательный анализ слов требуется время; поэтому, чтобы не «заморозить» игру в одном кадре, пока процесс анализа не завершится, мы создадим *сопрограмму* (coroutine), которая будет выполнять анализ на протяжении множества кадров (см. врезку «Использование сопрограмм» ниже).

ИСПОЛЬЗОВАНИЕ СОПРОГРАММ

Сопрограмма (coroutine) — это функция, которая может приостановить свое выполнение, чтобы дать другим функциям выполнить свои вычисления. Это особенность Unity C#, которая позволяет разработчикам управлять выполнением повторяющихся задач или задач, занимающих слишком много времени. В этой главе вы научитесь применять ее для второго случая, создав сопрограмму для анализа всех 75 000 слов из списка *2of12inf*.

Запуск сопрограммы осуществляется вызовом функции `StartCoroutine()`, которая доступна только внутри классов, наследующих `MonoBehaviour`. После запуска сопрограмма продолжает выполняться, пока не встретит инструкцию `yield`. Эта инструкция приостанавливает сопрограмму на определенный интервал времени и позволяет во время этой паузы выполниться другому коду. По истечении заданного времени сопрограмма продолжит выполнение со строки, следующей за инструкцией `yield`. Это значит, что в сопрограмме можно организовать бесконечный цикл `while(true) {}` и это не «заморозит» игру, если, конечно, внутри цикла `while` имеется инструкция `yield`. В этой игре мы напишем сопрограмму `ParseLines()`, которая будет делать паузу после анализа каждых 10 000 слов.

Обратите внимание, как сопрограмма используется в первом листинге в этой главе. Хотя использование сопрограммы в этом примере не является острой необходимостью, особенно если у вас быстрый компьютер, тем не менее данный прием безусловно пригодится вам, когда вы начнете создавать игры для мобильных или других устройств со слабыми процессорами. Анализ того же самого списка слов на старой модели iPhone может длиться от 10 до 20 секунд, поэтому очень важно делать перерывы в процессе анализа, чтобы дать приложению возможность выполнять другие задачи и не выглядеть зависшим.

Более подробную информацию о сопрограммах вы найдете в документации к Unity.

1. Создайте в папке __Scripts новый сценарий на C# с именем WordList и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class WordList : MonoBehaviour {
    private static WordList   S;                                          // a

    [Header("Set in Inspector")]
    public TextAsset          wordListText;
    public int                numToParseBeforeYield = 10000;
    public int                wordLengthMin = 3;
    public int                wordLengthMax = 7;

    [Header("Set Dynamically")]
    public int                currLine = 0;
    public int                totalLines;
    public int                longWordCount;
    public int                wordCount;

    // Скрытые поля
    private string[]          lines;                                      // b
    private List<string>      longWords;
    private List<string>      words;

    void Awake() {
        S = this; // Подготовка объекта-одиночки WordList
    }

    void Start () {
        lines = wordListText.text.Split('\n');                            // c
        totalLines = lines.Length;

        StartCoroutine( ParseLines() );                                   // d
    }

    // Все сопрограммы возвращают значение типа IEnumerator.
    public IEnumerator ParseLines() {                                     // e
```

```
    string word;
    // Инициализировать список для хранения длиннейших слов
    //   из числа допустимых
    longWords = new List<string>();                                    // f
    words = new List<string>();

    for (currLine = 0; currLine < totalLines; currLine++) {            // g
        word = lines[currLine];

        // Если длина слова равна wordLengthMax...
        if (word.Length == wordLengthMax) {
            longWords.Add(word); // ...сохранить его в longWords
        }

        // Если длина слова между wordLengthMin и wordLengthMax...
        if ( word.Length>=wordLengthMin && word.Length<=wordLengthMax ) {
            words.Add(word); // ...добавить его в список допустимых слов
        }

        // Определить, не пора ли сделать перерыв
        if (currLine % numToParseBeforeYield == 0) {                   // h
            // Подсчитать слова в каждом списке, чтобы показать,
            //   как протекает процесс анализа
            longWordCount = longWords.Count;
            wordCount = words.Count;
            // Приостановить выполнение сопрограммы до следующего кадра
            yield return null;                                         // i

            // Инструкция yield приостановит выполнение этого метода,
            //   даст возможность выполниться другому коду и возобновит
            //   выполнение сопрограммы с этой точки, начав следующую
            //   итерацию цикла for.
        }
    }
    longWordCount = longWords.Count;
    wordCount = words.Count;
}

// Эти методы позволяют другим классам
//   обращаться к скрытым полям List<string>                          // j
static public List<string> GET_WORDS() {
    return( S.words );
}

static public string GET_WORD(int ndx) {
    return( S.words[ndx] );
}

static public List<string> GET_LONG_WORDS() {
    return( S.longWords );
}

static public string GET_LONG_WORD(int ndx) {
    return( S.longWords[ndx] );
}
```

```
    static public int WORD_COUNT {
        get { return S.wordCount; }
    }

    static public int LONG_WORD_COUNT {
        get { return S.longWordCount; }
    }

    static public int NUM_TO_PARSE_BEFORE_YIELD {
        get { return S.numToParseBeforeYield; }
    }

    static public int WORD_LENGTH_MIN {
        get { return S.wordLengthMin; }
    }

    static public int WORD_LENGTH_MAX {
        get { return S.wordLengthMax; }
    }
}
```

a. Это скрытый объект-одиночка (впрочем, из-за того, что он скрытый, это не совсем объект-одиночка). Объявив объект-одиночку `S` скрытым (`private`), мы гарантируем, что обращаться к нему смогут только экземпляры класса `WordList`. Этот объект-одиночка используется методами доступа, которые обсуждаются в пункте **j** ниже.

b. Так как эти поля объявлены скрытыми, они не появятся в инспекторе. Эти переменные будут содержать так много данных, что могут существенно замедлить работу Unity, если инспектор попытается отобразить их. Чтобы этого не произошло, они объявлены скрытыми — доступными только для экземпляра `WordList`, а чтобы сделать их доступными для внешнего кода, в конец класса добавлены общедоступные методы доступа.

c. Разбивает текст в `wordListText` по символам перевода строки (`\n`), создавая в результате большой массив `string[]`, каждый элемент которого хранит отдельное слово из списка.

d. Запускает сопрограмму `ParseLines()`. За дополнительной информацией обращайтесь к врезке «Использование сопрограмм» выше.

e. Все сопрограммы должны определяться с возвращаемым значением типа `IEnumerator`. Это позволяет им приостановиться и дать возможность выполниться другим методам перед возвратом в сопрограмму, что очень важно для таких процессов, как загрузка огромных файлов или анализ большого объема данных (как в этом случае).

f. Массив строк со словами будет поделен на два списка: `longWords` — все слова, состоящие из `wordLengthMax` букв, и `words` — все слова, насчитывающие от `wordLengthMin` до `wordLengthMax` (включительно) букв. Например, если предположить, что `wordLengthMin` имеет значение 3, а `wordLengthMax` — значение 6,

тогда слово DESIGN попадет в список `longWords`, а слова DIE, DICE, GAME, BOARD и DESIGN — в список `words`. Анализ всего списка слов происходит только один раз, поэтому игроку придется подождать только один раз, а потом он сможет сыграть множество раундов подряд с множеством разных слов.

g. Этот цикл `for` перебирает все 75 000 с лишним слов в массиве `lines`. Через каждые `numToParseBeforeYield` слов вызывается инструкция `yield`, которая приостанавливает цикл `for` и дает возможность выполниться другому коду. Затем в следующем кадре выполнение цикла `for` возобновляется и продолжается, пока не будут обработаны следующие `numToParseBeforeYield` слов.

h. Определяет, не пора ли приостановить выполнение сопрограммы. Здесь используется оператор `%` деления по модулю (остаток от деления нацело), чтобы определить, когда были обработаны очередные 10 000 записей (или любое другое количество, которое вы укажете в `numToParseBeforeYield`).

i. Инструкция `yield` приостанавливает выполнение сопрограммы до следующего кадра, потому что возвращает `null`. Также можно приостановить сопрограмму на определенное количество секунд, например `yield return new WaitForSeconds(1);`, в данном случае сопрограмма приостановится не менее чем на 1 секунду (обратите внимание, что время приостановки сопрограммы выдерживается с удовлетворительной, но все же не абсолютной точностью). Это также означает, что повторяющиеся действия через определенные интервалы времени можно выполнять не только с применением метода `InvokeRepeating()`, но и с помощью сопрограммы.

j. Четыре метода, следующие за строкой с комментарием `// j`, — это статические общедоступные методы доступа к скрытым полям `words` и `longWords`. Любой код в игре может вызвать `WordList.GET_WORD(10)`, чтобы получить десятое слово из скрытого массива `words` в этом экземпляре-одиночке `WordList`. Кроме того, последние несколько методов доступа являются статическими общедоступными свойствами только для чтения и демонстрируют еще один способ доступа к скрытым переменным в `WordList`. По соглашениям статическим переменным и методам часто даются имена, содержащие только `СИМВОЛЫ_ВЕРХНЕГО_РЕГИСТРА`.

2. Сохраните сценарий и вернитесь в Unity.

3. Подключите сценарий **WordList** к главной камере **_MainCamera**.

4. Выберите **_MainCamera** в иерархии и в инспекторе присвойте переменной **wordListText** компонента **WordList (Script)** ссылку на файл **2of12inf**, который вы найдете в папке **Resources** в панели **Project** (Проект).

5. Щелкните на кнопке **Play** (Играть).

   Вы увидите, как начнут увеличиваться значения **currLine**, **longWordCount** и **wordCount** с шагом 10 000. Это объясняется тем, что числа могут обновляться, только когда приостанавливается сопрограмма `ParseLines()`.

Если остановить игру, изменить в инспекторе значение `numToParseBeforeYield` на 100 и снова запустить ее, вы увидите, что анализ списка слов происходит медленнее, потому что сопрограмма приостанавливается через каждые 100 слов. Но если подставить большое значение, например 100 000, числа обновятся только один раз, потому что число слов в списке меньше 100 000. Если вам интересно узнать, сколько времени тратит сопрограмма `ParseLines()` на каждый проход, попробуйте воспользоваться профилировщиком, как описывается во врезке «Профилировщик Unity».

ПРОФИЛИРОВЩИК UNITY

Профилировщик Unity — один из самых мощных инструментов для оптимизации производительности игр и один из многих, доступных в бесплатной версии Unity. В каждом кадре профилировщик собирает статистику по времени, затраченному на вызов каждой функции, обращение к графическому движку, обработку ввода пользователя и т. д. Отличный пример работы профилировщика можно увидеть, запустив его в этом проекте.

1. Проверьте безошибочную работу сценария WordList, представленного на предыдущих страницах.

2. Добавьте панель Profiler (Профилировщик) в одну группу с панелью Scene (Сцена). Это позволит одновременно наблюдать за происходящим в панелях Game (Игра) и Profiler (Профилировщик). Чтобы добавить панель Profiler (Профилировщик), щелкните на кнопке в правом верхнем углу панели Scene (Сцена) и в открывшемся меню выберите Add Tab > Profiler (Добавить вкладку > Профилировщик), как показано на рис. 34.2.

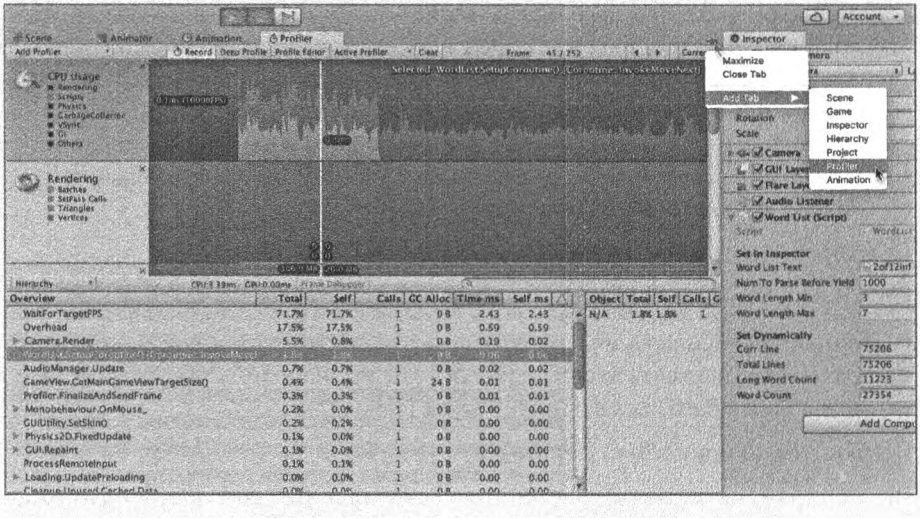


Рис. 34.2. Панель Profiler (Профилировщик)

3. Чтобы увидеть, как действует профилировщик, остановите игру щелчком на кнопке Pause (Пауза) вверху в окне Unity и затем щелкните на кнопке Play (Играть). Это заставит Unity подготовиться к запуску игры и приостановиться перед первым кадром. Если снова щелкнуть на кнопке Pause (Пауза), профилировщик начнет рисовать график. Приостановите игру, когда левый конец графика окажется примерно в паре сантиметров от левого края экрана.

   Когда игра приостановлена, профилировщик должен прекратить рисовать график, но сохранить график, накопленный за прошлые кадры. Каждому цвету под заголовком CPU Usage (Использование CPU) соответствует свой круг задач, для решения которых использовался CPU (главный процессор вашего компьютера). Если у вас быстрый компьютер, в графике за прошедшие кадры должен преобладать желтый цвет; желтый соответствует времени, потраченному движком Unity на вертикальную синхронизацию (VSync), то есть на ожидание, когда экран будет готов к отображению следующего кадра. Он мешает увидеть, сколько процессорного времени потратили ваши сценарии (светло-синий цвет), поэтому его нужно скрыть на графике.

4. Цветные метки под заголовком CPU Usage (Использование CPU) с левой стороны в окне профилировщика представляют разные виды процессов, выполняемых процессором. Прямо сейчас выключите их все, кроме синей метки Scripts (Сценарии). Для этого щелкните на каждом цветном квадратике, кроме Scripts (Сценарии). В результате должен остаться только синий график, как показано на рис. 34.2.

5. Наведите указатель мыши на область графика, нажмите левую кнопку мыши и, не отпуская ее, проведите указатель слева направо вдоль синего графика в разделе CPU Usage (Использование CPU). Вы должны увидеть, как появится белая вертикальная линия, перемещающаяся вслед за указателем мыши. Эта линия представляет один кадр на графике. При ее перемещении текст в нижней половине окна профилировщика будет обновляться и отражать количество процессорного времени, потраченного каждой функцией или фоновым процессом в этом кадре. В данном случае нас интересует сопрограмма WordList.SetupCoroutine() [Coroutine: InvokeMoveNext]. Она выполняется только в нескольких первых кадрах, поэтому вы не встретите ее в правой части графика; однако в начале графика должен быть ясно виден всплеск активности сценария (как показано на рис. 34.2), который соответствует большим затратам времени на выполнение сопрограммы `ParseLines()`.

6. В панели Profiler (Профилировщик), в полосе, разделяющей верхнюю и нижнюю половины, имеется поле поиска. (Если вы не видите его, попробуйте щелкнуть мышью где-нибудь в области графика CPU Usage (Использование CPU), в верхней половине.) Введите в это поле текст «ParseLines», чтобы отыскать метод WordList:ParseLines. Этот метод выполняется только в нескольких первых кадрах, поэтому вы не встретите его в правой части графика; однако в начале графика должен быть ясно виден всплеск активности сценария (как показано на рис. 34.2).

7. Перетащите белую вертикальную линию в область всплеска на графике, и в области данных под графиком появятся две строки WordList.ParseLines(). Щелкните

на строке WordList.ParseLines() [Coroutine: MoveNext]. В результате на графике подсветится вклад этой сопрограммы и потускнеет вклад всех остальных задач, как показано на рисунке. С помощью кнопок со стрелками влево и вправо, находящимися в правом верхнем углу панели Profiler (Профилировщик), можно переместиться на один кадр назад или вперед (соответственно) и посмотреть, сколько процессорного времени было потрачено сопрограммой в каждом кадре. Чтобы получить скриншот на рис. 34.2, я установил значение 1000 в поле numToParseBeforeYield и обнаружил, что в течение нескольких первых кадров сопрограмма тратит около 6,7% от общего процессорного времени, затраченного на каждый кадр (у вас эти цифры могут отличаться, потому что они зависят от модели компьютера и его быстродействия).

Помимо профилирования сценариев профилировщик может также помочь определить, какие аспекты отображения или моделирования физики в игре потребляют больше всего времени. Если вам когда-нибудь доведется столкнуться с проблемой низкой частоты кадров, попробуйте проверить ее с помощью профилировщика, чтобы понять причины. (В этом случае вам потребуется вновь включить отображение графиков потребления процессорного времени другими механизмами (то есть вновь щелкнуть на каждом цветном квадратике, которые мы выключили ранее, чтобы выделить график Scripts (Сценарии)).)

Чтобы увидеть совершенно отличающийся график профилирования, попробуйте запустить профилировщик с проектом *Hello World* из главы 19 «Hello World: ваша первая программа». Вы увидите, что в *Hello World* намного больше времени тратится на моделирование физики, чем на выполнение сценариев. (Возможно, вам снова придется выключить элемент VSync, чтобы он не загромождал график.)

Дополнительную информацию о профилировщике вы найдете в документации к Unity.

Поиграв с профилировщиком, верните значение 10 000 в поле numToParseBeforeYield.

Настройка игры

Наша следующая цель — определить класс `WordGame`, который будет управлять игрой, но перед этим мы должны внести несколько изменений в WordList. Во-первых, перенесем запуск анализа списка слов из метода `Start()` в метод `Init()`, который будет вызываться другим классом. Во-вторых, сценарий WordList должен уведомить будущий сценарий WordGame, когда закончит анализ слов. Для этого организуем отправку сообщения из WordList объекту _MainCamera с использованием метода `SendMessage()`. Как сценарий WordGame будет интерпретировать это сообщение, вы увидите чуть позже.

1. Измените имя метода `void Start()` в сценарии WordList на `public void Init()` и добавьте следующие строки, выделенные жирным, включая функцию `static public void INIT()` и строки в конце метода `ParseLines()`:

```
public class WordList : MonoBehaviour {
    ...
    void Awake() { ... }

    public void Init () { // Эта строка заменила "void Start()"
        lines = wordListText.text.Split('\n');
        totalLines = lines.Length;

        StartCoroutine( ParseLines() );
    }

    static public void INIT () {                                    // a
        S.Init();
    }

    // Все сопрограммы возвращают значение типа IEnumerator.
    public IEnumerator ParseLines() {
        ...
        for (currLine = 0; currLine < totalLines; currLine++) {
            ...
        }
        longWordCount = longWords.Count;
        wordCount = words.Count;

        // Послать игровому объекту gameObject сообщение об окончании анализа
        gameObject.SendMessage("WordListParseComplete");            // b
    }

    // Эти методы позволяют другим классам
    //   обращаться к скрытым полям List<string>
    static public List<string> GET_WORDS() { ... }
    ...
}
```

a. Этот метод `INIT()` объявлен статическим и общедоступным, чтобы его мог вызвать класс `WordGame`.

b. Вызов метода `SendMessage()` игрового объекта `_MainCamera` (поэтому `WordList` является компонентом сценария объекта `_MainCamera`). Этот метод вызовет метод `WordListParseComplete()` в любом сценарии, подключенном к игровому объекту, которому принадлежит метод `SendMessage()` (то есть к `_MainCamera`).

2. Создайте в папке `__Scripts` сценарий на C# с именем `WordGame` и подключите его к `_MainCamera`. Введите следующий код, чтобы воспользоваться изменениями, только что произведенными в `WordList`:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq; // Мы будем использовать LINQ

public enum GameMode {
    preGame,   // Перед началом игры
    loading,   // Список слов загружается и анализируется
```

```
    makeLevel, // Создается отдельный WordLevel
    levelPrep, // Создается уровень с визуальным представлением
    inLevel    // Уровень запущен
}

public class WordGame : MonoBehaviour {
    static public WordGame    S; // Одиночка

    [Header("Set Dynamically")]
    public GameMode           mode = GameMode.preGame;

    void Awake() {
        S = this; // Записать ссылку на объект-одиночку
    }

    void Start () {
        mode = GameMode.loading;
        // Вызвать статический метод INIT() класса WordList
        WordList.INIT();
    }

    // Вызывается методом SendMessage() из WordList
    public void WordListParseComplete() {
        mode = GameMode.makeLevel;
    }
}
```

3. Выберите _MainCamera в панели Hierarchy (Иерархия) и раскройте содержимое компонента WordGame (Script) в инспекторе. Щелкните на кнопке Play (Играть), и вы увидите, как значение preGame в поле mode сменится на loading. Затем, когда анализ списка слов закончится, значение loading сменит значение makeLevel. Это показывает, что все работает именно так, как мы предполагали.

Создание уровня с помощью класса WordLevel

Теперь пришла пора на основе слов из WordList создать уровень. Класс `WordLevel` будет включать:

- Длинное слово, на котором основывается уровень. (Если `maxWordLength` равно 6, это будет шестисимвольное слово, буквы из которого будут использоваться для конструирования других слов.)
- Индекс этого слова в массиве `longWords` класса `WordList`.
- Номер уровня в `int levelNum`. В этой главе при запуске каждого нового сеанса игра будет выбирать случайное слово[1].

[1] При желании можно вызвать `Random.InitState(1);` в методе `WordGame.Awake()`, чтобы установить 1 как начальное число последовательности в `Random` и гарантировать, что восьмой уровень всегда будет начинаться с одного и того же слова при условии, что выбор уровня производится с помощью `Random`. Другой способ достичь этой цели я опишу в разделе «Следующие шаги» в конце главы.

- Словарь `Dictionary<,>` с символами в слове и количеством раз их использования. Словари являются частью `System.Collections.Generic` вместе со списками.
- Список `List<>` всех других слов, которые можно сформировать из символов в словаре, описанном в предыдущем пункте.

Словарь `Dictionary<,>` — это обобщенный тип коллекции, хранящей множество пар ключ-значение, о котором подробно рассказывалось в главе 23 «Коллекции в C#». На каждом уровне создается словарь `Dictionary<,>` с символьными ключами и целочисленными значениями, соответствующими количеству раз использования символа в длинном слове. Например, вот как выглядит слово MISSISSIPPI в форме словаря:

```
Dictionary<char,int> charDict = new Dictionary<char,int>();
charDict.Add('M',1); // MISSISSIPPI содержит 1 букву M
charDict.Add('I',4); // MISSISSIPPI содержит 4 буквы I
charDict.Add('S',4); // MISSISSIPPI содержит 4 буквы S
charDict.Add('P',2); // MISSISSIPPI содержит 2 буквы P
```

Класс `WordLevel` имеет также два удобных статических метода:

- `MakeCharDict()`: на основе полученной строки заполняет словарь `charDict`.
- `CheckWordInLevel()`: проверяет, можно ли составить заданное слово из символов, имеющихся в словаре `charDict`.

1. Создайте в папке `__Scripts` новый сценарий на C# с именем `WordLevel` и введите следующий код. Обратите внимание, что класс `WordLevel` не наследует `MonoBehaviour`, поэтому его нельзя подключить ни к какому игровому объекту в виде компонента сценария и он не имеет функций `StartCoroutine()`, `SendMessage()` и многих других, характерных для Unity.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[System.Serializable] // Экземпляры WordLevel можно просматривать/изменять
                      // в инспекторе
public class WordLevel { // WordLevel НЕ наследует MonoBehaviour
    public int                      levelNum;
    public int                      longWordIndex;
    public string word;
    // Словарь со всеми буквами в слове
    public Dictionary<char,int>     charDict;
    // Все слова, которые можно составить из букв в charDict
    public List<string>             subWords;

    // Статическая функция, подсчитывает количество вхождений символов в строку
    //   и возвращает словарь Dictionary<char,int> с этой информацией
    static public Dictionary<char,int> MakeCharDict(string w) {
        Dictionary<char,int> dict = new Dictionary<char, int>();
        char c;
        for (int i=0; i<w.Length; i++) {
            c = w[i];
```

```
            if (dict.ContainsKey(c)) {
                dict[c]++;
            } else {
                dict.Add (c,1);
            }
        }
        return(dict);
    }

    // Статический метод, проверяет возможность составить слово
    //   из символов в level.charDict
    public static bool CheckWordInLevel(string str, WordLevel level) {
        Dictionary<char,int> counts = new Dictionary<char, int>();
        for (int i=0; i<str.Length; i++) {
            char c = str[i];
            // Если charDict содержит символ c
            if (level.charDict.ContainsKey(c)) {
                // Если counts еще не содержит ключа с символом c
                if (!counts.ContainsKey(c)) {
                    // ...добавить новый ключ со значением 1
                    counts.Add (c,1);
                } else {
                    // В противном случае прибавить 1 к текущему значению
                    counts[c]++;
                }

                // Если число вхождений символа c в строку str
                //   превысило доступное количество в level.charDict
                if (counts[c] > level.charDict[c]) {
                    // ... вернуть false
                    return(false);
                }
            } else {
                // Символ c отсутствует в level.word, вернуть false
                return(false);
            }
        }
        return(true);
    }
}
```

2. Чтобы задействовать класс `WordLevel`, внесите следующие изменения в сценарий WordGame:

```
public class WordGame : MonoBehaviour {
    static public WordGame    S; // Одиночка

    [Header("Set Dynamically")]
    public GameMode           mode = GameMode.preGame;
    public WordLevel          currLevel;

    ...

    public void WordListParseComplete() {
        mode = GameMode.makeLevel;
        // Создать уровень и сохранить в currLevel текущий WordLevel
```

```
        currLevel = MakeWordLevel();
    }

    public WordLevel MakeWordLevel(int levelNum = -1) {                    // a
        WordLevel level = new WordLevel();
        if (levelNum == -1) {
            // Выбрать случайный уровень
            level.longWordIndex = Random.Range(0,WordList.LONG_WORD_COUNT);
        } else {
            // Код в эту ветку будет добавлен далее в этой главе
        }
        level.levelNum = levelNum;
        level.word = WordList.GET_LONG_WORD(level.longWordIndex);
        level.charDict = WordLevel.MakeCharDict(level.word);

        StartCoroutine( FindSubWordsCoroutine(level) );                    // b
        return( level );                                                   // c
    }

    // Сопрограмма, отыскивающая слова, которые можно составить на этом уровне
    public IEnumerator FindSubWordsCoroutine(WordLevel level) {
        level.subWords = new List<string>();
        string str;

        List<string> words = WordList.GET_WORDS();                         // d

        // Выполнить обход всех слов в WordList
        for (int i=0; i<WordList.WORD_COUNT; i++) {
            str = words[i];
            // Проверить, можно ли его составить из символов в level.charDict
            if (WordLevel.CheckWordInLevel(str, level)) {
                level.subWords.Add(str);
            }
            // Приостановиться после анализа заданного числа слов в этом кадре
            if (i%WordList.NUM_TO_PARSE_BEFORE_YIELD == 0) {
                // приостановиться до следующего кадра
                yield return null;
            }
        }

        level.subWords.Sort ();                                            // e
        level.subWords = SortWordsByLength(level.subWords).ToList();

        // Сопрограмма завершила анализ, поэтому вызываем SubWordSearchComplete()
        SubWordSearchComplete();
    }

    // Использует LINQ для сортировки массива и возвращает его копию      // f
    public static IEnumerable<string> SortWordsByLength(IEnumerable<string> ws) {
        ws = ws.OrderBy(s => s.Length);
        return ws;
    }

    public void SubWordSearchComplete() {
        mode = GameMode.levelPrep;
    }
}
```

a. Получив значение по умолчанию –1, этот метод выберет случайное слово, чтобы сгенерировать экземпляр `WordLevel`.

b. Запускает сопрограмму, которая проверит все слова в `WordList` и выберет те из них, что состоят из символов, имеющихся в `level.charDict`.

c. Возвращает `WordLevel level` до того, как сопрограмма завершит проверку. Когда сопрограмма закончит поиск, она вызовет `SubWordSearchComplete()`.

d. Выполняется очень быстро, потому что списки передаются по ссылке. (Поэтому сценарию WordList не придется создавать копию списка `List<string> words` — он просто вернет ссылку на него.)

e. Эти две строки сортируют слова в списке `WordLevel.subWords`. Метод `Sort()` отсортирует слова по алфавиту (такой порядок принят по умолчанию для `List<String>`). Затем наш метод `SortWordsByLength()` отсортирует слова по количеству символов в них. В результате слова будут объединены в группы с одинаковой длиной и отсортированы по алфавиту внутри каждой группы.

f. Эта функция использует механизм запросов LINQ для сортировки полученного массива и возвращает копию. Синтаксис LINQ отличается от синтаксиса языка C# и не обсуждается в этой книге. Дополнительную информацию можно найти, выполнив поиск в интернете по фразе «C# LINQ».

   Хорошее описание LINQ можно также найти на веб-сайте *Unity Gems*. Ниже приводится ссылка на Internet Archive, чтобы гарантировать, что она останется действительной к моменту, когда вы будете читать эти строки:

   https://web.archive.org/web/20140209060811/http://unitygems.com/linq-1-time-linq/

Этот код создает экземпляр уровня, выбирает целевое слово и заполняет список `subWords` словами, которые можно составить из символов целевого слова. Сохраните все сценарии, вернитесь в Unity и щелкните на кнопке Play (Играть). Теперь в инспекторе вы должны увидеть заполненное поле currLevel компонента WordGame (Script) объекта _MainCamera.

3. Сохраните сцену! Если до сих пор вы ни разу не сохранили сцену и сделали это, только увидев этот пункт, значит, вам нужно чаще напоминать себе о необходимости сохранять сцену.

Компоновка экрана

После создания информационного представления уровня пришло время заняться созданием визуальных элементов на экране, представляющих большие плитки с буквами, которые можно использовать для составления слов, и маленькие — для отображения самих слов. Сначала создадим шаблон PrefabLetter, из которого будут создаваться экземпляры плиток с буквами.

Создание PrefabLetter

Выполните следующие шаги, чтобы создать шаблон PrefabLetter:

1. В главном меню выберите пункт GameObject > 3D Object > Quad (Игровой объект > 3D объект > Квадрат). Переименуйте вновь созданный объект квадрата в PrefabLetter.
2. В главном меню выберите пункт Assets > Create > Material (Ресурсы > Создать > Материал). Дайте новому материалу имя LetterMat и поместите его в папку Materials & Textures.
3. Перетащите LetterMat на объект PrefabLetter в иерархии. Щелкните на PrefabLetter и выберите шейдер Unlit > Transparent для материала LetterMat.
4. Выберите текстуру Rounded Rect 256 для материала LetterMat (вам может понадобиться щелкнуть на пиктограмме с треугольником рядом с именем LetterMat в инспекторе).
5. Дважды щелкните на PrefabLetter в иерархии, и вы увидите в сцене квадрат с закругленными углами. Если этого не произошло, попробуйте переместить камеру на другую сторону. Во врезке «Отбрасывание нелицевых граней» рассказывается, почему квадраты могут быть видны с одной стороны и не видны с другой. Чтобы квадраты были видны с любой стороны, выберите шейдер ProtoTools > UnlitAlpha для материала LetterMat.

ОТБРАСЫВАНИЕ НЕЛИЦЕВЫХ ГРАНЕЙ

Отбрасывание нелицевых граней (backface culling) — это прием оптимизации отображения. Полигоны отображаются, только когда они повернуты лицевой стороной к точке зрения. Эта оптимизация дает хорошие результаты при отображении замкнутых фигур, таких как сфера. При отображении сферы только половина полигонов, образующих ее поверхность, обращена к зрителю лицевой стороной, а другая половина (на дальней поверхности) обращена лицевой стороной в противоположную сторону. Специалисты по компьютерной графике установили, что для отображения целой сферы достаточно отобразить только полигоны, обращенные лицевой стороной к зрителю. Полигоны на противоположной стороне сферы — обращенные лицевой стороной от зрителя — отображать не имеет смысла. *Грани* (то есть полигоны), повернутые к зрителю *обратной*, *нелицевой* стороной, *исключаются* из отображения, то есть *отбрасываются*; отсюда появился термин «отбрасывание нелицевых граней».

Квадраты в Unity состоят из двух треугольных полигонов, повернутых лицевой стороной в одну сторону. Когда камера в Unity смотрит на квадрат сзади, оба полигона обычно исключаются из отображения и квадрат оказывается невидимым.

В Unity есть несколько шейдеров, не использующих оптимизацию отбрасывания нелицевых граней, включая шейдер UnlitAlpha из ProtoTools, который использовался в главе 31 «Прототип 3.5: SPACE SHMUP PLUS».

6. Щелкните правой кнопкой на PrefabLetter в иерархии и в контекстном меню выберите пункт 3D Object > 3D Text (3D объект > 3D Текст). В результате будет создан игровой объект New Text, вложенный в PrefabLetter.
7. Измените имя объекта New Text на 3D Text (с пробелом после «3D»).
8. Выберите 3D Text в иерархии и настройте его, как показано на рис. 34.3. Если буква W отобразится не в центре квадрата, значит, вы случайно ввели символ табуляции после W в поле Text (именно это случилось со мной, когда я работал над этим прототипом).
9. Перетащите PrefabLetter из иерархии в папку _Prefabs в панели Project (Проект) и удалите оставшийся экземпляр PrefabLetter из иерархии. Сохраните сцену.

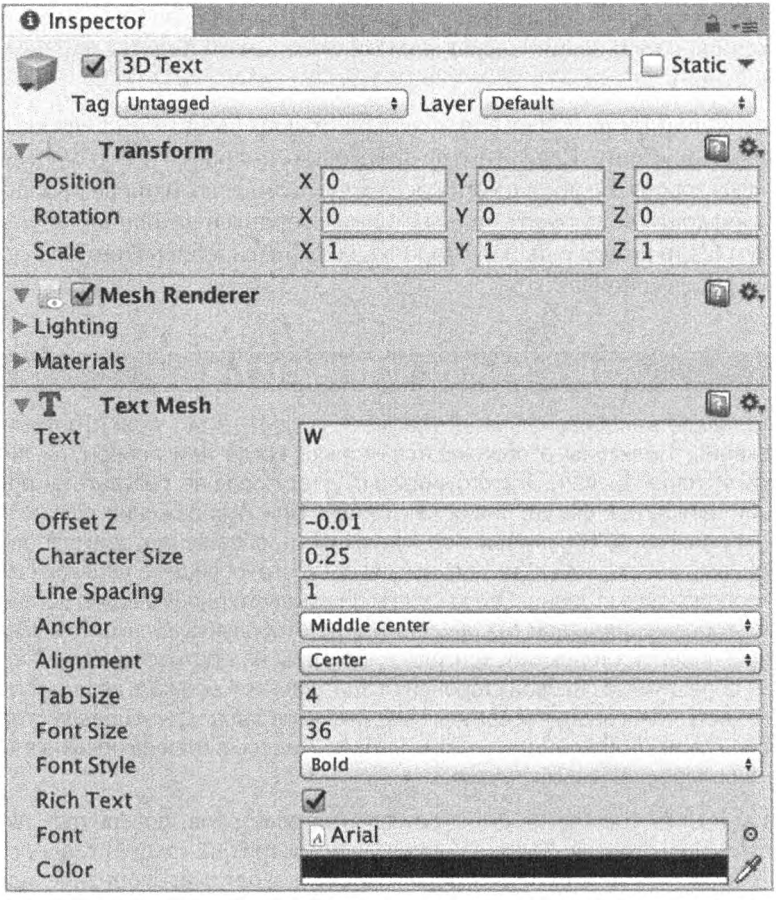


Рис. 34.3. Настройки объекта 3D Text в инспекторе, вложенного в шаблон PrefabLetter

Сценарий Letter

Теперь создадим для шаблона PrefabLetter отдельный сценарий на C#, который будет обслуживать настройки отображения символа, его цвет и многое другое.

1. Создайте в папке __Scripts новый сценарий на C# с именем Letter и подключите его к PrefabLetter.
2. Откройте сценарий в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Letter : MonoBehaviour {
    [Header("Set Dynamically")]
    public TextMesh         tMesh; // TextMesh отображает символ
    public Renderer         tRend; // Компонент Renderer объекта 3D Text. Он
                                   //   будет определять видимость символа
    public bool             big = false; // Большие и малые плитки действуют
                                         // по-разному

    private char            _c;    // Символ, отображаемый на этой плитке
    private Renderer        rend;

    void Awake() {
        tMesh = GetComponentInChildren<TextMesh>();
        tRend = tMesh.GetComponent<Renderer>();
        rend = GetComponent<Renderer>();
        visible = false;
    }

    // Свойство для чтения/записи буквы в поле _c, отображаемой объектом 3D Text
    public char c {
        get { return( _c ); }
        set {
            _c = value;
            tMesh.text = _c.ToString();
        }
    }

    // Свойство для чтения/записи буквы в поле _c в виде строки
    public string str {
        get { return( _c.ToString() ); }
        set { c = value[0]; }
    }

    // Разрешает или запрещает отображение 3D Text, что делает букву
    //   видимой или невидимой соответственно.
    public bool visible {
        get { return( tRend.enabled ); }
        set { tRend.enabled = value; }
    }

    // Свойство для чтения/записи цвета плитки
```

```
    public Color color {
        get { return(rend.material.color); }
        set { rend.material.color = value; }
    }

    // Свойство для чтения/записи координат плитки
    public Vector3 pos {
        set {
            transform.position = value;
            // Далее мы добавим дополнительный код
        }
    }
}
```

Этот класс определяет несколько свойств (имитации полей с методами доступа `get{}` и `set{}`) для выполнения разных действий при попытке присвоить переменной новое значение. С их помощью, например, класс `WordGame` сможет назначать символ `c` для отображения на плитке, не заботясь о преобразовании символа в строку и отображении его с помощью 3D Text. Такая инкапсуляция функциональности внутри класса является одной из основ объектно-ориентированного программирования. Обратите внимание, что когда `get{}` или `set{}` можно реализовать одной инструкцией, я часто сворачиваю их определения в одну строку.

Класс Wyrd: коллекция плиток с буквами

Класс `Wyrd` будет действовать как коллекция плиток с буквами, а его имя записано через `y`, чтобы отличить его от других экземпляров слова *word*, часто встречающихся в этом коде и в тексте книги. `Wyrd` — это еще один класс, который не наследует `MonoBehaviour` и который нельзя подключить к игровому объекту, но он содержит списки классов, подключенных к игровым объектам.

1. Создайте в папке __Scripts новый сценарий на C# с именем Wyrd.
2. Откройте сценарий Wyrd в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Wyrd {          // Wyrd не наследует MonoBehaviour
    public string       str; // Строковое представление слова
    public List<Letter> letters = new List<Letter>();
    public bool         found = false; // Получит true, если игрок нашел это слово

    // Свойство, управляющее видимостью компонента 3D Text каждой плитки Letter
    public bool visible {
        get {
            if (letters.Count == 0) return(false);
                return(letters[0].visible);
        }
        set {
            foreach( Letter l in letters) {
```

```
                l.visible = value;
            }
        }
    }

    // Свойство для назначения цвета каждой плитке Letter
    public Color color {
        get {
            if (letters.Count == 0) return(Color.black);
                return(letters[0].color);
        }
        set {
            foreach( Letter l in letters) {
                l.color = value;
            }
        }
    }

    // Добавляет плитку в список letters
    public void Add(Letter l) {
        letters.Add(l);
        str += l.c.ToString();
    }
}
```

Метод WordGame.Layout()

Метод `Layout()` сгенерирует экземпляры `Wyrd` и `Letter`, а также экземпляры больших плиток `Letter`, которые игрок сможет использовать для составления слов (большие плитки серого цвета с буквами внизу на рис. 34.1). Начнем с маленьких плиток, и на этом этапе разработки прототипа все буквы на плитках будут видимы (но в окончательной версии мы их скроем).

1. Добавьте в класс `WordGame` следующие строки, выделенные жирным:

```
public class WordGame : MonoBehaviour {
    static public WordGame    S; // Объект-одиночка

    [Header("Set in Inspector")]
    public GameObject         prefabLetter;
    public Rect               wordArea = new Rect( -24, 19, 48, 28 );
    public float              letterSize = 1.5f;
    public bool               showAllWyrds = true;

    [Header("Set Dynamically")]
    public GameMode           mode = GameMode.preGame;
    public WordLevel          currLevel;
    public List<Wyrd>         wyrds;

    private Transform         letterAnchor, bigLetterAnchor;

    void Awake() {
        S = this; // Записать ссылку на объект-одиночку
        letterAnchor = new GameObject("LetterAnchor").transform;
```

```
        bigLetterAnchor = new GameObject("BigLetterAnchor").transform;
    }

    ...

    public void SubWordSearchComplete() {
        mode = GameMode.levelPrep;
        Layout(); // Вызвать Layout() один раз после выполнения WordSearch
    }

    void Layout() {
        // Поместить на экран плитки с буквами каждого возможного
        //   слова текущего уровня
        wyrds = new List<Wyrd>();

        // Объявить локальные переменные, которые будут использоваться методом
        GameObject go;
        Letter lett;
        string word;
        Vector3 pos;
        float left = 0;
        float columnWidth = 3;
        char c;
        Color col;
        Wyrd wyrd;

        // Определить, сколько рядов плиток уместится на экране
        int numRows = Mathf.RoundToInt(wordArea.height/letterSize);

        // Создать экземпляр Wyrd для каждого слова в level.subWords
        for (int i=0; i<currLevel.subWords.Count; i++) {
            wyrd = new Wyrd();
            word = currLevel.subWords[i];

            // если слово длиннее, чем columnWidth, развернуть его
            columnWidth = Mathf.Max( columnWidth, word.Length );

            // Создать экземпляр PrefabLetter для каждой буквы в слове
            for (int j=0; j<word.Length; j++) {
                c = word[j]; // Получить j-й символ слова
                go = Instantiate<GameObject>( prefabLetter );
                go.transform.SetParent( letterAnchor );
                lett = go.GetComponent<Letter>();
                lett.c = c; // Назначить букву плитке Letter

                // Установить координаты плитки Letter
                pos = new Vector3(wordArea.x+left+j*letterSize, wordArea.y, 0);

                // Оператор % помогает выстроить плитки по вертикали
                pos.y -= (i%numRows)*letterSize;

                lett.pos = pos; // Позднее вокруг этой строки будет добавлен
                                //   дополнительный код

                go.transform.localScale = Vector3.one*letterSize;
```

```
            wyrd.Add(lett);
        }

        if (showAllWyrds) wyrd.visible = true;

        wyrds.Add(wyrd);

        // Если достигнут последний ряд в столбце, начать новый столбец
        if ( i%numRows == numRows-1 ) {
            left += ( columnWidth + 0.5f ) * letterSize;
        }
    }
  }
}
```

2. Перед тем как щелкнуть на кнопке Play (Играть), перетащите шаблон PrefabLetter из панели Project (Проект) на поле prefabLetter компонента WordGame (Script) в объекте _MainCamera. После этого щелкните на кнопке Play (Играть), и на экране должен появиться список слов, как показано на рис. 34.4[1].

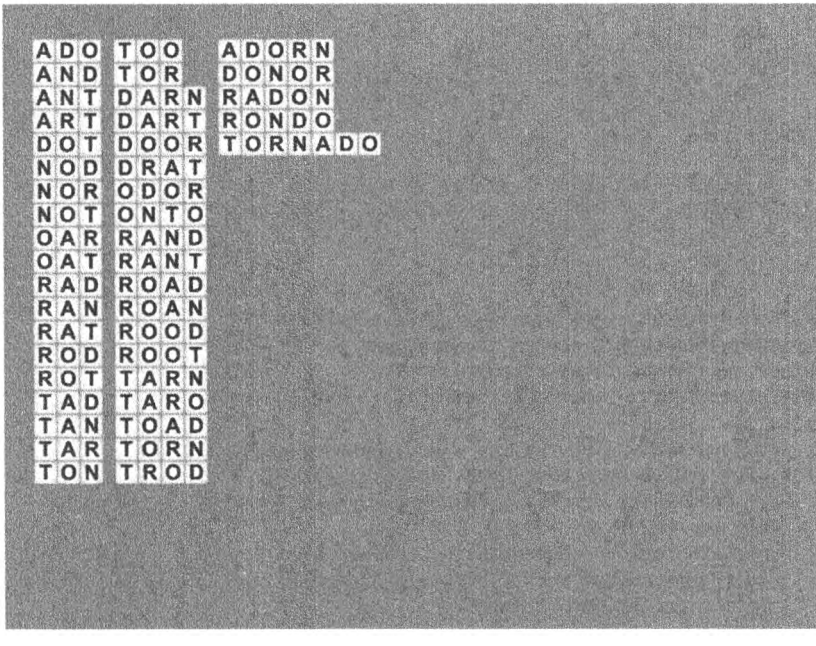


Рис. 34.4. Пример текущего состояния игры для уровня со словом TORNADO

[1] Сохранив эти сценарии и вернувшись в Unity, вы увидите два предупреждения о присутствии в сценариях неиспользуемых переменных (`col` b `bigLetterAnchor`). Пусть вас это не беспокоит, мы скоро их задействуем.

Добавление больших плиток с буквами

Следующим шагом добавим в Layout() размещение больших плиток с буквами вдоль нижнего края экрана.

1. Для этого добавьте следующий код:

```
public class WordGame : MonoBehaviour {
    static public WordGame    S; // Объект-одиночка

    [Header("Set in Inspector")]
    ...
    public bool               showAllWyrds = true;
    public float              bigLetterSize = 4f;
    public Color              bigColorDim = new Color( 0.8f, 0.8f, 0.8f );
    public Color              bigColorSelected = new Color( 1f, 0.9f, 0.7f );
    public Vector3            bigLetterCenter = new Vector3( 0, -16, 0 );

    [Header("Set Dynamically")]
    ...
    public List<Wyrd>         wyrds;
    public List<Letter>       bigLetters;
    public List<Letter>       bigLettersActive;

    ...

    void Layout() {
        ...
        // Создать экземпляр Wyrd для каждого слова в level.subWords
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...
        }

        // Поместить на экран большие плитки с буквами
        // Инициализировать список больших букв
        bigLetters = new List<Letter>();
        bigLettersActive = new List<Letter>();

        // Создать большую плитку для каждой буквы в целевом слове
        for (int i=0; i<currLevel.word.Length; i++) {
            // Напоминает процедуру создания маленьких плиток
            c = currLevel.word[i];
            go = Instantiate<GameObject>( prefabLetter );
            go.transform.SetParent( bigLetterAnchor );
            lett = go.GetComponent<Letter>();
            lett.c = c;
            go.transform.localScale = Vector3.one*bigLetterSize;

            // Первоначально поместить большие плитки ниже края экрана
            pos = new Vector3( 0, -100, 0 );
            lett.pos = pos;     // Позднее вокруг этой строки будет добавлен
                                //   дополнительный код

            col = bigColorDim;
```

```
            lett.color = col;
            lett.visible = true; // Всегда true для больших плиток
            lett.big = true;
            bigLetters.Add(lett);
        }
        // Перемешать плитки
        bigLetters = ShuffleLetters(bigLetters);

        // Вывести на экран
        ArrangeBigLetters();

        // Установить режим mode -- "в игре"
        mode = GameMode.inLevel;
    }

    // Этот метод перемешивает элементы в списке List<Letter> и возвращает
    // результат
    List<Letter> ShuffleLetters(List<Letter> letts) {
        List<Letter> newL = new List<Letter>();
        int ndx;
        while(letts.Count > 0) {
            ndx = Random.Range(0,letts.Count);
            newL.Add(letts[ndx]);
            letts.RemoveAt(ndx);
        }
        return(newL);
    }

    // Этот метод выводит большие плитки на экран
    void ArrangeBigLetters() {
        // Найти середину для вывода ряда больших плиток с центрированием
        // по горизонтали
        float halfWidth = ( (float) bigLetters.Count )/2f - 0.5f;
        Vector3 pos;
        for (int i=0; i<bigLetters.Count; i++) {
            pos = bigLetterCenter;
            pos.x += (i-halfWidth)*bigLetterSize;
            bigLetters[i].pos = pos;
        }
        // bigLettersActive
        halfWidth = ( (float) bigLettersActive.Count )/2f - 0.5f;
        for (int i=0; i<bigLettersActive.Count; i++) {
            pos = bigLetterCenter;
            pos.x += (i-halfWidth)*bigLetterSize;
            pos.y += bigLetterSize*1.25f;
            bigLettersActive[i].pos = pos;
        }
    }
}
```

2. Теперь, кроме маленьких плиток, внизу на экране должен также появиться ряд больших плиток с буквами из целевого слова, следующими в случайном порядке. А теперь добавим поддержку интерактивности.

Добавление интерактивности

В этой игре игрок должен иметь возможность собрать слово из имеющихся букв на больших плитках, нажимая клавиши с алфавитными символами, и подтвердить окончание ввода слова нажатием Return/Enter. Он также должен иметь возможность нажать Backspace/Delete, чтобы удалить букву в конце конструируемого слова, и перемешать оставшиеся невыбранные буквы нажатием клавиши пробела.

Когда игрок нажимает Enter, собранное им слово сравнивается со всеми допустимыми словами в `WordLevel`. Если сконструированное слово присутствует в `WordLevel`, игроку начисляется по одному очку за каждую букву в слове. Кроме того, если сконструированное слово содержит слова меньшего размера, также присутствующие в `WordLevel`, за них начисляются дополнительные очки и сумма умножается на порядковый номер слова. Рассмотрим пример со словом TORNADO: если игрок ввел первое слово TORNADO и нажал Return, он получит 36 очков согласно следующей логике вычислений:

TORNADO	7 × 1 очков	по 1 очку за каждую букву в первом слове = 7 очков
TORN	4 × 2 очков	по 1 очку за каждую букву x 2 за второе слово = 8 очков
TOR	3 × 3 очков	по 1 очку за каждую букву x 3 за третье слово = 9 очков
ADO	+ 3 × 4 очков	по 1 очку за каждую букву x 4 за четвертое слово = 12 очков
	всего 36 очков	

Взаимодействие с игроком реализовано в функции `Update()` класса `WordGame` и основывается на `Input.inputString`, строке с клавишами, нажатыми в этом кадре.

1. Добавьте в `WordGame` метод `Update()` и вспомогательные методы:

```
public class WordGame : MonoBehaviour {
    ...

    [Header("Set Dynamically")]
    ...
    public List<Letter>       bigLettersActive;
    public string             testWord;
    private string            upperCase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    ...

    void ArrangeBigLetters() { ... }

    void Update() {
        // Объявить пару вспомогательных переменных
        Letter ltr;
        char c;

        switch (mode) {
            case GameMode.inLevel:
```

```
// Выполнить обход всех символов, введенных игроком в этом кадре
foreach (char cIt in Input.inputString) {
    // Преобразовать cIt в верхний регистр
    c = System.Char.ToUpperInvariant(cIt);

    // Проверить, есть ли такая буква верхнего регистра
    if (upperCase.Contains(c)) { // Любая буква верхнего регистра
        // Найти доступную плитку с этой буквой в bigLetters
        ltr = FindNextLetterByChar(c);
        // Если плитка найдена
        if (ltr != null) {
            // ... добавить этот символ в testWord и переместить
            //   соответствующую плитку Letter в bigLettersActive
            testWord += c.ToString();
            // Переместить из списка неактивных в список активных
            // плиток
            bigLettersActive.Add(ltr);
            bigLetters.Remove(ltr);
            ltr.color = bigColorSelected; // Придать плитке
                                          // активный вид
            ArrangeBigLetters();          // Отобразить плитки
        }
    }

    if (c == '\b') { // Backspace
        // Удалить последнюю плитку Letter из bigLettersActive
        if (bigLettersActive.Count == 0) return;
        if (testWord.Length > 1) {
            // Удалить последнюю букву из testWord
            testWord = testWord.Substring(0,testWord.Length-1);
        } else {
            testWord = "";
        }

        ltr = bigLettersActive[bigLettersActive.Count-1];
        // Переместить из списка активных в список неактивных
        // плиток
        bigLettersActive.Remove(ltr);
        bigLetters.Add (ltr);
        ltr.color = bigColorDim; // Придать плитке неактивный вид
        ArrangeBigLetters();     // Отобразить плитки
    }

    if (c == '\n' || c == '\r') { // Return/Enter macOS/Windows
        // Проверить наличие сконструированного слова в WordLevel
        CheckWord();
    }

    if (c == ' ') { // Пробел
        // Перемешать плитки в bigLetters
        bigLetters = ShuffleLetters(bigLetters);
        ArrangeBigLetters();
    }
}
break;
```

```
    }
}

// Этот метод отыскивает плитку Letter с символом c в bigLetters.
// Если такой плитки нет, возвращает null.
Letter FindNextLetterByChar(char c) {
    // Проверить каждую плитку Letter в bigLetters
    foreach (Letter ltr in bigLetters) {
        // Если содержит тот же символ, что указан в c
        if (ltr.c == c) {
            // ...вернуть ее
            return(ltr);
        }
    }
    return( null ); // Иначе вернуть null
}

public void CheckWord() {
    // Проверяет присутствие слова testWord в списке level.subWords
    string subWord;
    bool foundTestWord = false;

    // Создать список List<int> для хранения индексов других слов,
    //   присутствующих в testWord
    List<int> containedWords = new List<int>();

    // Обойти все слова в currLevel.subWords
    for (int i=0; i<currLevel.subWords.Count; i++) {
        // Проверить, было ли уже найдено Wyrd
        if (wyrds[i].found) {                                          // a
            continue;
        }

        subWord = currLevel.subWords[i];
        // Проверить, входит ли это слово subWord в testWord
        if (string.Equals(testWord, subWord)) {                        // b
            HighlightWyrd(i);
            foundTestWord = true;
        } else if (testWord.Contains(subWord)) {
            containedWords.Add(i);
        }
    }

    if (foundTestWord) { // Если проверяемое слово присутствует в списке
        // ...подсветить другие слова, содержащиеся в testWord
        int numContained = containedWords.Count;
        int ndx;
        // Подсвечивать слова в обратном порядке
        for (int i=0; i<containedWords.Count; i++) {
            ndx = numContained-i-1;
            HighlightWyrd( containedWords[ndx] );
        }
    }

    // Очистить список активных плиток Letters независимо от того,
    //   является ли testWord допустимым
    ClearBigLettersActive();
```

```
    }

    // Подсвечивает экземпляр Wyrd
    void HighlightWyrd(int ndx) {
        // Активировать слово
        wyrds[ndx].found = true; // Установить признак, что оно найдено
        // Выделить цветом
        wyrds[ndx].color = (wyrds[ndx].color+Color.white)/2f;
        wyrds[ndx].visible = true; // Сделать компонент 3D Text видимым
    }

    // Удаляет все плитки Letters из bigLettersActive
    void ClearBigLettersActive() {
        testWord = ""; // Очистить testWord
        foreach (Letter ltr in bigLettersActive) {
            bigLetters.Add(ltr);     // Добавить каждую плитку в bigLetters
            ltr.color = bigColorDim; // Придать ей неактивный вид
        }
        bigLettersActive.Clear(); // Очистить список
        ArrangeBigLetters();      // Повторно вывести плитки на экран
    }
}
```

a. Если *i*-й экземпляр `Wyrd` на экране уже был найден, пропустить его и начать следующую итерацию для проверки очередного экземпляра. Такое возможно благодаря тому, что экземпляры `Wyrd` на экране и слова в списке `subWords` хранятся в одном и том же порядке.

b. Проверяет, является ли слово `subWord` сконструированным словом `testWord`. Если является, оно подсвечивается на экране. Если это не сконструированное слово `testWord`, тогда проверяется, содержится ли данное слово `subWord` в `testWord` (например, AND содержится в SAND), и если содержится, добавляется в список `containedWords`.

2. Сохраните сценарий WordGame и вернитесь в Unity.

3. В инспекторе сбросьте флажок showAllWyrds в компоненте WordGame (Script) объекта _MainCamera и щелкните на кнопке Play (Играть).

На экране должна появиться действующая версия игры со случайно выбранным уровнем. Можете поиграть в игру, используя клавиатуру, как описано выше.

Отображение очков

В этот проект уже импортированы классы `Scoreboard` и `FloatingScore`, написанные в предыдущих главах, поэтому отображение очков, заработанных игроком, реализуется очень просто.

1. Создайте объект Canvas для отображения текстовых полей пользовательского интерфейса, выбрав в меню пункт GameObject > UI > Canvas (Игровой объект > ПИ > Холст).

2. Перетащите Scoreboard из папки _Prefab в панели Project (Проект) на объект Canvas в панели Hierarchy (Иерархия), чтобы создать объект Scoreboard, вложенный в Canvas.
3. Убедитесь, что поле prefabFloatingScore компонента Scoreboard (Script) в объекте Scoreboard ссылается на шаблон PrefabFloatingScore из папки _Prefabs. (Чтобы узнать больше о том, как работает Scoreboard, обращайтесь к главе 32 «Прототип 4: Prospector Solitaire».)
4. Создайте в папке __Scripts новый сценарий на C# с именем ScoreManager и подключите его к Scoreboard.
5. Откройте сценарий ScoreManager в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ScoreManager : MonoBehaviour {
    static private ScoreManager S; // Еще один скрытый объект-одиночка

    [Header("Set in Inspector")]
    public List<float>      scoreFontSizes = new List<float> { 36, 64, 64, 1 };
    public Vector3          scoreMidPoint = new Vector3( 1, 1, 0 );
    public float            scoreTravelTime = 3f;
    public float            scoreComboDelay = 0.5f;

    private RectTransform   rectTrans;

    void Awake() {
        S = this;
        rectTrans = GetComponent<RectTransform>();
    }

    // Этот метод можно вызвать как ScoreManager.SCORE() из любого места
    static public void SCORE(Wyrd wyrd, int combo) {
        S.Score(wyrd, combo);
    }

    // Добавить очки за это слово
    // int combo - номер этого слова в комбинации
    void Score(Wyrd wyrd, int combo) {
        // Создать список List<Vector2> с точками, определяющими кривую Безье
        //   для FloatingScore.
        List<Vector2> pts = new List<Vector2>();

        // Получить позицию плитки с первой буквой в wyrd
        Vector3 pt = wyrd.letters[0].transform.position;                    // a
        pt = Camera.main.WorldToViewportPoint(pt);

        pts.Add(pt); // Сделать pt первой точкой кривой Безье               // b

        // Добавить вторую точку кривой Безье
```

```
        pts.Add( scoreMidPoint );

        // Сделать Scoreboard последней точкой кривой Безье
        pts.Add(rectTrans.anchorMax);

        // Определить значение для FloatingScore
        int value = wyrd.letters.Count * combo;
        FloatingScore fs = Scoreboard.S.CreateFloatingScore(value, pts);

        fs.timeDuration = scoreTravelTime;
        fs.timeStart = Time.time + combo * scoreComboDelay;
        fs.fontSizes = scoreFontSizes;

        // Удвоить эффект InOut из Easing
        fs.easingCurve = Easing.InOut+Easing.InOut;

        // Вывести в FloatingScore текст вида "3 x 2"
        string txt = wyrd.letters.Count.ToString();
        if (combo > 1) {
            txt += " x "+combo;
        }
        fs.GetComponent<Text>().text = txt;
    }

}
```

a. В первый момент объект FloatingScore должен отображаться непосредственно поверх экземпляра `wyrd`. Сначала получаем мировые трехмерные координаты 0-й буквы в `wyrd`. В следующей строке используем _MainCamera для преобразования их из трехмерных мировых координат в координаты видимой области `ViewportPoint`. Координаты X и Y видимой области `ViewportPoint` изменяются в диапазоне от 0 до 1, определяют расстояние до точки от верхнего левого угла относительно ширины и высоты и используются для определения координат элементов пользовательского интерфейса.

b. Когда к вектору `Vector3 pt` добавляется `List<Vector2> pts`, координата Z отбрасывается.

6. Сохраните сценарий ScoreManager.

7. Откройте сценарий WordGame и добавьте в `CheckWord()` следующий код, который реализует подсчет очков (выделен жирным):

```
public class WordGame : MonoBehaviour {
    ...
    public void CheckWord() {
        ...
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...
            // Проверить, входит ли это слово subWord в testWord
            if (string.Equals(testWord, subWord)) {
                HighlightWyrd(i);
```

```
                ScoreManager.SCORE( wyrds[i], 1 ); // Подсчитать очки      // a
                foundTestWord = true;
            } else if (testWord.Contains(subWord)) {
                ...
            }
        }

        if (foundTestWord) { // Если проверяемое слово присутствует в списке
            ...
            for (int i=0; i<containedWords.Count; i++) {
                ndx = numContained-i-1;
                HighlightWyrd( containedWords[ndx] );
                ScoreManager.SCORE( wyrds[ containedWords[ndx] ], i+2 );   // b
            }
        }
        ...
    }

    ...
}
```

a. Эта строка вызывает статический метод `ScoreManager.SCORE()` для отображения поля с очками за слово `testWord`, собранное игроком.

b. Здесь метод `ScoreManager.SCORE()` вызывается для отображения поля с очками за каждое слово, входящее в `testWord`. Второй параметр (`i+2`) — это порядковый номер слова.

8. Сохраните сценарий **WordGame**, вернитесь в Unity и щелкните на кнопке **Play** (Играть).

Теперь на экране должно появиться число очков за правильно сконструированное слово, а также числа с очками за каждое дополнительное слово, содержащееся в сконструированном слове. Однако белые числа плохо видны на фоне белых плиток Letter. Мы исправим этот недостаток позднее, когда начнем расцвечивать игру.

Добавление анимации в плитки

По аналогии с отображением очков мы легко можем добавить анимационный эффект перемещения плиток, воспользовавшись функциями интерполяции, находящимися в импортированном сценарии **Utils**.

1. Добавьте следующий код в сценарий Letter:

```
public class Letter : MonoBehaviour {
    [Header("Set in Inspector")]
    public float            timeDuration = 0.5f;
    public string           easingCuve = Easing.InOut; // Функция сглаживания
                                                       // из Utils.cs
```

```
[Header("Set Dynamically")]
public TextMesh         tMesh; // TextMesh отображает символ
public Renderer         tRend; // Компонент Renderer объекта 3D Text. Он
                               //   будет определять видимость символа
public bool             big = false; // Большие и малые плитки действуют
                                     // по-разному
// Поля для линейной интерполяции
public List<Vector3>    pts = null;
public float            timeStart = -1;

private char            _c;    // Символ, отображаемый на этой плитке
...

// Свойство для чтения/записи координат плитки
// Теперь настраивает кривую Безье для плавного перемещения в новые координаты
public Vector3 pos {
    set {
        // transform.position = value; // Закомментируйте эту строку

        // Найти среднюю точку на случайном расстоянии от фактической
        // средней точки между текущей и новой (value) позициями

        Vector3 mid = (transform.position + value)/2f;

        // Случайное расстояние не превышает 1/4 расстояния
        //   до фактической средней точки
        float mag = (transform.position - value).magnitude;
        mid += Random.insideUnitSphere * mag*0.25f;

        // Создать List<Vector3> точек, определяющих кривую Безье
        pts = new List<Vector3>() { transform.position, mid, value };

        // Если timeStart содержит значение по умолчанию -1,
        //   установить текущее время
        if (timeStart == -1 ) timeStart = Time.time;
    }
}

// Немедленно перемещает в новую позицию
public Vector3 posImmediate {                                          // a
    set {
        transform.position = value;
    }
}

// Код, реализующий анимационный эффект
void Update() {
    if (timeStart == -1) return;

    // Стандартная линейная интерполяция
    float u = (Time.time-timeStart)/timeDuration;
    u = Mathf.Clamp01(u);
    float u1 = Easing.Ease(u,easingCuve);
    Vector3 v = Utils.Bezier(u1, pts);
```

```
        transform.position = v;

        // Если интерполяция закончена, записать -1 в timeStart
        if (u == 1) timeStart = -1;
    }
}
```

a. Так как запись в свойство pos теперь запускает анимационный эффект перемещения в новое местоположение, было добавлено свойство posImmediate, позволяющее переместить плитку Letter в другое место немедленно, без анимационного эффекта.

2. Сохраните сценарий Letter, вернитесь в Unity и щелкните на кнопке Play (Играть).

   Теперь все плитки с буквами должны перемещаться в новые координаты с воспроизведением анимационного эффекта. Однако довольно странно видеть, как все плитки начинают перемещаться одновременно и из центра экрана.

3. Изменим немного метод WordGame.Layout(), чтобы устранить эту странность:

```
public class WordGame : MonoBehaviour {
    ...

    void Layout() {
        ...
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...
            // Создать экземпляр PrefabLetter для каждой буквы в слове
            for (int j=0; j<word.Length; j++) {
                ...
                // Оператор % помогает выстроить плитки по вертикали
                pos.y -= (i%numRows)*letterSize;

                // Переместить плитку lett немедленно за верхний край экрана
                lett.posImmediate = pos+Vector3.up*(20+i%numRows);
                // Затем начать ее перемещение в новую позицию pos
                lett.pos = pos; // Позднее вокруг этой строки будет добавлен
                                //   дополнительный код
                // Увеличить lett.timeStart для перемещения слов в разные времена
                lett.timeStart = Time.time + i*0.05f;

                go.transform.localScale = Vector3.one*letterSize;
                wyrd.Add(lett);
            }
            ...
        }
        ...
        // Создать большую плитку для каждой буквы в целевом слове
        for (int i=0; i<currLevel.word.Length; i++) {
            ...
            // Первоначально поместить большие плитки ниже края экрана
            pos = new Vector3( 0, -100, 0 );

            lett.posImmediate = pos;
```

```
            lett.pos = pos; // Позднее вокруг этой строки будет добавлен
                            //   дополнительный код
            // Увеличить lett.timeStart, чтобы большие плитки
            //   с буквами появились последними
            lett.timeStart = Time.time + currLevel.subWords.Count*0.05f;
            lett.easingCuve = Easing.Sin+"-0.18"; // Bouncy easing

            col = bigColorDim;
            ...
        }
        ...
    }
    ...
}
```

4. Сохраните сценарий WordGame, вернитесь в Unity и щелкните на кнопке Play (Играть).

Теперь начальная раскладка плиток в игре должна происходить плавно, с воспроизведением привлекательного анимационного эффекта.

Добавление цвета

Теперь, реализовав движение плиток, добавим в игру немного цвета:

1. Добавьте в WordGame следующий код, выделенный жирным, для расцвечивания плиток в словах, исходя из их длины:

```
public class WordGame : MonoBehaviour {
    static public WordGame    S; // Объект-одиночка

    [Header("Set in Inspector")]
    ...
    public Vector3            bigLetterCenter = new Vector3( 0, -16, 0 );
    public Color[]            wyrdPalette;

    [Header("Set Dynamically")]
    ...

    void Layout() {
        ...
        // Создать экземпляр Wyrd для каждого слова в level.subWords
        for (int i=0; i<currLevel.subWords.Count; i++) {
            ...
            // Создать экземпляр PrefabLetter для каждой буквы в слове
            for (int j=0; j<word.Length; j++) {
                ...
            }

            if (showAllWyrds) wyrd.visible = true;

            // Определить цвет слова, исходя из его длины
```

```
            wyrd.color = wyrdPalette[ word.Length - WordList.WORD_LENGTH_MIN ];
            wyrds.Add(wyrd);
            ...
        }
        ...
    }

    ...
}
```

Эти последние несколько изменений оказались настолько простыми, потому что у нас уже имеется необходимый код поддержки (например, свойства `Wyrd.color` и `Letter.color`, а также класс `Easing` с функциями сглаживания в сценарии *Utils.cs*).

Теперь нужно определить восемь цветов для `wyrdPalette`. Используем для этого изображение Color Palette, включенное в пакет, импортированный в самом начале работы над проектом. Выбор цвета предполагается производить с помощью инструмента «пипетка», для чего необходимо одновременно видеть изображение Color Palette и панель Inspector (Инспектор) с настройками _MainCamera. Чтобы вывести на экран и то и другое, воспользуемся возможностью Unity одновременно отображать две панели Inspector (Инспектор).

2. Щелкните на кнопке настройки панели (в красном кружке), как показано на рис. 34.5, и в открывшемся меню выберите пункт Add Tab > Inspector (Добавить вкладку > Инспектор), чтобы добавить панель Inspector (Инспектор) во вкладку Game (Игра).

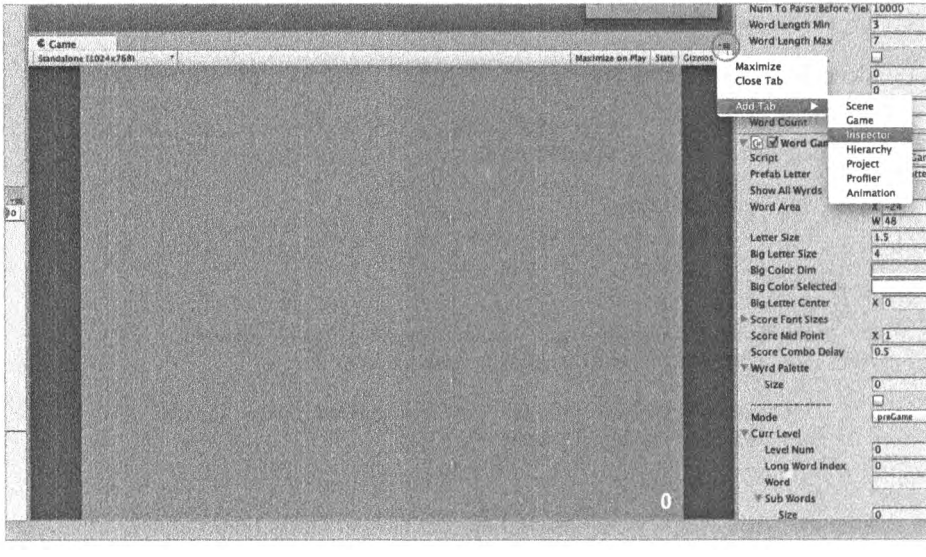


Рис. 34.5. Использование кнопки настройки панели для добавления инспектора в панель Game (Игра)

3. Выберите изображение Color Palette в папке Materials & Textures панели Project. Теперь на экране появятся две панели Inspector (Инспектор). (Возможно, вам понадобится растянуть область предварительного просмотра изображения в инспекторе, чтобы получить вид, как показано на рис. 34.6.)
4. Щелкните на значке с изображением замка вверху на этой панели инспектора (в красном кружке на рис. 34.6), чтобы заблокировать ее.
5. Выберите _MainCamera в панели Hierarchy (Иерархия). Настройки _MainCamera отобразятся только в одной, незаблокированной панели инспектора, а заблокированная продолжит показывать изображение Color Palette.

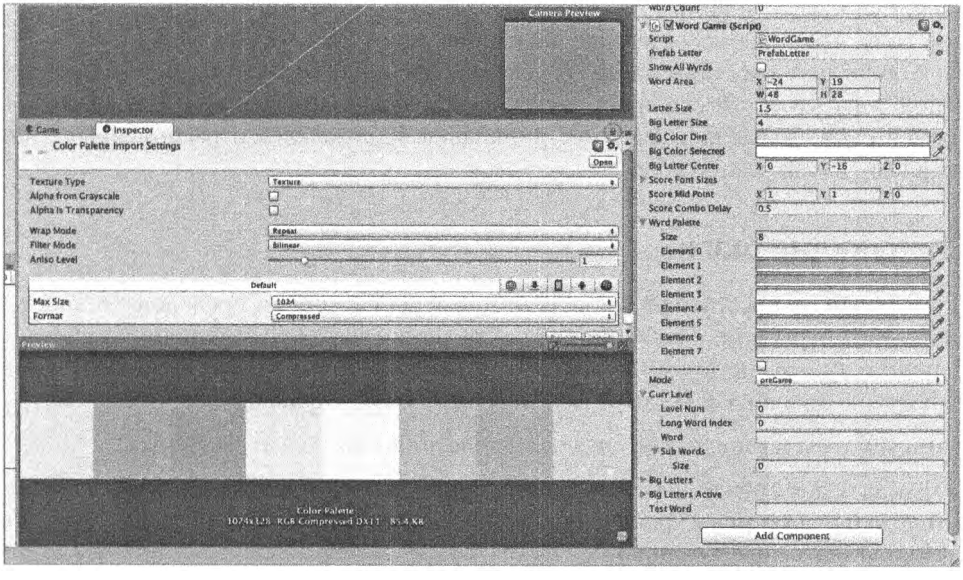


Рис. 34.6. Значок с изображением замка в одной панели инспектора (в красном кружке) и значок с изображением пипетки — в другой (в синем кружке)

6. В инспекторе с настройками _MainCamera раскройте массив wyrdPalette, щелкнув на пиктограмме с треугольником рядом с его именем, и введите в поле Size число 8.
7. Поочередно для каждого элемента wyrdPalette щелкните на значке с пипеткой (в синем кружке на рис. 34.6) и затем на одном из цветов в изображении Color Palette. Таким способом вы перенесете в wyrdPalette все восемь цветов из Color Palette, но все они по умолчанию получат значение 0 для альфа-канала (и поэтому будут невидимы). Признаком значения 0 в альфа-канале служит черная полоска, тянущаяся вдоль нижнего края каждого из элементов в wyrdPalette.
8. Для каждого элемента wyrdPalette щелкните на цветной полоске и установите значение альфа-канала (или A) равным 255, чтобы сделать цвет полностью не-

прозрачным. После этого полоска вдоль нижнего края каждого из элементов в wyrdPalette изменит цвет с черного на белый.

9. Как обычно, сохраните сцену.

Если теперь запустить сцену, она должна выглядеть примерно так, как показано на скриншоте в начале главы.

Итоги

В этой главе мы создали простую игру в слова и украсили ее несложной анимацией. Если вы неуклонно следовали за всеми учебными примерами, то наверняка заметили, что процесс создания прототипов становится все более простым. Благодаря более полному пониманию Unity, которое у вас есть сейчас, и возможности использовать готовые сценарии, такие как Scoreboard, FloatingScore и Utils, вы больше времени можете уделить аспектам, уникальным для каждой игры, и меньше — повторному изобретению колеса.

Следующие шаги

В предыдущих прототипах вы видели примеры настройки последовательностей состояний игры для обработки разных этапов и перехода с уровня на уровень. В данном прототипе, прямо сейчас, нет ничего подобного. Теперь вы должны самостоятельно добавить в эту игру похожую структуру управления.

Вот о чем следует подумать, когда вы начнете заниматься этим:

- Когда игрок сможет перейти на следующий уровень? Должен ли он угадать все слова или для смены уровня достаточно набрать определенное количество очков или угадать целевое слово?
- Как вы будете генерировать уровни? Будет ли всегда выбираться случайное слово, как сейчас, или вы измените процедуру случайного выбора так, чтобы уровень 5 всегда давал бы одно и то же слово (чтобы игроки могли сравнивать заработанные очки на уровне 5)? Вот как можно организовать случайный выбор, если вы решите изменить его:

```
using UnityEngine;
using System.Collections;

public class LevelPicker : MonoBehaviour {
    static private System.Random     rng;

    [Header("Set in Inspector")]
    public int      randomSeed = 12345;

    void Awake() {
        rng = new System.Random(randomSeed);
```

```
    }

    static public int Next(int max=-1) {
        // Вернуть следующее число между 0 и max-1 из rng.
        // Если получено число -1, параметр max игнорируется
        if (max == -1) {
            return rng.Next();
        } else {
            return rng.Next( max );
        }
    }
}
```

- Как лучше обрабатывать уровни с очень большим или очень маленьким количеством слов в списке `subWords`? Для некоторых коллекций из семи букв можно составить так много слов, что для них не хватит места на экране, тогда как для других будет достаточно одной колонки. Может быть, игра должна пропускать подобные случаи? Если да, как бы вы реализовали функцию `PickNthRandom` для пропуска таких вариантов?

Теперь у вас есть достаточный объем знаний о программировании и прототипировании, чтобы ответить на эти вопросы и воплотить ответы на них в настоящую игру. Вы многое уже умеете, поэтому смело идите вперед!

35 Прототип 7: DUNGEON DELVER

Игра *Dungeon Delver*, которую вы создадите в этой главе, является упрощенным вариантом оригинальной игры *Legend of Zelda* для игровой приставки Nintendo Entertainment System. В своей преподавательской деятельности я не раз убеждался, что воссоздание старых игр очень помогает дизайнерам в их учебе, и *The Legend Of Zelda* всегда была одним из лучших образцов для повторения.

Это последний прототип в книге, и поэтому он самый сложный. Кроме того, в этом прототипе используется более последовательный компонентно-ориентированный подход, чем в предыдущих, поэтому отдельные сценарии станут еще короче. Закончив эту главу, вы получите превосходный каркас приключенческой игры, который сможете продолжить развивать сами.

Dungeon Delver — обзор игры

Эта глава значительно длиннее большинства других, но в ней также создается самая большая игра. Это единственный новый прототип, появившийся во втором издании книги, — приключенческая игра, основанная на *The Legend of Zelda* для игровой приставки Nintendo Entertainment System. События в игре развиваются вокруг главного персонажа по имени Дрей (Dray) по мере того, как он исследует подземелье, сражается со скелетами и ищет крюк (кошку).

На рис. 35.1 показано, как будет выглядеть игра *Dungeon Delver* в конце этой главы. *Dungeon Delver* начинается с копии первого подземелья, встречающегося в *The Legend of Zelda* для NES. Далее в главе мы перейдем ко второму подземелью (обсуждавшемуся в главе 9 «Прототипирование на бумаге»). Закончив читать эту главу, загляните в раздел Chapter 35 на веб-сайте этой книги, где вы найдете редактор уровней для этой игры и инструкции по созданию своих подземелий.

Компонентно-ориентированная архитектура

Работая над этим прототипом, мы будем стараться максимально придерживаться *компонентно-ориентированной архитектуры* (см. главу 27 «Объектно-ориентированное мышление»). Для этого мы должны заранее решить, как должна работать

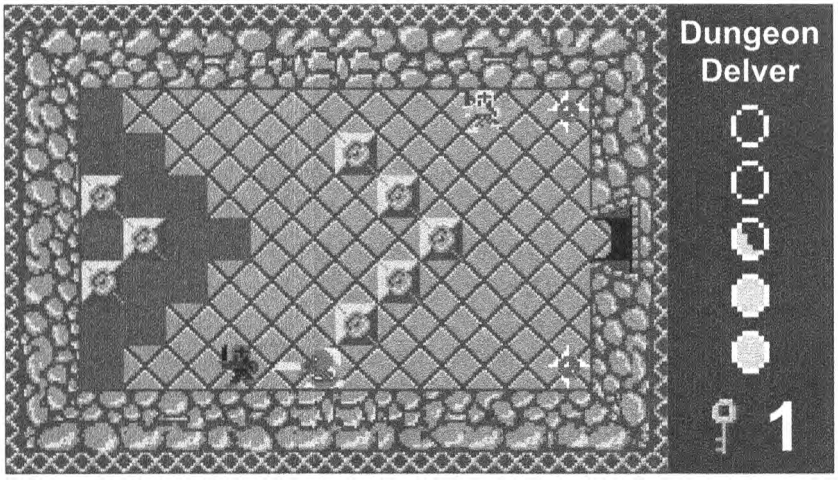


Рис. 35.1. Так будет выглядеть игра Dungeon Delver

игра. *The Legend of Zelda*, как и большинство других игр того времени, была основана на мозаичной графике, в том смысле, что карты в них конструировались из ограниченного набора плиток, которые могли многократно повторяться на протяжении всего этапа. На рис. 35.2 можно видеть набор доступных плиток (слева) и часть карты, составленной из них (справа).

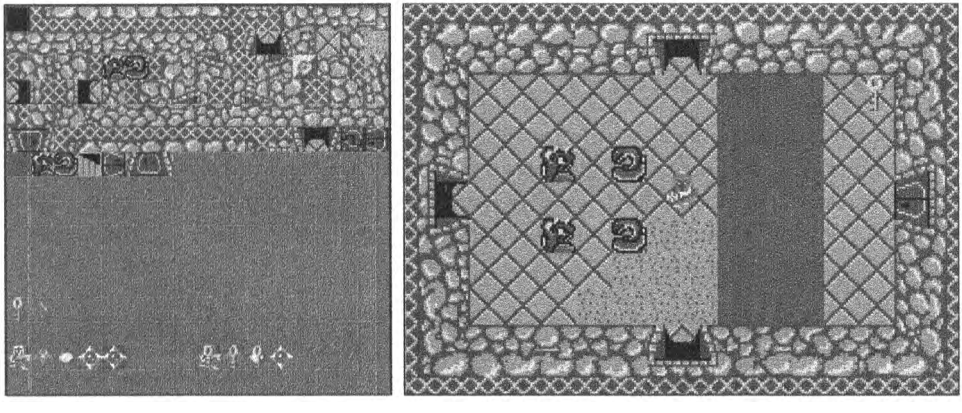


Рис. 35.2. Атлас плиток (слева) и комната, сконструированная из них (справа). Сетка, нанесенная на изображение слева, помогает более отчетливо увидеть границы плиток

В Unity нет своего механизма управления плитками, поэтому нам придется его создать. Это означает, что нам понадобится шаблон для плиток и сценарий для камеры, который будет их упорядочивать. Решив пойти более компонентно-ориентированным путем, мы должны написать сценарий для каждой плитки (*Tile.cs*), выполня-

ющий как можно больше работы, чтобы сценарий для камеры (*TileCamera.cs*) мог просто задавать местоположение каждой плитки. Вот как планируется разделить обязанности между сценариями:

1. **TileCamera.cs**.
   - **Позиционирование себя** — первоначально камера отображает вход в подземелье и следует за игроком (Дреем) в другие комнаты.
   - **Чтение данных карты** — TileCamera читает карту из файла DelverData с картой, информацию о столкновениях из DelverCollisions и спрайты с изображениями плиток из DelverTiles.
   - **Позиционирование плиток** — TileCamera создает экземпляры игровых объектов Tile и назначает им позиции на карте.
2. **Tile.cs** — для нормальной работы плитке требуется, чтобы камера TileCamera назначила ей местоположение.
   - **Позиционирование себя** — плитка Tile должна поместить себя в назначенное место.
   - **Отображение соответствующего спрайта** — плитка Tile должна выполнить поиск в файле DelverData, которым управляет TileCamera, и найти спрайт для отображения.
   - **Управление коллайдером BoxCollider** — плитка Tile должна выполнить поиск в файле DelverCollisions, которым управляет TileCamera, и настроить свой коллайдер BoxCollider, как указано в файле.

Начало: прототип 7

Импортируемый пакет для этого проекта включает множество ресурсов, материалов и сценариев. Так как вы уже имеете опыт конструирования объектов и нарезки спрайтов в Unity, в этой главе я не буду просить вас делать все это. Вместо этого вы должны импортировать группу шаблонов, которые в этой игре будут играть роль художественных ресурсов.

НАСТРОЙКА ПРОЕКТА ДЛЯ ЭТОЙ ГЛАВЫ

Следуя стандартной процедуре настройки проекта, создайте новый проект в Unity. Чтобы вспомнить, как это делается, обращайтесь к приложению А «Стандартная процедура настройки проекта». При создании проекта вам будет предложено выбрать настройки по умолчанию для дву- или трехмерной игры. Выберите настройки для двумерной версии.

- **Имя проекта:** Dungeon Delver
- **Загрузите и импортируйте пакет:** находится в разделе Chapter 35 на сайте http://book.prototools.net.

- **Имя сцены:** _Scene_Eagle (эта сцена будет отображать подземелье, идентичное подземелью *Eagle* из *The Legend of Zelda*).
- **Папки проекта:** все необходимые папки импортируются из пакета.
- **Имена сценариев на C#:** никаких других сценариев, кроме импортированных.

Папка __Scripts в импортируемом пакете включает сценарий *Spiker.cs*, в котором большая часть кода закомментирована. Завершив эту главу, вы сможете раскомментировать этот код и подключить сценарий Spiker к шаблону Spiker.

Все изображения персонажей, плиток, вещей и т. д. в этой главе были созданы моим потрясающим коллегой и другом Эндрю Деннисом (Andrew Dennis), который преподает компьютерную графику и анимацию в Университете штата Мичиган[1].

Настройка камер

Это первый проект, где мы будем использовать несколько камер. Первая, главная камера Main Camera, будет отображать основной игровой процесс, а вторая, камера пользовательского интерфейса GUI Camera, будет отображать графический пользовательский интерфейс (ГПИ) игры. Это позволит управлять каждой камерой независимо, что упростит реализацию ГПИ.

Панель Game (Игра)

Прежде чем заняться настройками камер, нужно подготовить панель Game (Игра).

В раскрывающемся списке соотношения сторон, в верхней части панели Game (Игра), выберите пункт 1080p (1920x1080). Если у вас этот пункт отсутствует в списке, выполните следующие шаги:

1. Щелкните на раскрывающемся списке выбора соотношения сторон в верхней части панели Game (Игра) — второй раскрывающийся список слева — и щелкните на кнопке + внизу этого списка.
2. В открывшемся диалоге:
   - В поле Label введите значение 1080p.
   - В списке Type выберите Fixed Resolution.
   - В поле W введите 1920.
   - В поле H введите 1080.

[1] Страница факультета, где преподает Эндрю: http://gamedev.msu.edu/andrew-dennis/.

3. Щелкните на кнопке OK, чтобы сохранить настройки.
4. Выберите пункт 1080p (1920x1080) в раскрывающемся списке соотношения сторон.

Главная камера

Выберите Main Camera в иерархии и в инспекторе:

- Настройте компонент Transform: P:[23.5, 5, -10] R:[0, 0, 0] S:[1, 1, 1].
- В компоненте Camera:
  - В поле Clear Flags выберите Solid Color.
  - В поле Background установите черный цвет (RGBA:[0, 0, 0, 255]).
  - В поле Projection выберите Orthographic.
  - В поле Size введите 5.5.
  - В поле Viewport Rect введите значения X:0, Y:0, W:0.8, H:1.

Камера пользовательского интерфейса

Выполните следующие шаги, чтобы создать камеру с именем GUI Camera.

1. Создайте новую камеру, выбрав в меню пункт GameObject > Camera (Игровой объект > Камера).
2. Переименуйте созданный объект Camera в GUI Camera.
3. Выберите GUI Camera в иерархии и в инспекторе:
   - В поле Tag выберите Untagged — это гарантирует, что `Camera.main` будет продолжать ссылаться на Main Camera.
   - Настройте компонент Transform: P:[−100, 0, −10] R:[0, 0, 0] S:[1, 1, 1].
   - В компоненте Camera:
     - В поле Clear Flags выберите Solid Color.
     - В поле Background пока установите серый цвет (RGBA:[128, 128, 128, 255]).
     - В поле Projection выберите Orthographic.
     - В поле Size введите 5.5.
     - В поле Viewport Rect введите значения X:0.8, Y:0, W:0.2, H:1.
- Audio Listener — в сцене может присутствовать только один компонент Audio Listener, и в этой сцене он находится в Main Camera.
  - Щелкните на кнопке с шестеренкой справа от компонента Audio Listener и в открывшемся меню выберите пункт Remove Component (Удалить компонент).

В результате этих настроек панель Game (Игра) будет поделена на две меньших панели. Main Camera заполнит большую часть экрана, а GUI Camera — примерно 20%.

Данные для игры Dungeon Delver

В папке Resources, в панели Project (Проект) присутствуют три файла, которые хранят всю информацию, необходимую для отображения подземелья в этой игре (расширения файлов, такие как *.png*, не отображаются в панели Project (Проект)):

- **DelverTiles.png** — файл изображения с текстурой Texture2D, в которой хранятся все изображения, которые можно использовать для отображения подземелья.
- **DelverCollisions.txt** — текстовый файл, хранящий информацию о поддержке столкновений для каждого спрайта плиток в *DelverTiles.png*. Далее я объясню, что означают разные буквы в этом файле.
- **DelverData.txt** — текстовый файл, хранящий информацию о границах каждого спрайта в файле *DelverTiles.png*.

Создание спрайтов из файла DelverTiles

Для подготовки атласа спрайтов DelverTiles выполните следующие шаги:

1. Выберите файл DelverTiles в папке Resources в панели Project (Проект).
2. В разделе Import Settings (Настройки импортирования), в панели Inspector (Инспектор) выполните настройки, как показано на рис. 35.3:
   - В поле Texture Type выберите Sprite (2D and UI).
   - В поле Sprite Mode выберите Multiple.
   - В поле Pixels Per Unit введите 16. Это означает, что в сцене спрайт шириной 16 пикселов будет иметь ширину 1 м. Так как все плитки имеют размеры 16 × 16, на карте каждая плитка будет занимать 1 квадратный метр (то есть 1 квадратную единицу Unity).
   - Снимите флажок Generate Mip Maps. Термин *Mip Maps* обозначает метод ускорения отображения далеких предметов за счет сохранения нескольких версий изображения с разными разрешениями. Они не понадобятся нам в этом проекте.
   - В поле Wrap Mode выберите Clamp. Если координаты полигона в текстуре меньше 0 или больше 1, в режиме Clamp повторяться будут краевые пикселы изображения (а не все изображение целиком).
   - В поле Filter Mode выберите Point (no filter). Это поможет вам сохранить вид игры, характерный для 8-битных приставок.

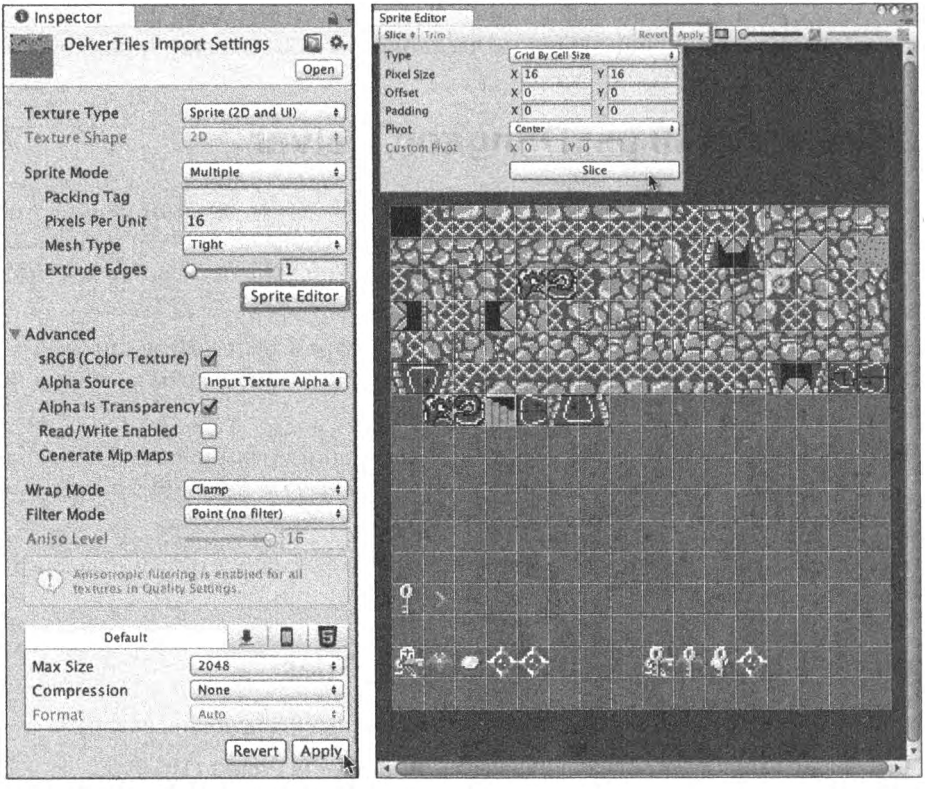


Рис. 35.3. Настройки импортирования и редактор спрайтов для DelverTiles с требуемыми настройками

- В поле **Compression** (в разделе **Default**, в нижней части панели) выберите **None**. Сжатие часто бывает необходимо при работе с большими изображениями (и кстати, файлы *png*, которые вы импортировали, прекрасно справляются со сжатием без потери качества), но в Unity сжатие приводит к существенной потере качества изображений, особенно таких маленьких 8-битных, как эти.
- Все другие вкладки, находящиеся ниже в этом блоке, содержат параметр **Compression** для настройки сжатия на конкретных платформах, включая **PC, Mac & Linux Standalone**, **WebGL** и все другие платформы, поддержку которых вы установили (у кого-то из вас также может присутствовать вариант для **iOS**). Проверьте их все и везде снимите флажок **Override**.

3. Щелкните на кнопке **Apply** (Применить) справа внизу, под указателем мыши на рис. 35.3.
4. Щелкните на кнопке **Sprite Editor** (Редактор спрайтов), выделенной красной рамкой на рис. 35.3. После этого откроется редактор спрайтов **Sprite Editor**, изображенный на рис. 35.3 справа.

5. Щелкните на кнопке Slice (Нарезка) в левом верхнем углу редактора спрайтов и введите следующие настройки:
   - В поле Type выберите Grid by Cell Size.
   - В поле Pixel Size введите: X:16 Y:16.
6. Щелкните на кнопке Slice (Нарезать) в правом нижнем углу диалога Slice (Нарезка). В результате из единственного изображения будет создано множество спрайтов размерами 16 × 16 пикселов. Они получат порядковые номера от 0 (в верхнем левом углу) до 15 (в правом верхнем углу), и затем счет продолжится во втором горизонтальном ряду.
7. Щелкните на кнопке Apply (Применить) вверху, в окне редактора спрайтов, и затем закройте это окно. Всего будет создано 256 спрайтов — от DelverTiles_0 до DelverTiles_255, которые теперь можно увидеть в панели Project (Проект), раскрыв раздел Resources/DelverTiles щелчком на пиктограмме с треугольником.

Текстовый файл DelverData

Файл DelverData хранит информацию в шестнадцатеричном виде о каждой плитке, составляющей подземелье.

Щелкните дважды на файле Resources/DelverData в панели Project (Проект), чтобы открыть его в MonoDevelop.

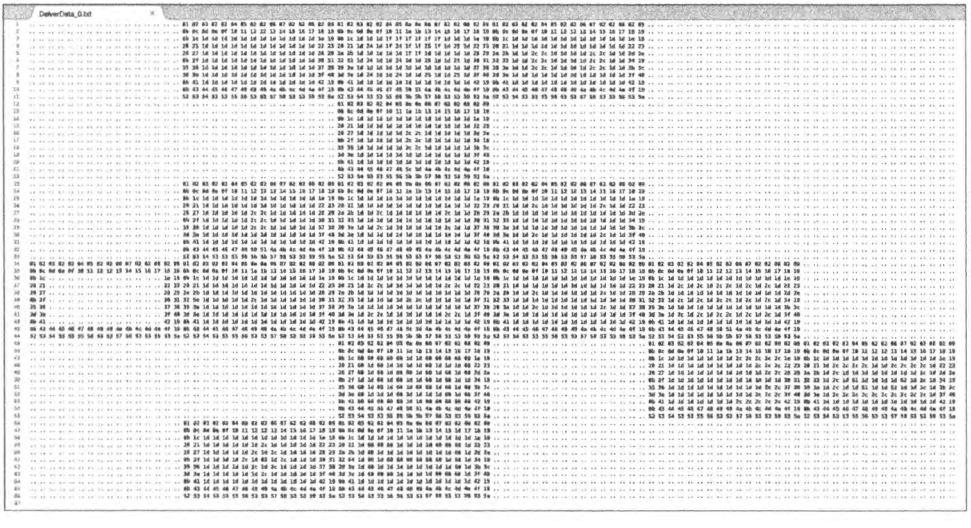


Рис. 35.4. Файл DelverData.txt

На рис. 35.4 показано содержимое файла *DelverData.txt*. Как видите, это перевернутая версия подземелья *Eagle* из игры *The Legend of Zelda*, которое описывалось

в главе 9 «Прототипирование на бумаге»[1]. Изображение перевернуто, потому что текстовые файлы всегда читаются сверху вниз, тогда как начало оси Y на нашей карте подземелья будет находиться внизу.

Этот файл включает двузначные шестнадцатеричные числа (см. врезку ниже), разделенные пробелами. Чтобы легче было увидеть очертания подземелья, я заменил шестнадцатеричные значения `"00"` на `".."`.

ШЕСТНАДЦАТЕРИЧНЫЕ ЧИСЛА

Если прежде вам доводилось заниматься веб-разработкой, вы наверняка видели шестнадцатеричные числа в обозначениях цветов (например, `FF0000` соответствует ярко-красному цвету). В файле *DelverData.txt* используются двузначные шестнадцатеричные числа в диапазоне от 0 до 255 (в десятичной системе счисления).

Шестнадцатеричная система счисления, так же как десятичная, является позиционной (когда значение каждой цифры определяется ее местоположением в числе, — например, единицы, десятки и т. д.). В обычной десятичной системе счисления используются цифры от 0 до 9, а в шестнадцатеричной — от 0 до 15. Для представления цифр от 10 до 15 используются символы от `a` до `f`:

Десятичная	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Шестнадцатеричная	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10

Буквы `a-f` можно записывать в верхнем или в нижнем регистре. Я предпочел использовать в файле DelverData буквы нижнего регистра, потому что, как мне кажется, они легче читаются. В программах на C# шестнадцатеричные числа должны начинаться с префикса `0x`, чтобы компилятор мог отличить их от десятичных чисел (для большей ясности в этой врезке я буду добавлять к шестнадцатеричным числам префикс `0x` и оформлять их моноширинным шрифтом, а десятичные числа будут оформляться обычным шрифтом).

По аналогии с десятичными числами цифра в старшем разряде должна быть умножена на число, которое на 1 больше, чем можно представить самой большой цифрой. То есть шестнадцатеричное число `0 x 10` имеет цифру 1 в *старшем* разряде и может быть представлено как 1 × 16 + 0 × 1 (так же, как 10 в десятичной системе можно представить в виде 1 × 10 + 0 × 1). То есть шестнадцатеричное число `0 × 10` равно десятичному числу 16.

Один из замечательных аспектов использования шестнадцатеричной нумерации для обозначения плиток в изображении DelverTiles состоит в том, что всего в изо-

[1] Эта версия DelverData не включает никакой информации о стенах, которые можно разрушить бомбой, потому что этот трюк не используется в данной главе. Однако, закончив читать главу, вы сможете сами добавить эту информацию. Кроме того, эта версия не содержит информации о местоположении врагов и предметов, но далее вы увидите, как добавить ее в карту другого подземелья.

бражении заключено 256 плиток и двузначным шестнадцатеричным числом тоже можно выразить 256 разных значений. Это также означает, что первая цифра в шестнадцатеричном числе всегда будет обозначать номер ряда, а вторая — номер плитки в ряду. Например, чтобы получить номер плитки с красной статуей на карте, можно подсчитать количество рядов (начиная с 0), поместить получившуюся цифру в старший разряд, а затем подсчитать плитки в ряду до указанного места и поместить получившуюся цифру в младший разряд (рис. 35.5).

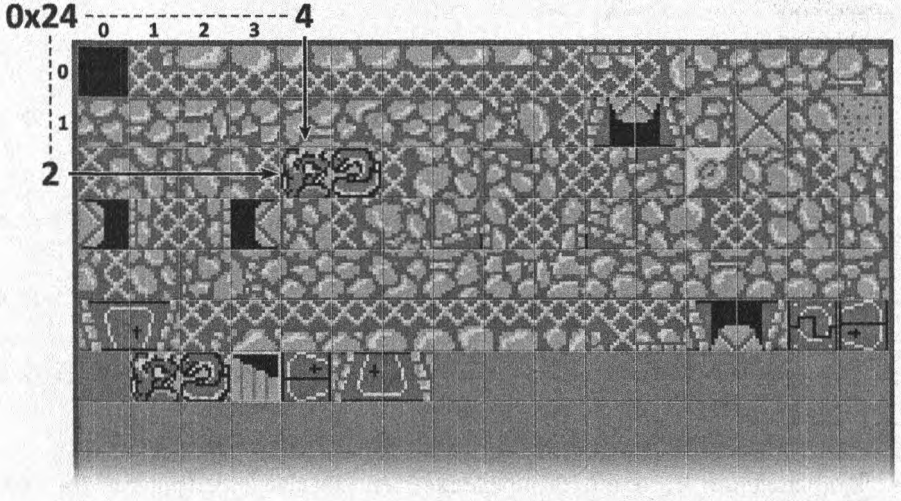


Рис. 35.5. Демонстрация определения номера плитки в DelverTiles

Генерирование карты из данных

Теперь у нас есть изображение для плиток и текстовый файл, описывающий расположение плиток, и мы можем объединить их в карту. Далее нам предстоит создать два класса, `TileCamera` и `Tile`. Они тесно связаны между собой, поэтому мы будем постоянно перепрыгивать между ними по мере добавления кода.

Класс Tile — подготовка

Начнем с класса `Tile`. По большому счету, класс `Tile` должен принимать целое число от `TileCamera`, определяющее изображение, отображаемое на плитке. Пока это все, что нам нужно.

1. Создайте новый спрайт в иерархии (**GameObject > 2D Object > Sprite** (Игровой объект > 2D объект > Спрайт)) с именем **Tile**.
2. Создайте в папке **__Scripts** новый сценарий на C# с именем **Tile**.

3. Подключите сценарий Tile к игровому объекту Tile в иерархии.
4. Откройте сценарий Tile в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Tile : MonoBehaviour {

    [Header("Set Dynamically")]
    public int              x;
    public int              y;
    public int              tileNum;

    public void SetTile(int eX, int eY, int eTileNum = -1) {                // a
        x = eX;
        y = eY;
        transform.localPosition = new Vector3(x, y, 0);
        gameObject.name = x.ToString("D3")+"x"+y.ToString("D3");            // b

        if (eTileNum == -1) {
            eTileNum = TileCamera.GET_MAP(x,y);                             // c
        }
        tileNum = eTileNum;
        GetComponent<SpriteRenderer>().sprite = TileCamera.SPRITES[tileNum]; // d
    }
}
```

a. В объявлении этого метода определяется необязательный параметр `eTileNum`. Если вызвать этот метод без параметра `eTileNum` или передать в нем –1, тогда по умолчанию его значение будет получено вызовом `TileCamera.GET_MAP()`.

b. Вызов метода `ToString("D3")` целых чисел `x` и `y` вернет строку в заданном формате. Спецификатор формата `"D"` означает, что строка результата должна представлять число в десятичной системе счисления (то есть в системе с основанием 10), а параметр спецификатора `"3"` требует, чтобы строка результата содержала не менее трех символов (с добавлением ведущих нулей, если необходимо). То есть если предположить, что x = 23 и y = 5, тогда эта строка кода вернет `"023x005"`. Дополнительную информацию о различных форматах можно найти в интернете, поискав по фразе «C# Строки стандартных числовых форматов».

c. Если метод получил в параметре `eTileNum` значение –1, номер изображения для плитки определяется из `TileCamera.MAP`. В листинге вызов этого метода окрашен в красный цвет, потому что класс `TileCamera` еще не определен.

d. Когда мы определим массив `TileCamera.SPRITES`, эта инструкция будет присваивать данному экземпляру `Tile` соответствующий спрайт.

5. Сохраните сценарий и вернитесь в Unity. Вы увидите в консоли два сообщения об ошибках, потому что мы пока не определили класс `TileCamera`.

6. Перетащите игровой объект Tile из иерархии в папку _Prefabs в панели Project (Проект), чтобы создать из него шаблон.
7. Удалите экземпляр Tile из иерархии.

Это пока все, что нужно. Текущая реализация позволит нам вызвать `SetTile()` из `TileCamera`, и экземпляр `Tile` автоматически поместит себя в нужные координаты и установит свое имя.

Класс TileCamera — парсинг файлов с данными и спрайтами

Класс `TileCamera` отвечает за извлечение и сохранение всех спрайтов из изображения *DelverTiles.png* и чтение файла *DelverData.txt* для определения местоположений плиток. Начнем с чтения двух файлов. Важно, чтобы оба файла, DelverData и DelverTiles, находились в папке Resources панели Project (Проект). «Resources» — это одно из специальных имен папок в проектах Unity. Unity включит в скомпилированный проект все файлы в папке *Resources*, независимо от их присутствия в сцене, а кроме того, любой файл из папки *Resources* можно загрузить в коде с помощью класса `Resources`, входящего в состав библиотеки `UnityEngine`.

1. Создайте в папке __Scripts новый сценарий на C# с именем TileCamera. Его пока нельзя подключить ни к какому игровому объекту из-за ошибки компилятора, которую можно видеть в консоли. Если прямо сейчас заглянуть в *Tile.cs*, можно увидеть, что имя `TileCamera` окрасилось синим цветом, но имена статических полей, следующих за ним, подсвечены красным, потому что они пока не определены.
2. Откройте сценарий TileCamera в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TileCamera : MonoBehaviour {
    static private int          W, H;
    static private int[,]       MAP;
    static public Sprite[]      SPRITES;
    static public Transform     TILE_ANCHOR;
    static public Tile[,]       TILES;

    [Header("Set in Inspector")]
    public TextAsset            mapData;
    public Texture2D            mapTiles;
    public TextAsset            mapCollisions; // Будет использоваться позже
    public Tile                 tilePrefab;

    void Awake() {
        LoadMap();
    }

    public void LoadMap() {
```

```
        // Создать TILE_ANCHOR. Он будет играть роль родителя для всех плиток Tile.
        GameObject go = new GameObject("TILE_ANCHOR");
        TILE_ANCHOR = go.transform;

        // Загрузить все спрайты из mapTiles
        SPRITES = Resources.LoadAll<Sprite>(mapTiles.name);                    // a

        // Прочитать информацию для карты
        string[] lines = mapData.text.Split('\n');                             // b
        H = lines.Length;
        string[] tileNums = lines[0].Split(' ');
        W = tileNums.Length;

        System.Globalization.NumberStyles hexNum;                              // c
        hexNum = System.Globalization.NumberStyles.HexNumber;
        // Сохранить информацию для карты в двумерный массив для ускорения доступа
        MAP = new int[W,H];
        for (int j=0; j<H; j++) {
            tileNums = lines[j].Split(' ');
            for (int i=0; i<W; i++) {
                if (tileNums[i] == "..") {
                    MAP[i,j] = 0;
                } else {
                    MAP[i,j] = int.Parse( tileNums[i], hexNum );               // d
                }
            }
        }
        print("Parsed "+SPRITES.Length+" sprites.");                           // e
        print("Map size: "+W+" wide by "+H+" high");
    }

    static public int GET_MAP( int x, int y ) {                                // f
        if ( x<0 || x>=W || y<0 || y>=H ) {
            return -1; // Предотвратить исключение IndexOutOfRangeException
        }
        return MAP[x,y];
    }

    // Перегруженная float-версия GET_MAP()
    static public int GET_MAP( float x, float y ) {
        int tX = Mathf.RoundToInt(x);
        int tY = Mathf.RoundToInt(y - 0.25f);                                  // g
        return GET_MAP(tX,tY);
    }

    static public void SET_MAP( int x, int y, int tNum ) {                     // f
        // Сюда можно поместить дополнительную защиту или точку останова.
        if ( x<0 || x>=W || y<0 || y>=H ) {
            return; // Предотвратить исключение IndexOutOfRangeException
        }
        MAP[x,y] = tNum;
    }
}
```

a. Изображение `mapTiles` (*DelveTiles.png*) находится в папке *Resources*, поэтому все спрайты можно загрузить с помощью метода `Resources.LoadAll<Sprite>()`.

b. Если присвоить имя файла *DelverData.txt* полю `TextAsset mapData`, к его содержимому можно обратиться через свойство `mapData.text`. Эта инструкция разбивает содержимое файла на отдельные строки по символу возврата каретки `'\n'` и помещает каждую строку в отдельный элемент массива `lines`. Общее число строк в массиве сохраняется в `H`. Затем первая строка разбивается на подстроки по символу пробела `' '`, и каждый двузначный шестнадцатеричный код сохраняется в отдельный элемент массива `tileNums`. Количество элементов в `tileNums` сохраняется в `W`.

c. Для преобразования строк с двузначными шестнадцатеричными кодами из `tileNums` в целые числа нам понадобится передать константу `System.Globalization.NumberStyles.HexNumber` в метод `int.Parse()`. Константа имеет настолько длинное имя, что, используя его, я не смог бы уместить в одну строку код в `// d`, поэтому я сохранил ее значение в переменной `hexNum`.

d. Метод `int.Parse()` пытается преобразовать строку в целое число. Передавая `NumberStyles hexNum` во втором параметре, мы сообщаем ему, что строка содержит шестнадцатеричное представление числа. В инструкции `if` можно видеть, что каждый элемент `tileNums` сначала проверяется на равенство со строкой `".."`, которая напрямую преобразуется в целое число 0.

e. Если в отладчике поставить в этой строке точку останова до запуска игры, вы сможете увидеть все значения, хранящиеся в статических полях `MAP` и `SPRITES`.

f. Статические общедоступные методы `GET_MAP()` и `SET_MAP()` обеспечивают защищенный доступ для чтения/записи к `MAP`, предотвращая появление исключения `IndexOutOfRangeException`.

g. Выражение `y - 0.25f` учитывает сложную перспективу в этой игре, когда верхняя половина тела персонажа игрока может находиться за пределами плитки, но считается, что он находится на этой плитке. Важность этого правила прояснится далее в этой главе, когда Grappler (крюк) должен будет определять, попал ли он на небезопасную плитку.

3. Сохраните сценарий TileCamera и вернитесь в Unity.

4. Теперь, когда все ошибки компиляции исчезли, подключите сценарий TileCamera к Main Camera.

5. В инспекторе настройте поля компонента TileCamera главной камеры Main Camera:

   - **mapData**: присвойте ссылку на DelverData из папки Resources в панели Project (Проект).
   - **mapTiles**: присвойте ссылку на DelverTiles из папки Resources в панели Project (Проект).
   - **mapCollisions**: присвойте ссылку на DelverCollisions из папки Resources в панели Project (Проект). Поле mapCollisions будет использоваться далее в этой главе.
   - **tilePrefab**: присвойте ссылку на Tile из папки _Prefabs в панели Project (Проект). Так как это поле имеет тип `Tile`, а не `GameObject`, вам придется перета-

щить шаблон Tile из панели Project (Проект) в поле в инспекторе. Щелчок на пиктограмме с изображением мишени в инспекторе рядом с полем tilePrefab в этом случае не поможет.

6. Сохраните сцену.
7. Щелкните на кнопке Play (Играть), и в консоли должны появиться следующие две строки:

```
Parsed 256 sprites.
Map size: 96 wide by 66 high
```

Отображение карты

Чтобы отобразить карту, нужно написать еще один метод.

TileCamera — ShowMap()

Добавим в класс `TileCamera` метод, отображающий сразу всю карту. Это не самая эффективная операция в проекте, но она будет выполняться достаточно быстро.

1. Добавьте в сценарий TileCamera следующие строки, выделенные жирным.

```
public class TileCamera : MonoBehaviour {
    ...
    public void LoadMap() {
        ...
        print("Parsed "+SPRITES.Length+" sprites.");
        print("Map size: "+W+" wide by "+H+" high");

        ShowMap();                                                    // a
    }

    /// <summary>
    /// Генерирует плитки сразу для всей карты.
    /// </summary>
    void ShowMap() {
        TILES = new Tile[W,H];

        // Просмотреть всю карту и создать плитки, где необходимо
        for (int j=0; j<H; j++) {
            for (int i=0; i<W; i++) {
                if (MAP[i,j] != 0) {
                    Tile ti = Instantiate<Tile>(tilePrefab);          // b
                    ti.transform.SetParent( TILE_ANCHOR );
                    ti.SetTile(i, j);                                 // c
                    TILES[i,j] = ti;
                }
            }
        }
    }

    static public int GET_MAP( int x, int y ) { ... }
    ...
}
```

a. В конец `LoadMap()` добавлен вызов `ShowMap()`, который вставит плитки в сцену.

b. Это другой способ использования `Instantiate`, отличный от показанных выше. Так как нам нужен экземпляр класса `Tile`, а не `GameObject`, к которому он подключен, мы создаем экземпляр шаблона `tilePrefab` как `Tile` и сохраняем ссылку на него в локальной переменной `ti`. Сам игровой объект `GameObject`, к которому подключена эта плитка, также будет создан в сцене, просто нам не придется обращаться к нему в коде.

c. Метод `SetTile()` переменной `ti` вызывается только с координатами (без необязательного параметра `eTileNum`, благодаря чему он получит значение для `tileNum` из `TileCamera.MAP`).

2. Сохраните сценарий, вернитесь в Unity и щелкните на кнопке **Play** (Играть).

Вы увидите, как за короткое время в панели **Scene** (Сцена) будет построена вся карта подземелья целиком. Чтобы получить более полное представление о происшедшем, щелкните дважды на **TILE_ANCHOR** в иерархии.

Главная камера **Main Camera** должна отобразить только одну комнату в подземелье, как показано на рис. 35.6. (Если комната не отобразилась, проверьте еще раз настройки **transform.position** главной камеры **Main Camera**, они должны соответствовать приводившимся выше в этой главе.)

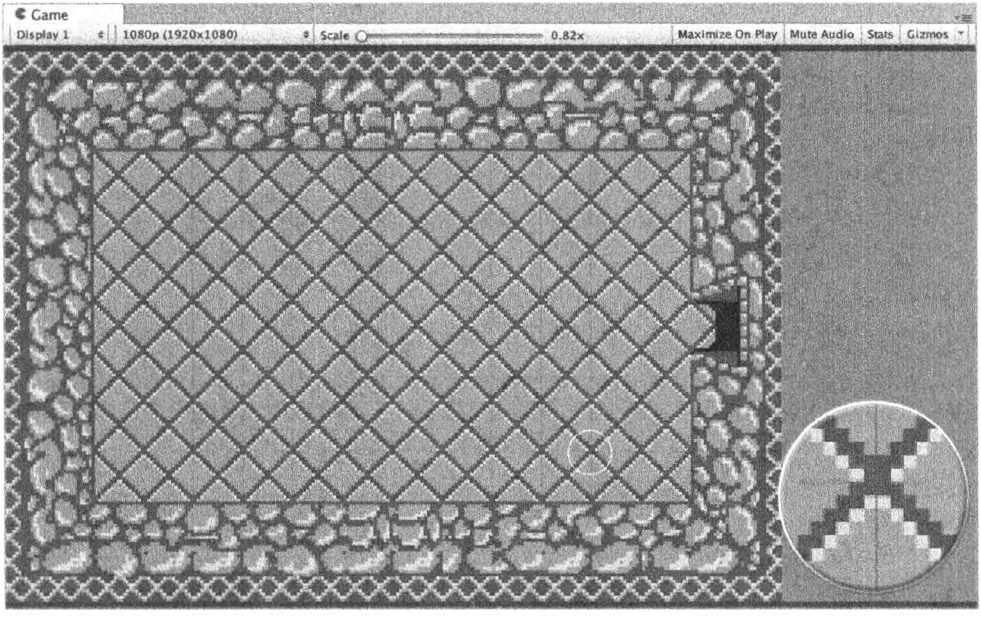


Рис. 35.6. Результат добавления метода ShowMap с увеличенным фрагментом, демонстрирующим проблему сглаживания

Решение проблемы сглаживания

По умолчанию Unity выполняет *сглаживание* (anti-aliasing), чтобы улучшить отображение графики (что происходит почти всегда), но это может вызывать проблемы с графикой в 8-битном стиле, как в проекте *Dungeon Delver*.

Сглаживание — это метод отображения с избыточной дискретизацией, когда Unity создает более крупное изображение экрана в памяти и затем понижает его разрешение, чтобы графика получилась более гладкой. По умолчанию используется разновидность метода избыточной дискретизации *2x Multi Sampling*. 2x Multi Sampling — это метод избыточной дискретизации, когда Unity создает изображение со стороной в два раза больше и затем сжимает его до обычного размера, затеняя пикселы в ходе этого процесса. Этот прием дает очень хорошие результаты при работе с трехмерной графикой, но может искажать двумерную графику, как показано в увеличенном фрагменте на рис. 35.6, где видно, как края плиток наезжают друг на друга, вместо того чтобы образовать ровный стык. В проекте *Dungeon Delver* следует отключить сглаживание, чтобы получить желаемый внешний вид.

1. В меню Unity выберите пункт Edit > Project Settings > Quality (Правка > Параметры проекта > Качество). В панели Inspector (Инспектор) откроется диспетчер настроек качества QualitySettings.

   По умолчанию вы должны увидеть, что выделена строка Ultra (темно-серым цветом). Также вы должны увидеть, что в колонке Standalone (у колонки для варианта автономной сборки Standalone в заголовке есть стрелка, направленная вниз), в строке Ultra, присутствует зеленый флажок. Это означает, что для сборки типа Standalone по умолчанию выбран уровень качества Ultra (вы можете установить качество по умолчанию для всех типов сборки, щелкнув на пиктограмме с треугольником в каждой колонке в строке Default).

2. Щелкните на строке Ultra, чтобы выбрать ее, и найдите параметр Anti Aliasing[1] в разделе Rendering.

3. Выберите в поле Anti Aliasing значение Disabled.

4. Сохраните сцену и снова попробуйте запустить ее.

Теперь края плиток не должны перекрывать друг друга. Чтобы получить дополнительную информацию о диспетчере настроек качества QualitySettings, щелкните на кнопке со знаком вопроса в верхнем углу диспетчера в инспекторе.

В версии Unity 2017 появилась также возможность включать и отключать сглаживание для отдельных камер во всех настройках качества. Выберите обе камеры, Main

[1] Многие пишут термин «anti-aliasing» (сглаживание) через дефис, но в Unity, в диспетчере настроек качества QualitySettings, этот термин отображается без дефиса.

Camera и GUI Camera, и установите флажок Allow MSAA. (MSAA — это сокращение от MultiSample Anti-Aliasing — сглаживание множественной выборкой.)

Добавление героя

Героем этой игры будет Дрей (Dray), рыцарь в доспехах. Для имитации 8-битной технологии оригинальной игры *Legend of Zelda* для героя предусмотрено отображение с четырех сторон, и его передвижение будет реализовано с применением анимационного эффекта. Позднее мы добавим также позу атакующего удара мечом или другим оружием.

Здесь мы впервые в книге столкнемся с анимацией спрайтов. Система поддержки анимационных эффектов в Unity позволяет создавать очень сложные, многослойные двумерные анимации. Это, конечно, очень здорово, но в действительности это не совсем то, что нам нужно в такой простой игре, как *Dungeon Delver*. Как следствие, мы будем использовать компоненты Animator и Animation не совсем обычным способом.

Соглашение об именовании спрайтов с изображением Дрея

Имена спрайтам с изображением Дрея должны даваться в соответствии со специальным соглашением, следование которому позволит нам использовать их в коде:

1. Раскройте папку _Images в панели Project (Проект), щелкнув на пиктограмме с треугольником.
2. Щелкните на пиктограмме с треугольником рядом с именем текстуры Dray.

Далее вы увидите, что я выбрал конкретные имена для спрайтов, импортированных из этого изображения (я сделал это в редакторе спрайтов, открыв его в диспетчере настроек импортирования Import Settings, в инспекторе, когда импортировал изображение Dray). Эти имена, наложенные на изображение Dray, показаны на рис. 35.7.

Одним из важнейших аспектов в создании этой игры является соглашение о применении нумерации в именах спрайтов, как показано на рис. 35.7. На протяжении всего этого проекта число 0 будет соответствовать направлению вправо, 1 — вверх, 2 — влево и 3 — вниз. Я выбрал такой порядок нумерации, потому что если представить стрелку, указывающую вправо (в положительном направлении вдоль оси X), и повернуть ее на 90 градусов относительно оси Z, она будет указывать вверх. Если повернуть стрелку на 180 градусов (или 2 × 90°), она будет указывать влево, и если повернуть стрелку на 270 градусов (или 3 × 90°), она будет указывать вниз. Как вы увидите далее, такое соответствие ориентации с нумерацией весьма эффективно используется во всей игре.

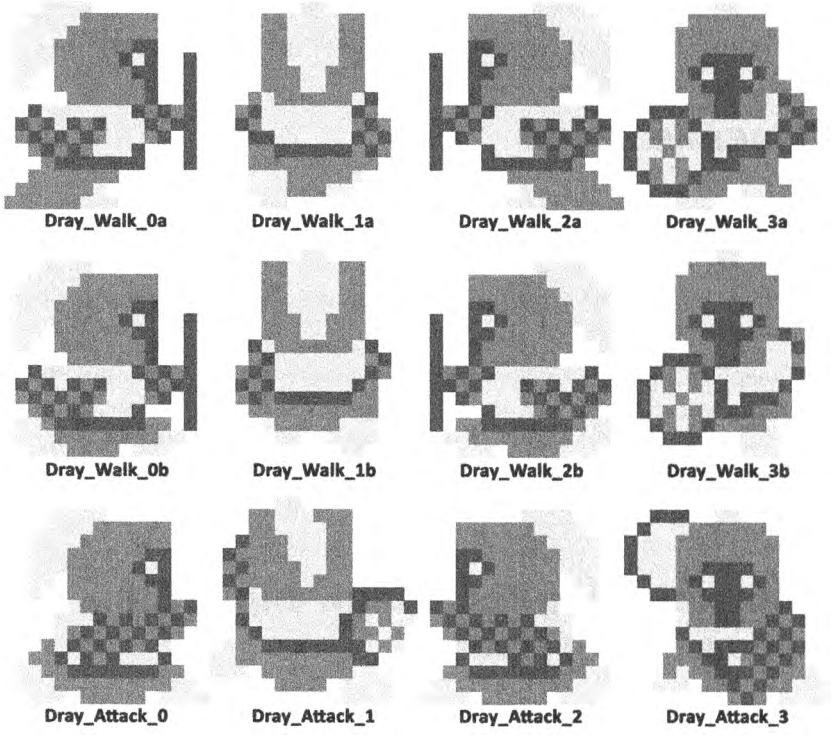


Рис. 35.7. Именованные спрайты в изображении Dray

Ваша первая анимация

Чтобы создать первую анимацию, выполните следующие шаги:

1. Создайте в панели Project (Проект) новую папку с именем _Animations (Assets > Create > Folder (Ресурсы > Создать > Папку)).
2. В панели Project (Проект) выберите Dray_Walk_0a и Dray_Walk_0b в списке спрайтов Dray, в папке _Images. (Для этого щелкните на первом спрайте, нажмите клавишу Shift и, удерживая ее, щелкните на втором спрайте.)
3. Перетащите их вместе из панели Project (Проект) в панель Hierarchy (Иерархия) и отпустите кнопку мыши. Откроется диалог, в котором вам будет предложено ввести имя анимации.
4. Введите имя Dray_Walk_0.anim и сохраните анимацию в папке *_Animations*.

   В результате будут созданы:

   - В папке _Animations, в панели Project (Проект):
     - Dray_Walk_0: анимация, которую вы сохранили с именем *Dray_Walk_0.anim*. Она включает два спрайта, изображающих Дрея, идущего вправо.

- Dray_Walk_0a: *аниматор*, хранящий несколько разных анимаций и управляющий порядком их отображения.

➢ В иерархии:

- Игровой объект с именем Dray_Walk_0a с подключенным компонентом аниматора Dray_Walk_0a. Это главный игровой объект, представляющий героя, которым будет управлять игрок.

Прежде чем продолжить, нужно внести кое-какие изменения.

5. Выберите игровой объект Dray_Walk_0a в иерархии и переименуйте его в Dray.
6. Выберите аниматор Dray_Walk_0a в папке _Animations в панели Project (Проект) и переименуйте его в Dray_Animator.
7. Щелкните дважды на Dray_Animator в панели Project (Проект). Откроется панель Animator (Аниматор), она показана вверху на рис. 35.8. Если панель Animator

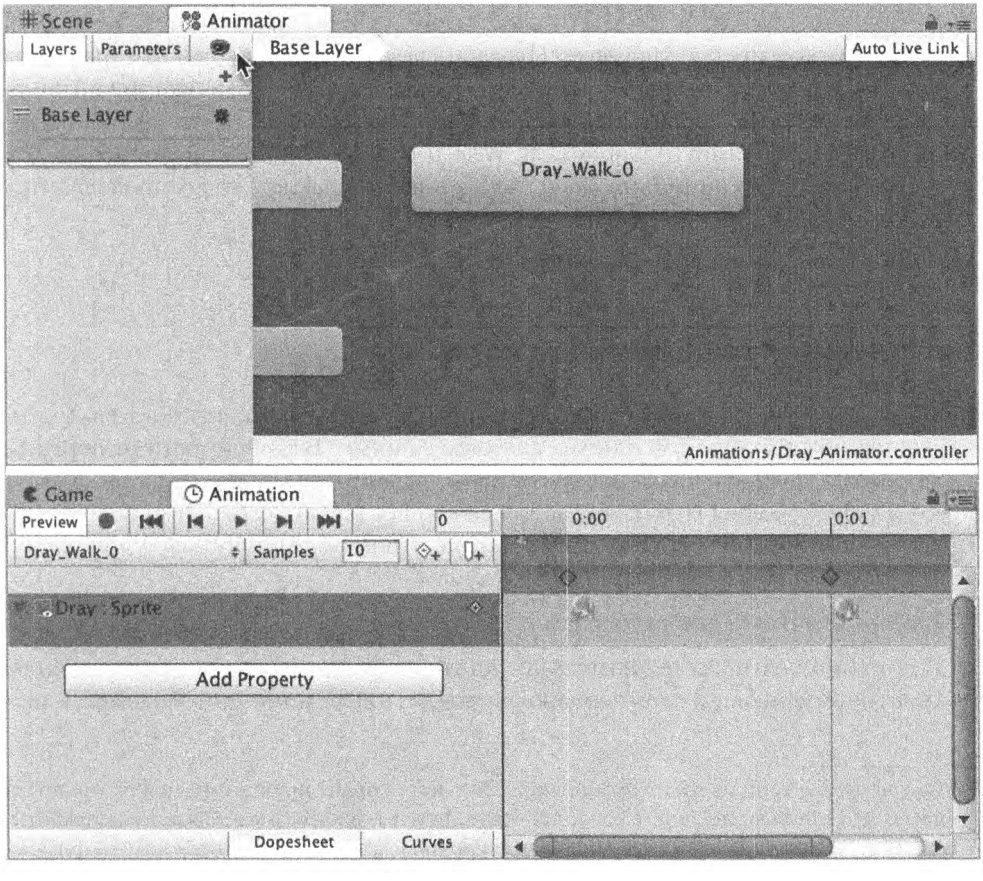


Рис. 35.8. Панели Animator (Аниматор) и Animation (Анимация)

(Аниматор) появится не как вторая вкладка в той же области экрана, где находится панель Scene (Сцена), то наведите указатель мыши на вкладку панели Animator (Аниматор), нажмите левую кнопку и, удерживая ее, перетащите правее вкладки панели Scene (Сцена).

8. Если панель Animator (Аниматор) слишком мала, чтобы увидеть происходящее в ней (как у меня на рис. 35.8), щелкните на кнопке с изображением открытого глаза (под указателем мыши на рис. 35.8), чтобы скрыть часть панели Animator (Аниматор), которую мы не будем использовать. Можно также нажать клавишу Option/Alt на клавиатуре и, удерживая ее, перетащить внутреннюю область в панели Animator (Аниматор), чтобы все необходимое оказалось в поле зрения.
9. Выберите объект Dray в иерархии и настройте координаты в компоненте Transform: P:[23.5, 5, 0]. В результате объект Dray переместится в центр поля зрения камеры Main Camera в панели Game (Игра).
10. Щелкните на кнопке Play (Играть).

    Unity отобразит панель Game (Игра), и вы увидите, как Дрей быстро бежит на месте в центре открытой комнаты[1]. Кроме того, в панели Animator (Аниматор) вы должны увидеть, как заполняется и опустошается синий индикатор в оранжевом прямоугольнике Dray_Walk_0, отражающий ход выполнения. Это показывает, что анимация Dray_Walk_0 в настоящее время воспроизводится снова и снова.
11. Остановите игру и **сохраните сцену**.

Организация спрайтов по слоям

В процессе перемещения Дрея по комнате можно заметить, что иногда он отображается поверх спрайтов плиток, а иногда за ними. В зависимости от порядка, в котором создавались разные элементы игры, вы могли не заметить такого эффекта у себя в панели Game (Игра); тем не менее вам все равно необходимо определить правильный порядок отображения спрайтов, чтобы не столкнуться с проблемой неправильного наложения спрайтов в будущем.

1. Выберите объект Dray в иерархии.
2. В его компоненте Sprite Renderer щелкните на раскрывающемся списке Sorting Layer, определяющем слой сортировки, где в данный момент отображается зна-

[1] Мы еще не настроили слои сортировки для наших спрайтов, поэтому может так получиться, что, когда вы щелкнете на кнопке Play (Играть), Дрей скроется за плитками карты. В этом случае щелкните на TILE_ANCHOR в иерархии и затем в инспекторе снимите флажок рядом с именем TILE_ANCHOR. Это сделает TILE_ANCHOR и все вложенные в него объекты неактивными, и вы увидите Дрея.

чение Default, и выберите пункт Add Sorting Layer (Добавить слой сортировки). Откроется диспетчер тегов и слоев Tags & Layers с раскрытым разделом Sorting Layers (Слои сортировки).

3. Щелкните на кнопке + в правом нижнем углу списка Sorting Layers (Слои сортировки) и дайте вновь созданному слою имя Dray.
4. Повторите эту процедуру и создайте еще три слоя: Ground, Enemies и Items.
5. Разместите слои в таком порядке, сверху вниз: Ground, Default, Enemies, Dray, Items. При таком размещении слой Ground всегда будет отображаться позади всех остальных, Enemies — между слоями Default и Dray, Dray — поверх всех слоев, кроме Items, и Items — поверх всех других слоев.
6. Выберите объект Dray в иерархии и в его компоненте Sprite Renderer, в инспекторе, выберите в раскрывающемся списке Sorting Layer пункт Dray. (Не пытайтесь изменить значение в списке Layer (Слой) в верхней части панели Inspector (Инспектор), — он определяет физический слой.)
7. Выберите шаблон Tile в папке _Prefabs, в панели Project (Проект), и в его компоненте Sprite Renderer, в инспекторе, выберите в раскрывающемся списке Sorting Layer пункт Ground. В результате все экземпляры Tile всегда будут отображаться позади любых других спрайтов.
8. Сохраните сцену.

По мере добавления других объектов в игру мы также будем определять для них свой слой сортировки.

Настройка анимации Dray_Walk_0

Прямо сейчас Дрей слишком быстро перебирает ногами и пока может бежать только вправо. Исправим оба этих недостатка в панели Animation (Анимация).

1. Щелкните на кнопке с тремя маленькими горизонтальными полосками в правом верхнем углу панели Game (Игра) — на рис. 35.8 она обведена красной рамкой. В открывшемся меню выберите пункт Add Tab > Animation (Добавить вкладку > Анимация), чтобы открыть панель Animation (Анимация) как вкладку в панели Game (Игра), как показано на рис. 35.8 внизу.
2. Чтобы панель Animation (Анимация) у вас выглядела как на рис. 35.8, выберите Dray в иерархии и раскройте раздел Dray : Sprite в панели Animation (Анимация), щелкнув на пиктограмме с треугольником рядом с названием раздела.

   В панели Animation (Анимация) присутствует раскрывающийся список, в котором в данный момент отображается Dray_Walk_0 (он находится непосредственно под кнопкой записи анимации с изображением красного кружка). На протяжении оставшейся части этого раздела я буду называть этот список *переключателем анимации*. Справа от переключателя анимации находится поле Samples, определяющее скорость анимации (как количество кадров в секунду).

3. Введите в поле Samples число 10 и нажмите Return/Enter. Согласно этому значению, Дрей будет делать примерно 5 шагов в секунду.
4. Чтобы увидеть, как это будет выглядеть на экране:
   - Перейдите в панель Scene (Сцена), щелкнув на вкладке Scene (Сцена), чтобы отобразить ее поверх панели Animator (Аниматор).
   - Щелкните дважды на объекте Dray в иерархии, чтобы переместить его в центр сцены.
   - Щелкните на кнопке Animation Play (Запустить анимацию), которая находится в панели Animation (Анимация), непосредственно над переключателем анимаций.
5. Щелкните на кнопке Animation Play (Запустить анимацию), чтобы остановить воспроизведение анимации.

Если цвет полосы шкалы времени в верхней части панели Animation (Анимация) изменится с голубого на красный, это значит, что включился *режим записи*. До версии Unity 2017 это происходило всегда после щелчка на кнопке Animation Play (Запустить анимацию), но теперь, похоже, эту проблему исправили. Однако если вы в какой-то момент окажетесь в режиме записи, щелкните на кнопке записи анимации с красным кружком, находящейся левее кнопки Animation Play (Запустить анимацию), чтобы выйти из него.

Добавление дополнительных анимаций для объекта Dray

Нам нужно добавить в объект Dray дополнительные анимации, чтобы правильно отображать его перемещение во всех четырех направлениях.

1. Щелкните на списке переключателя анимации и выберите пункт Create New Clip (Создать новый клип).
2. Дайте новому клипу имя *Dray_Walk_1.anim* и сохраните его в папке *_Animations*.
3. В поле Samples для новой анимации Dray_Walk_1 введите число 10 и нажмите Return/Enter.
4. В панели Project (Проект) выберите спрайты Dray_Walk_0a и Dray_Walk_0b и перетащите их в область плана-графика в панели Animation (Анимация), где на рис. 35.8 показаны изображения Дрея.

   Теперь панель Animation (Анимация) должна выглядеть как на рис. 35.8, за исключением того, что теперь спрайты изображают движение вверх (направление 1). При этом панель Animation (Анимация) может переключиться в режим записи. Если это произошло, щелкните на кнопке с красным кружком в панели Animation (Анимация), чтобы остановить запись.
5. Повторите шаги с 1-го по 4-й и создайте анимации Dray_Walk_2 и Dray_Walk_3, используя соответствующие имена и спрайты.
6. Закончив, сохраните сцену.

Состояния анимаций в аниматоре

Если вернуться в панель Animator (Аниматор), выбрав в главном меню Unity пункт Window > Animator (Окно > Аниматор), можно увидеть четыре состояния объекта Dray. Это компонент Animator игрового объекта Dray в иерархии, и он объединяет игровой объект Dray с четырьмя анимациями Dray_Walk_#.

Перемещение Дрея

Прежде чем начать перемещать объект Drey, добавим в него компонент твердого тела Rigidbody.

1. Выберите игровой объект Dray в иерархии.
2. Подключите к нему компонент Rigidbody (Component > Physics > Rigidbody (Компонент > Физика > Твердое тело)).
   - Снимите флажок Use Gravity.
   - В разделе Constraints:
     - Установите флажок Freeze Position Z.
     - Установите флажок Freeze Rotation X, Y и Z.
3. Сохраните сцену.

   Теперь добавим в объект Dray сценарий, управляющий движением.
4. Создайте в папке __Scripts новый сценарий на C# с именем Dray и подключите его к игровому объекту Dray в иерархии.
5. Откройте сценарий Dray в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Dray : MonoBehaviour {
    [Header("Set in Inspector")]
    public float            speed = 5;

    [Header("Set Dynamically")]
    public int              dirHeld = -1; // Направление, соответствующее
                                          //   удерживаемой клавише
    private Rigidbody       rigid;

    void Awake () {
        rigid = GetComponent<Rigidbody>();
    }

    void Update () {
        dirHeld = -1;
        if ( Input.GetKey(KeyCode.RightArrow) ) dirHeld = 0;              // a
        if ( Input.GetKey(KeyCode.UpArrow   ) ) dirHeld = 1;
```

```
        if ( Input.GetKey(KeyCode.LeftArrow ) ) dirHeld = 2;
        if ( Input.GetKey(KeyCode.DownArrow ) ) dirHeld = 3;

        Vector3 vel = Vector3.zero;
        switch (dirHeld) {                                          // b
            case 0:
                vel = Vector3.right;
                break;
            case 1:
                vel = Vector3.up;
                break;
            case 2:
                vel = Vector3.left;
                break;
            case 3:
                vel = Vector3.down;
                break;
        }

        rigid.velocity = vel * speed;
    }
}
```

a. Из-за особого порядка следования четырех однострочных инструкций `if` в начале функции `Update()`, если одновременно нажать несколько клавиш со стрелками, последнее выполнившееся условие отменит все предыдущие (например, если одновременно нажать клавиши со стрелками вниз и вправо, Дрей будет двигаться только вниз). Эта игра достаточно простая, поэтому я выбрал такой порядок действий, впрочем, закончив читать эту главу, вы, возможно, пожелаете изменить это.

b. Эту инструкцию `switch` мы целиком заменим в следующем разделе.

6. Сохраните сценарий **Dray**, вернитесь в Unity и щелкните на кнопке **Play** (Играть). Теперь вы сможете перемещать Дрея по сцене, нажимая клавиши со стрелками.

Более интересный подход к реализации движения

Поскольку для определения направления движения используется целочисленная переменная `dirHeld`, значение которой изменяется в диапазоне от 0 до 3, появляется несколько интересных способов ее применения. Например, инструкция `switch` с комментарием `// b` в предыдущем листинге, использующая значение `dirHeld` для выбора одного из четырех направлений, содержит много повторяющегося кода. Взгляните на следующий листинг, где показано, как сократить объем кода, зная, что `dirHeld` может принимать только значения 0–3.

Снова откройте сценарий **Dray** в MonoDevelop и измените код, как показано ниже:

```
public class Dray : MonoBehaviour {
    ...
    private Rigidbody       rigid;
```

```
    private Vector3[]       directions = new Vector3[] {
        Vector3.right, Vector3.up, Vector3.left, Vector3.down };          // a

    void Awake () {
        rigid = GetComponent<Rigidbody>();
    }

    void Update () {
        dirHeld = -1;
        if ( Input.GetKey(KeyCode.RightArrow) ) dirHeld = 0;
        if ( Input.GetKey(KeyCode.UpArrow   ) ) dirHeld = 1;
        if ( Input.GetKey(KeyCode.LeftArrow ) ) dirHeld = 2;
        if ( Input.GetKey(KeyCode.DownArrow ) ) dirHeld = 3;

        Vector3 vel = Vector3.zero;
        // Полностью удалите инструкцию switch, бывшую здесь
        if (dirHeld > -1) vel = directions[dirHeld];                      // b

        rigid.velocity = vel * speed;
    }
}
```

a. Этот массив `directions` векторов `Vector3` позволяет легко получить любое из четырех направлений.

b. Эта одна строка заменила 14-строчную инструкцию `switch` из предыдущего листинга!

Вернувшись в Unity и снова запустив игру, вы увидите, что та же функциональность получена меньшим количеством строк кода. Далее в этой главе мы продолжим использовать `dirHeld` подобным способом, чтобы упростить код.

Лучший способ обработки нажатых клавиш

По аналогии с массивом `directions` можно организовать хранение кодов клавиш, управляющих движением. Измените определение класса `Dray`, как показано ниже.

```
public class Dray : MonoBehaviour {
    [Header("Set in Inspector")]
    public float            speed = 5;

    [Header("Set Dynamically")]
    public int              dirHeld = -1; // Направление, соответствующее
                                          //   удерживаемой клавише
    private Rigidbody       rigid;

    private Vector3[]       directions = new Vector3[] {
        Vector3.right, Vector3.up, Vector3.left, Vector3.down };

    private KeyCode[] keys = new KeyCode[] { KeyCode.RightArrow,
        KeyCode.UpArrow, KeyCode.LeftArrow, KeyCode.DownArrow };          // a

    void Awake () {
```

```
        rigid = GetComponent<Rigidbody>();
    }

    void Update () {
        dirHeld = -1;
        // Удалите четыре инструкции "if ( Input.GetKey...", бывшие в этом месте
        for (int i=0; i<4; i++) {
            if ( Input.GetKey(keys[i]) ) dirHeld = i;                    // b
        }

        Vector3 vel = Vector3.zero;
        if (dirHeld > -1) vel = directions[dirHeld];

        rigid.velocity = vel * speed;
    }
}
```

a. Использование массива позволяет легко сослаться на любую из четырех клавиш.

b. Этот цикл `for` выполняет обход всех возможных кодов клавиш в массиве `keys` и определяет, какая из них нажата.

Добавив новый массив с кодами клавиш, мы вновь воспользовались знанием природы значения `dirHeld`, изменяющегося в диапазоне 0–3, и смогли усовершенствовать анализ нажатой клавиши.

Анимация движения Дрея

Снова откройте сценарий Dray в MonoDevelop и добавьте в него следующий код:

```
public class Dray : MonoBehaviour {
    ...
    private Rigidbody       rigid;
    private Animator        anim;                                        // a
    ...
    void Awake () {
        rigid = GetComponent<Rigidbody>();
        anim = GetComponent<Animator>();                                 // a
    }

    void Update () {
        ...
        rigid.velocity = vel * speed;

        // Анимация
        if (dirHeld == -1) {                                             // b
            anim.speed = 0;
        } else {
            anim.CrossFade( "Dray_Walk_"+dirHeld, 0 );                   // c
            anim.speed = 1;
        }
    }
}
```

a. Сохранение ссылки на компонент Animator игрового объекта Dray в скрытом поле `anim`.

b. Если не нажата ни одна из клавиш управления направлением движения (то есть `dir == -1`), установить скорость анимации равной 0, остановив текущую анимацию.

c. Если Дрей движется в каком-то направлении, присоединить число из `dirHeld` к концу `"Dray_Walk_"`, что даст в результате имя анимации, подключенной к объекту Dray![1]

Функция `anim.CrossFade()` требует от аниматора `anim` переключиться на новую анимацию и дает на переход 0 секунд. Если `anim` уже отображает требуемую анимацию, данный вызов не произведет никакого эффекта.

Сохраните сценарий Dray и вернитесь в Unity. Запустите игру и посмотрите, как при движении Дрея в разных направлениях воспроизводятся разные анимационные эффекты.

Добавление анимации атаки

Теперь пришло время дать Дрею возможность атаковать врага! С этой целью прежде всего реализуем анимацию атаки. (Нанесение ущерба врагам в результате атаки мы реализуем далее в этой главе.)

Создание анимации атаки

Чтобы создать анимацию атаки, выполните следующие шаги:

1. Выберите объект Dray в иерархии и перейдите в панель Animation (Анимация), выбрав в меню Unity пункт Window > Animation (Окно > Анимация).
2. В списке переключателя анимаций выберите пункт Create New Clip (Создать новый клип) и сохраните новый клип с именем Dray_Attack_0 в папке *_Animations*.
3. Выберите Dray_Attack_0 в списке переключателя анимаций.
4. Введите в поле Samples число 10 и нажмите Return/Enter.
5. В панели Project (Проект) выберите спрайт Dray_Attack_0 в изображении Dray, в папке _Images.
6. Перетащите спрайт Dray_Attack_0 из панели Project (Проект) в область планаграфика в панели Animation (Анимация).

[1] Если у вас анимации не воспроизводятся, проверьте еще раз, правильно ли вы задали их имена: Dray_Walk_0, Dray_Walk_1, Dray_Walk_2 и Dray_Walk_3. Также проверьте имена анимаций в панели Animator (Аниматор) — имена в панели Animator (Аниматор) и в файлах *.anim* не должны отличаться.

7. Повторите шаги со 2-го по 6-й и создайте анимации с именами Dray_Attack_1, Dray_Attack_2 и Dray_Attack_3, использовав соответствующие спрайты.

Теперь в панели Animator (Аниматор) должно появиться четыре новые анимации атаки.

Программирование анимаций атаки

На этот раз нам придется приложить больше усилий, чтобы добавить новую анимацию в класс Dray. В дополнение к полю dirHeld (хранящему направление движения, пока удерживается нажатой клавиша со стрелкой) мы должны добавить новое поле, facing (направление движения Дрея). В момент атаки Дрей будет стоять на месте, и мы должны гарантировать, что направление не изменится в этот момент.

1. Откройте сценарий Dray в MonoDevelop и добавьте в начало новые строки с перечислением и определениями полей.

```
public class Dray : MonoBehaviour {
    public enum eMode { idle, move, attack, transition }                    // a

    [Header("Set in Inspector")]
    public float            speed = 5;
    public float            attackDuration = 0.25f; // Продолжительность атаки
                                                    // в секундах
    public float            attackDelay = 0.5f;     // Задержка между атаками

    [Header("Set Dynamically")]
    public int              dirHeld = -1; // Направление, соответствующее
                                          //   удерживаемой клавише
    public int              facing = 1;   // Направление движения Дрея
    public eMode            mode = eMode.idle;                              // a

    private float           timeAtkDone = 0;                                // b
    private float           timeAtkNext = 0;                                // c

    private Rigidbody       rigid;
    private Animator        anim;
    ...
}
```

   a. Перечисление eMode и поле mode позволяют хранить и определять состояние Дрея.

   b. timeAtkDone — время, когда должна завершиться анимация атаки.

   c. timeAtkNext — время, когда Дрей сможет повторить атаку.

2. Появление новых полей существенно меняет реализацию метода Update(), поэтому в следующем листинге он приводится целиком. Он реализует те же идеи, что были описаны выше, но немного иначе. Замените метод Update() в своем классе Dray следующим кодом (новые строки выделены жирным):

```
void Update () {
    //----Обработка ввода с клавиатуры и управление режимами eMode----
    dirHeld = -1;
    for (int i=0; i<4; i++) {
        if ( Input.GetKey(keys[i]) ) dirHeld = i;
    }

    // Нажата клавиша атаки
    if (Input.GetKeyDown(KeyCode.Z) && Time.time >= timeAtkNext) {          // a
        mode = eMode.attack;
        timeAtkDone = Time.time + attackDuration;
        timeAtkNext = Time.time + attackDelay;
    }

    // Завершить атаку, если время истекло
    if (Time.time >= timeAtkDone) {                                         // b
        mode = eMode.idle;
    }

    // Выбрать правильный режим, если Дрей не атакует
    if (mode != eMode.attack) {                                             // c
        if (dirHeld == -1) {
            mode = eMode.idle;
        } else {
            facing = dirHeld;                                               // d
            mode = eMode.move;
        }
    }

    //----Действия в текущем режиме----
    Vector3 vel = Vector3.zero;
    switch (mode) {                                                         // e
        case eMode.attack:
            anim.CrossFade( "Dray_Attack_"+facing, 0 );
            anim.speed = 0;
            break;

        case eMode.idle:
            anim.CrossFade( "Dray_Walk_"+facing, 0 );
            anim.speed = 0;
            break;

        case eMode.move:
            vel = directions[dirHeld];
            anim.CrossFade( "Dray_Walk_"+facing, 0 );
            anim.speed = 1;
            break;
    }

    rigid.velocity = vel * speed;
}
```

a. Если нажата клавиша атаки (клавиша Z) и прошло достаточно много времени после предыдущей атаки, записать в `mode` режим `eMode.attack`.

Дополнительно установить значения полей `timeAtkDone` и `timeAtkNext`, чтобы экземпляр `Dray` мог определить, когда прекратить анимацию атаки и когда можно повторить атаку.

`b`. После переключения в режим атаки Дрей будет оставаться в нем, пока не завершится анимация атаки (что произойдет спустя `attackDuration` секунд (по умолчанию 0,25 секунды)). Через этот промежуток времени поле режима `mode` получит значение `eMode.idle`.

`c`. Если Дрей не находится в состоянии атаки, этот код выберет режим `idle` или `move`, в зависимости от того, нажата ли в этот момент клавиша управления движением (то есть если `dirHeld > -1`).

`d`. Поле `facing` устанавливается только здесь. Направление взгляда Дрея меняется только во время движения. Это гарантирует, что Дрей будет обращен лицом в правильном направлении во время атаки или после остановки.

`e`. После определения режима `mode` эта инструкция `switch` выбирает, что должно произойти в этом режиме. Здесь обрабатываются `anim` и `vel`.

Теперь, когда Дрей всегда повернут лицом в правильном направлении и может производить атакующий выпад, мы легко добавим применение оружия во время атаки.

Меч Дрея

Главное оружие Дрея — меч, которым он может наносить удары в любом направлении. Текстура со спрайтами меча была импортирована в составе пакета, в начале этой главы.

1. Выберите объект Dray в иерархии. Щелкните на нем правой кнопкой мыши и выберите в контекстном меню пункт Create Empty (Создать пустой). Переименуйте вновь созданный пустой игровой объект в SwordController.
2. Настройте компонент Transform объекта SwordController: P:[0, 0, 0] R:[0, 0, 0] S:[1, 1, 1].
3. Раскройте текстуру Swords в папке _Images, щелкнув на пиктограмме с треугольником.
4. Перетащите спрайт Swords_0 из панели Project (Проект) в панель Hierarchy (Иерархия). Вложите его в объект SwordController (который сам вложен в объект Dray).
   - В иерархии переименуйте экземпляр Swords_0 в Sword.
   - Настройте компонент Transform объекта Sword: P:[0.75, 0, 0] R:[0, 0, 0] S:[1, 1, 1].
   - В раскрывающемся списке Sorting Layer компонента Sprite Renderer объекта Sword выберите слой сортировки Enemies (находящийся выше слоя Ground, но ниже слоя Dray).

5. Добавьте в Sword компонент Box Collider (Component > Physics > Box Collider (Компонент > Физика > Коробчатый коллайдер)). Он по умолчанию должен приобрести оптимальные размеры, но если это не так, введите в поля раздела Size значения [1, 0.4375, 0.2].
   - Установите флажок Is Trigger в коллайдере.
6. Выберите SwordController в иерархии.
   - Щелкните на кнопке Add Component (Добавить компонент) в инспекторе.
   - В открывшемся меню выберите пункт New Script (Новый сценарий).
   - Дайте новому сценарию имя SwordController.
   - В панели Project (Проект) перетащите сценарий SwordController в папку _Scripts.
7. Откройте сценарий SwordController и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SwordController : MonoBehaviour {
    private GameObject   sword;
    private Dray         dray;

    void Start () {
        sword = transform.Find("Sword").gameObject;                    // a
        dray = transform.parent.GetComponent<Dray>();
        // Деактивировать меч
        sword.SetActive(false);                                        // b
    }

    void Update () {
        transform.rotation = Quaternion.Euler( 0, 0, 90*dray.facing );  // c
        sword.SetActive(dray.mode == Dray.eMode.attack);               // d
    }
}
```

a. Эти две строки отыскивают ссылки на вложенный игровой объект Sword и на экземпляр класса Dray, подключенный к родительскому игровому объекту.

b. После вызова метода SetActive(false) игровой объект не отображается, не обрабатывается механизмом определения столкновений, его сценарий не выполняется и т. д. После деактивации объект Sword становится невидимым.

c. Эта строка поворачивает меч в ту сторону, куда смотрит Дрей. Так как Sword вложен в этот игровой объект SwordController, который имеет смещение 0.75 вдоль локальной оси X, поворот объекта SwordController в направлении взгляда Дрея обеспечивает правильное отображение меча в его руках.

d. В каждом вызове Update меч активируется, если Дрей находится в состоянии атаки.

8. Сохраните все сценарии в MonoDevelop[1], вернитесь в Unity и щелкните на кнопке Play (Играть). Теперь вы можете двигать Дрея по комнате клавишами со стрелками и атаковать, нажимая Z. Меч должен появляться только в момент атаки и всегда быть нацеленным в правильном направлении.

Враги: скелеты

Скелеты — основные враги Дрея. Подобно скелетам в *The Legend of Zelda*, в этой игре скелеты тоже будут хаотически блуждать по комнатам, в которых они находятся. Скелеты могут проходить друг сквозь друга и наносить урон Дрею.

Изображения скелетов

Чтобы получить изображения скелетов:

1. Выберите два спрайта с именами Skeletos_0 и Skeletos_1 (вы найдете их в панели Project (Проект) > _Images > Skeletos).
2. Перетащите их оба в иерархию. В результате будет создана новая анимация. Сохраните ее с именем *Skeletos.anim* в папке *_Animations*.
3. В иерархии переименуйте игровой объект Skeletos_0 в Skeletos.
4. Настройте компонент Transform объекта Skeletos: P:[19, 7, 0].
5. Подключите к объекту Skeletos компонент Rigidbody (Component > Physics > Rigidbody (Компонент > Физика > Твердое тело)).
   - Снимите флажок Use Gravity.
   - В разделе Constraints:
     - Установите флажок Freeze Position Z.
     - Установите флажок Freeze Rotation X, Y и Z.
6. Добавьте в Skeletos сферический коллайдер (Component > Physics > Sphere Collider (Компонент > Физика > Сферический коллайдер)).
7. В раскрывающемся списке Sorting Layer компонента Sprite Renderer объекта Skeletos выберите слой сортировки Enemies.
8. Выберите объект Skeletos в иерархии и перейдите в панель Animation (Анимация), выбрав в меню Unity пункт Window > Animation (Окно > Анимация).
9. Введите в поле Samples число 5 и нажмите Return/Enter.
10. Сохраните сцену.

[1] Если пункт File > Save All (Файл > Сохранить все) выглядит неактивным, значит, вы уже сохранили все сценарии. Молодцы!

Щелкните на кнопке Play (Играть), и в комнате с Дреем вы должны увидеть скелет, бегущий на месте.

Базовый класс врагов Enemy

Все враги в игре *Dungeon Delver* будут наследовать общий базовый класс `Enemy`.

1. Создайте в папке __Scripts новый сценарий на C# с именем Enemy.
2. Откройте его в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Enemy : MonoBehaviour {
    protected static Vector3[] directions = new Vector3[] {                 // a
        Vector3.right, Vector3.up, Vector3.left, Vector3.down };

    [Header("Set in Inspector: Enemy")]                                     // b
    public float                maxHealth = 1;                              // c

    [Header("Set Dynamically: Enemy")]
    public float                health;                                     // c

    protected Animator          anim;                                       // c
    protected Rigidbody         rigid;                                      // c
    protected SpriteRenderer    sRend;                                      // c

    protected virtual void Awake() {                                        // d
        health = maxHealth;
        anim = GetComponent<Animator>();
        rigid = GetComponent<Rigidbody>();
        sRend = GetComponent<SpriteRenderer>();
    }
}
```

a. Массив `directions` используется врагами так же, как его использует Дрей. Он хранит одни и те же элементы для всех экземпляров врагов, поэтому был объявлен статическим. Также он объявлен защищенным (`protected`), чтобы доступ к нему имели все подклассы, наследующие класс `Enemy`.

b. Я изменил привычные заголовки `[Header(...)]`, чтобы в инспекторе было видно, какие поля унаследованы из базового класса `Enemy`, а какие определены в классе `Skeletos` и в других подклассах `Enemy`.

c. В классе `Enemy` также объявляется несколько полей, общих для всех подклассов врагов, в том числе поля, хранящие уровень здоровья и ссылки на часто используемые компоненты.

d. Метод `Awake()` устанавливает уровень здоровья по умолчанию и кэширует ссылки на компоненты в переменных `anim`, `rigid` и `sRend`. Объявление метода виртуальным и защищенным (`protected virtual`) позволит переопределить его в подклассах (как вы вскоре увидите).

Skeletos — подкласс Enemy

Создайте подкласс Skeletos, выполнив следующие шаги:

1. Создайте в папке __Scripts новый сценарий на C# с именем Skeletos.
2. Подключите его к игровому объекту Skeletos в иерархии.
3. Откройте сценарий в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Skeletos : Enemy {                                          // a
    [Header("Set in Inspector: Skeletos")]                               // b
    public int      speed = 2;
    public float    timeThinkMin = 1f;
    public float    timeThinkMax = 4f;

    [Header("Set Dynamically: Skeletos")]
    public int      facing = 0;
    public float    timeNextDecision = 0;

    void Update () {
        if (Time.time >= timeNextDecision) {                             // c
            DecideDirection();
        }
        // Поле rigid унаследовано от класса Enemy и инициализируется
        // в Enemy.Awake()
        rigid.velocity = directions[facing] * speed;
    }

    void DecideDirection() {                                             // d
        facing = Random.Range(0,4);
        timeNextDecision = Time.time + Random.Range(timeThinkMin,timeThinkMax);
    }
}
```

a. Skeletos — наследует класс Enemy (не MonoBehaviour).

b. Я изменил привычные заголовки [Header(...)], чтобы в инспекторе было видно, какие поля унаследованы из базового класса Enemy, а какие являются частью класса Skeletos.

c. Если прошло достаточно времени с момента последнего изменения направления движения, вызывается DecideDirection(), чтобы решить, куда двигаться дальше.

d. В DecideDirection() выбирается случайное направление, а также случайный интервал времени до следующей смены направления.

4. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и щелкните на кнопке Play (Играть).

Вы должны увидеть скелет, блуждающий по комнате и свободно проходящий сквозь стены! Нам нужен сценарий, который будет удерживать скелеты и других врагов в их комнатах.[1]

Сценарий InRoom

Подземелье поделено на несколько комнат, каждая по 16 метров в ширину и 11 метров в глубину. Сценарий InRoom реализует несколько вспомогательных служб. Чтобы определить попытку скелета выйти из комнаты, нужно знать, где в комнате он находится, а поскольку все комнаты в подземелье имеют одинаковые размеры, сделать это легко.

1. Создайте в папке __Scripts новый сценарий на C# с именем InRoom.
2. Откройте его в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class InRoom : MonoBehaviour {
    static public float     ROOM_W = 16;                                    // a
    static public float     ROOM_H = 11;
    static public float     WALL_T = 2;

    // Местоположение этого персонажа в локальных координатах комнаты
    public Vector2 roomPos {                                                // b
        get {
            Vector2 tPos = transform.position;
            tPos.x %= ROOM_W;
            tPos.y %= ROOM_H;
            return tPos;
        }
        set {
            Vector2 rm = roomNum;
            rm.x *= ROOM_W;
            rm.y *= ROOM_H;
            rm += value;
            transform.position = rm;
        }
    }

    // В какой комнате находится этот персонаж?
    public Vector2 roomNum {                                                // c
        get {
            Vector2 tPos = transform.position;
```

[1] Если ваш скелет долго танцует на месте, проверьте, насколько правильно вы выполнили шаг 2 (подключение сценария). Однако пребывание на месте в течение некоторого времени — вполне допустимое состояние для скелета, поэтому попробуйте подождать чуть дольше.

```
            tPos.x = Mathf.Floor( tPos.x / ROOM_W );
            tPos.y = Mathf.Floor( tPos.y / ROOM_H );
            return tPos;
        }
        set {
            Vector2 rPos = roomPos;
            Vector2 rm = value;
            rm.x *= ROOM_W;
            rm.y *= ROOM_H;
            transform.position = rm + rPos;
        }
    }
}
```

a. Эти статические поля определяют ширину и высоту комнаты (в единицах Unity/метрах/плитках, что суть одно и то же). `WALL_T` определяет толщину стен.

b. Свойство `roomPos` позволяет определить или изменить местоположение игрового объекта относительно левого нижнего угла комнаты (которому соответствуют координаты X:0, Y:0).

c. Свойство `roomNum` позволяет определить или изменить комнату, в которой находится игровой объект (левой нижней комнате в подземелье соответствуют координаты X:0, Y:0). Если у игрового объекта изменить только это свойство, он окажется в другой комнате в той же позиции `roomPos`, что и в предыдущей комнате.

Эту базовую версию сценария InRoom можно подключить к разным игровым объектам и получить возможность определять их местоположения в любых комнатах. Также InRoom дает возможность изменять местоположение игрового объекта внутри комнаты и перемещать его в другие комнаты.

Удержание игровых объектов внутри комнат

Как упоминалось выше, нам нужно удержать скелет внутри комнаты. Для этого добавим метод `LateUpdate()`, который будет проверять, не вышел ли скелет за пределы основной области комнаты. Метод `LateUpdate()` вызывается в каждом кадре после вызова метода `Update()` всех игровых объектов[1]. Метод `LateUpdate()` отлично подходит для выполнения заключительных операций, таких как возврат увлекшихся персонажей в комнату, где они должны находиться.

1. Подключите сценарий InRoom к объекту Skeletos в иерархии.
2. Откройте сценарий InRoom в MonoDevelop и добавьте следующие строки, выделенные жирным:

[1] Например, если пять игровых объектов имеют метод `Update()` и два из них — метод `LateUpdate()`, сначала будут вызваны методы `Update()` всех пяти объектов и только потом методы `LateUpdate()` двух объектов, имеющих их.

```
public class InRoom : MonoBehaviour {
    static public float     ROOM_W = 16;
    static public float     ROOM_H = 11;
    static public float     WALL_T = 2;

    [Header("Set in Inspector")]
    public bool             keepInRoom = true;
    public float            gridMult = 1;                                    // a

    void LateUpdate() {
        if (keepInRoom) {                                                    // b
            Vector2 rPos = roomPos;
            rPos.x = Mathf.Clamp( rPos.x, WALL_T, ROOM_W-1-WALL_T );         // c
            rPos.y = Mathf.Clamp( rPos.y, WALL_T, ROOM_H-1-WALL_T );
            roomPos = rPos;                                                  // d
        }
    }

    // Местоположение этого персонажа в локальных координатах комнаты
    public Vector2 roomPos { ... }
    ...
}
```

a. Поле `gridMult` будет использовано далее в этой главе.

b. Если флажок **keepInRoom** установлен, тогда в каждом кадре следующие строки будут удерживать координаты `roomPos` этого игрового объекта в пределах стен, ограничивающих комнату.

c. `Mathf.Clamp()` гарантирует, что координата `rPos.x` будет иметь значение между минимальным значением `WALL_T` и максимальным значением `ROOM_W-1-WALL_T`, то есть помешает скелету проходить сквозь стены комнаты.

d. После наложения ограничений новые координаты `rPos` присваиваются свойству `roomPos`, в результате чего выполнится метод записи `set` свойства `roomPos` и игровой объект вернется обратно в комнату (если он попытался покинуть ее).

3. Сохраните сценарий **InRoom** и вернитесь в Unity, чтобы протестировать его. Теперь скелет все так же будет блуждать по комнате, но уже не сможет проникать сквозь стены, как прежде.

Сценарий **InRoom** очень удобно использовать для определения столкновений врагов со стенами комнаты, но нам также нужно реализовать возможность определения столкновений персонажей с некоторыми плитками (например, статуями и обелисками, присутствующими в некоторых комнатах). Для этого реализуем поддержку столкновений на уровне плиток.

Столкновения с плитками

Информация о местоположениях всех плиток в подземелье хранится в текстовом файле **DelverData**, а изображения плиток хранятся в файле **DelverTiles**. У нас имеется еще один текстовый файл, **DelverCollisions**, в котором хранится информация о воз-

можности столкновения с плитками разных типов в некотором закодированном виде (рис. 35.9).

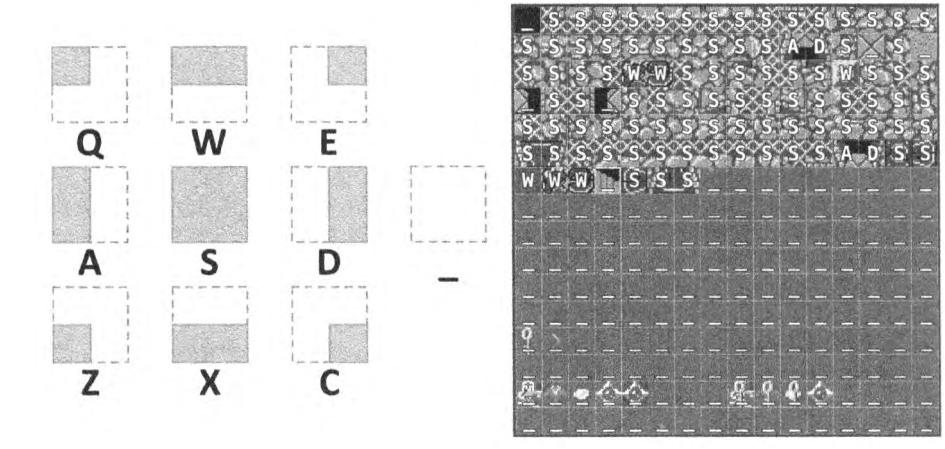


Рис. 35.9. Коды из текстового файла DelverCollisions (*слева*), наложенные на изображение в DelverTiles (*справа*)

Слева на рис. 35.9 можно видеть коды, используемые в текстовом файле DelverCollisions, соответствующие форме коллайдеров Box Collider для всех плиток из файла DelverTiles. Каждая буква слева представляет определенный вид столкновения. Серая пунктирная граница определяет всю плитку, а зеленая область — часть плитки, охватываемую коллайдером. Например, плитка с кодом W имеет коллайдер, охватывающий только верхнюю ее половину. Этот код присваивается всем столбам и статуям, находящимся в середине комнаты. В *Dungeon Delver* используются следующие коды: _ (не поддерживает столкновения), S (столкновения определяются со всей плиткой), A (с левой половиной), D (с правой половиной) и W (с верхней половиной), но я также добавил другие коды (Q, E, Z, X и C) для любых других игр, которые вы, возможно, пожелаете создать.

Справа на рис. 35.9 можно видеть содержимое текстового файла DelverCollisions, наложенное на изображения плиток в файле DelverTiles.

Чтобы воспользоваться информацией из DelverCollisions, добавим коллайдер в шаблон Tile.

1. Выберите игровой объект Tile в папке _Prefabs в панели Project (Проект).
2. Подключите к нему коллайдер Box Collider (Component > Physics > Box Collider (Компонент > Физика > Коробчатый коллайдер))[1].

[1] В отличие от большинства игровых объектов с коллайдерами, шаблон Tile не нуждается в компоненте Rigidbody, потому что его экземпляры никуда не перемещаются. Если позднее вы решите добавить что-нибудь вроде скользящих блоков, тогда вам придется добавить Rigidbody в Tile.

Определение столкновений с плитками

Далее, добавим несколько строк в сценарии TileCamera и Tile, которые будут использовать информацию из DelverCollisions.

1. Откройте сценарий TileCamera в MonoDevelop и добавьте следующие строки, выделенные жирным:

```
public class TileCamera : MonoBehaviour {
    static public int           W, H;
    static private int[,]       MAP;
    static public Sprite[]      SPRITES;
    static public Transform     TILE_ANCHOR;
    static public Tile[,]       TILES;
    static public string        COLLISIONS;                                    // a

    [Header("Set in Inspector")]
    ...
    void Awake() {
        COLLISIONS = Utils.RemoveLineEndings( mapCollisions.text );           // b
        LoadMap();
    }
    ...
}
```

   a. Статическое общедоступное строковое поле `COLLISIONS` может быть получено из любого сценария. Тип `String` прекрасно подходит здесь, потому что позволяет обращаться к отдельным символам строки с помощью индексов. Это даст нам возможность извлекать символьные коды столкновений по значению `tileNum`.

   b. Здесь содержимое текстового ресурса `mapCollisions` передается в вызов метода класса `Utils`, который отсекает символы завершения строки (в результате получается строка с 256 символами, не имеющая символов разрыва строк), оставляя нам массив символов (в форме строки), размер которого соответствует размеру массива спрайтов.

2. Сохраните сценарий TileCamera.

3. Откройте сценарий Tile и добавьте следующие строки, выделенные жирным:

```
public class Tile : MonoBehaviour {

    [Header("Set Dynamically")]
    public int              x;
    public int              y;
    public int              tileNum;

    private BoxCollider     bColl;                                             // a

    void Awake() {
        bColl = GetComponent<BoxCollider>();                                   // a
    }

    public void SetTile(int eX, int eY, int eTileNum = -1) {
        ...
```

```
        GetComponent<SpriteRenderer>().sprite = TileCamera.SPRITES[tileNum];

        SetCollider();                                                    // b
    }

    // Настроить коллайдер для этой плитки
    void SetCollider() {
        // Получить информацию о коллайдере из Collider DelverCollisions.txt
        bColl.enabled = true;
        char c = TileCamera.COLLISIONS[tileNum];                          // c
        switch (c) {
            case 'S': // Вся плитка
                bColl.center = Vector3.zero;
                bColl.size = Vector3.one;
                break;
            case 'W': // Верхняя половина
                bColl.center = new Vector3( 0, 0.25f, 0 );
                bColl.size = new Vector3( 1, 0.5f, 1 );
                break;
            case 'A': // Левая половина
                bColl.center = new Vector3( -0.25f, 0, 0 );
                bColl.size = new Vector3( 0.5f, 1, 1 );
                break;
            case 'D': // Правая половина
                bColl.center = new Vector3( 0.25f, 0, 0 );
                bColl.size = new Vector3( 0.5f, 1, 1 );
                break;

            // vvvvvvvv-------- Дополнительные коды --------vvvvvvvv       // d
            case 'Q': // Левая верхняя четверть
                bColl.center = new Vector3( -0.25f, 0.25f, 0 );
                bColl.size = new Vector3( 0.5f, 0.5f, 1 );
                break;
            case 'E': // Правая верхняя четверть
                bColl.center = new Vector3( 0.25f, 0.25f, 0 );
                bColl.size = new Vector3( 0.5f, 0.5f, 1 );
                break;
            case 'Z': // Левая нижняя четверть
                bColl.center = new Vector3( -0.25f, -0.25f, 0 );
                bColl.size = new Vector3( 0.5f, 0.5f, 1 );
                break;
            case 'X': // Нижняя половина
                bColl.center = new Vector3( 0, -0.25f, 0 );
                bColl.size = new Vector3( 1, 0.5f, 1 );
                break;
            case 'C': // Правая нижняя четверть
                bColl.center = new Vector3( 0.25f, -0.25f, 0 );
                bColl.size = new Vector3( 0.5f, 0.5f, 1 );
                break;
            // ^^^^^^^^-------- Дополнительные коды --------^^^^^^^^       // d

            default: // Все остальное: _, |, и др.                         // e
                bColl.enabled = false;
                break;
        }
    }
}
```

a. Поле `bColl` хранит ссылку на коллайдер этой плитки.

b. В конце метода `SetTile()` вызывается метод `SetCollider()`.

c. Здесь с помощью `tileNum` из `TileCamera.COLLISIONS` извлекается символ, определяющий вид столкновения.

d. Строки между комментариями `// d` (для символов `Q`, `E`, `Z`, `X` и `C`) не нужны в игре *Dungeon Delver*, но могут пригодиться в других проектах.

e. Заключительная инструкция `default` необходима для обработки символов `'_'`.

4. Сохраните сценарий Tile и вернитесь в Unity.

Добавления коллайдера в объект Dray

Наконец, чтобы увидеть результаты столкновений с отдельными плитками, добавим коллайдер в объект Dray.

1. Выберите объект Dray в иерархии.
2. Добавьте в него сферический коллайдер (Component > Physics > Sphere Collider (Компонент > Физика > Сферический коллайдер)).
   - Введите в поле Radius коллайдера значение 0.4.

Сохраните сцену и запустите ее. Подвигайте главного персонажа и посмотрите, что получится, если заставить его пройти сквозь стену. В отличие от скелетов, которые принудительно удерживаются в комнате, главный персонаж должен иметь возможность выходить в дверные проемы. Однако наблюдаются две проблемы:

- Довольно сложно попасть прямо в дверной проем.
- Фактически главный персонаж не переходит в другие комнаты.

Решим их по очереди.

Выравнивание по сетке

Оригинальная игра *Legend of Zelda* имеет остроумную систему, которая выравнивает местоположение главного персонажа по сетке и вместе с тем дает игрокам чувство свободы перемещения. Игрок имеет полную свободу движения, но чем дольше его персонаж движется в одном направлении, тем точнее его движения соответствуют сетке с шагом 0,5 единицы. Чтобы реализовать нечто подобное в *Dungeon Delver*, воспользуемся сценарием InRoom, написанным выше, куда добавим код для получения информации о ближайшем узле сетки.

1. Подключите сценарий InRoom к игровому объекту Dray в иерархии.
   - Снимите флажок keepInRoom в настройках Dray.

2. Откройте сценарий InRoom в MonoDevelop и добавьте следующий метод в конец определения класса. `GetRoomPosOnGrid()` будет вычислять координаты узла сетки в комнате, ближайшего к игровому объекту (размер стороны ячейки сетки по умолчанию равен 1 метру).

```
public class InRoom : MonoBehaviour {
    ...
    // В какой комнате находится этот персонаж?
    public Vector2          roomNum { ... }

    // Вычисляет координаты узла сетки, ближайшего к данному персонажу
    public Vector2 GetRoomPosOnGrid(float mult = -1) {
        if (mult == -1) {
            mult = gridMult;
        }
        Vector2 rPos = roomPos;
        rPos /= mult;
        rPos.x = Mathf.Round( rPos.x );
        rPos.y = Mathf.Round( rPos.y );
        rPos *= mult;
        return rPos;
    }
}
```

3. Сохраните сценарий InRoom и вернитесь в Unity.

Интерфейс IFacingMover

На самом деле нам нужно, чтобы не только Дрей, но и все создания в *Dungeon Delver* выравнивались по сетке в процессе движения, и в конечном счете мы должны применить будущий сценарий GridMove ко всем. И Дрей, и скелеты имеют некоторые сходные черты (например, могут быть обращены лицом в одном из четырех направлений, проявляют одинаковое поведение во время движения или когда стоят на месте, и т. д.), но единственным общим предком для классов `Dray` и `Skeletos` является класс `MonoBehaviour`. Это именно тот момент, когда стоит задуматься об использовании *интерфейса* C#. Более подробное введение в интерфейсы вы найдете в разделе «Интерфейсы» приложения Б «Полезные идеи».

Если говорить кратко, интерфейс — это гарантия включения в класс объявлений конкретных методов и/или свойств. На любой класс, реализующий интерфейс, можно ссылаться в коде как на тип интерфейса, а не как на конкретный тип класса. Этот прием имеет несколько отличий от приема наследования, наиболее важными из которых являются:

- Класс может одновременно реализовать несколько разных интерфейсов, но наследовать только один суперкласс.
- Любые классы, какие бы суперклассы они ни наследовали, могут реализовать один и тот же интерфейс.

Интерфейс можно рассматривать как своего рода обещание: любой класс, наследующий интерфейс, обещает иметь конкретные методы и/или свойства, которые можно вызывать безо всякой опаски.

Интерфейс `IFacingMover`, который мы реализуем здесь, очень прост и его легко можно применить к обоим классам, `Skeletos` и `Dray`.

1. Создайте в папке `__Scripts` новый сценарий на C# с именем `IFacingMover` (обычно интерфейсам даются имена, начинающиеся с буквы `I`).
2. Откройте сценарий `IFacingMover` в MonoDevelop и введите следующий код. Обратите внимание, что интерфейс `IFacingMover` *не наследует* `MonoBehaviour` и интерфейс — это не *класс*.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface IFacingMover {                                    // a
    int GetFacing();                                               // b
    bool moving { get; }                                           // c
    float GetSpeed();
    float gridMult { get; }                                        // d
    Vector2 roomPos { get; set; }                                  // e
    Vector2 roomNum { get; set; }
    Vector2 GetRoomPosOnGrid( float mult = -1 );                   // f
}
```

a. Объявление общедоступного интерфейса `IFacingMover`.

b. Интерфейсы включают методы и свойства, которые должны быть общедоступными в любом классе, реализующем эти интерфейсы. Эта строка объявляет, что любой класс, реализующий интерфейс `IFacingMover`, должен иметь общедоступный метод `int GetFacing()`.

c. Кроме методов, интерфейс может также обещать наличие реализации свойств. Эта строка объявляет, что любой класс, реализующий интерфейс `IFacingMover`, должен иметь свойство `bool moving`, доступное только для чтения.

d. Свойство `gridMult`, доступное только для чтения, позволит реализациям `IFacingMover` передавать значение поля `gridMult` из `InRoom` в сценарии, такие как `GridMove`, без необходимости напрямую обращаться к `InRoom`.

e. Свойства `gridMult` и `roomPos`, а также метод `GetRoomPosOnGrid()` понадобятся для реализации доступа к обоим классам, `Dray` и `Skeletos`, из `InRoom`. Несмотря на то что в `InRoom` мы будем обращаться к экземплярам `Dray` и `Skeletos`, эти элементы позволят интерпретировать их как экземпляры `IFacingMover`, а не `Dray` и `Skeletos`.

f. Этот метод интерфейса включает определение значения по умолчанию для параметра `mult`. Если опустить этот параметр в вызове метода или передать

в нем –1, `GetRoomPosOnGrid()` будет использовать свойство `gridMult`. Интересно отметить: если в классе, реализующем интерфейс `IFacingMover`, определить для параметра `mult` другое значение по умолчанию, будет использоваться значение, объявленное в классе.

Реализация интерфейса IFacingMover в классе Dray

Выполните следующие шаги, чтобы реализовать интерфейс `IFacingMover` в классе `Dray`:

1. Откройте сценарий `Dray` и добавьте следующий код, реализующий интерфейс `IFacingMover`.

```
public class Dray : MonoBehaviour, IFacingMover {                    // a
    ...
    private Rigidbody       rigid;
    private Animator        anim;
    private InRoom          inRm;                                    // b

    ...

    void Awake () {
        rigid = GetComponent<Rigidbody>();
        anim = GetComponent<Animator>();
        inRm = GetComponent<InRoom>();                               // b
    }

    void Update () { ... }

    // Реализация интерфейса IFacingMover
    public int GetFacing() {                                         // c
        return facing;
    }

    public bool moving {                                             // d
        get {
            return (mode == eMode.move);
        }
    }

    public float GetSpeed() {                                        // e
        return speed;
    }

    public float gridMult {
        get { return inRm.gridMult; }
    }

    public Vector2 roomPos {                                         // f
        get { return inRm.roomPos; }
        set { inRm.roomPos = value; }
    }
```

```
    public Vector2 roomNum {
        get { return inRm.roomNum; }
        set { inRm.roomNum = value; }
    }

    public Vector2 GetRoomPosOnGrid( float mult = -1 ) {
        return inRm.GetRoomPosOnGrid( mult );
    }
}
```

a. Фрагмент ", IFacingMover" объявляет, что данный класс реализует интерфейс IFacingMover.

b. Переменная inRm открывает доступ к подключенному классу InRoom и инициализируется в методе Awake().

c. Реализация метода public int GetFacing(), объявленного в интерфейсе IFacingMover.

d. Реализация свойства public bool moving { get; }, доступного только для чтения, объявленного в интерфейсе IFacingMover.

e. Реализация метода float GetSpeed(), объявленного в интерфейсе IFacingMover.

f. Свойство roomPos действует точно так же, как любое другое свойство, доступное для чтения/записи.

2. Сохраните сценарий Dray и вернитесь в Unity, чтобы убедиться, что компиляция выполнилась без ошибок.

Может показаться, что мы проделали массу лишней работы, но посмотрите, что получится после реализации того же интерфейса в классе Skeletos.

3. Откройте сценарий Skeletos и добавьте следующие строки, выделенные жирным.

```
public class Skeletos : Enemy, IFacingMover {                          // a
    ...
    public float    timeNextDecision = 0;

    private InRoom  inRm;                                              // b

    protected override void Awake () {                                 // c
        base.Awake();
        inRm = GetComponent<InRoom>();
    }

    void Update () { ... }

    void DecideDirection() { ... }

    // Реализация интерфейса IFacingMover
    public int GetFacing() {
        return facing;
```

```
    }

    public bool moving { get { return true; } }                    // d

    public float GetSpeed() {
        return speed;
    }

    public float gridMult {
        get { return inRm.gridMult; }
    }

    public Vector2 roomPos {
        get { return inRm.roomPos; }
        set { inRm.roomPos = value; }
    }

    public Vector2 roomNum {
        get { return inRm.roomNum; }
        set { inRm.roomNum = value; }
    }

    public Vector2 GetRoomPosOnGrid( float mult = -1 ) {
        return inRm.GetRoomPosOnGrid( mult );
    }
}
```

a. Реализация интерфейса `IFacingMover` в `Skeletos` выглядит практически так же.

b. В `Skeletos` тоже нужно объявить и инициализировать переменную `inRm`.

c. Метод `Awake()` в `Skeletos` должен быть объявлен как `protected override`, чтобы работать сообща с методом `protected virtual Awake()` в суперклассе `Enemy`. Сначала этот метод `Awake()` вызывает метод `Awake()` базового класса (`Enemy:Awake()`), а затем присваивает значение переменной `inRm`.

d. Скелеты — экземпляры класса `Skeletos` — находятся в постоянном движении, поэтому реализация `bool moving { get; }` всегда возвращает `true`, в отличие от одноименного свойства в классе `Dray`.

4. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

Теперь экземпляры обоих классов, `Dray` и `Skeletos`, можно обрабатывать одним и тем же кодом, как экземпляры `IFacingMover`, и не нужно писать код отдельно для каждого класса. Чтобы убедиться в этом, реализуем сценарий GridMove.

Сценарий GridMove

Мы сможем подключить сценарий GridMove к любому игровому объекту, класс которого реализует интерфейс `IFacingMover`.

1. Создайте в папке __Scripts новый сценарий на C# с именем GridMove и подключите его к объекту Dray.

2. Откройте сценарий GridMove в MonoDevelop и введите следующий код.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GridMove : MonoBehaviour {
    private IFacingMover    mover;

    void Awake() {
        mover = GetComponent<IFacingMover>();                                    // a
    }

    void FixedUpdate() {
        if ( !mover.moving ) return; // Если объект не перемещается - выйти
        int facing = mover.GetFacing();

        // Если объект перемещается, применить выравнивание по сетке
        // Сначала получить координаты ближайшего узла сетки
        Vector2 rPos = mover.roomPos;
        Vector2 rPosGrid = mover.GetRoomPosOnGrid();
        // Этот код полагается на интерфейс IFacingMover (который использует
        // InRoom) для определения шага сетки

        // Затем подвинуть объект в сторону линии сетки
        float delta = 0;
        if (facing == 0 || facing == 2) {
            // Движение по горизонтали, выравнивание по оси y
            delta = rPosGrid.y - rPos.y;
        } else {
            // Движение по вертикали, выравнивание по оси x
            delta = rPosGrid.x - rPos.x;
        }
        if (delta == 0) return; // Объект уже выровнен по сетке

        float move = mover.GetSpeed() * Time.fixedDeltaTime;
        move = Mathf.Min( move, Mathf.Abs(delta) );
        if (delta < 0) move = -move;

        if (facing == 0 || facing == 2) {
            // Движение по горизонтали, выравнивание по оси y
            rPos.y += move;
        } else {
            // Движение по вертикали, выравнивание по оси x
            rPos.x += move;
        }

        mover.roomPos = rPos;
    }
}
```

a. Метод `GetComponent<IFacingMover>()` отыскивает любой компонент, подключенный к текущему игровому объекту, который реализует интерфейс `IFacingMover`. Он вернет ссылку на экземпляр `Dray` или `Skeletos`, а если

в будущем появятся другие классы, реализующие интерфейс `IFacingMover`, он будет возвращать и их.

Основная реализация GridMove сосредоточена в методе `FixedUpdate()`, потому что он вызывается, когда физический движок выполняет расчеты и фактически перемещает разные игровые объекты.

3. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и щелкните на кнопке Play (Играть).

   Подвигайте Дрея в разных направлениях, и вы заметите, что он постепенно смещается к ближайшей линии сетки с размером ячейки в одну единицу. Теперь намного проще попасть в створ дверного проема.

4. Остановите игру.

5. Выберите объект Dray и в инспекторе введите в поле gridMult компонента InRoom число 0,5.

   Снова запустите игру, и вы увидите, что теперь главный персонаж двигается по сетке с размером ячейки в половину единицы, как в игре *The Legend of Zelda*.

6. Подключите сценарий GridMove к игровому объекту Skeletos в иерархии.

7. Сохраните сцену и снова щелкните на кнопке Play (Играть).

Внимательно понаблюдав за скелетами, можно заметить, что они перемещаются по сетке с размером ячейки в одну единицу.

Переход из комнаты в комнату

Теперь, когда Дрей без труда попадает в створ дверей, пришла пора направить его в другие комнаты подземелья. Так как Дрей — единственный персонаж, который может переходить из комнаты в комнату, большую часть необходимого для этого кода можно включить в класс `Dray`; однако глобальная информация о комнатах, такая как местоположение дверей и общий размер карты, все еще должна находиться в классе `InRoom`.

1. Откройте сценарий `InRoom` в MonoDevelop и введите следующий код:

```
public class InRoom : MonoBehaviour {
    static public float     ROOM_W = 16;
    static public float     ROOM_H = 11;
    static public float     WALL_T = 2;

    static public int        MAX_RM_X = 9;                          // a
    static public int        MAX_RM_Y = 9;

    static public Vector2[] DOORS = new Vector2[] {                 // b
        new Vector2(14,   5),
        new Vector2(7.5f, 9),
        new Vector2(1,    5),
```

```
        new Vector2(7.5f, 1)
    };

    [Header("Set in Inspector")]
    public bool              keepInRoom = true;
    ...
}
```

a. Статические поля MAX_RM_X и MAX_RM_Y определяют максимальные размеры карты. Значение 9 обусловлено возможностями редактора уровней *Delver Level Editor*, о котором рассказывается в конце этой главы. Если у вас появится желание увеличить размер карты, измените это значение.

b. Статический массив DOORS хранит информацию об относительном расположении дверей в комнатах.

2. Сохраните сценарий InRoom.

3. Откройте сценарий Dray в MonoDevelop и введите следующий код:

```
public class Dray : MonoBehaviour, IFacingMover {
    ...
    [Header("Set in Inspector")]
    ...
    public float            attackDelay = 0.5f;     // Задержка между атаками
    public float            transitionDelay = 0.5f; // Задержка перехода между
                                                    // комнатами // a

    [Header("Set Dynamically")]
    ...

    private float           timeAtkDone = 0;
    private float           timeAtkNext = 0;

    private float           transitionDone = 0;                          // a
    private Vector2         transitionPos;

    private Rigidbody       rigid;
    ...

    void Update () {
        if ( mode == eMode.transition ) {                                // b
            rigid.velocity = Vector3.zero;
            anim.speed = 0;
            roomPos = transitionPos; // Оставить Дрея на месте
            if (Time.time < transitionDone) return;
            // Следующая строка выполнится, только если Time.time >= transitionDone
            mode = eMode.idle;
        }

        //----Обработка ввода с клавиатуры и управление режимами eMode----
        dirHeld = -1
        ...
    }

    void LateUpdate() {
```

```
        // Получить координаты узла сетки, с размером ячейки
        //   в половину единицы, ближайшего к данному персонажу
        Vector2 rPos = GetRoomPosOnGrid( 0.5f ); // Размер ячейки в пол-единицы // c

        // Персонаж находится на плитке с дверью?
        int doorNum;
        for (doorNum=0; doorNum<4; doorNum++) {
            if (rPos == InRoom.DOORS[doorNum]) {
                break;                                                    // d
            }
        }

        if ( doorNum > 3 || doorNum != facing ) return;                   // e

        // Перейти в следующую комнату
        Vector2 rm = roomNum;
        switch (doorNum) {                                                // f
            case 0:
                rm.x += 1;
                break;
            case 1:
                rm.y += 1;
                break;
            case 2:
                rm.x -= 1;
                break;
            case 3:
                rm.y -= 1;
                break;
        }

        // Проверить, можно ли выполнить переход в комнату rm
        if (rm.x >= 0 && rm.x <= InRoom.MAX_RM_X) {                       // g
            if (rm.y >=0 && rm.y <= InRoom.MAX_RM_Y) {
                roomNum = rm;
                transitionPos = InRoom.DOORS[ (doorNum+2) % 4 ];          // h
                roomPos = transitionPos;
                mode = eMode.transition;                                  // i
                transitionDone = Time.time + transitionDelay;
            }
        }
    }

    // Реализация интерфейса IFacingMover
    public int GetFacing() { ... }
    ...
}
```

a. Обязательно добавьте эти строки.

b. Эти строки задерживают Дрея на месте на короткое время перед переходом в другую комнату, чтобы предотвратить переход в опасную комнату до того, как камера успеет переместиться в нее.

c. Принудительное выравнивание по сетке с ячейкой в половину единицы, потому что `InRoom.DOORS` также выравниваются с этим шагом.

d. Этот цикл выполняет обход всех дверей и прерывается, обнаружив дверь, в которой стоит игрок. Если игрок не находится ни в одной из дверей, по завершении цикла переменная `doorNum` будет иметь значение 4.

e. Если `doorNum` > 3 (то есть Дрей не находится ни в одной из дверей) или если Дрей не повернут в сторону выхода (то есть `doorNum != facing`), метод прекращает выполнение в этой точке.

f. Этот код выполняется, только если Дрей действительно должен пройти через двери. Инструкция `switch` изменяет координаты комнаты `roomNum` в зависимости от дверей, через которые выполняется переход.

g. Здесь проверяется наличие комнаты за дверью. Например, эта проверка не позволит Дрею выйти из подземелья через вход (в этом случае `roomNum.y` получит значение –1).

h. Выражение `(doorNum+2) % 4` выбирает противоположную дверь в комнате (например, если Дрей вышел в дверь `DOORS[3]`, он войдет через дверь `DOORS[1]`). Затем результат присваивается переменной `transitionPos` и в следующей строке ее значение переписывается в переменную `roomPos`, определяющую позицию Дрея — на выходе из дверей в следующей комнате.

i. Главный персонаж переводится в режим перехода `transition`, что вызывает короткую задержку перед дальнейшим перемещением и дает игроку возможность увидеть новую комнату перед переходом в нее.

4. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и щелкните на кнопке Play (Играть).

Пока камера не следует за Дреем в новую комнату, поэтому происходящее можно наблюдать только в панели Scene (Сцена), но теперь вы можете видеть, что Дрей получил возможность переходить из комнаты в комнату через двери.

Перемещение камеры вслед за Дреем

Теперь, когда Дрей получил возможность переходить из комнаты в комнату, мы должны заставить камеру следовать за ним.

1. Создайте в папке __Scripts новый сценарий на C# с именем CamFollowDray.
2. Подключите сценарий CamFollowDray к главной камере Main Camera в иерархии.
3. Откройте CamFollowDray в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CamFollowDray : MonoBehaviour {
    static public bool TRANSITIONING = false;

    [Header("Set in Inspector")]
```

```
    public InRoom       drayInRm;                                    // a
    public float        transTime = 0.5f;

    private Vector3     p0, p1;

    private InRoom      inRm;                                        // b
    private float       transStart;

    void Awake() {
        inRm = GetComponent<InRoom>();
    }

    void Update () {
        if (TRANSITIONING) {                                         // c
            float u = (Time.time - transStart) / transTime;
            if (u >= 1) {
                u = 1;
                TRANSITIONING = false;
            }
            transform.position = (1-u)*p0 + u*p1;
        } else {                                                     // d
            if (drayInRm.roomNum != inRm.roomNum) {
                TransitionTo( drayInRm.roomNum );
            }
        }
    }

    void TransitionTo( Vector2 rm ) {                                // e
        p0 = transform.position;
        inRm.roomNum = rm;
        p1 = transform.position + (Vector3.back * 10);
        transform.position = p0;

        transStart = Time.time;
        TRANSITIONING = true;
    }
}
```

a. Вы должны будете присвоить значение полю `drayInRm` в инспекторе.

b. CamFollowDray тоже использует свой экземпляр InRoom.

c. Если экземпляр `CamFollowDray` находится в режиме перехода, он перемещает камеру из старой комнаты (`p0`) в новую (`p1`) в течение 0,5 секунды (по умолчанию).

d. Если экземпляр `CamFollowDray` не в режиме перехода, выполняется проверка, находится ли Дрей в одной комнате с этим объектом (с главной камерой Main Camera).

e. Когда вызывается метод `TransitionTo()`, экземпляр `CamFollowDray` запоминает свою текущую позицию в `p0`, затем временно перемещается в новую комнату и запоминает позицию в `p1`. Выражение «`+ (Vector3.back * 10)`» необходимо, потому что простое присваивание значения переменной `roomNum`

в InRoom запишет в координату Z игрового объекта число 0. После этого экземпляр CamFollowDray возвращается в исходную позицию, инициализирует линейную интерполяцию из точки p0 в точку p1 и присваивает переменной TRANSITIONING значение true.

4. Сохраните сценарий CamFollowDray и вернитесь в Unity.
5. Выберите главную камеру Main Camera в иерархии.
6. Подключите к ней сценарий InRoom.
   - В инспекторе снимите флажок keepInRoom.
7. В инспекторе присвойте полю drayInRoom компонента CamFollowDray главной камеры Main Camera ссылку на игровой объект Dray. Это позволит компоненту CamFollowDray получить ссылку на компонент InRoom, подключенный к объекту Dray.
8. Сохраните сцену и щелкните на кнопке Play (Играть).

Теперь должна появиться возможность перемещаться между тремя нижними комнатами в подземелье, но теперь встает проблема запертой двери в средней комнате.

Отпирание дверей

Чтобы отпереть дверь, нужен ключ, а еще нужно заменить плитку запертой двери плиткой открытой двери. Код определяет столкновение Дрея с плиткой запертой двери. Если у Дрея есть ключ, когда он сталкивается с закрытой дверью, счетчик ключей должен уменьшиться на единицу. С верхними и нижними дверьми ситуация немного сложнее (в направлениях 1 и 3), потому что каждая из них представлена парой плиток, но мы справимся с этим.

Интерфейс IKeyMaster

Несмотря на то что Дрей — единственный персонаж, способный открывать двери в этом подземелье, я решил реализовать интерфейс IKeyMaster, чтобы показать, как можно реализовать несколько интерфейсов в одном классе.

1. Создайте в папке _Scripts новый сценарий на C# с именем IKeyMaster.
2. Откройте сценарий IKeyMaster в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface IKeyMaster {
    int keyCount { get; set; }                                    // a
    int GetFacing();                                              // b
}
```

a. Свойство keyCount позволит узнавать и изменять количество ключей.

b. Метод GetFacing() уже реализован в классе Dray (он объявлен также в интерфейсе IFacingMover).

3. Сохраните сценарий IKeyMaster и откройте сценарий Dray. Добавьте в него следующие строки, выделенные жирным.

```
public class Dray : MonoBehaviour, IFacingMover, IKeyMaster {              // a
    ...
    [Header("Set Dynamically")]
    public int              dirHeld = -1; // Направление, соответствующее
                                          // удерживаемой клавише
    public int              facing = 1;   // Направление, куда смотрит Дрей
    public eMode            mode = eMode.idle;
    public int              numKeys = 0;                                    // b

    private float           timeAtkDone = 0;
    ...
    // Реализация интерфейса IFacingMover
    public int GetFacing() {                                                // c
        return facing;
    }
    ...
    public Vector2 GetRoomPosOnGrid( float mult = -1 ) {
        return inRm.GetRoomPosOnGrid( mult );
    }

    // Реализация интерфейса IKeyMaster
    public int keyCount {                                                   // d
        get { return numKeys; }
        set { numKeys = value; }
    }
}
```

a. IKeyMaster добавлен в список интерфейсов, реализованных в классе Dray.

b. Поле public int numKeys хранит количество ключей, имеющихся у Дрея. Оно объявлено общедоступным, чтобы дать возможность изменять его в инспекторе. Также его можно было бы объявить скрытым (private) и снабдить атрибутом [SerializeField].

c. Метод GetFacing() уже реализован в классе Dray (потому что он объявлен и в интерфейсе IFacingMover).

d. Счетчик ключей keyCount реализован как простое общедоступное свойство.

4. Сохраните сценарий Dray. Теперь можно реализовать класс GateKeeper.

Класс GateKeeper

Класс GateKeeper отпирает двери, заменяя плитки с запертыми дверьми плитками с отпертыми дверьми из TileCamera.MAP. Метод Tile.SetTile() может принимать два параметра: (eX и eY с координатами плитки) или три (в дополнительном

параметре eTileNum можно передать плитку, которая должна быть установлена в указанные координаты). Мы немного изменим этот метод, чтобы он не только отображал спрайт, переданный в eTileNum, но и изменял содержимое TileCamera.MAP, сохраняя в нем новую плитку.

Реализация простой защиты для TileCamera.MAP

Эта книга посвящена разработке прототипов игр, поэтому основное внимание в ней уделяется созданию действующих прототипов игр, а не защите классов. Но я должен отметить, что прямые манипуляции статическим общедоступным массивом MAP в TileCamera из класса Tile — не самое лучшее решение. Именно поэтому массив TileCamera.MAP объявлен скрытым (private), а для работы с ним добавлены методы GET_MAP() и SET_MAP(), которые обеспечивают безопасную обработку значений eX и eY за пределами MAP и тем самым предотвращают появление исключений IndexOutOfRangeException.

Методы доступа, такие как SET_MAP(), добавляются еще и потому, что помогают определять источник ошибок. Если в будущем обнаружится, что содержимое MAP изменяется не так, как ожидалось, вы всегда сможете поставить точку останова в функции SET_MAP() и запустить отладчик. После этого выполнение будет приостанавливаться при каждом вызове SET_MAP(), и вы сможете просмотреть содержимое панели Call Stack (Стек вызовов) в MonoDevelop, чтобы определить, откуда был вызван метод SET_MAP() и какие аргументы он получил. Увидев неожиданный вызывающий метод или необычные аргументы, вы поймете причину своих бед.

Изменение TileCamera.MAP в Tile.SetTile()

Откройте сценарий Tile в MonoDevelop и внесите в метод SetTile() следующие изменения, выделенные жирным:

```
public class Tile : MonoBehaviour {
    ...
    public void SetTile(int eX, int eY, int eTileNum = -1) {
        ...
        if (eTileNum == -1) {
            eTileNum = TileCamera.GET_MAP[x,y];
        } else {
            TileCamera.SET_MAP(x, y, eTileNum); // Заменить плитку, если необходимо
        }
        tileNum = eTileNum;
        ...
    }
    ...
}
```

Реализация сценария GateKeeper

Создайте сценарий GateKeeper, выполнив следующие шаги:

1. Создайте в папке __Scripts сценарий на C# с именем GateKeeper.

2. Подключите его к игровому объекту Dray в иерархии.

3. Откройте сценарий GateKeeper в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GateKeeper : MonoBehaviour {
    // Следующие константы зависят от файла изображения по умолчанию DelverTiles.
    // Если вы переупорядочите спрайты в DelverTiles,
    // возможно, вам придется изменить эти константы!
    //--------Индексы плиток с запертыми дверьми                                    // a
    const int               lockedR = 95;
    const int               lockedUR = 81;
    const int               lockedUL = 80;
    const int               lockedL = 100;
    const int               lockedDL = 101;
    const int               lockedDR = 102;

    //-------- Индексы плиток с открытыми дверьми
    const int               openR = 48;
    const int               openUR = 93;
    const int               openUL = 92;
    const int               openL = 51;
    const int               openDL = 26;
    const int               openDR = 27;

    private IKeyMaster      keys;

    void Awake() {
        keys = GetComponent<IKeyMaster>();
    }

    void OnCollisionStay( Collision coll ) {                                      // b
        // Если ключей нет, можно не продолжать
        if (keys.keyCount < 1) return;

        // Интерес представляют только плитки
        Tile ti = coll.gameObject.GetComponent<Tile>();
        if (ti == null) return;

        // Открывать, только если Дрей обращен лицом к двери
        // (предотвратить случайное использование ключа)
        int facing = keys.GetFacing();
        // Проверить, является ли плитка закрытой дверью
        Tile ti2;
        switch (ti.tileNum) {                                                     // c
            case lockedR:
                if (facing != 0) return;                                          // d
                ti.SetTile( ti.x, ti.y, openR );
                break;

            case lockedUR:
                if (facing != 1) return;
```

```
            ti.SetTile( ti.x, ti.y, openUR );
            ti2 = TileCamera.TILES[ti.x-1, ti.y];
            ti2.SetTile( ti2.x, ti2.y, openUL );
            break;

        case lockedUL:
            if (facing != 1) return;
            ti.SetTile( ti.x, ti.y, openUL );
            ti2 = TileCamera.TILES[ti.x+1, ti.y];
            ti2.SetTile( ti2.x, ti2.y, openUR );
            break;

        case lockedL:
            if (facing != 2) return;
            ti.SetTile( ti.x, ti.y, openL );
            break;

        case lockedDL:
            if (facing != 3) return;
            ti.SetTile( ti.x, ti.y, openDL );
            ti2 = TileCamera.TILES[ti.x+1, ti.y];
            ti2.SetTile( ti2.x, ti2.y, openDR );
            break;

        case lockedDR:
            if (facing != 3) return;
            ti.SetTile( ti.x, ti.y, openDR );
            ti2 = TileCamera.TILES[ti.x-1, ti.y];
            ti2.SetTile( ti2.x, ti2.y, openDL );
            break;

        default:
            return; // Выйти, чтобы исключить уменьшение счетчика ключей
    }
    keys.keyCount--;
  }
}
```

a. Эти целочисленные константы определяют порядковые номера плиток с запертыми и открытыми дверьми (например, `lockedR` имеет значение 95, и 95-й спрайт в текстуре DelverTiles соответствует запертой двери, ведущей вправо).

b. Метод `OnCollisionStay()` сразу же завершится и вернет управление, если у Дрея нет ни одного ключа, если объект, с которым столкнулся Дрей, не является плиткой `Tile`, или если Дрей не обращен лицом в сторону запертой двери. Это предотвратит уменьшение счетчика ключей без фактического отпирания двери и позволит игроку пройти мимо двери, не отпирая ее.

c. Переменные не могут играть роль вариантов в инструкции `switch`, и это одна из причин, почему все номера плиток объявлены константами (`const`) в начале класса.

d. Если Дрей не обращен лицом в сторону двери, метод завершается.

4. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.
5. Щелкните на кнопке Play (Играть) в Unity и попробуйте перейти в комнату справа. Если попробовать выйти в запертую дверь на севере, у вас ничего не получится (пока).
6. Пока Unity находится в режиме игры, выберите объект Dray в иерархии и введите в поле numKeys компонента Dray (Script) число 6 (этого количества ключей достаточно, чтобы обойти все подземелье). Если теперь подойти к запертой двери, вы сможете миновать ее и исследовать остальное подземелье[1]!

Пользовательский интерфейс для отображения количества ключей и уровня здоровья

Наши игроки не смогут увидеть количество имеющихся у них ключей, заглянув в инспектор Unity, значит, мы должны добавить несколько элементов пользовательского интерфейса.

1. Создайте новый объект Canvas, выбрав в главном меню Unity пункт GameObject > UI > Canvas (Игровой объект > ПИ > Холст). В результате на верхнем уровне иерархии будут созданы объекты Canvas и Event System.
2. Выберите Canvas в иерархии.
3. Выполните следующие настройки в инспекторе:
   - В раскрывающемся списке Render Mode выберите пункт Screen Space — Camera.
   - Щелкните на маленьком изображении мишени справа от поля Render Camera и в появившемся диалоге на вкладке Scene (Сцена) выберите GUI Camera.

   Когда вы импортировали пакет в начале этой главы, в папку *_Prefabs* была скопирована панель пользовательского интерфейса с именем DelverPanel.
4. Перетащите объект DelverPanel из папки _Prefabs на объект Canvas в иерархии, чтобы вложить DelverPanel в Canvas. После этого справа на экране, в изображении, которое формируется камерой пользовательского интерфейса GUI Camera, должна появиться панель. По умолчанию элементы пользовательского интерфейса показывают, что у игрока 0 ключей и уровень здоровья составляет половину.

Теперь напишем сценарий, управляющий этим пользовательским интерфейсом.

---

[1] Вы все еще не сможете выйти в комнату слева. Дверь, ведущая в нее, особая — ее нельзя открыть ключом.

Поддержка уровня здоровья

В настоящий момент класс Dray хранит количество ключей, но не имеет никакой информации об уровне здоровья и не предусматривает возможности получения ранений главным героем. Исправим первый из этих недостатков.

1. Откройте сценарий Dray в MonoDevelop и введите следующий код:

```
public class Dray : MonoBehaviour, IFacingMover, IKeyMaster {
    public enum eMode { idle, move, attack, transition }

    [Header("Set in Inspector")]
    public float            speed = 5;
    public float            attackDuration = 0.25f; // Продолжительность атаки
                                                    // в секундах
    public float            attackDelay = 0.5f;     // Задержка между атаками
    public float            transitionDelay = 0.5f; // Задержка перехода между
                                                    // комнатами
    public int              maxHealth = 10;                                 // a

    [Header("Set Dynamically")]
    public int              dirHeld = -1; // Направление, соответствующее
                                          // удерживаемой клавише
    public int              facing = 1;   // Направление, куда смотрит Дрей
    public eMode            mode = eMode.idle;
    public int              numKeys = 0;

    [SerializeField]                                                        // b
    private int             _health;

    public int health {                                                     // c
        get { return _health; }
        set { _health = value; }
    }

    private float timeAtkDone = 0;
    private float timeAtkNext = 0;
    ...

    void Awake () {
        rigid = GetComponent<Rigidbody>();
        anim = GetComponent<Animator>();
        inRm = GetComponent<InRoom>();
        health = maxHealth;                                                 // d
    }
}
```

a. В пользовательском интерфейсе отображается пять кругов, каждый из которых представляет 2 пункта здоровья, то есть уровень здоровья главного персонажа может достигать 10 пунктов.

b. Атрибут [SerializeField] делает поле _health видимым (и доступным для изменения) в инспекторе Unity, хотя оно и скрытое.

c. Свойство `health` открывает любым сценариям доступ к скрытому полю `private int _health` для чтения/записи. Основное назначение этого свойства — помощь в отладке; вы сможете установить точку останова в методе `set`, если значение `_health` меняется по непонятным причинам, и выяснить их природу.

d. В момент создания экземпляра `Dray` свойству `health` присваивается максимальное значение.

2. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

Подключение пользовательского интерфейса к объекту Dray

Мы должны написать сценарий, отображающий значения свойств `health` и `numKeys` экземпляра `Dray`.

1. Создайте в папке __Scripts новый сценарий на C# с именем GuiPanel.
2. Подключите его к игровому объекту DelverPanel (вложен в объект Canvas в иерархии).
3. Откройте сценарий GuiPanel в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class GuiPanel : MonoBehaviour {
    [Header("Set in Inspector")]
    public Dray         dray;
    public Sprite       healthEmpty;
    public Sprite       healthHalf;
    public Sprite       healthFull;

    Text                keyCountText;
    List<Image>         healthImages;

    void Start () {
        // Счетчик ключей
        Transform trans = transform.Find("Key Count");                    // a
        keyCountText = trans.GetComponent<Text>();

        // Индикатор уровня здоровья
        Transform healthPanel = transform.Find("Health Panel");
        healthImages = new List<Image>();
        if (healthPanel != null) {                                        // b
            for (int i=0; i<20; i++) {
                trans = healthPanel.Find("H_"+i);
                if (trans == null) break;
                healthImages.Add( trans.GetComponent<Image>() );
            }
        }
```

```
    }

    void Update () {
        // Показать количество ключей
        keyCountText.text = dray.numKeys.ToString();                    // c

        // Показать уровень здоровья
        int health = dray.health;
        for (int i=0; i<healthImages.Count; i++) {                      // d
            if (health > 1) {
                healthImages[i].sprite = healthFull;
            } else if (health == 1) {
                healthImages[i].sprite = healthHalf;
            } else {
                healthImages[i].sprite = healthEmpty;
            }
            health -= 2;
        }
    }
}
```

a. Этот код целиком и полностью зависит от правильности имен объектов, вложенных в DelverPanel. Этот метод отыскивает объект с именем Key Count, вложенный в DelverPanel, и возвращает его компонент Transform. Затем переменной `keyCountText` присваивается ссылка на компонент Text этого дочернего компонента Transform. Здесь не выполняется никаких предохранительных проверок, поэтому, если имя Key Count изменится, вызов `GetComponent` возбудит ошибку обращения к пустой ссылке.

b. Сначала выполняется поиск объекта с именем Health Panel, вложенный в DelverPanel, и, если он существует, последовательно отыскиваются объекты с именами от H_0 до H_19, вложенные в Health Panel. В процессе поиска ссылки на их компоненты Image добавляются в список `healthImages`. Если какой-то из дочерних объектов не будет найден (например, H_5), выполнение цикла `for` прерывается.

c. Значение поля `numKeys` из экземпляра `dray` присваивается свойству `text` переменной `keyCountText`.

d. Реализация отображения индикатора уровня здоровья выглядит намного сложнее. Сначала извлекается текущее значение `health` из экземпляра `dray` и сохраняется в локальной переменной `int health`. Затем выполняется обход элементов списка `healthImage`, начиная с нижнего (H_0). Если значение `health` больше 1, отображается спрайт `healthFull`. Если значение `health` равно 1, отображается спрайт `healthHalf`, и, если значение `health` меньше 1, отображается спрайт `healthEmpty`. В конце каждой итерации значение локальной переменной `health` уменьшается на 2 и запускается следующая итерация. При такой организации каждый элемент в списке `healthImage` отображает до 2 единиц уровня здоровья.

4. Сохраните сценарий GuiPanel и вернитесь в Unity.

5. Выберите DelverPanel в иерархии и в инспекторе настройте компонент GuiPanel (Script):
   - Перетащите в поле dray объект Dray из иерархии.
   - В поле healthEmpty выберите спрайт Health_0 из изображения Health в папке _Images в панели Project (Проект).
   - В поле healthHalf выберите спрайт Health_1 из изображения Health в папке _Images в панели Project (Проект).
   - В поле healthFull выберите спрайт Health_2 из изображения Health в папке _Images в панели Project (Проект).
6. Щелкните на кнопке Play (Играть), и вы увидите, как индикатор уровня здоровья заполнится доверху.
7. Выберите Dray в иерархии, не останавливая игру в Unity, и в инспекторе попробуйте вводить разные значения в поля numKeys и _health компонента Dray (Script). Эти изменения должны отражаться в панели пользовательского интерфейса. Сохраните сцену.

Реализация нанесения ущерба Дрею врагами

Настал момент добавить опасностей в игровой мир, дав врагам возможность наносить раны Дрею в момент столкновения с ними. Кроме того, столкновение с врагом будет отбрасывать Дрея немного назад и на короткое время делать его неуязвимым.

Реализация DamageEffect

Сценарий DamageEffect будет использоваться для определения степени вреда, наносимого врагом Дрею, и того, должен ли контакт с врагом вызывать отбрасывания Дрея. Позднее этот же сценарий мы применим к оружию Дрея, чтобы определить эффект воздействия оружия на врагов.

1. Создайте в папке __Scripts новый сценарий на C# с именем DamageEffect.
2. Подключите его к игровому объекту Skeletos в иерархии.
3. Откройте сценарий DamageEffect в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DamageEffect : MonoBehaviour {
    [Header("Set in Inspector")]
    public int      damage = 1;
    public bool     knockback = true;
}
```

4. Сохраните сценарий DamageEffect в MonoDevelop и вернитесь в Unity. Теперь в компоненте DamageEffect (Script) объекта Skeletos, в инспекторе, должны появиться два поля.

По умолчанию величина наносимого ущерба равна 1, что равноценно половине одного круга на индикаторе уровня здоровья. Значения по умолчанию подобраны для скелетов, поэтому вам не придется ничего менять в инспекторе.

Изменения в классе Dray

Нам также нужно изменить класс Dray и использовать в нем сценарий DamageEffect, только что подключенный к объекту Skeletos.

1. Откройте сценарий Dray в MonoDevelop и добавьте следующие строки, выделенные жирным.

```
public class Dray : MonoBehaviour, IFacingMover, IKeyMaster {
    public enum eMode { idle, move, attack, transition, knockback }        // a

    [Header("Set in Inspector")]
    ...
    public float            attackDelay = 0.5f;     // Задержка между атаками
    public float            transitionDelay = 0.5f; // Задержка перехода между
                                                    // комнатами
    public int              maxHealth = 10;
    public float            knockbackSpeed = 10;                            // b
    public float            knockbackDuration = 0.25f;
    public float            invincibleDuration = 0.5f;

    [Header("Set Dynamically")]
    ...
    public int              numKeys = 0;
    public bool             invincible = false;                             // c

    [SerializeField]
    private int             _health;
    ...

    private float           transitionDone = 0;
    private Vector2         transitionPos;
    private float           knockbackDone = 0;                              // d
    private float           invincibleDone = 0;
    private Vector3         knockbackVel;

    private SpriteRenderer  sRend;                                          // e
    private Rigidbody       rigid;
    ...
    void Awake () {
        sRend = GetComponent<SpriteRenderer>();                             // e
        rigid = GetComponent<Rigidbody>();
        ...
    }

    void Update () {
```

```
        // Проверить состояние неуязвимости и необходимость выполнить отбрасывание
        if (invincible && Time.time > invincibleDone) invincible = false;  // f
        sRend.color = invincible ? Color.red : Color.white;
        if ( mode == eMode.knockback ) {
            rigid.velocity = knockbackVel;
            if (Time.time < knockbackDone) return;
        }

        if ( mode == eMode.transition ) { ... }
        ...
    }

    void LateUpdate() { ... }

    void OnCollisionEnter( Collision coll ) {
        if (invincible) return; // Выйти, если Дрей пока неуязвим          // g
        DamageEffect dEf = coll.gameObject.GetComponent<DamageEffect>();
        if (dEf == null) return; // Если компонент DamageEffect отсутствует - выйти

        health -= dEf.damage;// Вычесть величину ущерба из уровня здоровья // h
        invincible = true; // Сделать Дрея неуязвимым
        invincibleDone = Time.time + invincibleDuration;

        if (dEf.knockback) { // Выполнить отбрасывание                     // i
            // Определить направление отбрасывания
            Vector3 delta = transform.position - coll.transform.position;
            if (Mathf.Abs(delta.x) >= Mathf.Abs(delta.y)) {
                // Отбрасывание по горизонтали
                delta.x = (delta.x > 0) ? 1 : -1;
                delta.y = 0;
            } else {
                // Отбрасывание по вертикали
                delta.x = 0;
                delta.y = (delta.y > 0) ? 1 : -1;
            }

            // Применить скорость отскока к компоненту Rigidbody
            knockbackVel = delta * knockbackSpeed;
            rigid.velocity = knockbackVel;

            // Установить режим knockback и время прекращения отбрасывания
            mode = eMode.knockback;
            knockbackDone = Time.time + knockbackDuration;
        }
    }

    // Реализация интерфейса IFacingMover
    public int GetFacing() { ... }
    ...
}
```

a. В перечисление `eMode` добавлен новый режим `knockback`.

b. Переменные `knockbackSpeed`, `knockbackDuration` и `invincibleDuration` доступны для правки в инспекторе.

c. Переменная `public bool invincible` хранит значение `true` в период, когда Дрей невосприимчив к повреждениям. Я выбрал для этой цели логический флаг вместо создания отдельного режима в `eMode`, потому что Дрей может оставаться неуязвимым, находясь в разных режимах (если бы Дрей был неуязвим только во время выполнения отбрасывания, этот флаг был бы не нужен).

d. Несколько новых скрытых полей, необходимых для реализации отскока и неуязвимости.

e. Чтобы показать игроку, что Дрей получил ранение и пребывает в состоянии неуязвимости, герой на экране будет окрашиваться в красный цвет на время, пока он неуязвим. Для этого нам понадобится ссылка на компонент **Sprite Renderer** объекта **Dray**.

f. В начало метода `Update()` добавлен новый код, проверяющий окончание режима отбрасывания или состояния неуязвимости. Если Дрей неуязвим, `sRend` окрашивается в красный цвет. Если Дрей отбрасывается, свойству `rigid.velocity` присваивается скорость отбрасывания `knockbackVel`.

g. Метод `OnCollisionEnter()` вызывается движком Unity всякий раз, когда объект **Dray** сталкивается с коллайдером другого игрового объекта.

   Метод немедленно возвращает управление, если Дрей находится в состоянии неуязвимости или объект, с которым он столкнулся, не имеет компонента **DamageEffect** (например, Дрей часто будет сталкиваться со стенами, но стены не наносят ранений). Любые сценарии, подключенные к одному и тому же игровому объекту **Dray**, могут иметь свои методы `OnCollisionEnter()`, и Unity будет вызывать каждый из них.

h. Свойство `health` уменьшается на величину переменной `damage` из сценария **DamageEffect**, и на короткое время Дрей становится неуязвимым.

i. Если объект, с которым столкнулся Дрей, вызывает отбрасывание, выполняется содержимое этой условной инструкции `if`. Здесь определяется разность позиций Дрея и игрового объекта, с которым он столкнулся, и по разности вычисляется направление отскока — по вертикали или по горизонтали — и скорость `knockbackVelocity`. После этого Дрей переводится в режим отскока на период, пока `Time.time` не превысит `knockbackDone`.

2. Сохраните все сценарии в MonoDevelop, вернитесь в Unity и щелкните на кнопке **Play** (Играть).

Если теперь Дрей столкнется со скелетом, он получит повреждение, отскочит назад и станет неуязвимым. Чтобы протестировать неуязвимость, выберите объект **Dray** в иерархии и в инспекторе увеличьте до 10 секунд значение поля **invincibleDuration** компонента **Dray (Script)**. Затем попробуйте столкнуться со скелетом несколько раз.

Мы планируем использовать `OnCollisionEnter()` для учета ущерба, наносимого Дрею, и `OnTriggerEnter()` — для учета ущерба, наносимого скелетам и другим врагам. Это возможно, потому что все виды оружия Дрея будут иметь установлен-

ный флаг isTrigger, а различия между триггерами и коллайдерами позволят нам использовать сценарий DamageEffects для определения величины ущерба, наносимого и Дрею, и врагам.

Реализация нанесения урона врагам

Сейчас Дрей уже может взмахнуть мечом, атакуя врага; теперь пришло время придать мечу разящую силу.

1. Выберите объект Sword в иерархии (вложен в объект SwordController внутри объекта Dray).
2. Подключите к нему сценарий DamageEffect.
   - В поле damage компонента DamageEffect (Script) введите число 2. Меч должен быть достаточно мощным оружием.
3. Сохраните сцену.

Изменения в классе Enemy

Учитывая широкое использование интерфейсов в этой главе, можно создать интерфейс `IDamageable` и реализовать его в сценариях Dray и Enemy, а также в Damage, который затем подключить к обоим видам игровых объектов (как мы поступили со сценарием GridMove). Но я решил пойти другим путем по двум важным причинам:

- Столкновение Дрея с врагами обрабатывается в методе `OnCollisionEnter()`, тогда как столкновение врагов с мечом Дрея обрабатывается в методе `OnTriggerEnter()` (потому что коллайдер объекта Sword является триггером).
- Все враги являются подклассами `Enemy`, поэтому необходимый код достаточно добавить только в этот базовый класс.

1. Откройте сценарий Enemy в MonoDevelop и добавьте следующие строки:

```
public class Enemy : MonoBehaviour {
    ...
    [Header("Set in Inspector: Enemy")]
    public float               maxHealth = 1;
    public float               knockbackSpeed = 10;                    // a
    public float               knockbackDuration = 0.25f;
    public float               invincibleDuration = 0.5f;

    [Header("Set Dynamically: Enemy")]
    public float               health;
    public bool                invincible = false;                     // a
    public bool                knockback = false;

    private float              invincibleDone = 0;                     // a
    private float              knockbackDone = 0;
    private Vector3            knockbackVel;
```

```
    ...

    protected virtual void Awake() { ... }

    protected virtual void Update() {                                    // b
        // Проверить состояние неуязвимости и необходимость выполнить отскок
        if (invincible && Time.time > invincibleDone) invincible = false;
        sRend.color = invincible ? Color.red : Color.white;
        if ( knockback ) {
            rigid.velocity = knockbackVel;
            if (Time.time < knockbackDone) return;
        }

        anim.speed = 1;                                                  // c
        knockback = false;
    }

    void OnTriggerEnter( Collider colld ) {                              // d
        if (invincible) return; // Выйти, если Дрей пока неуязвим
        DamageEffect dEf = colld.gameObject.GetComponent<DamageEffect>();
        if (dEf == null) return; // Если компонент DamageEffect отсутствует - выйти

        health -= dEf.damage; // Вычесть величину ущерба из уровня здоровья
        if (health <= 0) Die();                                          // e
        invincible = true; // Сделать Дрея неуязвимым
        invincibleDone = Time.time + invincibleDuration;

        if (dEf.knockback) { // Выполнить отбрасывание
            // Определить направление отскока
            Vector3 delta = transform.position - colld.transform.root.position;
            if (Mathf.Abs(delta.x) >= Mathf.Abs(delta.y)) {
                // Отбрасывание по горизонтали
                delta.x = (delta.x > 0) ? 1 : -1;
                delta.y = 0;
            } else {
                // Отбрасывание по вертикали
                delta.x = 0;
                delta.y = (delta.y > 0) ? 1 : -1;
            }

            // Применить скорость отбрасывания к компоненту Rigidbody
            knockbackVel = delta * knockbackSpeed;
            rigid.velocity = knockbackVel;

            // Установить режим knockback и время прекращения отбрасывания
            knockback = true;
            knockbackDone = Time.time + knockbackDuration;
            anim.speed = 0;
        }
    }

    void Die() {                                                         // f
        Destroy(gameObject);
    }
}
```

a. В класс Enemy добавляются почти те же поля и изменения, что недавно были добавлены в класс Dray. Единственное отличие: признак отскока knockback здесь реализован как логический флаг, тогда как в сценарии Dray это был один из режимов в перечислении Dray.eMode.

b. Объявление метода Update() защищенным и виртуальным (protected virtual) позволит переопределить его в подклассах, таких как Skeletos. Мы сделаем это в следующем листинге.

c. Эти две строки выполняются, только когда истекает время выполнения отскока.

d. Для описания реакции на столкновение используется метод OnTriggerEnter(), потому что коллайдер меча Дрея настроен как триггер. Обратите внимание, что методу OnTriggerEnter() передается Collider, а не Collision. В остальном этот метод очень похож на аналогичный метод в классе Dray.

e. Если уровень здоровья врага уменьшается до нуля или ниже, вызывается новый метод Die().

f. Прямо сейчас метод Die() делает не так много, но позднее мы дополним его, заставив врага выбрасывать свои предметы на месте гибели.

2. Сохраните сценарий Enemy.

3. Откройте сценарий Skeletos в MonoDevelop и внесите следующие небольшие изменения:

```
public class Skeletos : Enemy, IFacingMover {
    ...
    protected override void Awake () { ... }

    override protected void Update () {                                    // a
        base.Update();
        if (knockback) return;

        if (Time.time >= timeNextDecision) {
            DecideDirection();
        }

        // Поле rigid унаследовано от класса Enemy и инициализируется
        // в Enemy.Awake()
        rigid.velocity = directions[facing] * speed;
    }
    ...
}
```

a. Добавьте спецификаторы override protected в начало объявления метода Update().

Первая строка в методе Update() вызывает метод Enemy.Update() базового класса. Если этот экземпляр Skeletos выполняет отскок назад, метод завершается сразу

после вызова `base.Update()`, чтобы предотвратить изменение направления или скорости движения скелета, пока тот не завершит отскок.

4. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.
5. Выберите объект Skeletos в иерархии и в инспекторе введите значение 4 в поле maxHealth компонента Skeletos (Script). В результате, чтобы убить скелета, Дрею придется нанести два удара мечом (каждый удар которым уменьшает уровень здоровья врага на 2).
6. Сохраните сцену и щелкните на кнопке Play (Играть).

Теперь Дрей может атаковать скелетов, наносить им ущерб и заставлять отскакивать назад.

Сбор предметов

Теперь, когда Дрей получил способность уничтожать врагов, можно заняться сбором ключей и аптечек для увеличения уровня здоровья. Начнем с ключей.

1. Перетащите спрайт Key из папки _Images в панели Project (Проект) в панель Hierarchy (Иерархия). В результате будет создан игровой объект с компонентом Sprite Renderer, отображающим спрайт Key.
2. Настройте параметры объекта Key в инспекторе:
   - Transform: P:[28, 3, 0].
   - Sprite Renderer: Sorting Layer: Items.
3. Добавьте в объект Key коробчатый коллайдер Box Collider.
   - Установите флажок Is Trigger в настройках этого коллайдера.
4. Сохраните сцену.
5. Создайте в папке __Scripts новый сценарий на C# с именем PickUp.
6. Подключите его к игровому объекту Key в иерархии.
7. Откройте сценарий PickUp в MonoDevelop и введите следующий код:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PickUp : MonoBehaviour {
    public enum eType { key, health, grappler }

    public static float     COLLIDER_DELAY = 0.5f;

    [Header("Set in Inspector")]
    public eType            itemType;
```

```
    // Awake() и Activate() деактивируют коллайдер на 0,5 секунды
    void Awake() {
        GetComponent<Collider>().enabled = false;
        Invoke("Activate", COLLIDER_DELAY);
    }

    void Activate() {
        GetComponent<Collider>().enabled = true;
    }
}
```

8. Сохраните сценарий PickUp.

9. Откройте сценарий Dray и добавьте следующие строки, выделенные жирным:

```
public class Dray : MonoBehaviour, IFacingMover, IKeyMaster {

    ...

    void OnCollisionEnter( Collision coll ) { ... }

    void OnTriggerEnter( Collider colld ) {
        PickUp pup = colld.GetComponent<PickUp>();                          // a
        if (pup == null) return;

        switch (pup.itemType) {
            case PickUp.eType.health:
                health = Mathf.Min( health+2, maxHealth );
                break;

            case PickUp.eType.key:
                keyCount++;
                break;
        }

        Destroy( colld.gameObject );
    }

    // Реализация интерфейса IFacingMover
    public int GetFacing() { ... }
    ...
}
```

   a. Если игровой объект, столкнувшийся с этим триггером, не имеет компонента сценария PickUp, этот метод завершается, не выполняя никаких действий.

10. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.

11. Выберите объект Key в иерархии, а в инспекторе выберите значение Key в поле itemType компонента PickUp (Script).

12. Сохраните сцену и щелкните на кнопке Play (Играть).

Теперь вы сможете подобрать ключ, увидеть, как увеличится счетчик ключей в пользовательском интерфейсе, и затем воспользоваться этим ключом, чтобы открыть первую дверь.

Бросание предметов врагами на месте гибели

Наша следующая задача — заставить одних врагов ронять на месте своей гибели ключи, а других — аптечки.

Бросание ключей

Выполните следующие шаги, чтобы заставить врагов оставлять ключи:

1. Откройте сценарий Enemy в MonoDevelop и добавьте следующие строки:

```
public class Enemy : MonoBehaviour {
    ...
    [Header("Set in Inspector: Enemy")]
    ...
    public float                invincibleDuration = 0.5f;
    public GameObject           guaranteedItemDrop = null;

    [Header("Set Dynamically: Enemy")]
    ...
    void Die() {
        GameObject go;
        if ( guaranteedItemDrop != null ) {
            go = Instantiate<GameObject>( guaranteedItemDrop );
            go.transform.position = transform.position;
        }
        Destroy(gameObject);
    }
}
```

2. Сохраните сценарий Enemy и вернитесь в Unity.
3. Создайте шаблон Key, перетащив его из панели Hierarchy (Иерархия) в папку _Prefab, в панели Project (Проект).
4. Выберите Skeletos в иерархии и в инспекторе выберите шаблон Key (из папки _Prefabs) в поле guaranteedItemDrop компонента Skeletos (Script).
5. Сохраните сцену и щелкните на кнопке Play (Играть).

Если теперь убить скелета, он должен выронить на месте гибели ключ, который вы сможете подобрать.

Бросание случайных предметов

Мы также должны реализовать бросание врагами на месте гибели случайных предметов. Если поле `guaranteedItemDrop` не имеет значения[1], тогда нужно выбрать слу-

[1] В отличие от полей некоторых типов, таких как `Vector3`, которым в инспекторе нельзя назначить пустое значение, в поле guaranteedItemDrop типа `GameObject` можно выбрать None (GameObject) — значение, которое Unity распознает как пустую ссылку `null`.

чайный элемент из массива доступных предметов, который также будет содержать пустые элементы, чтобы не гарантировать бросание предмета.

Сначала подготовим предметы.

Создание аптечки

Выполните следующие шаги, чтобы создать аптечку:

1. В панели Project (Проект) выберите спрайты Health_2 и Health_3 в текстуре Health, которая находится в папке _Images.
2. Перетащите их в панель Hierarchy (Иерархия), чтобы создать новый игровой объект и анимацию.
3. Сохраните только что созданную анимацию с именем Health.anim в папке *_Animations*.
4. Выберите в иерархии игровой объект Health_2 и в инспекторе настройте следующие поля:
   - **Name:** измените имя Health_2 на Health.
   - **Transform:** P:[28, 7, 0].
   - **Sprite Renderer:** в раскрывающемся списке Sorting Layer выберите пункт Items.
   - Добавьте в объект Health компонент Box Collider.
   - **Box Collider:** установите флажок Is Trigger.
   - Добавьте в объект Health компонент PickUp (Script).
   - **PickUp (Script):** выберите значение Health в поле itemType.
5. Оставив выделенным игровой объект Health в иерархии, откройте панель Animation (Анимация), выбрав в меню пункт Window > Animation (Окно > Анимация), и в поле Samples для анимации Health введите число 4 (это заставит элемент Health мигать с умеренной частотой — четыре раза в секунду).
6. Перетащите игровой объект Health из иерархии в папку _Prefabs в панели Project (Проект), чтобы создать шаблон Health.
7. Сохраните сцену и щелкните на кнопке Play (Играть).
8. Заставьте Дрея столкнуться со скелетом пару раз, чтобы получить повреждения. Затем подберите элемент аптечки Health, и вы увидите, как уровень здоровья немного увеличился.

Реализация бросания случайных предметов

Выполните следующие шаги, чтобы реализовать бросание случайных предметов.

1. Откройте сценарий Enemy в MonoDevelop и добавьте следующие строки, выделенные жирным:

```
public class Enemy : MonoBehaviour {
    ...
    [Header("Set in Inspector: Enemy")]
    ...
    public float                invincibleDuration = 0.5f;
    public GameObject[]         randomItemDrops;                    // a
    public GameObject           guaranteedItemDrop = null;
    ...
    void OnTriggerEnter( Collider colld ) { ... }

    void Die() {
        GameObject go;
        if ( guaranteedItemDrop != null ) {
            go = Instantiate<GameObject>( guaranteedItemDrop );
            go.transform.position = transform.position;
        } else if (randomItemDrops.Length > 0) {                    // b
            int n = Random.Range( 0, randomItemDrops.Length );
            GameObject prefab = randomItemDrops[n];
            if (prefab != null) {
                go = Instantiate<GameObject>( prefab );
                go.transform.position = transform.position;
            }
        }
        Destroy(gameObject);
    }
}
```

a. Массив `randomItemDrops` может хранить любое количество возможных предметов (в том числе и пустые значения None (GameObject) — `null`), которые враг может оставить на месте гибели.

b. Если поле `guaranteedItemDrop` не содержит ссылки на предмет, гарантированно оставляемый данным врагом, и массив `randomItemDrops` не пуст, мы выбираем случайный элемент массива и присваиваем его переменной `prefab`. Если `prefab` содержит действительную ссылку (не `null`), создается экземпляр шаблона, на который ссылается `prefab`.

2. Сохраните сценарий Enemy и вернитесь в Unity.
3. Выберите в иерархии игровой объект Skeletos.
   - Удалите значение в поле guaranteedItemDrop, чтобы в нем осталось None (Game Object).
   - Распахните поле randomItemDrops компонента Skeletos (Script), щелкнув на пиктограмме с треугольником.
   - В разделе randomItemDrops введите в поле Size число 1.
   - В поле Element 0 выберите шаблон Health (из папки _Prefabs).
4. Преобразуйте объект Skeletos в шаблон, перетащив его из иерархии в папку _Prefabs в панели Project (Проект).
5. Сохраните сцену и щелкните на кнопке Play (Играть).

Теперь, когда Дрей убьет скелета, тот оставит на месте гибели элемент аптечки Health.

6. Выберите шаблон Skeletos в папке _Prefabs, в панели Project (Проект) (не в иерархии).
   - В разделе randomItemDrops введите в поле Size число 2.
   - Удалите значение Health в поле Element 1.
   - Введите в поле Size число 3. После этого массив randomItemDrops получит два пустых элемента (со значением None (Game Object)), и скелеты (с незаполненным полем guaranteedItemDrop) будут оставлять на месте гибели аптечки Health с вероятностью 1/3.

Эти изменения в шаблоне Skeletos должны автоматически отразиться на экземпляре Skeletos в иерархии.

Реализация крюка

Последний элемент, который мы должны реализовать, — это крюк, с помощью которого игрок сможет перебираться через прежде непроходимые красные плитки.

1. В пакете, импортированном в начале главы, в папке _Prefabs имеется шаблон Grappler. Выберите этот шаблон в папке _Prefabs, в панели Project (Проект).
   - **Sprite Renderer:** выберите в раскрывающемся списке Sorting Layer пункт Items.
2. Перетащите Grappler на игровой объект Dray в иерархии, чтобы сделать его дочерним по отношению к Dray (и братским по отношению к SwordController).
3. Создайте в папке __Scripts новый сценарий на C# с именем Grapple. (Сценарий получил имя Grapple, описывающее действие, а не предмет, чтобы его проще было отличить от предмета Grappler; разница в именах также обусловлена тем, что сценарий Grapple будет подключаться к объекту Dray, а не к Grappler.)
4. Подключите сценарий Grapple к объекту Dray.
5. Откройте его в MonoDevelop и введите следующий код. Этот сценарий немного длиннее всех остальных в этой главе.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Grapple : MonoBehaviour {
    public enum eMode { none, gOut, gInMiss, gInHit }                    // a

    [Header("Set in Inspector")]
    public float            grappleSpd = 10;
    public float            grappleLength = 7;
    public float            grappleInLength = 0.5f;
```

```
public int              unsafeTileHealthPenalty = 2;
public TextAsset        mapGrappleable;

[Header("Set Dynamically")]
public eMode            mode = eMode.none;
// Номера плиток, на которые можно забросить крюк
public List<int>        grappleTiles;                                    // b
public List<int>        unsafeTiles;

private Dray            dray;
private Rigidbody       rigid;
private Animator        anim;
private Collider        drayColld;

private GameObject      grapHead;                                        // c
private LineRenderer    grapLine;
private Vector3         p0, p1;
private int             facing;

private Vector3[] directions = new Vector3[] {
    Vector3.right, Vector3.up, Vector3.left, Vector3.down };

void Awake() {
    string gTiles = mapGrappleable.text;                                 // d
    gTiles = Utils.RemoveLineEndings( gTiles );
    grappleTiles = new List<int>();
    unsafeTiles = new List<int>();
    for (int i=0; i<gTiles.Length; i++) {
        switch (gTiles[i]) {
            case 'S':
                grappleTiles.Add(i);
                break;

            case 'X':
                unsafeTiles.Add(i);
                break;
        }
    }

    dray = GetComponent<Dray>();
    rigid = GetComponent<Rigidbody>();
    anim = GetComponent<Animator>();
    drayColld = GetComponent<Collider>();

    Transform trans = transform.Find("Grappler");
    grapHead = trans.gameObject;
    grapLine = grapHead.GetComponent<LineRenderer>();
    grapHead.SetActive(false);
}

void Update () {
    if (!dray.hasGrappler) return;                                       // e

    switch (mode) {
        case eMode.none:
```

```
            // Если нажата клавиша применения крюка
            if ( Input.GetKeyDown(KeyCode.X) ) {
                StartGrapple();
            }
            break;
    }
}

void StartGrapple() {                                                    // f
    facing = dray.GetFacing();
    dray.enabled = false;                                                // g
    anim.CrossFade( "Dray_Attack_"+facing, 0 );
    drayColld.enabled = false;
    rigid.velocity = Vector3.zero;

    grapHead.SetActive(true);

    p0 = transform.position + ( directions[facing] * 0.5f );
    p1 = p0;
    grapHead.transform.position = p1;
    grapHead.transform.rotation = Quaternion.Euler(0,0,90*facing);

    grapLine.positionCount = 2;                                          // h
    grapLine.SetPosition(0,p0);
    grapLine.SetPosition(1,p1);
    mode = eMode.gOut;
}

void FixedUpdate() {
    switch (mode) {
        case eMode.gOut: // Крюк брошен                                  // i
            p1 += directions[facing] * grappleSpd * Time.fixedDeltaTime;
            grapHead.transform.position = p1;
            grapLine.SetPosition(1,p1);

            // Проверить, попал ли крюк куда-нибудь
            int tileNum = TileCamera.GET_MAP(p1.x,p1.y);
            if ( grappleTiles.IndexOf( tileNum ) != -1 ) {
                // Крюк попал на плитку, за которую можно зацепиться!
                mode = eMode.gInHit;
                break;
            }
            if ( (p1-p0).magnitude >= grappleLength ) {
                // Крюк улетел на всю длину веревки, но никуда не попал
                mode = eMode.gInMiss;
            }
            break;

        case eMode.gInMiss: // Игрок промахнулся, вернуть крюк на удвоенной
                            // скорости // j
            p1 -= directions[facing] * 2 * grappleSpd * Time.fixedDeltaTime;
            if ( Vector3.Dot( (p1-p0), directions[facing] ) > 0 ) {
                // Крюк все еще перед Дреем
                grapHead.transform.position = p1;
                grapLine.SetPosition(1,p1);
```

```
            } else {
                StopGrapple();
            }
            break;

        case eMode.gInHit: // Крюк зацепился, поднять Дрея на стену    // k
            float dist = grappleInLength + grappleSpd * Time.fixedDeltaTime;
            if (dist > (p1-p0).magnitude) {
                p0 = p1 - ( directions[facing] * grappleInLength );
                transform.position = p0;
                StopGrapple();
                break;
            }
            p0 += directions[facing] * grappleSpd * Time.fixedDeltaTime;
            transform.position = p0;
            grapLine.SetPosition(0,p0);
            grapHead.transform.position = p1;
            break;
    }
}

void StopGrapple() {                                                    // l
    dray.enabled = true;
    drayColld.enabled = true;

    // Проверить безопасность плитки
    int tileNum = TileCamera.GET_MAP(p0.x,p0.y);
    if (mode == eMode.gInHit && unsafeTiles.IndexOf(tileNum) != -1) {
        // Дрей попал на небезопасную плитку
        dray.ResetInRoom( unsafeTileHealthPenalty );
    }

    grapHead.SetActive(false);

    mode = eMode.none;
}

void OnTriggerEnter( Collider colld ) {                                 // m
    Enemy e = colld.GetComponent<Enemy>();
    if (e == null) return;

    mode = eMode.gInMiss;
}
}
```

a. Крюк может пребывать в четырех состояниях:

- `none`: неактивный;
- `gOut`: крюк брошен и летит вперед;
- `gInMiss`: крюк никуда не попал и возвращается назад; Дрей остается на месте;
- `gInHit`: крюк зацепился за какую-то плитку и теперь Дрей подтягивается к ней на веревке.

b. Список `grappleTiles` хранит индексы плиток, за которые крюк может зацепиться и перейти в состояние `gInHit`. Список `unsafeTiles` хранит индексы плиток, небезопасных для Дрея, на которые тот может опуститься, подтянувшись на веревке после `gInHit`; это важно для комнат, где Дрей, подтянувшись на веревке, может оказаться на красной плитке.

c. `grapHead` — это ссылка на игровой объект, представляющий начало крюка. `grapLine` — это ссылка на компонент `LineRenderer` объекта `Grappler`.

d. В методе `Awake()` выполняется чтение текстового файла mapGrappleable и заполняются списки `grappleTiles` и `unsafeTiles`. Кроме того, этот метод отыскивает ссылки на различные компоненты и кэширует их.

e. Если у Дрея есть крюк и тот находится в состоянии `none`, метод `Update()` проверяет нажатие клавиши броска крюка (X). Вызов `dray.hasGrappler` подсвечен красным, потому что мы пока не добавили в класс `Dray` код, использующий объект Grappler.

f. `StartGrapple()` выполняет бросок крюка из позиции Дрея в направлении его взгляда.

g. Кроме того, `StartGrapple()` деактивирует компонент сценария Dray, чтобы игрок не мог управлять персонажем, пока крюк не завершит полет. Также в особое состояние переводится компонент Animator и отключается коллайдер объекта Dray.

h. Здесь выполняется настройка компонента LineRenderer, на который ссылается `grapLine`. Крюк всегда летит по прямой, поэтому для отображения веревки компоненту LineRenderer требуются только две точки.

i. Когда Дрей бросает крюк, тот должен удаляться с постоянной скоростью, поэтому использован метод `FixedUpdate()`. `grapHead` перемещается вперед, и вслед за ним перемещается точка `p1` компонента LineRenderer. Чтобы определить попадание крюка в плитку, вместо коллайдера используется непосредственная проверка вхождения плитки из `TileCamera.MAP` в список `grappleTiles`. Если крюк попал в плитку, присутствующую в списке `grappleTile`, класс `Grapple` переходит в состояние `gInHit`; если крюк улетел слишком далеко, но так и не попал в плитку, за которую можно зацепиться, происходит переход в состояние `gInMiss`.

j. В состоянии `gInMiss` крюк возвращается к герою на удвоенной скорости. Проверка скалярного произведения позволяет определить, находится ли точка `p1` впереди Дрея (дополнительную информацию об использовании операции скалярного произведения вы найдете в приложении Б «Полезные идеи»).

k. В состоянии `gInHit` Дрей начинает подтягиваться на веревке в направлении `grapHead`. Так как компонент коллайдера в объекте Dray в это время отключен, герой беспрепятственно пересекает любые, иначе непреодолимые, красные плитки.

l. `StopGrapple()` вновь активирует сценарий `dray` и коллайдер `drayColld`. Затем, если сценарий Grapple находился в состоянии `gInHit`, проверяется позиция Дрея, и если тот оказался на небезопасной плитке, он получает ранение и переносится в позицию двери, через которую вошел в комнату. Метод `Dray.ResetInRoom()`, выполняющий это действие, пока не написан.

m. Если `grapHead` столкнулся с врагом, метод `OnTriggerEnter()` заставляет крюк вернуться к герою, устанавливая состояние `gInMiss`. Ниже мы подключим к объекту Grappler сценарий DamageEffect, чтобы тот мог наносить урон врагам.

6. Сохраните сценарий Grapple.

Добавление возможности пользоваться крюком

Выполните следующие шаги, чтобы дать Дрею возможность пользоваться крюком:

1. Откройте сценарий Dray в MonoDevelop и добавьте следующие строки, выделенные жирным:

```
public class Dray : MonoBehaviour, IFacingMover, IKeyMaster {
    ...
    [Header("Set Dynamically")]
    ...
    public bool             invincible = false;
    public bool             hasGrappler = false;
    public Vector3          lastSafeLoc;                                    // a
    public int              lastSafeFacing;

    [SerializeField]
    private int             _health;
    ...

    void Awake () {
        ...
        health = maxHealth;
        lastSafeLoc = transform.position; // Начальная позиция безопасна.
        lastSafeFacing = facing;
    }
    ...

    void LateUpdate() {
        ...
        // Проверить, можно ли выполнить переход в комнату rm
        if (rm.x >= 0 && rm.x <= InRoom.MAX_RM_X) {
            if (rm.y >=0 && rm.y <= InRoom.MAX_RM_Y) {
                roomNum = rm;
                transitionPos = InRoom.DOORS[ (doorNum+2) % 4 ];
                roomPos = transitionPos;
                lastSafeLoc = transform.position;                           // b
                lastSafeFacing = facing;
                mode = eMode.transition;
                transitionDone = Time.time + transitionDelay;
            }
```

```
        }
    }
    ...

    void OnTriggerEnter( Collider colld ) {
        ...
        switch (pup.itemType) {
            ...
            case PickUp.eType.key:
                keyCount++;
                break;

            case PickUp.eType.grappler:                                    // c
                hasGrappler = true;
                break;
        }
        ...
    }

    public void ResetInRoom(int healthLoss = 0) {                          // d
        transform.position = lastSafeLoc;
        facing = lastSafeFacing;
        health -= healthLoss;

        invincible = true; // Сделать Дрея неуязвимым
        invincibleDone = Time.time + invincibleDuration;
    }

    // Реализация интерфейса IFacingMover
    ...
}
```

a. `hasGrappler` получит значение `true`, когда Дрей приобретет крюк. `lastSafeLoc` и `lastSafeFacing` хранят местоположение и направление взгляда Дрея, когда тот последний раз вошел в комнату. Если, подтянувшись на веревке за крюком, Дрей окажется на небезопасной плитке, он переместится в эту позицию и развернется в этом направлении.

b. `lastSafeLocation` и `lastSafeFacing` получают новые значения всякий раз, когда Дрей входит в очередную комнату.

c. Когда Дрей подбирает крюк, переменной `hasGrappler` присваивается значение `true`.

d. Вызов `ResetInRoom()` перемещает Дрея в последнее безопасное местоположение и поворачивает его лицом в направлении, в котором он был повернут при входе в комнату.

Возможно, вы заметили, что в метод `Update()` класса `Dray` не было добавлено ничего для броска крюка, потому что эту функцию взял на себя метод `Update()` в сценарии Grapple. Хорошо это или плохо — спорный вопрос. С одной стороны, часто предпочтительнее сосредоточить в одном месте весь код, взаимодействующий с игроком; с другой, включив его в сценарий Grapple, мы получили возмож-

ность добавлять в будущем другие виды оружия и инструментов и привязывать их к клавишам, не меняя код взаимодействия с игроком в сценарии Dray. Эти соображения приобретают особую важность при переходе от прототипирования проекта к его разработке.

2. Сохраните все сценарии в MonoDevelop и вернитесь в Unity.
3. Выберите объект Dray в иерархии.
   - **Grapple (Script):** выберите в поле mapGrappleable текстовый файл DelverGrappleable из папки Resources в панели Project (Проект). Этот текстовый файл используется сценарием Grapple для заполнения списков `grappleTiles` и `unsafeTiles`.
4. Сохраните сцену.

Добавление возможности поражения врагов крюком

Если подключить сценарий DamageEffect к игровому объекту Grappler, он сможет наносить повреждения врагам. В результате Дрей получит слабое дистанционное оружие.

1. Выберите объект Grappler в иерархии (вложен в объект Dray).
2. Подключите к нему компонент DamageEffect (Script).
   - Введите в поле damage число 1.
   - Снимите флажок knockback.
3. Сохраните сцену.

Добавление возможности подобрать крюк

Дрей должен иметь возможность подобрать крюк в процессе исследования уровня. К счастью, в пакете, который мы импортировали в начале главы, уже имеется шаблон GrapplerPickUp, но нам нужно подключить к нему сценарий PickUp, чтобы его можно было подобрать.

1. Перетащите GrapplerPickUp из папки _Prefabs в панели Project (Проект) в панель Hierarchy (Иерархия).
2. Настройте его компонент Transform: P:[19, 3, 0], R:[0, 0, 0], S:[2, 2, 2].
3. Добавьте в экземпляр GrapplerPickUp компонент PickUp (Script) и в поле itemType этого компонента выберите значение Grappler.
4. Щелкните на кнопке Apply (Применить) в верхней части инспектора. В результате изменения, произведенные в экземпляре GrapplerPickUp, будут переписаны в шаблон GrapplerPickUp, находящийся в папке _Prefabs в панели Project (Проект).

5. Выберите шаблон GrapplerPickUp в панели Project (Проект) и проверьте, что изменения действительно попали в него.

Тестирование крюка

Для тестирования крюка выполните следующие шаги:

1. Щелкните на кнопке Play (Играть). Нажмите клавишу X на клавиатуре. В этот момент ничего не должно произойти.
2. Подберите крюк. В результате поле `hasGrappler` объекта `Dray` должно получить значение `true`.

   Теперь можно нажать клавишу X, чтобы бросить крюк. Он должен цепляться за стены и подтаскивать героя к ним. Также с помощью крюка можно подбирать предметы, такие как ключи и аптечки. Кроме того, им можно наносить повреждения скелетам — в два раза меньшие, чем мечом, — но при этом они не будут отскакивать назад.
3. Приостановите игру (щелкните на кнопке Pause (Пауза), вверху в окне Unity) и настройте в объекте Dray компонент Transform: P:[39.5, 40, 0]. В результате Дрей окажется прямо под комнатой с несколькими красными плитками рядом со стенами (в шее орла подземелья *Eagle*[1] из *The Legend of Zelda*).
4. Возобновите игру (щелкните на кнопке Pause (Пауза) еще раз) и пройдите вверх через дверь в комнату с красными плитками (переход установит переменные `lastSafeLocation` и `lastSafeFacing`).
5. Попробуйте подтягиваться к стенам с помощью крюка, чтобы в конце оказаться на красной плитке. Вы увидите, что уровень здоровья Дрея немного снижается и он возвращается на безопасное место рядом с дверью.
6. Снова приостановите игру, настройте в объекте Dray компонент Transform: P:[40, 49.5, 0] и возобновите игру.
7. Повернитесь вправо и запустите крюк в стену.

   Выражение `y - 0.25f` в методе `GET_MAP(float, float)` класса `TileCamera` (соответствующая строка в листинге сценария TileCamera, который приводится в разделе «Класс TileCamera — парсинг файлов с данными и спрайтами», отмечена комментарием `// g`) сделает это местоположение безопасным для Дрея (без вычитания 0.25f позиция была бы округлена и он оказался бы на красной плитке).
8. Остановите игру и сохраните сцену.

Покончив с тестированием крюка, перейдем к созданию другого подземелья.

[1] См. рис. 9.3. — *Примеч. пер.*

Реализация нового подземелья — Hat

Наша следующая цель — реализовать подземелье, служившее примером в главе 9 «Прототипирование на бумаге»[1]. В процессе ее достижения мы также реализуем способ внедрения врагов в файл DelverData.

Подготовка сцены

Подготовьте сцену, выполнив следующие шаги:

1. Выберите в меню пункт File > Save Scene As (Файл > Сохранить сцену как) и сохраните новую копию сцены с именем _Scene_Hat.
2. После сохранения копии сцены с другим именем среда Unity иногда оставляет активной прежнюю сцену. Взгляните на заголовок окна: в нем должен отображаться текст _Scene_Hat. Если это не так, щелкните дважды на _Scene_Hat в панели Project (Проект), чтобы открыть сцену.
3. Проверьте, чтобы вместе со сценой были созданы шаблоны Skeletos, Key и Health. Все они должны присутствовать в папке _Prefabs в панели Project (Проект). Если их там нет, перетащите одноименные объекты из панели Hierarchy (Иерархия) в папку _Prefabs.
4. Удалите игровые объекты Skeletos, Key, Health и GrapplerPickUp из иерархии. После этого в панели Hierarchy (Иерархия) на верхнем уровне должно остаться только шесть игровых объектов: Main Camera, Directional Light,[2] GUI Camera, Dray, Canvas и Event System (объекты Dray и Canvas имеют дочерние объекты).
5. Выберите Main Camera в иерархии.
   - **Transform:** P:[55.5, 5, –10].
   - **TileCamera (Script):** в поле mapData компонента TileCamera (Script) выберите текстовый файл DelverData_Hat из папки Resources в панели Project (Проект).
6. Выберите объект Dray в иерархии.
   - **Transform:** P:[55.5, 1, 0].
   - **Dray (Script):** снимите флажок hasGrappler.
7. Сохраните сцену и щелкните на кнопке Play (Играть).

Вы увидите, что игра загрузила совершенно новое подземелье. В первой комнате можно заметить изображение недоступного ключа. Переместившись в комнату

[1] См. рис. 9.4. — *Примеч. пер.*

[2] Иногда при создании нового проекта Unity не добавляет объект Directional Light в начальную сцену. Если у вас он отсутствует, добавьте его сами. Впрочем, присутствие или отсутствие источника направленного света Directional Light не влияет на работу шейдера Sprite Renderer в этом проекте (начиная с версии Unity 2017).

влево, вы увидите два изображения скелетов, но они не движутся. В данный момент они являются лишь особыми плитками, которые выглядят как враги и предметы. Нам нужно написать такой код, который превратит эти особые плитки в настоящие предметы, потому что это подземелье невозможно исследовать, не имя ключей и крюка, а без врагов исследование становится простым и неинтересным.

Замена плиток врагами и предметами

Добавим в сценарий TileCamera код, который будет превращать специальные плитки на карте во врагов, предметы и простые напольные плитки. Заглянув в изображение DelverTiles в папке Resources в панели Project (Проект), можно увидеть в нижней четверти плитки, которые выглядят как разные предметы, враги и враги с ключами. Это специальные плитки, которые мы заменим в окончательной карте обычными напольными плитками с предметами или врагами на них.

1. Откройте сценарий TileCamera в MonoDevelop и добавьте следующие строки, выделенные жирным:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[System.Serializable]
public class TileSwap {                                                    // a
    public int                  tileNum;
    public GameObject           swapPrefab;
    public GameObject           guaranteedItemDrop;
    public int                  overrideTileNum = -1;
}

public class TileCamera : MonoBehaviour {
    ...
    [Header("Set in Inspector")]
    ...
    public Tile                 tilePrefab;
    public int                  defaultTileNum;                            // b
    public List<TileSwap>       tileSwaps;                                 // c

    private Dictionary<int,TileSwap> tileSwapDict;                         // c
    private Transform           enemyAnchor, itemAnchor;

    void Awake() {
        COLLISIONS = Utils.RemoveLineEndings( mapCollisions.text );
        PrepareTileSwapDict();                                             // d
        enemyAnchor = (new GameObject("Enemy Anchor")).transform;
        itemAnchor = (new GameObject("Item Anchor")).transform;
        LoadMap();
    }

    public void LoadMap() {
        ...
```

```
        MAP = new int[W,H];
        for (int j=0; j<H; j++) {
            tileNums = lines[j].Split(' ');
            for (int i=0; i<W; i++) {
                if (tileNums[i] == "..") {
                    MAP[i,j] = 0;
                } else {
                    MAP[i,j] = int.Parse( tileNums[i], hexNum );
                }
                CheckTileSwaps(i,j);                                    // e
            }
        }
        ...
    }

    void ShowMap() { ... }

    void PrepareTileSwapDict() {                                        // d
        tileSwapDict = new Dictionary<int, TileSwap>();
        foreach (TileSwap ts in tileSwaps) {
            tileSwapDict.Add(ts.tileNum, ts);
        }
    }

    void CheckTileSwaps(int i, int j) {                                 // e
        int tNum = GET_MAP(i,j);
        if ( !tileSwapDict.ContainsKey(tNum) ) return;
        // Мы должны заменить плитку
        TileSwap ts = tileSwapDict[tNum];
        if (ts.swapPrefab != null) {                                    // f
            GameObject go = Instantiate(ts.swapPrefab);
            Enemy e = go.GetComponent<Enemy>();
            if (e != null) {
                go.transform.SetParent( enemyAnchor );
            } else {
                go.transform.SetParent( itemAnchor );
            }
            go.transform.position = new Vector3(i,j,0);
            if (ts.guaranteedItemDrop != null) {                        // g
                if (e != null) {
                    e.guaranteedItemDrop = ts.guaranteedItemDrop;
                }
            }
        }

        // Заменить другой плиткой
        if (ts.overrideTileNum == -1) {                                 // h
            SET_MAP( i, j, defaultTileNum );
        } else {
            SET_MAP( i, j, ts.overrideTileNum );
        }
    }
    ...
}
```

a. Сериализуемый класс `TileSwap` включает всю информацию, необходимую для замены особой плитки обычной напольной плиткой с врагом или предметом на ней.

   - `tileNum`: индекс особой плитки для замены.
   - `swapPrefab`: шаблон для создания врага или предмета из этой особой плитки
   - `guaranteedItemDrop`: некоторые особые плитки содержат информацию о том, что враг должен гарантированно оставлять ключ на месте гибели. Все, что указано в поле `guaranteedItemDrop`, будет сохранено в поле `guaranteedItemDrop` созданного экземпляра класса `Enemy`.
   - `overrideTileNum`: большинство особых плиток будет заменяться плиткой `defaultTileNum` (обычная напольная плитка). В это поле можно записать целочисленный индекс конкретной плитки, которая должна заменить особую плитку. Мы воспользуемся этим полем для замены особых плиток с врагами Spiker в правой верхней комнате подземелья красными плитками вместо обычных напольных.

b. `defaultTileNum` — это индекс напольной плитки, которая в большинстве случаев используется для замены особых плиток. В этой игре поле `defaultTileNum` получает значение 29, которое соответствует индексу спрайта DelverTiles_29 с изображением желтой напольной плитки.

c. Списки сериализуются по умолчанию, но словари — нет. Однако поиск в словарях осуществляется проще (их ключи организованы в виде хеш-таблиц, которые обеспечивают очень быстрый поиск). Мы будем вводить информацию `TileSwap` в список `tileSwaps`, а затем в методе `Awake()` вызывать метод `PrepareTileSwapDict()` для преобразования этого списка в словарь `tileSwapDict`.

d. `PrepareTileSwapDict()` выполняет обход всех элементов списка `tileSwaps` и добавляет их в словарь `tileSwapDict`, роль ключей в котором играют значения `tileNum` особых плиток.

e. `CheckTileSwaps()` принимает координаты плитки на карте, извлекает ее индекс из `MAP` и производит замену, если плитка включена в `tileSwapDict`. `CheckTileSwaps()` использует методы `GET_MAP()` и `SET_MAP()`, чтобы упростить отладку (всегда проще добавить точку останова в функцию доступа, такую как `SET_MAP()`, чем непосредственно следить за изменениями в поле).

f. Если `tileSwapDict` содержит элемент с ключом `tileNum (tNum)`, `CheckTileSwaps()` извлечет из `tileSwapDict` соответствующий экземпляр класса `TileSwap`. Затем, если он имеет непустое поле `swapPrefab`, то создаст экземпляр врага или предмета и поместит его в позицию текущей плитки.

g. Если экземпляр `TileSwap` имеет непустое поле `guaranteedItemDrop` и игровой объект, созданный из шаблона `swapPrefab`, имеет компонент `Enemy`, зна-

чение `guaranteedItemDrop` переписывается из экземпляра `TileSwap` в поле `guaranteedItemDrop` созданного экземпляра `Enemy`.

h. В заключение вносятся изменения в `TileCamera.MAP`, где прежняя особая плитка замещается плиткой с индексом `defaultTileNum` (желтая напольная плитка). Если `TileSwap ts` содержит в поле `overrideTileNum` значение, отличное от –1, тогда в `MAP` записывается это значение вместо значения по умолчанию.

2. Сохраните сценарий TileCamera и вернитесь в Unity.

3. Выберите главную камеру Main Camera в иерархии и выполните следующие настройки:

   - **TileCamera (Script):** введите в поле defaultTileNum число 29.
   - **TileCamera (Script):** в разделе tileSwaps введите в поле Size число 6.
   - **TileCamera (Script):** выполните настройки в разделе tileSwaps, как показано на рис. 35.10.

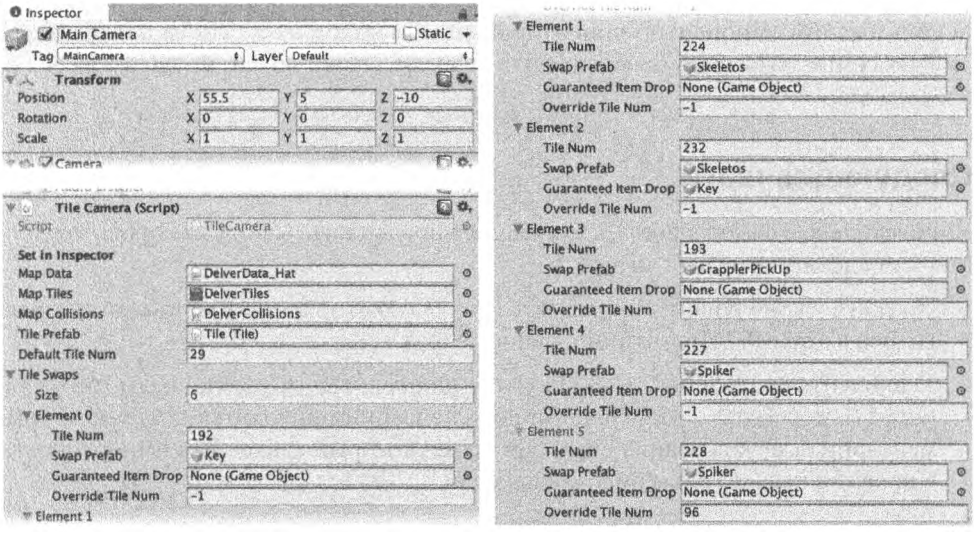


Рис. 35.10. Настройки списка tileSwaps в компоненте TileCamera (Script) главной камеры Main Camera

Элементы 4 и 5 в списке tileSwaps содержат шаблон Spiker, который был импортирован вместе с пакетом в начале главы. Прямо сейчас враги типа Spiker почти ничего не делают, но они демонстрируют использование поля `overrideTileNum` для замены особых плиток не плитками по умолчанию (в данном случае красной плиткой). Вы можете увидеть это в правой верхней комнате в подземелье *Hat*.

4. Сохраните сцену и щелкните на кнопке Play (Играть). Теперь вы сможете поиграть и пройти весь уровень от начала до конца, подбирая ключи (в основном оставляемые побежденными врагами) и крюк[1]!

Редактор уровней для игры Dungeon Delver

Если у вас появится желание создать свои уровни для *Dungeon Delver*, загрузите редактор уровней *Delver Level Editor* с веб-сайта книги http://book.prototools.net. Вы найдете его в разделе Chapter 35. В состав пакета с редактором *Delver Level Editor* включены инструкции по конструированию своих уровней и импортированию их в только что созданный прототип игры *Dungeon Delver*.

Итоги

Это был заключительный учебный пример! Работая над этим прототипом, вы познакомились с множеством новых понятий, таких как интерфейсы, и активнее использовали компонентно-ориентированный подход. Эта основа поможет вам создать самые разные приключенческие игры, и я советую вам продолжить ее исследование.

Следующие шаги

Вот несколько советов для тех, кто пожелает продолжить работу над проектом, они помогут сделать игру еще интереснее:

1. Создайте свои уровни! Попробуйте сначала создать их прототипы, как описывалось в главе 9, «Прототипирование на бумаге».
2. Реализуйте врага Spiker. Шаблон уже импортирован в проект, но его требуется немного доработать. Инструкции по реализации приводятся в длинном комментарии, в начале файла сценария Spiker, который также был импортирован в проект в начале главы вместе с пакетом.
3. Добавьте других врагов. В импортированном пакете есть еще несколько спрайтов с изображениями врагов. Попробуйте добавить новых врагов с новым поведением.
4. Добавьте в крюк возможность оглушать врагов при попадании в них. Это сделает его очень похожим на бумеранг в игре *The Legend of Zelda*.

[1] Если вам не хватит ключей, попробуйте уничтожить несколько скелетов — у некоторых из них есть ключи. Если уровень здоровья героя упадет слишком низко, не волнуйтесь; мы не реализовали окончание игры, если уровень здоровья Дрея упадет до нуля. Это один из аспектов, которые вы теперь сможете реализовать самостоятельно.

5. Также из игры *The Legend of Zelda* можно взять волшебный меч, который стреляет, когда герой имеет максимальный уровень здоровья. Для этого можно использовать второе изображение меча в файле Swords.
6. Спроектируйте и реализуйте новое оружие/предмет. Идею можно подсмотреть во множестве приключенческих игр. *Стреляющий крюк* из *The Legend of Zelda* всегда был одним из моих излюбленных предметов, именно поэтому я и включил его в этот прототип.
7. Сейчас крюк летит по прямой между двумя плитками и проверяет возможность столкновения только с одной из них. Вы можете изменить обработку состояния `eMode.gOut` в `Grapple:FixedUpdate()` и проверять возможность зацепления за любую плитку, лежащую на линии между начальной и конечной плитками (не забудьте проверить случаи броска по вертикали и по горизонтали).
8. Придумайте сами, что еще можно сделать! Вы прочитали целую книгу, добравшись сюда. Теперь ваши возможности безграничны!

Спасибо вам!

Еще раз спасибо, что прочитали эту книгу. Я искренне надеюсь, что она поможет вам достичь своей мечты.

Джереми Гибсон Бонд

Джереми Гибсон Бонд

Unity и C#. Геймдев от идеи до реализации

2-е издание

Перевел с английского А. Киселев

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*Д. Дрошнев*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России Изготовитель ООО «Прогресс книга»
Место нахождения и фактический адрес 194044, Россия, г Санкт-Петербург,
Б Сампсониевский пр , д 29А, пом. 52 Тел. +78127037373

Дата изготовления: 02 2019 Наименование книжная продукция Срок годности не ограничен

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58 11 12 —
Книги печатные профессиональные, технические и научные

Импортер в Беларусь ООО «ПИТЕР М», 220020, РБ, г Минск, ул Тимирязева, д 121/3, к 214, тел /факс 208 80 01

Подписано в печать 12 02 19 Формат 70×100/16 Бумага офсетная Усл п л 74,820 Тираж 1200 Заказ 1293

Отпечатано в АО «Первая Образцовая типография» Филиал «Чеховский Печатный Двор»
142300, Московская область, г Чехов, ул Полиграфистов, 1
Сайт www chpd ru, E-mail sales@chpd ru
тел 8(499) 270-73-59

ИЗДАТЕЛЬСКИЙ ДОМ
ПИТЕР®
WWW.PITER.COM

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

МЫ ПРЕДЛАГАЕМ:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт – гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами. Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePub или PDF – самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – *Анна Титова, (812) 703-73-73, titova@piter.com*
Москва – *Сергей Клебанов, (495) 234-38-15, klebanov@piter.com*

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы,(адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ВПЕРВЫЕ ВВЕДЕНИЕ В ГЕЙМДИЗАЙН, ПРОТОТИПИРОВАНИЕ И ГЕЙМДЕВ ОБЪЕДИНЕНЫ В ОДНУ КНИГУ

Если вы собираетесь заняться разработкой игр, то в первую очередь вам необходима информация о современных методах и профессиональных инструментах. Эти незаменимые знания можно получить в книге Джереми Гибсона Бонда. Кросс-платформенная разработка Unity позволяет создать игру, а затем с легкостью портировать куда угодно — от Windows и Linux до популярных мобильных платформ.

Часть I: Геймдев и прототипирование

- Обобщение 50-летней теории геймдева
- Мозговой штурм и итеративное проектирование
- Управление игровыми проектами и командами. Методы и инструменты
- Тестирование и обратная связь

Часть II: Цифровое прототипирование

- Знакомство с программированием, позволяющее освоить C# с нуля
- Подробное изучение возможностей Unity
- Отладка и методы решения часто встречающихся проблем

Часть III: Примеры прототипов и инструкции

- Семь прототипов игр: простая игра в жанре экшн, космический шутер, пасьянсы, игра в слова, приключение и др.
- Особенности компиляции игр для ПК, интернета или любой платформы, поддерживаемой Unity
- Сценарии и идеи, которые облегчат создание игр

Начните путешествие в мир игровой индустрии прямо сейчас! Заявите гордо: «Я — геймдизайнер». Ведь если вас услышат другие, то вы будете стараться соответствовать своим словам. А что дальше? Как стать геймдизайнером? Ответы на эти вопросы дает книга Джереми Гибсона Бонда — геймдизайнера и профессора, который больше 10 лет учит других создавать великолепные игры и делает это сам. Вы погрузитесь в увлекательный мир игровой индустрии, построите 8 реальных прототипов и овладеете всеми необходимыми инструментами.

«Книга сочетает в себе важные философские и практические понятия, необходимые всем, кто хочет стать настоящим разработчиком игр. Она познакомит вас с высокоуровневыми теориями проектирования, главными понятиями из мира разработки игр и основами программирования. Джереми использовал свой многолетний опыт, чтобы научить вас мыслить категориями геймдева и создавать игры».

Мишель Пун (Michelle Pun), игровой продюсер в Osmo. Ведущий геймдизайнер в Disney и Zynga

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг и интернет-магазин

instagram.com/**piterbooks**
youtube.com/**ThePiterBooks**
vk.com/**piterbooks**
facebook.com/**piterbooks**

ISBN: 978-5-4461-0715-5