Эта книга - 100%-ная гарантия вашей уверенной работы с Unity. Освой сам Unity "с нуля"!

Корнилов А. В.

UNITY

Полное руководство

Виртуальный DVD 10Гб, с Unity-проектами из книги www.nit.com.ru

книга + виртуальный DVD 10 Гб с Unity-проектами, примерами из книги и ассетами.

Все, что нужно для создания собственного проекта: от идеи до реализации

2-е издание

ПОЛНОЕ РУКОВОДСТВО

Корнилов А. В.

UNITY

Полное руководство

2-е издание

(обновленное и дополненное)

"Наука и Техника"

Санкт-Петербург

УДК 004.42 ББК 32.973

Корнилов А. В.

UNITY. Полное руководство. 2-е издание (+виртуальный DVD 10 Гб с Unity-проектами, примерами из книги и ассетами) — СПб.: Наука и Техника, 2021. — 496 с., ил.

ISBN 978-5-94387-721-6

В этой книге мы расскажем, как с использованием **Unity** (популярной межплатформенной среды разработки компьютерных игр) вы сможете САМИ создавать свои игры и трехмерные миры, причем без лишних затрат и профессиональных навыков программирования.

Книга поделена на три части. **Первая часть** посвящена изучению интерфейса и основных возможностей Unity. Мы поговорим о двух- и трехмерных проектах; рассмотрим ключевые особенности Unity; узнаем, как использовать ассеты; подробном изучим интерфейс Unity; узнаем об игровых объектах, сценах, камерах, источниках света; создадим свои первые Unity-проекты.

Во **второй части** мы поговорим о других важных частях Unity – о графике; о физике; рассмотрим основы скриптинга (написания сценариев – скриптов); узнаем как работать со звуком; как настроить навигацию в игре; как использовать анимацию и многое другое.

Ну и в заключительной, **третьей, части** полученных знаний из первых двух частей нам хватит для создания полноценной игры. Рассмотрен весь цикл создания игры – от этапа планирования игрового мира до настройки игрового интерфейса. Все этапы сопровождаются примерами программных кодов и скриншотами. Если вы уже сейчас хотите создавать свои игры – то эта книга определенно для вас! Читайте и творите!

Виртуальный DVD 10 Гб с Unity-проектами, примерами из книги и ассетами можно скачать на сайте издательства в разделе «Материалы к книгам».

ISBN 978-5-94387-721-6

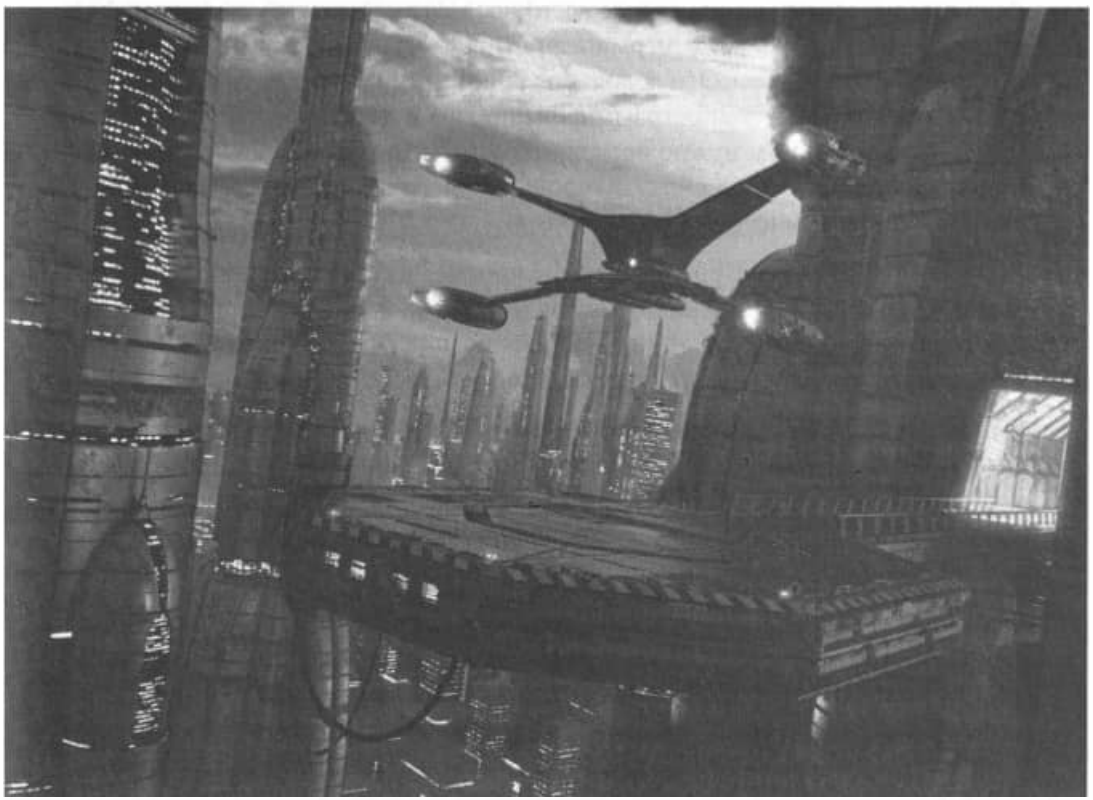
9 78- 5- 94387- 721- 6

Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru

Содержание

Введение



Почему именно Unity?

Если вы читаете эту книгу, скорее всего, вы мечтаете создать свою компьютерную игру. Раньше, когда игры были относительно простыми, разработка игры начиналась с нуля. Это довольно кропотливый процесс, требующий не только особых навыков программирования, но и целой разношерстной команды – как минимум нужно несколько разработчиков (создать компьютерную игру одному не под силу) и хороший художник, владеющий не только кистью, но и графическими редакторами.

В современном мире игры настолько сложные, то с нуля их никто практически не создает. Как правило, используется какой-то фреймворк (движок) для создания компьютерных игр. Использование фреймворка существенно упрощает (следовательно, ускоряет) процесс разработки игры.

Одно из таких средств и стало предметом этой книги. Unity — межплатформенная среда разработки компьютерных игр. С ее помощью можно создавать игры, работающие под более чем 20 различными операционными системами – настольные компьютеры (стационарные и ноутбуки), игровые консоли, мобильные устройства, Интернет-приложения и др.

Основная задача Unity – демократизировать разработку игр. Теперь разработка собственной игры доступна каждому желающему, в том числе студенту (для них вообще особые условия) и для старта вам не понадобится какая-то космическая сумма денег (возможно, она понадобится в будущем для содержания собственной IT-компании, специализирующейся на разработке

игр). Большинство необходимых средств бесплатно и на что, возможно, придется потратиться – на пару наборов ассетов (ресурсов) для игры, чтобы не разрабатывать их самостоятельно.

Первая версия Unity вышла в 2005 году и с того времени активно развивается.

Преимущества Unity:

- **Она бесплатна**. Да, есть и платные версии Unity, но если вы начинающий разработчик игр, для начала вам будет достаточно бесплатной версии, которая и будет рассмотрена в этой книге. Все, что мы сделаем в этой книге, будет сделано в бесплатной версии.
- **Кроссплатформенная поддержка**. Это означает, что с помощью Unity можно построить игру для любой платформы, в том числе Windows, Linux, iOS, macOS и т.д.
- **Магазин ассетов**. Нужна функциональность, выходящая за рамки стандартной поставки? Необходимые ассеты можно найти в Asset Store – там вы можете купить скрипты, модели, звуки и многое другое для ваших игр.
- **Визуальный редактор**. В отличие от других игровых платформ, где вам придется писать тонны кода, прежде чем вы что-либо увидите на экране, в Unity вам нужно просто импортировать ассет (например, модель человека или машины) на сцену. Визуальный стиль разработки привлекает новичков, да и профессионалов, которые желают экономить время, требуемое на разработку игры. Unity подобна LEGO, где из набора ассетов вы можете построить все, на что способна ваша фантазия.

У всего есть и недостатки и они не обошли стороной и Unity. При работе с этой средой возможны сложности при работе с многокомпонентными схемами и затруднения при подключении внешних библиотек. Но от этого среда не стала менее популярной.

На Unity написаны тысячи игр, приложений и симуляций, которые охватывают множество платформ и жанров. При этом Unity используется как крупными разработчиками, так и небольшими независимыми студиями.

Приборы и материалы

Вот довольно скромный список того, что вам нужно, чтобы работать с этой книге и чтобы разработать собственную игру:

- Персональный компьютер, работающий под управлением Windows 7 SP+ (или с более новой версией Windows) или Mac под управлением Mountain Lion или новее. Это все, что необходимо для установки Unity. О системных требованиях мы поговорим далее.
- Собственно, сама Unity, которую можно бесплатно скачать с сайта **https://store.unity.com/.**
- Blender 2.92 или более новый. Большинство ассетов, которые мы будем использовать в этой книге, являются файлами Blender, который должен быть установлен на вашем компьютере. Не беспокойтесь, он бесплатный. Скачать Blender можно по адресу **https://www.blender.org/download**.
- Устройство, работающее под управлением iOS 9 или выше или Android 6.0 (более древние нет смысла использовать в 2021-ом году) или выше (необязательно). Такие мобильные устройства понадобятся лишь в том случае, если вы планируете разработку мобильных игр.
- Видеокарта, совместимая с DX10 - DX12. В большинстве случаев примеры из этой книги можно будет запускать и на более старой видеокарте, однако, если вы планируете разработку реальной игры, придется потратиться на видеокарту.

На этом все. Теперь о системных требованиях. Более конкретные системные требования выглядят так:

- Операционная система должна быть 64-разрядной. Да, вы можете вести разработку в Windows 7 SP1+, но она должна быть 64-разрядной. Связано это хотя бы с тем, что 32-разрядные операционные системы могут адресовать 2^{32} битов, то есть 4 Гб ОЗУ. Да, версия Unity 2019.1 до сих пор поддерживает Windows 7 SP1, так что если у вас до сих пор древняя «операционка», ее не придется переустанавливать, хотя в 2021-ом году давно пора перейти на Windows 10, учитывая, что поддержка Windows 7 была прекращена в 2020-ом году.
- Минимальный объем оперативной памяти – 8 Гб (на 2020 год, в скором времени системные требования могут измениться и «минималка» будет на уровне 16 Гб).
- Процессор должен поддерживать набор инструкций SSE2. Это минимальные требования. Рекомендуемые – Intel Core i7-3770 @ 3.4 GHz или AMDFX-8350 @ 4.0 GHz или лучше. Если компьютера еще нет, и вы только подумываете о приобретении – покупайте или стационарный

компьютер или игровой ноутбук – на таких ноутбуках лучше проработана система охлаждения, и он не будет выключаться при перегревах. Также на игровых ноутбуках часто устанавливают полноценные процессоры, а не их урезанные мобильные версии. Но и стоят такие машины недешево.

- Объем видеопамяти – 2 Гб или выше. О совместимости с DirectX было сказано ранее (DX10 или DX11/DX12). Конкретные примеры минимальных рекомендуемых карт – NVIDIA GeForce GTX 780 или AMD Radeon R9 290X. Встроенные видеокарты ноутбуков мало подходят для разработки игр, хотя бы по той причине, что они будут использовать оперативную память ноутбука и из, скажем 8 Гб, у вас будет доступно 6 Гб, поскольку 2 из них будет отведено под видеопамять. Покупайте ноутбуки только с двумя видеокартами – в обычном режиме будет использовать встроенная видеокарта (вроде какой-то Intel 6xx), а в процессе игры – вторая, более мощная видеокарта.
- Желательно иметь SSD-накопитель, чтобы ускорить работу дисковой подсистемы. Объем диска может быть минимальным – хватит и 128 Гб. Этого будет достаточно, чтобы установить операционную систему и Unity, а все остальное можно хранить на обычном HDD.

Конечно, это не означает, что если у вас менее мощный компьютер, нужно сразу же отправляться в магазин и покупать мощную машину. Не спешите. Начать разработку можно и на менее мощном компьютере, скажем, с процессором Intel Core i3. Из тех рекомендаций обязательный только объем оперативной памяти – 8 Гб, при меньшем объеме процесс разработки игры обещает быть менее приятным.

Что умеет Unity? Основные возможности

Отличительная черта Unity – простой и понятный даже новичку интерфейс визуального редактора. Разработчик может не только просмотреть, как будет выглядеть так или иная сцена, но и выполнить отладку игры прямо в редакторе. Да, используя инспектор (область **Inspector**), вы можете в процессе выполнения игры менять свойства всех объектов сцены и смотреть в режиме реального времени, как применяются изменения! При использовании любой другой среды процесс выглядит иначе: внесли изменения, запустили, если что-то не так – прекратили выполнение, опять внесли изменения, опять запустили и т.д. В этом плане Unity сильно экономит время разработчика.

До версии 2017.2 Unity поддерживала два языка программирования – C# и диалект JavaScript (правда, в Unity он называется UnityScript). Ранее была поддержка диалекта Python (язык Boo), но в пятой версии ее убрали, поскольку она не пользовалась спросом. Отсюда следует, что вы, как разработчик, должны владеть, либо C#, либо JavaScript. Ни один из этих языков программирования в данной книге не рассматривается – это довольно распространенные языки и им посвящены десятки книг. Вы можете выбрать любую. Однако в более новых версиях Unity от поддержки UnityScript отказались. Получается, или использовать версию 2017, которая на данный момент актуальна и поддерживается сообществом (в плане ассетов и других ресурсов), или же программировать на C#. Лучше выбрать второй вариант, тем более что все скрипты в этой книге написаны именно на C#, но если вы отлично знаете JavaScript, у вас могут возникнуть сомнения. Установив версию 2017, можно программировать на UnityScript, однако, имейте в виду, что у вас не получится перенести код в более новую версию Unity. И это основной недостаток, из-за которого использование UnityScript строго не рекомендуется.

Теперь поговорим о проекте в Unity. Проект делится на сцены (уровни) – это отдельные файлы, содержащие свои игровые миры со своим набором объектов, сценариев, и настроек. Сцены, как правило, содержат объекты (модели), так и пустые игровые объекты, которые не имеют своей модели – так называемые «пустышки». В свою очередь, объекты содержат наборы компонентов, с которыми и взаимодействуют скрипты.

У каждого объекта есть свое название (при этом в Unity допускается наличие двух и более объектов с одинаковыми названиями), также у объекта есть тег и слой, на котором он должен быть показан.

Еще у каждого объекта на сцене обязательно имеется компонент **Transform**, хранящий координаты местоположения, поворота и размеров объекта по всем трем осям. Чтобы сделать модель видимой, используется компонент **Mesh Renderer**.

Среда Unity поддерживает физику твердых тел и тканей, а также физику вроде Ragdoll (тряпичная кукла). Редактор поддерживает систему наследования объектов: дочерние объекты будут повторять все изменения позиции, поворота и масштаба родительского объекта. Скрипты в редакторе прикрепляются к объектам в виде отдельных компонентов.

При импорте текстуры в Unity можно сгенерировать alpha-канал, mip-уровни, normal-map, light-map, карту отражений – это все, конечно, хорошо. Но есть и ограничения – нельзя прикрепить текстуру на модель. Будет соз-

дан материал, которому будет назначен шейдер, и затем материал прикрепится к модели. Редактор Unity поддерживает написание и редактирование шейдеров, также он имеет компонент для создания анимации, но анимацию при желании можно предварительно создать в трехмерном редакторе и импортировать вместе с моделью, а после – разбить на файлы.

Среда Unity 3D поддерживает систему LOD (Level Of Detail). Суть этой системы заключается в том, что на дальнем расстоянии от игрока модели с высокой детализацией автоматически заменяются моделями с меньшей детализацией. Система LOD существенно повышает производительность и вообще позволяет играм запускаться на оборудовании со скромными характеристиками.

Также имеется поддержка системы Occlusion culling. Работает она так: если объект не попадает в поле действия камеры, то у него не визуализируется геометрия, что снижает нагрузку на центральный процессор и также повышает производительность.

Среда Unity требует для компиляции программы Visual Studio Community. Эта среда должна у вас быть установлена. Если она не установлена, ее можно будет установить при установке Unity. Компиляцию программы выполняет не Unity, а компилятор Visual Studio.

При компиляции проекта создается исполняемый (.exe) файл игры (для Windows – мы будем рассматривать только эту платформу), а в отдельной папке — данные игры (включая все игровые уровни и динамически подключаемые библиотеки).

Еще одно преимущество Unity – поддержка множества самых разных форматов моделей, звуков, текстур, материалов, скриптов – все это можно запаковывать в формат .unityassets и передавать другим разработчикам, или выкладывать в свободный доступ. Этот же формат используется во внутреннем магазине Unity Asset Store, в котором разработчики могут бесплатно и за деньги выкладывать в общий доступ различные элементы, нужные при создании игр. Чтобы использовать Unity Asset Store, необходимо иметь аккаунт разработчика Unity. Unity имеет все нужные компоненты для создания мультиплеера. Также можно использовать подходящий пользователю способ контроля версий. К примеру, Tortoise SVN или Source Gear. В состав Unity входит Unity Asset Server — инструментарий для совместной разработки на базе Unity, являющийся дополнением, добавляющим контроль версий и ряд других серверных решений.

На Unity написано, и продолжают создаваться огромное количество игр самых разных жанров. Вот лишь небольшой список игр за 2020 и 2021 года:

- Battletoads
- Bendy and the Dark Revival
- Cloudpunk
- Cobra Kai: The Karate Kid Saga Continues
- Cooking Mama: Cookstar
- Desperados III
- Fall Guys: Ultimate Knockout
- Frog Fractions: Game of the Decade Edition Genshin Impact
- Hellpoint
- Helltaker
- Iron Man VR
- Jump Rope Challenge
- Kingdom Hearts: Melody of Memory
- League of Legends: Wild Rift
- Legends of Runeterra
- NASCAR Heat 5
- Oddworld: Soulstorm
- Ooblets
- Ori and the Will of the Wisps
- PGA Tour 2K21
- Phasmophobia
- Pokémon Mystery Dungeon: Rescue Team DX
- Realm of the Mad God
- Shin Megami Tensei III: Nocturne HD Remaster
- Temtem
- Tony Stewart's Sprint Car Racing

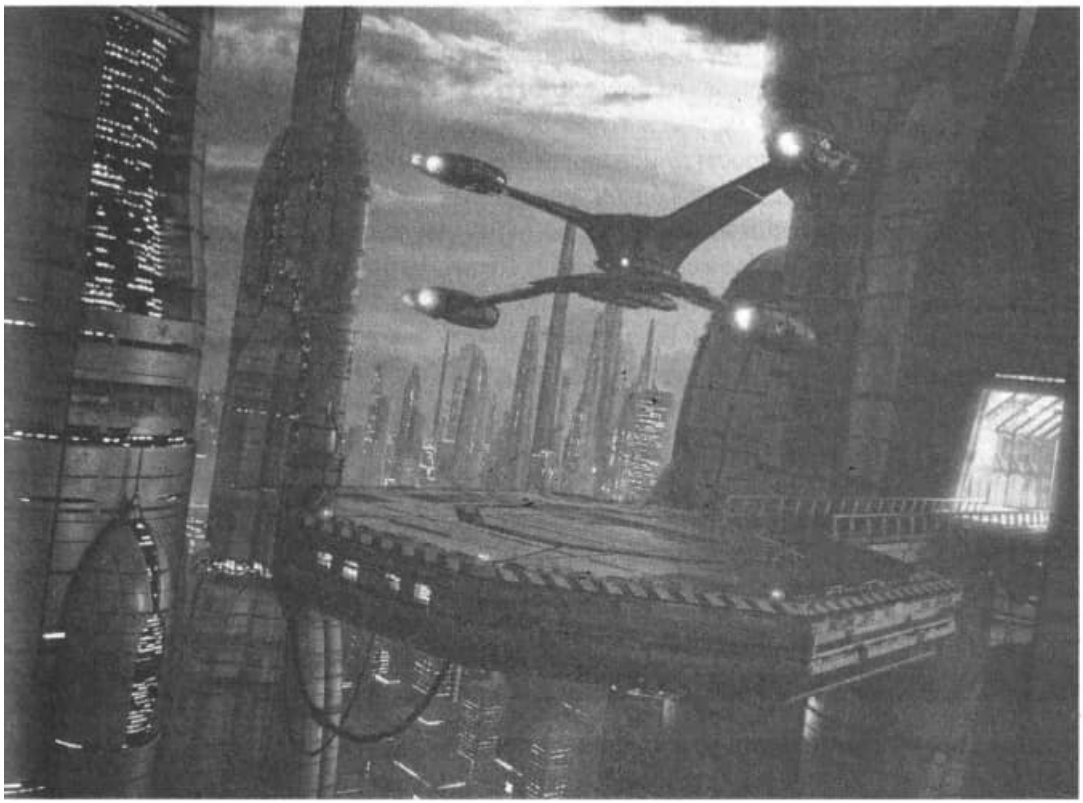
- Tony Stewart's All-American Racing
- Umurangi Generation
- Wasteland 3
- YesterMorrow
- EverHood
- Kerbal Space Program 2
- FAU-G: Fearless and United Guards
- Valheim
- ...

И еще многие-многие другие. На данный момент Unity стали также активно использовать крупные игровые компании-разработчики, делающие проекты высочайшего класса.

ЧАСТЬ I.
РАБОТАЕМ В UNITY

Глава 1.

Основы Unity



Первая часть посвящена изучению интерфейса и основных возможностей Unity. Так, в первой главе мы поговорим о загрузке и установке Unity, о двух- и трехмерных проектах, рассмотрим основные элементы Unity. Конечно же, будут затронуты вопросы лицензирования – без этого никак. Во второй – об использовании ассетов – кубиков, из которых состоит ее Величество Игра. Третья глава посвящена более подробному изучению интерфейса Unity, в ней будут рассмотрены основные окна среды. Игровые объекты, сцены, камеры, источники света – все это рассматривается в четвертой главе. По результатам прочтения первой части книги игру вы, конечно, еще не сможете создать, но уже будете ориентироваться в Unity на твердом среднем уровне.

1.1. Лицензирование: что и сколько стоит

Скачать Unity, как уже отмечалось во введении (если вы, конечно, его прочитали), можно по адресу **https://store.unity.com/**. Там же можно ознакомиться с ценовой политикой (рис. 1.1).

Существует несколько типов лицензий:

- Коммерческие – Plus, Pro и Enterprise
- Для индивидуального использования – Student, Personal и Unity Learn

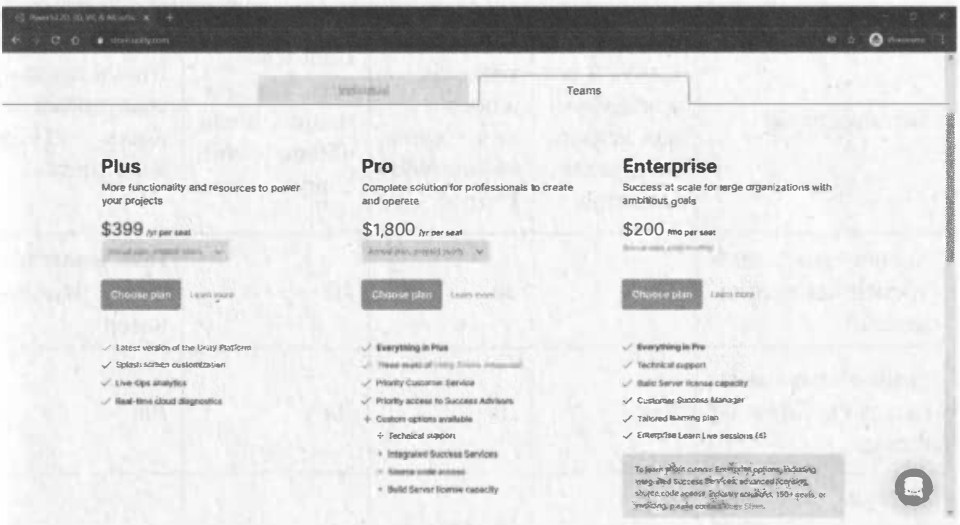


Рис. 1.1. Политика ценообразования

Примечание. Unity Learn – это не сколько тип лицензирования, сколько мастер-классы с экспертами Unity, возможности этого «тарифа» включены в коммерческие планы Plus, Pro и Enterprise.

Для небольших студий, чей ежегодный доход не превышает 100 000 долларов, Unity 3D абсолютно бесплатна по лицензии Personal. Другими словами, если вы хотите попытаться создать игру в одиночку или же у вас даже есть уже стартап, на первых порах можно обойтись бесплатной версией.

Лицензию Plus нужно покупать, если ежегодный доход превысил 100 000 долларов – до 200 000. Но тип лицензии зависит не только от вашего дохода, но и от количества одновременных игроков. К тому же расширенная поддержка доступна только, начиная с Pro-версии.

Таблица 1.1 подробно описывает особенности разных типов лицензий.

Таблица 1.1. Типы лицензии

Характеристика	Personal	Plus	Pro	Enterprise
Ежегодный доход компании	100 000$	200 000$	Неограничен	Неограничен

Заставка игры	«Made With Unity» и необязательная пользовательская анимация	Пользовательская анимация и/или «Made With Unity»	Пользовательская анимация и/или «Made With Unity»	Пользовательская анимация и/или «Made With Unity»
Количество одновременных пользователей	20	50	200	Пользовательских мультиплеер
Темная тема (называется Pro Editor UI theme)	Нет	Да	Да	Да
Доступ Unity Teams Advanced для совместной работы над проектом	Нет	25 Гб хранилища	25 Гб, 3 места	25 Гб, 3 места
Высококачественный набор ассетов	Нет	Нет	Да	Да
Отчеты по производительности	Нет	Да	Да	Да
Премиум поддержка	Нет	Нет	За отдельную плату	За отдельную плату
Доступ к исходному коду	Нет	Нет	За отдельную плату	За отдельную плату
Стоимость в год, $	Бесплатно	399	1800	-
Стоимость в мес., $	Бесплатно	33	150	200$ за одно рабочее место

Примечание. Подробная информация (сравнение) тарифных планов находится по адресу **https://store.unity.com/compare-plans?currency=USD**. Информация постоянно обновляется, поэтому посетите эту страничку, даже если вы ознакомились с табл. 1.1. Также обратите внимание, что нужно лицензировать каждое рабочее место. Другими словами, если в вашей команде

три рабочих места для Unity-разработчиков, то все цены из таблицы 1.1 нужно умножить на 3.

Итак, проанализируем ограничения бесплатной версии. Первым делом, у вас будет заставка «Made With Unity», отказаться от которой невозможно. В лучшем случае вы можете добавить только свою анимацию, но от заставки разработчиков вы не избавитесь.

Серьезную многопользовательскую игру вы также не построите, поскольку 20 одновременных игроков по сегодняшним меркам это маловато. Лет 15 назад, когда многопользовательская игра велась в основном в компьютерном клубе, то 20 было вполне достаточно – в небольших клубах компьютеров даже меньше было. А сейчас, когда игра ведется по Интернету, то 20 – очень мало. 50 игроков тоже маловато, 200 – уже кое-что, но только на первых порах, пока ваша игра не станет популярной. Когда это произойдет, придется перейти на Enterprise-версию, где подразумевается использование пользовательского мультиплеера.

Также сборка в облаке будет происходить не так быстро, как у платных версий. И чем дороже версия, тем приоритетнее сборка в облаке.

И еще у вас не будет темной темы оформления, которая является признаком всех платных версий и очень удобна, если вы привыкли работать по ночам, так как не так слепит глаза, как светлая тема по умолчанию.

Тем не менее, не смотря на все ограничения, бесплатную версию можно попробовать в самом начале – вы ничего не теряете. Впрочем, бесплатная версия не такая уж и плохая. Как будет показано далее, даже бесплатная версия поддерживает облачное хранилище, что позволит получать доступ к вашим проектам с любого компьютера. При первом запуске Unity вам будет предложено создать учетную запись – Unity ID, которую вы можете использовать для доступа к своим облачным проектам.

Особый тарифный план предусмотрен для студентов (https://store.unity.com/academic/unity-student). Если вкратце, то вот его преимущества:

- Неограниченный доступ к Learn Premium;
- Пять рабочих мест в Unity Teams Advanced;
- Богатый набор высококачественных бесплатных ассетов;
- Темная тема UI.

Для получения этого тарифного плана вы должны зарегистрироваться на GitHub Student Developer Pack, чтобы пройти проверку, что вы действительно являетесь студентом.

1.2. Установка

Устанавливается Unity 3D не сложнее обычного приложения. В самом начале нужно принять условия использования (можно их даже прочитать), а также выбрать устанавливаемые компоненты.

Процесс установки самой последней версии Unity немного отличается от того, как это делалось ранее. Ранее пользователь загружал инсталлятор, который устанавливал саму Unity и Visual Studio. Сейчас же происходит установка Unity Hub, который управляет разными версиями Unity и проектами. Но обо всем по порядку.

Скачайте с сайта **https://store.unity.com** установочный файл UnityHubSetup.exe и запустите его. Установите Unity Hub, как обычное приложение (рис. 1.2 и 1.3).

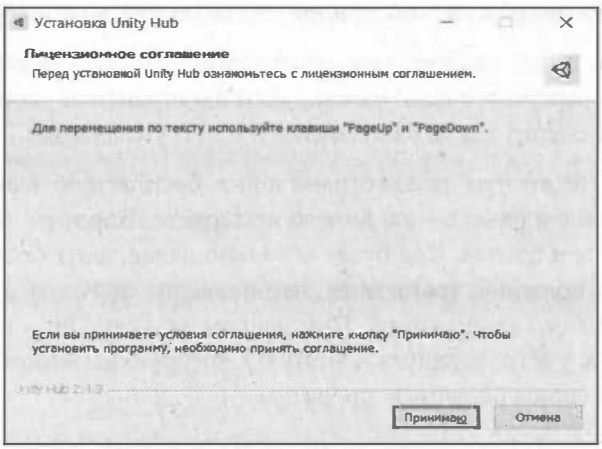


Рис. 1.2. Лицензионное соглашение. Нажмите кнопку "Принимаю"

Обратите внимание: это не каталог для установки Unity, а каталог для установки UnityHub. После этого браундмауэр Windows или любой другой (если у вас установлен сторонний) предложит разрешить или запретить доступ новому приложению. В случае с брандмауэром Windows нужно выбрать, как частные, так и общественные сети и нажать кнопку **Разрешить доступ** (рис. 1.4).

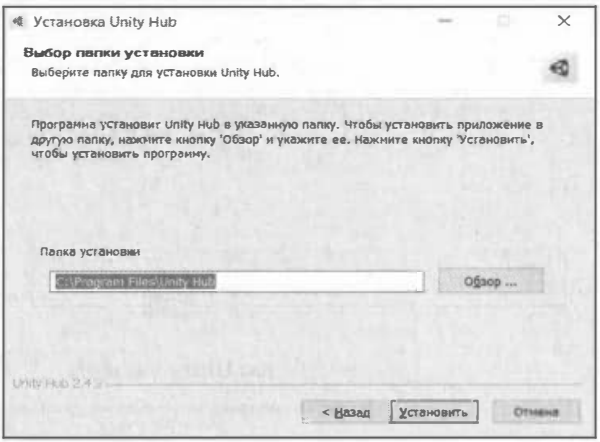


Рис. 1.3. Каталог для установки

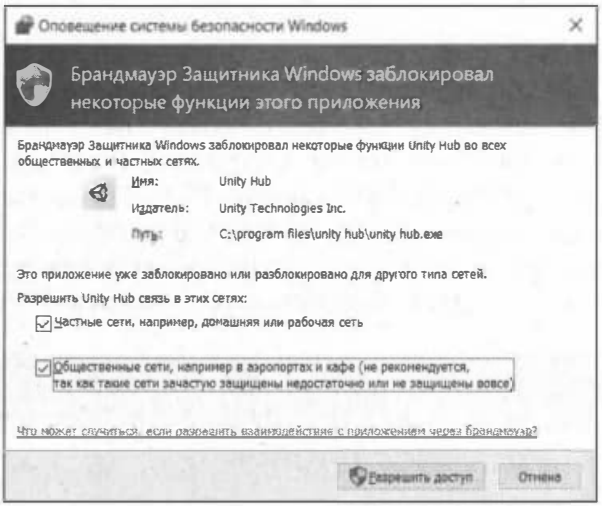


Рис. 1.4. Настройка брандмауэра для корректной работы Unity Hub

Далее откроется окно **Unity Hub** (рис. 1.5), который сразу же предложит установить актуальную версию Unity. В этой книге я решил использовать версию 2020.3. Если нужна другая версия Unity (вы можете установить предыдущие версии, например, 2018 или 2019), то нажмите ссылку **Skip Install Wizard**, затем в появившемся окне нажмите кнопку **Add** в разделе **Install** и у вас будет возможность выбора версии Unity.

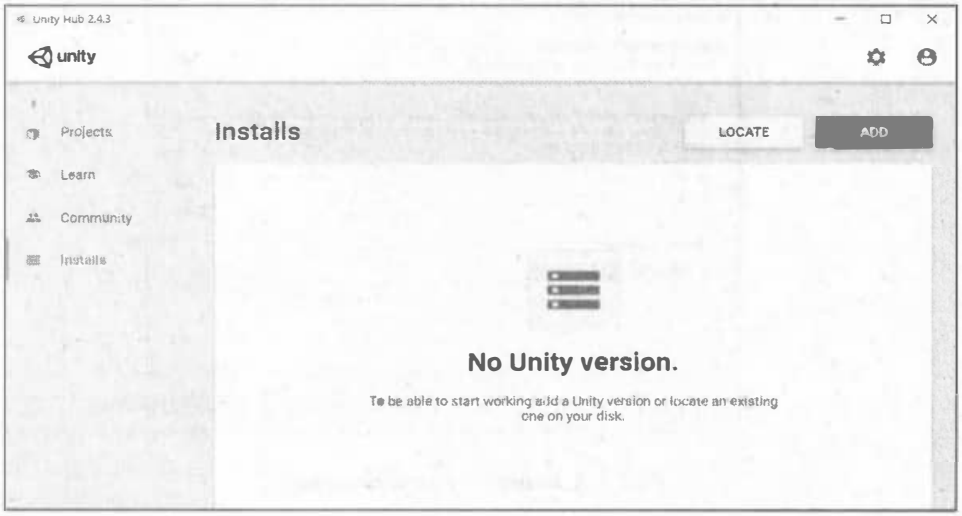


Рис. 1.5. Unity Hub (раздел Install)

Далее следует выбрать устанавливаемую версию. На данный момент доступны версии, показанные на рис. 1.6. Выберем наиболее стабильную версию – 2020.3. Другие версии – 2019.4 или 2018.4 вы должны выбирать только в том случае, если вы знаете, что делаете. Например, у вас есть ассеты, которые по каким-то причинам не работают в версии 2020.3. Но на таком уровне познаний, думаю, вы уже не читали бы эту книгу. Версию Pre-Release выбирать не стоит – это «сырая» версия, которая станет релизом в скором времени. Пусть ее тестирует кто-то другой. Выберите версию и нажмите кнопку **Next**.

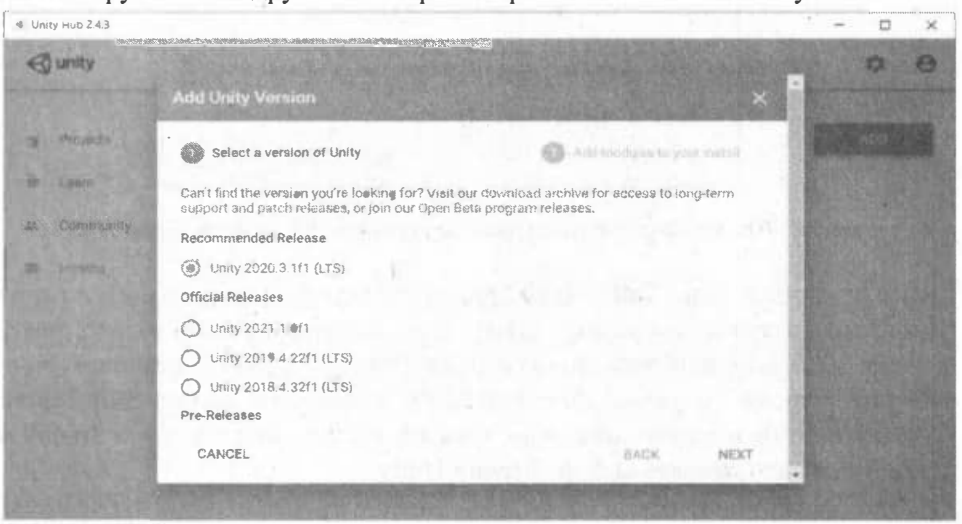


Рис. 1.6. Выбор версии Unity

Внимание! Выбор версии Unity – это довольно серьезная вещь. Да, всегда хочется использовать самую последнюю версию, и мы ее будем использовать, пока вы изучаете Unity. Но на практике иногда нужно выбирать более старую версию, поскольку не всегда желаемые наборы ассетов могут работать в самой последней версии. Разработчики ассетов не всегда поспевают за разработчиками Unity, особенно, если речь идет о бесплатных наборах. Поэтому если есть бесплатный набор ассетов, который вас устраивает и который вы хотите использовать в своей игре, но он не работает с версией 2020.3, смело устанавливайте ту версию, которая вам нужна. Благо Unity Hub позволяет с легкостью установить несколько версий Unity на одном компьютере и вызывать их по мере необходимости. С коммерческими ассетами все гораздо проще – разработчики, как правило, обновляют их вовремя, и вы можете использовать последние версии Unity, но на этапе обучения и создания первой вашей игры вряд ли вы будете покупать коммерческие ассеты, тем более что некоторые из них стоят довольно дорого. Установка предыдущей версии Unity в данной ситуации – выход для вас.

Затем необходимо выбрать список устанавливаемых компонентов (рис. 1.7). В любом случае вам необходим Microsoft Visual Studio Community – это бесплатная версия Visual Studio (исключения могут составить лишь случаи, если эта среда уже у вас установлена). Остальные компоненты выберите по своему усмотрению. Например, если планируете разработку игр под Android, то выберите **Android Build Support**.

Примечание. Устанавливать Unity нужно на диск, на котором достаточно свободного пространства. В идеале, если бы это был SSD-диск, чтобы Unity быстрее работала и быстрее компилировался проект. Однако не всегда получается так сделать. Как правило, компьютеры оснащаются небольшими SSD-дисками (128-256 Гб) и не все ПО можно установить на них при всем нашем желании. Поэтому приходится устанавливать на жесткий диск (HDD). В любом случае текстуры и вообще все, что касается трехмерной графики, весит довольно много, поэтому чем больше свободного места будет, тем лучше.

Внимание! При установке версий 2017 и 2018 в списке устанавливаемых компонентов будет набор стандартных ассетов

Standard Assets. Для версий 2019.3 и 2020.3 такой набор не предусмотрен. Далее в этой книге будет показано, как импортировать стандартные ассеты в версию 2020.3 и как заставить их работать в новой версии Unity. Учтите, об этом не сказано в руководстве и только что вы сэкономили несколько тысяч рублей на покупке аналогичного набора для версии 2020.3!

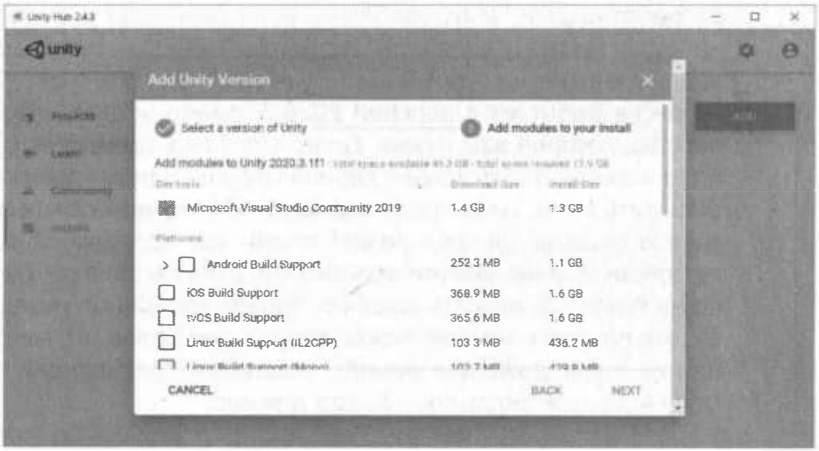


Рис. 1.7. Выбор компонентов для установки

Далее пролистайте страницу и включите флажок **I have read and agree with the above terms and conditions** – этим вы соглашаетесь с условиями распространения Visual Studio (неплохо бы эти условия еще и прочитать перед этим по приведенной ссылке), см. рис. 1.8. Нажмите кнопку **Done**.

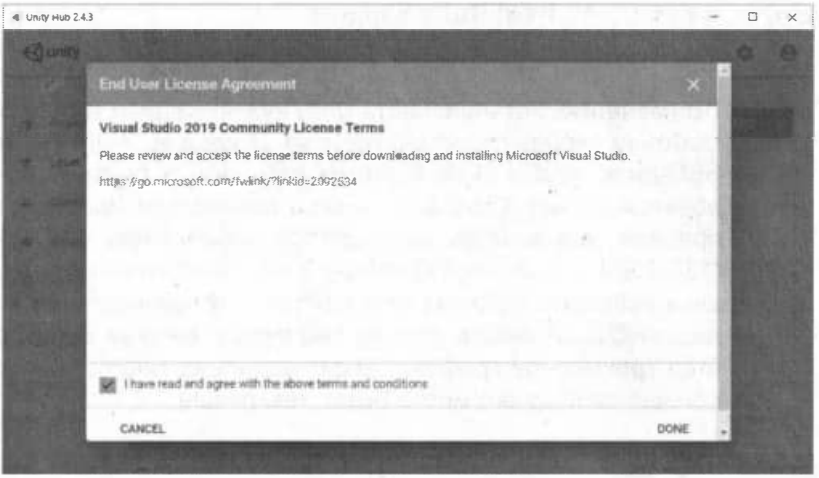


Рис. 1.8. Нажмите кнопку "Done"

Начнется установка всего, что вы выбрали до этого. Индикатор установки (синий на фоне голубого вверху плитки с номером выбранной версии) информирует о ходе процесса установки (рис. 1.9).

Рис. 1.9. Ход установки

Также в процессе установки (если это выбрано) будет загружен (рис. 1.10) и установлен (рис. 1.11) Visual Studio. Когда все будет готово, вы увидите плитку с номером выбранной версии Unity без какого-либо индикатора прогресса (рис. 1.12).

Рис. 1.10. Загрузка Visual Studio

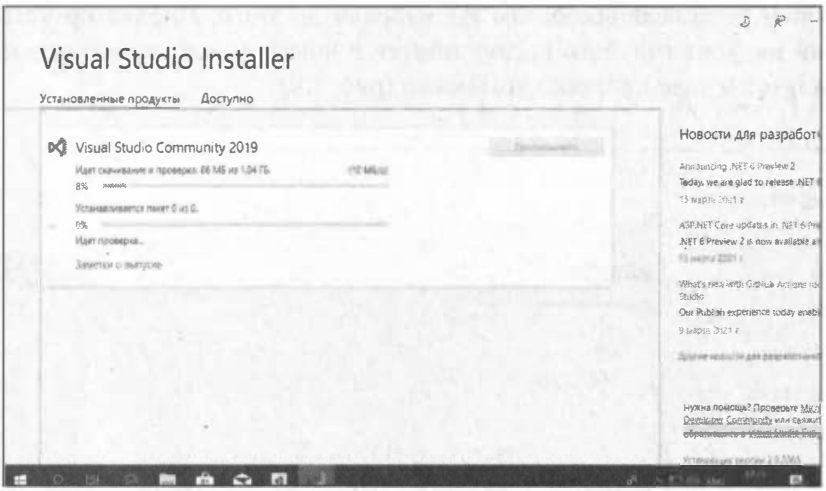


Рис. 1.11. Установка Visual Studio

Рис. 1.12. Все готово

1.3. 2D или 3D?

Прежде, чем мы приступим к созданию вашего первого проекта, нужно поговорить о 2D и 3D играх. Используя Unity, вы можете создать, как двухмерную, так и трехмерную игру. При создании нового проекта вам будет предоставлен выбор – какую игру вы хотите создать. Другими словами, еще

до создания проекта вы должны определиться, какую именно игру вы будете создавать.

Выбор режима – 2D или 3D – перед началом проекта определяет некоторые настройки редактора Unity. Например, будут ли изображения импортированы как текстуры или же как спрайты. С другой стороны, хотя выбор шаблона осуществляется при создании проекта, вы должны знать, что если вы сделали неправильный выбор, то не нужно заново создавать проект. Вы можете переключаться между 2D и 3D режимом в любое время, независимо от того, какой режим выбрали при создании проекта.

Следующие рекомендации помогут вам сделать правильный выбор.

1.3.1. Полноценные 3D игры

Полноценные трехмерные игры используют трехмерную геометрию, с материалами и текстурами, отображаемыми на поверхностях объектов так, чтобы обеспечить целостность окружения, персонажей и объектов, из которых состоит ваш игровой мир. Камера может двигаться внутри и вокруг сцены совершенно свободно, с реалистичным отображением света и теней по всему миру. Популярный компьютерный шутер Counter-Strike – типичный пример полноценной трехмерной игры.

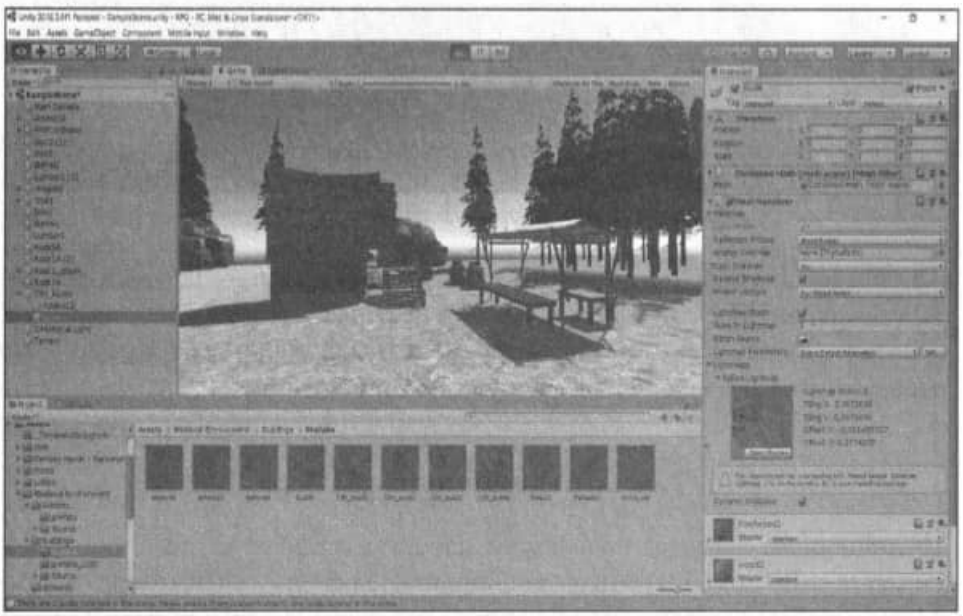


Рис. 1.13. Полноценная трехмерная сцена, созданная в Unity

Трехмерные игры отображают сцену с использованием перспективы, то есть объекты кажутся тем больше размером, чем ближе они к камере. Для всех игр, соответствующих, этому описанию стоит использовать режим 3D.

1.3.2. Ортографические 3D игры

Такие игры используют трехмерную графику, но при этом вместо перспективы применяется ортографическая камера. Данная технология используется в играх, где отображение происходит с высоты птичьего полета. Типичные примеры – всевозможные «строилки», например, SimCity, Мегаполис и т.д. Такие игры еще называют 2.5D – не полностью трехмерными.

Рис. 1.14. Пример ортографической 3D-игры

При создании подобной игры вам нужно использовать редактор в режиме 3D, поскольку не смотря на то, что перспективы у вас не будет, но вы по-прежнему будете работать с трехмерными моделями. Однако нужно будет переключить камеру и вид сцены в режим **Orthographic**.

1.3.3. Традиционная 2D игра

Традиционные 2D игры используют плоскую графику, которая иногда называется спрайтом. Такие игры в принципе не используют трехмерную графику. Спрайты показываются на экране как плоские картинки, а камера не

имеет никакой перспективы. Типичный приведен на рис. 1.15. Думаю, вы узнали данную игру.

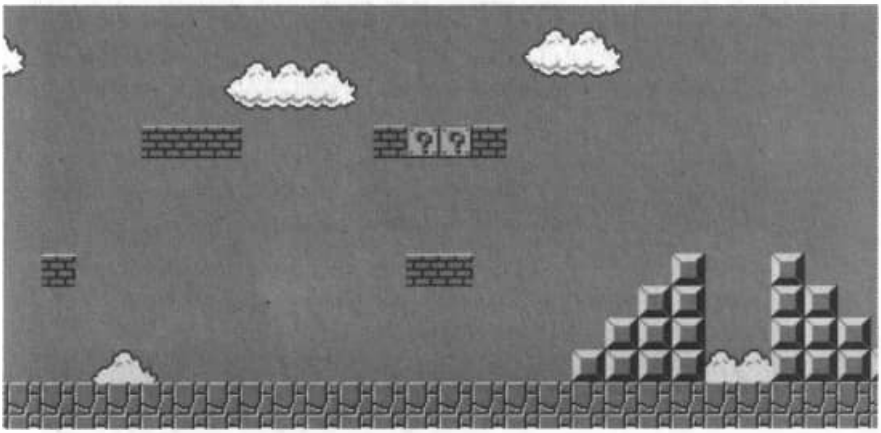


Рис. 1.15. Mario: 2D игра

Для создания такой игры нужно выбрать режим 2D.

1.3.4. 2D игра с 3D графикой

Еще один симбиоз игр – это двухмерная игра с трехмерной графикой. Например, некоторые игры используют трехмерные модели персонажей и окружения, но сам геймплей ограничен двумя измерениями. Камера может использовать "боковое скольжение" и игрок может двигаться только в двух измерениях, но игра по-прежнему использует 3D модели в качестве препятствий, а камера имеет перспективу. По сути, это обычная двухмерная игра, а 3D графика больше используется из стилистических соображений, никакого функционала в ней нет. Поскольку вам все равно нужно будет управлять трехмерными моделями, выбирайте режим 3D для создания таких игр.

Если подытожить, то режим 2D стоит выбирать только для создания традиционных «плоских» игр вроде Super Mario.

1.4. Создание проекта

Последовательность действий по созданию проекта следующая:

Рис. 1.16. 3D модели, 2D геймплей

1. Откройте Unity Hub и перейдите в раздел **Projects**.
2. Нажмите кнопку **New**, из появившегося меню выберите версию Unity, для которой создается проект. На данном этапе у вас должна быть установлена одна версия – 2020.3, о чем мы говорили ранее.
3. Выберите предлагаемый шаблон. Пусть это будет 3D игра (рис. 1.17).
4. Введите название проекта.
5. Выберите папку для вашего проекта. На диске, на котором находится эта папка, должно быть достаточно свободного пространства. Проекты компьютерных игр (даже самых простых) легко могут занимать несколько гигабайтов дискового пространства – все зависит от ассетов, которые вы импортируете в проект. Поэтому нужно позаботиться, чтобы места хватило.
6. Нажмите кнопку **Create**.

> **Примечание.** Также создать новый проект можно командой меню **File, New Project** из основного окна Unity.

Рис. 1.17. Создание нового проекта

После того, как проект будет создан, откроется окно Unity. В следующем разделе мы поговорим о назначении элементов основного окна.

Рис. 1.18. Главное окно Unity

1.5. Быстрый старт

1.5.1. Элементы главного окна

По умолчанию Unity создает полностью пустой проект. Это базовая точка для создания игры и начало начал: с этого начинается создание любой игры. Только что созданный проект не содержит вообще ничего: ни текстур, ни сеток – он полностью пуст. В папке **Assets** (ресурсы) – пусто.

Не стоит торопиться, нужно первым делом ознакомиться с интерфейсом редактора. Главное окно редактора состоит из нескольких вкладок, называемых **Видами** (Views). В Unity есть несколько типов видов – все они предназначены для конкретных целей, описанных в данном разделе.

Рассмотрим основные элементы главного окна (рис. 1.19):

1. Панель инструментов
2. Панель иерархии (**Hierarchy**)
3. Рабочая область, здесь есть вид сцены (**Scene**), вид игры (вкладка **Game**), вкладка магазина ассетов (**Asset Store**):
   - Вкладка **Scene** – позволяет перемещаться по сцене и редактировать ее. Сцена может отображаться в 2D или 3D перспективе, в зависимости от потребностей разработчика
   - Вкладка **Game** – позволяет просмотреть, как будет выглядеть игра – без компиляции и запуска игры
   - Вкладка **Asset Store** – магазин ассетов, здесь можно приобрести ассеты для импорта в проект
4. Панель инспектора объектов (**Inspector**). Здесь можно изменить свойства объектов
5. Панель **Project**, отображающая библиотеку ассетов, доступных для использования в вашем проекте

В нижней части окна отображается панель **Project** (она также называется Обозревателем проекта), отображающая папки проекта. Если вы загляните в папку Assets, там будет пусто. Данная панель отображает содержимое папки проекта – той, которую вы выбрали для хранения проекта при его создании. При желании эту папку также можно просмотреть с помощью файлового менеджера. Если помните расположение, можете зайти непосредственно в папку, или же используйте команду контекстного меню **Show in Explorer**. Щелкните правой кнопкой мыши на папке **Assets** на панели **Project** и выберите эту команду.

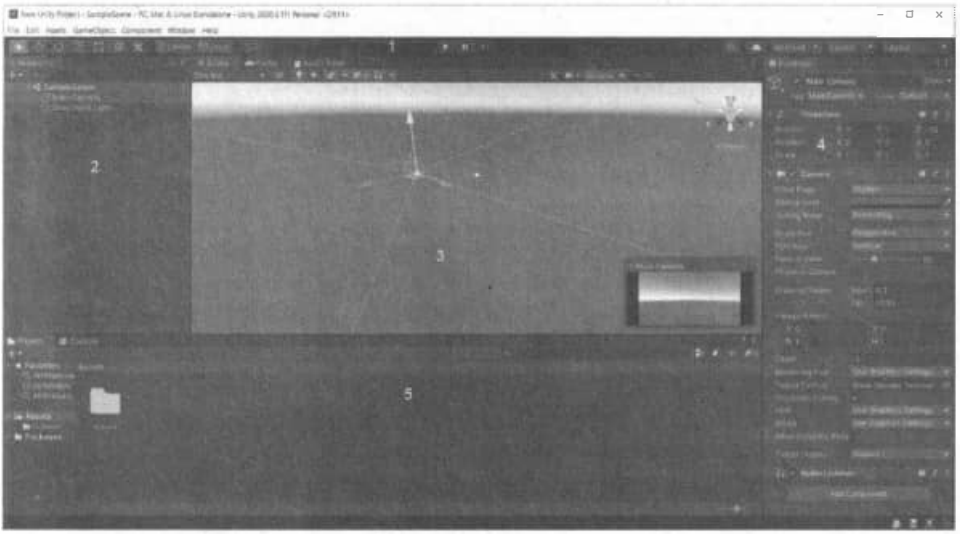


Рис. 1.19. Основные элементы окна Unity

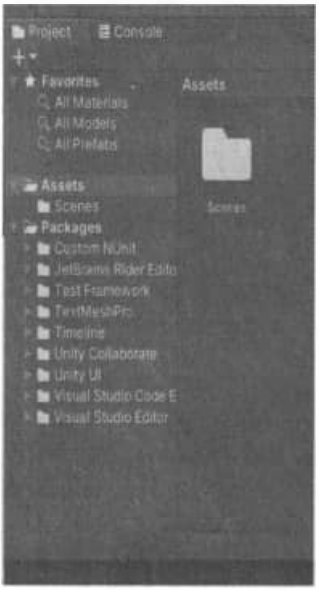


Рис. 1.20. Проект пуст

Примечание. Если ваш интерфейс отличается от приведенного на рис. 1.21, используйте список **Layout** (рис. 1.21) в верхнем правом углу для смены типа разметки основного окна. Выберите удобную для себя разметку.

Команда контекстного меню **Show in Explorer** приведет к открытию папки проекта (рис. 1.22), которую можно будет исследовать посредством стандартного файлового менеджера. Однако не пытайтесь изменить что-то в этой папке вручную. Не нужно перемещать, переименовывать, удалять файлы, поскольку вы повредите целостность проекта Unity.

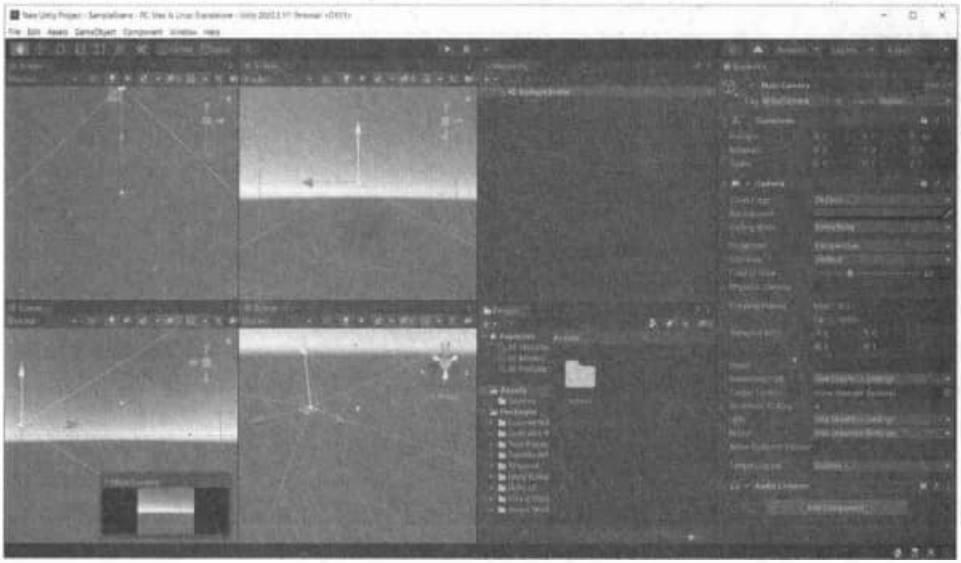


Рис. 1.21. Выбрана другая разметка

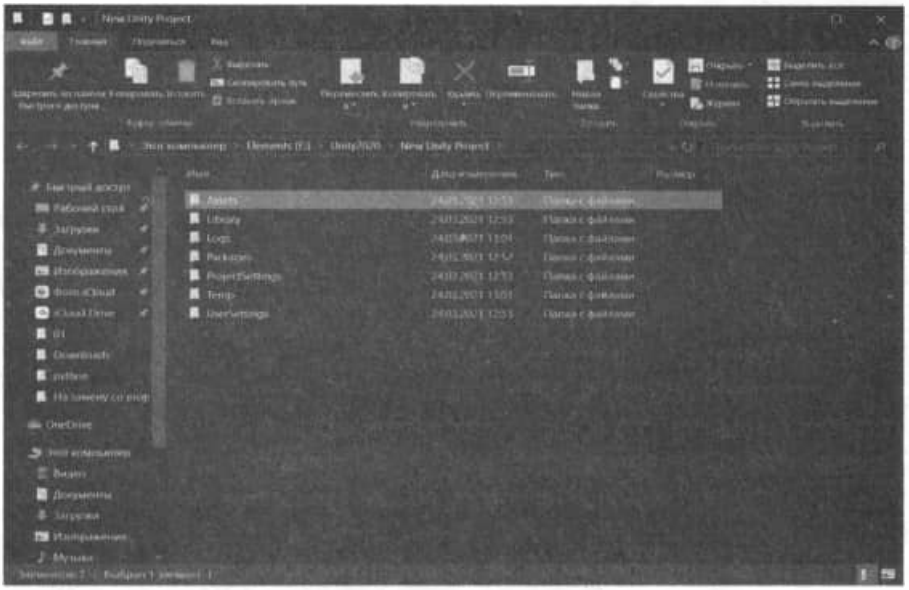


Рис. 1.22. Папка проекта в Проводнике

1.5.2. Импорт ресурсов

Смотрите на ресурсы (assets) как на ингредиенты для своей игры – это кирпичики, из которых вы будете строить игру. К ресурсам относят: 3D-модели

(дома, деревья и т.д.), текстуры (обычно это JPEG/PNG-файл), музыкальные и звуковые эффекты для придания реализма и создания атмосферы игры, а также сцены, которые являются 3D-пространствами или виртуальными мирами. Поэтому игра не может существовать без ресурсов. Без них она будет бессмысленной.

К сожалению, Unity – это игровой движок, а не программа для создания ресурсов. Ресурсы можно создать в Blender, 3DS Max, Photoshop, GIMP и других подобных программах. Для создания трехмерных моделей можно использовать 3DS Max или Blender (бесплатный), для создания текстур – Photoshop или GIMP (бесплатный). Создание аудио-ресурсов можно производить в Audacity (тоже бесплатная). Использование этих программ, к сожалению, выходит за рамки этой книги. Но не расстраивайтесь – на полках книжного магазина можно найти множество книг по тому же 3DS Max и Photoshop.

Unity считает, что ресурсы уже готовы на момент создания нового проекта. Давайте их импортируем. Для этого используется команда меню **Assets, Import Package**. Далее выберите тип импортируемого ресурса – **Vehicles, Cameras, Effects** и т.д. Или выберите команду **Custom Package** для выбора пользовательского пакета.

Если же у вас в этом меню пусто и доступен только пункт **Custom Package**, тогда нужно загрузить стандартные ресурсы. Для этого перейдите по следующей ссылке:

https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-for-unity-2018-4-32351

> **Примечание.** К книге прилагается виртуальный диск, размещенный на сайте издательства. На этом диске вы найдете все используемые в книге ассеты – на тот случай, если они будут удалены из магазина ассетов и больше будут недоступны.

Нажмите кнопку **Add to My Assets** и примите условия использования, нажав кнопку **Accept**. Далее нужно нажать кнопку **Open in Unity**.

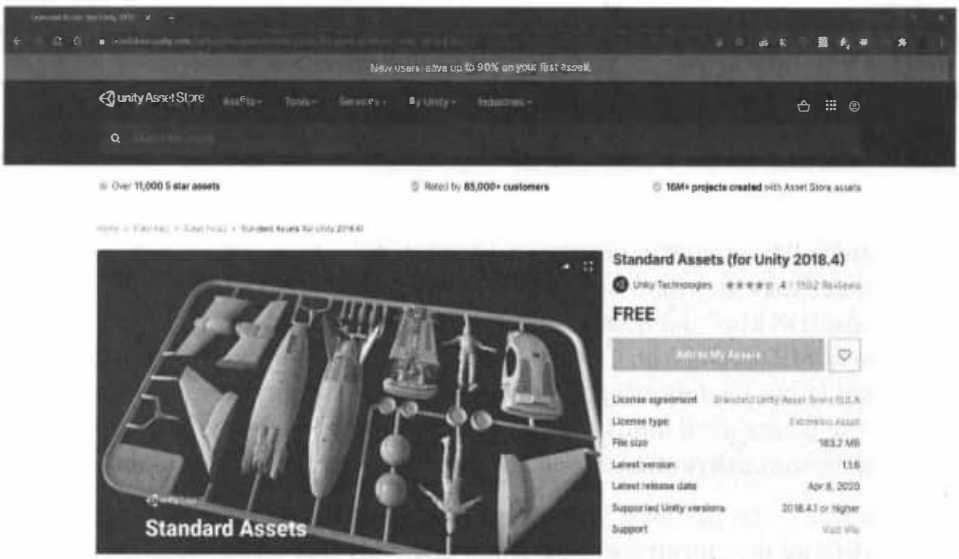


Рис. 1.23. Стандартные ресурсы

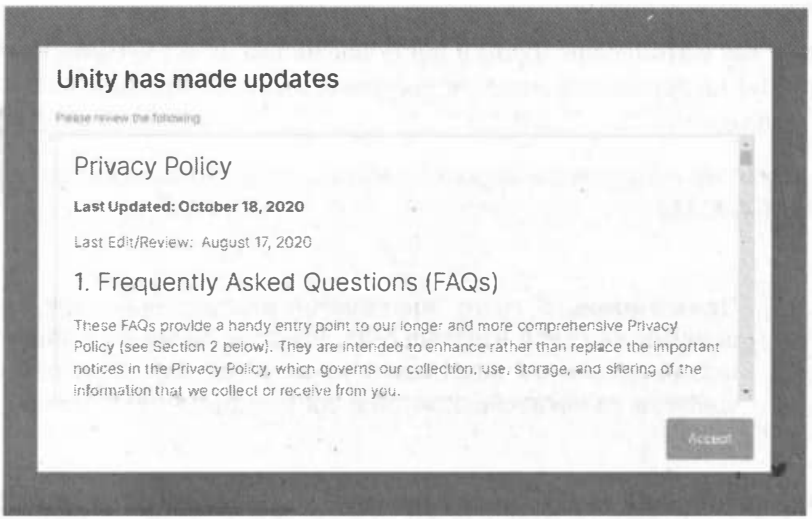


Рис. 1.24. Условия использования стандартных ресурсов

Когда пакет ресурсов откроется в **Unity**, нажмите кнопку **Download** для ее загрузки. На рис. 1.26 ресурсы ранее уже загружались автором этой книги, поэтому кнопка **Download** называется **Update**. Если же вы ранее не загружали эти ресурсы, то в нижнем правом углу у вас будет кнопка **Donwload**.

Рис. 1.25. Нажмите кнопку "Open in Unity"

Примечание. Чтобы увидеть список доступных пакетов, нужно сначала войти в учетную запись Unity ID, нажав кнопку **Sing in**. Если вы еще не зарегистрировали учетную запись, вам предложат это сделать. Регистрация бесплатна и ни к чему вас не обязывает. Зато все купленные или загруженные ранее ресурсы будут доступны через эту учетную запись.

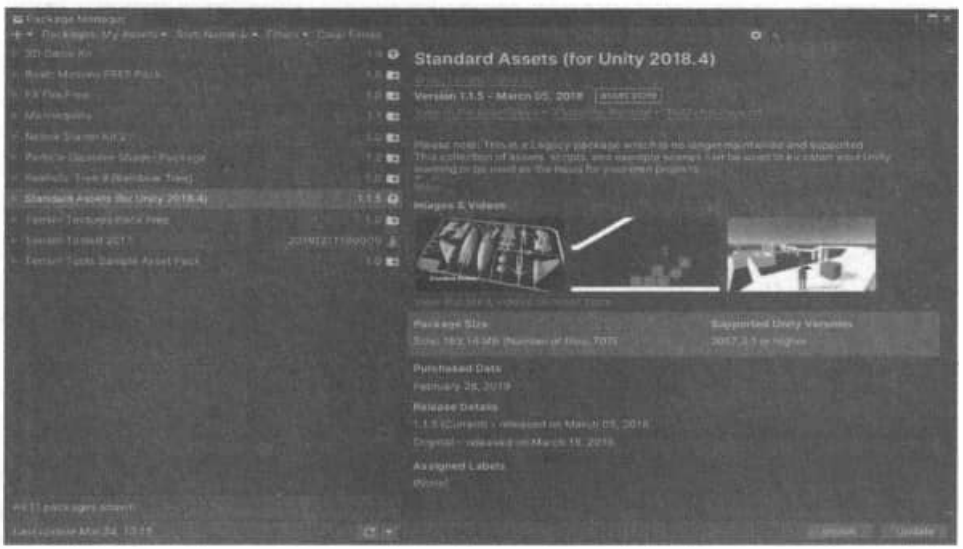


Рис. 1.26. Нажмите кнопку "Download/Update" и дождитесь загрузки ресурсов

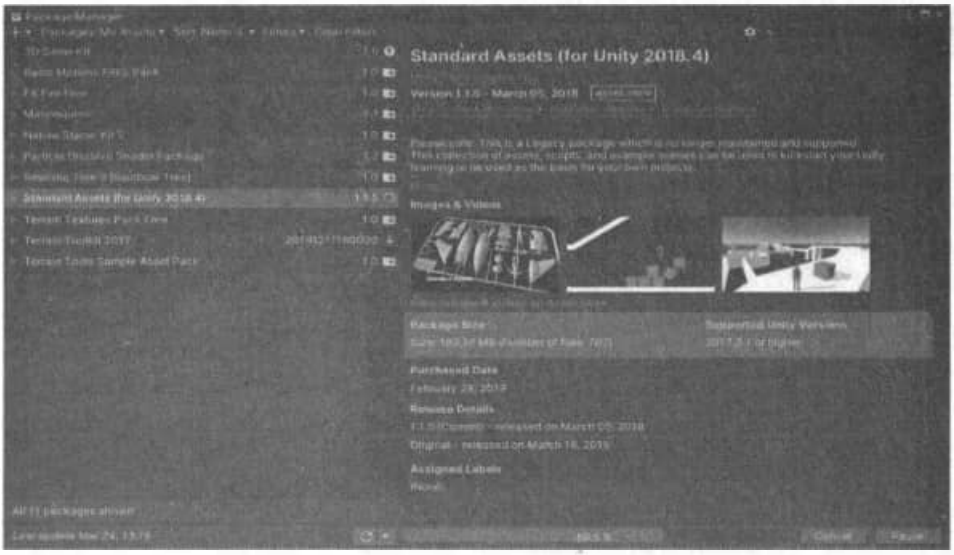


Рис. 1.27. Загрузка ресурсов

После того, как ресурсы будут загружены, их нужно импортировать. Нажмите кнопку **Import** (рис. 1.28). Процесс импорта ресурсов может быть довольно длительным – нужно распаковать их и импортировать.

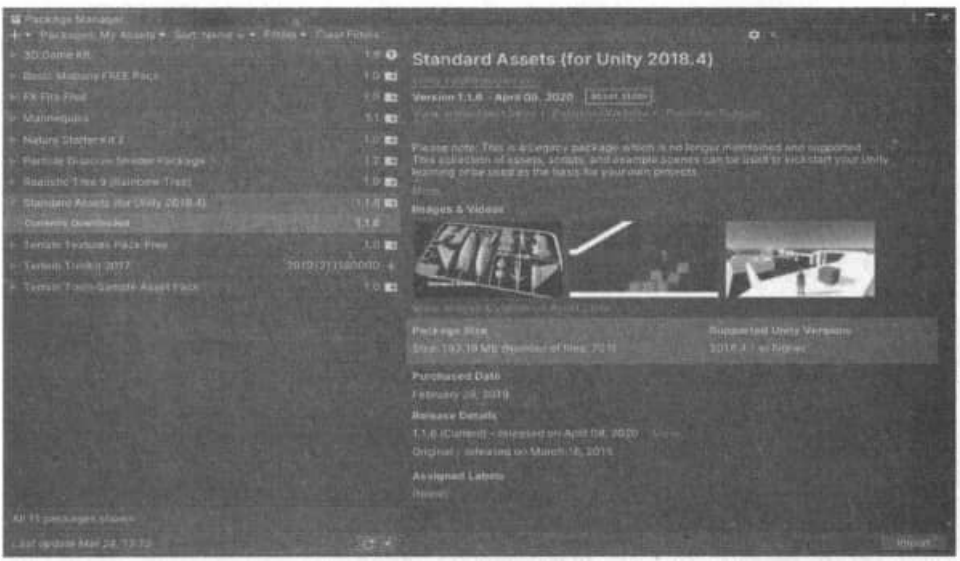


Рис. 1.28. Нажмите кнопку "Import"

Выберите все ресурсы и нажмите кнопку **Import**. В реальном проекте лучше так не делать, поскольку весь пакет ресурсов будет импортирован в ваш проект. Однако в нашем случае, когда хочется попробовать все и по максимуму, можно импортировать все ресурсы.

Рис. 1.29. Импорт ресурсов

1.5.3. Помещение объекта на сцену и изменение его свойств

Когда ресурсы будут импортированы, найдите нужный вам ресурс, например, в **Assets, Standard Assets, Vehicles, Car, Models** находится модель автомобиля. Перетащите ее в область сцены (рис. 1.30).

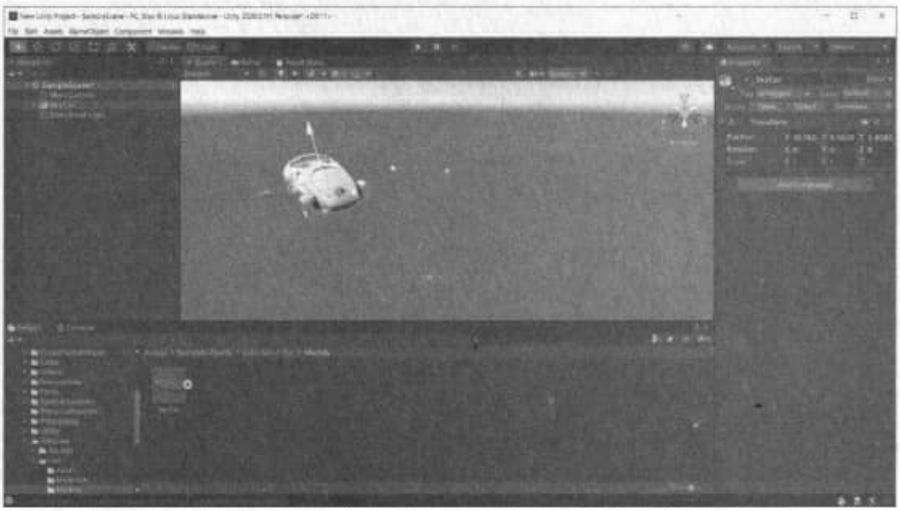


Рис. 1.30. Автомобиль помещен на сцену

Используя оси координат (рис. 1.31), можно вращать сцену, просматривая ее со всех сторон. Вращение колесика мышки позволяет изменить масштаб сцены.

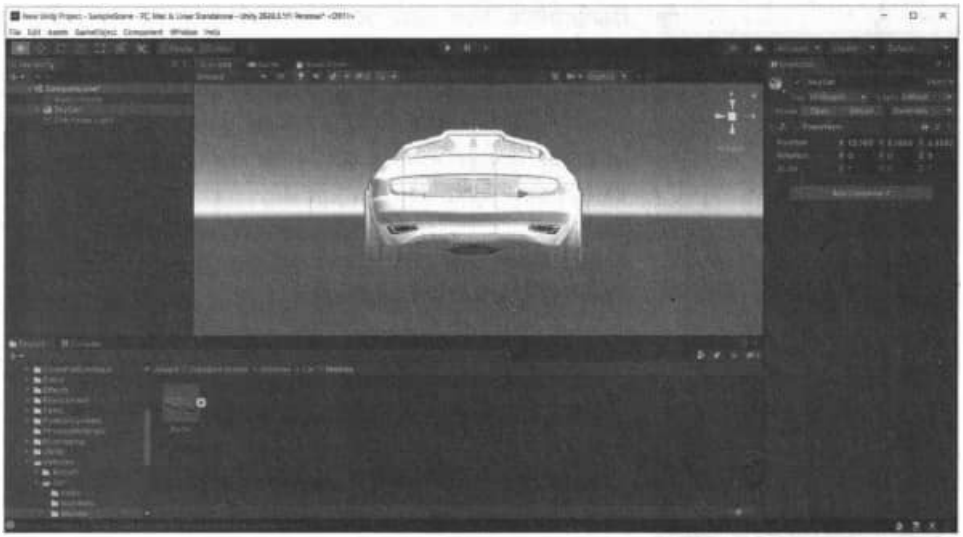


Рис. 1.31. Вращение сцены

Рис. 1.32. Изменен масштаб, добавлена модель самолета

Изменять свойства объекта можно с помощью панели **Inspector** (рис. 1.33). Для изменения свойств объекта его сначала нужно выделить с помощью мышки.

Рис. 1.33. Панель Inspector: отображает свойства выделенного объекта

Для удаления объекта его нужно выделить и нажать кнопку **Delete**.

1.5.4. Создание новой сцены

В Unity сцена означает уровень игры. Одна сцена – один уровень. Сцена и уровень мира могут использоваться здесь взаимозаменяемо. Они просто относятся к 3D-пространству, то есть к пространству-времени игрового мира – месту, где происходят события и существуют вещи.

Для создания новой сцены используется команда меню **File, New Scene**. Можно также нажать Ctrl + N на клавиатуре. Unity создаст пустую сцену. Визуализация сцены отображается на вкладке **Scene** основного окна. Эта вкладка занимает большую часть пространства окна.

Рис. 1.34. Новая сцена создана

Панель **Hierarchy** покажет вам все, что происходит на сцене: модели, свет, камеры и т.д. Эта панель отображает название каждого игрового объекта на сцене. В мире Unity игровой объект (GameObject) относится к одной независимой сущности, которая существует на сцене, независимо от того видима она или нет. Выделить объект можно путем или щелчка мышью по самому объекту на сцене или на панели **Hierarchy**, если объект на сцене не виден. Редактировать свойства выделенного объекта можно на панели **Inspector**, как было показано ранее.

После создания сцены давайте создадим пол (этаж), иначе все объекты повиснут в пространстве. При создании космического шутера может быть это и допустимо, но во всех остальных случаях нужно позаботиться о поле или местности. Здесь все зависит от создаваемой игры. Можно использовать префаб Floor* из стандартного набора ассетов или же создать объект **Terrain**, если вам нужна местность (например, вы создаете игру, действие которой происходит в лесу или на какой-то далекой планете – с помощью текстур вы легко можете изменять тип местности).

Пол, как полноценную модель, можно построить с помощью 3Ds Max, Blender или Maya. Но на данном этапе гораздо быстрее и проще использовать пакет **Unity Standard Assets** и использовать стандартный пол. Для этого загляните в раздел ресурсов **Prototyping, Prefabs** (рис. 1.35).

Да, пусть это будет не очень интересный и пустой пол, но на его создание вы не потратили никакого времени (рис. 1.35). Как только пол добавлен, он появится на панели **Hierarchy**, а на панели **Inspector** можно будет изменить его свойства.

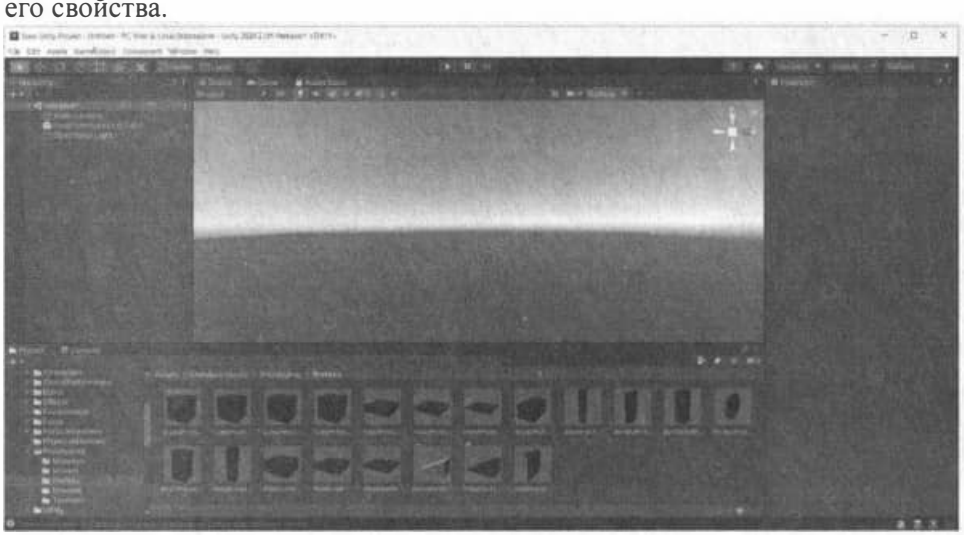


Рис. 1.35. Пол добавлен

По умолчанию наш пол называется примерно так FloorPrototype64x01x64 (название зависит от выбранного объекта). Давайте переименуем его в WorldFloor. Для этого сначала выберите этот объект на панели иерархии, а затем переименуйте на панели инспектора.

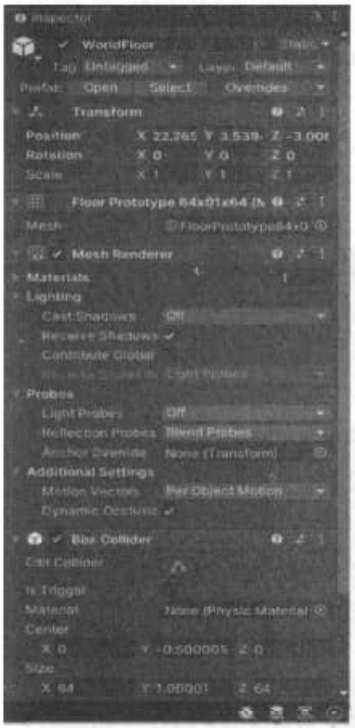


Рис. 1.36. Переименование объекта

1.5.5. Трансформация и навигация

В прошлом пункте мы создали пол, но пока в нашем проекте нет ничего интересного. Давайте добавим дополнительные объекты – здания, лестницы, колонны и прочее.

Каждая точка и местоположение в сцене уникально идентифицируется координатой, измеренной как смещение (X, Y, Z) от центра мира (начала координат). Текущая позиция всегда отображается в инспекторе объектов. По сути, параметры **Position** (позиция), **Rotation** (вращение) и **Scale** (масштаб) сгруппированы в один компонент, который называется **Transfom** (трансформация).

Параметр **Position** показывает, насколько далеко объект находится от центра мира по трем осям.

Параметр **Rotation** показывает, насколько объект будет повернут вокруг собственной центральной оси.

Параметр **Scale** определяет масштаб объекта. Он позволяет задать, насколько объект будет отличаться от своего оригинального размера. Например, 0.5 – это половина размера.

Вместе параметры **Position, Rotation, Scale** определяют трансформацию объекта. Для изменения позиции выделенного объекта вы можете просто ввести новые значения в поля X, Y, Z параметра **Position**. Для перемещения объекта к центру мира просто введите координаты (0, 0 и 0). При желании, конечно, можно устанавливать позицию объекта вручную путем ввода координат, но гораздо проще делать это с помощью мышкой. Просто возьмите и переместите объект в заданную позицию. Чтобы упростить этот процесс используйте инструмент **Move Tool** – кнопка с изображением стрелок (или нажмите клавишу **W**). После активации этого инструмента на сцене поя-

вится трехцветная ось координат: красный, зеленый и синий соответствуют осям X, Y, Z соответственно.

Чтобы переместить объект, наведите курсор на одну из трех осей (или плоскостей между осями), а затем нажмите и удерживайте мышь, перемещая ее, чтобы сдвинуть объект в этом направлении. Вы можете повторять этот процесс столько, сколько это необходимо, чтобы ваши объекты располагались в нужном месте.

С помощью клавиатуры можно активировать и другие инструменты, например, **E** – активирует инструмент вращения, а **R** – масштаба.

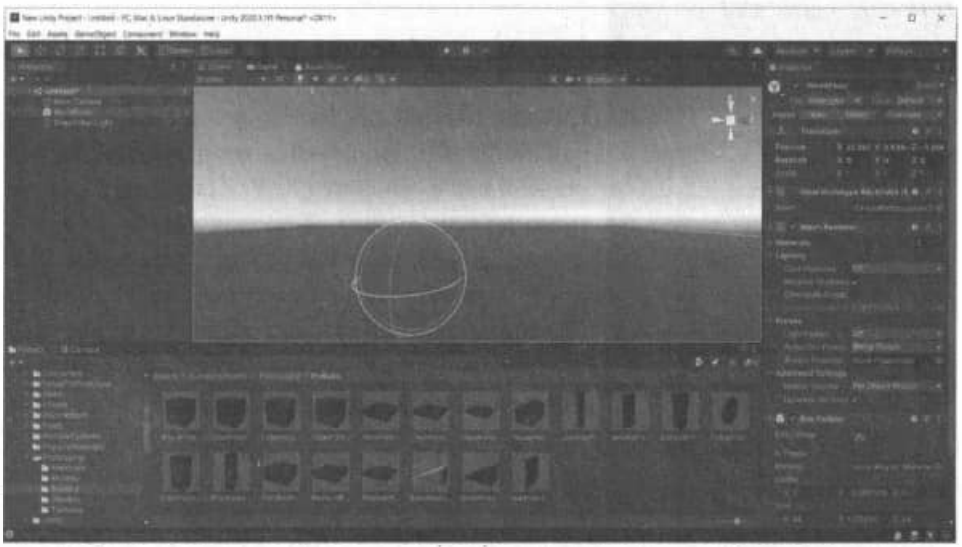


Рис. 1.37. Активирован инструмент вращения

Однако в дополнение к перемещению, вращению и масштабированию объектов вам часто нужно перемещаться вокруг себя, чтобы увидеть мир с разных позиций, ракурсов и перспектив. Это означает, что вам часто придется перемещать камеру предварительного просмотра сцены в мире. Уменьшайте или увеличивайте масштаб, изменяйте угол обзора, чтобы видеть, правильно ли объекты совмещаются друг с другом. Для этого активируйте инструмент **Hand Tool** (кнопка с изображением руки). После этого нажмите **Alt** и, «ухватившись» мышкой за пространство, вы можете изменять угол обзора. Колесико мыши в этом режиме изменяет масштаб.

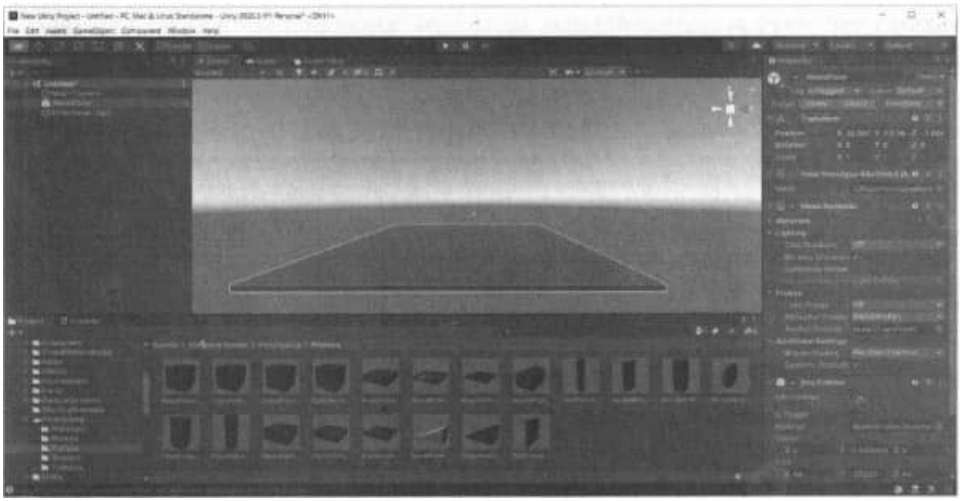


Рис. 1.38. Hand Tool в действии

1.5.6. Построение сцены

Теперь давайте добавим на наш пол некоторые другие модели – вид пустого пространства не радует. Найти модели домов можно в **Assets, Standard Assets, Prototyping, Prefabs**. Используйте модель **HousePrototype16x16x24.** Поместите на пол несколько объектов. Убедитесь, что объекты помещены на пол, а не висят в воздухе (рис. 1.39). Для этого используйте инструменты трансформации и вращения.

Рис. 1.39. Прототип дома помещен на пол

Можно добавить дополнительные элементы вроде StepsPrototype. Опять-таки, при помещении ступенек убедитесь, что они прилегают к зданию, а не просто стоят рядом.

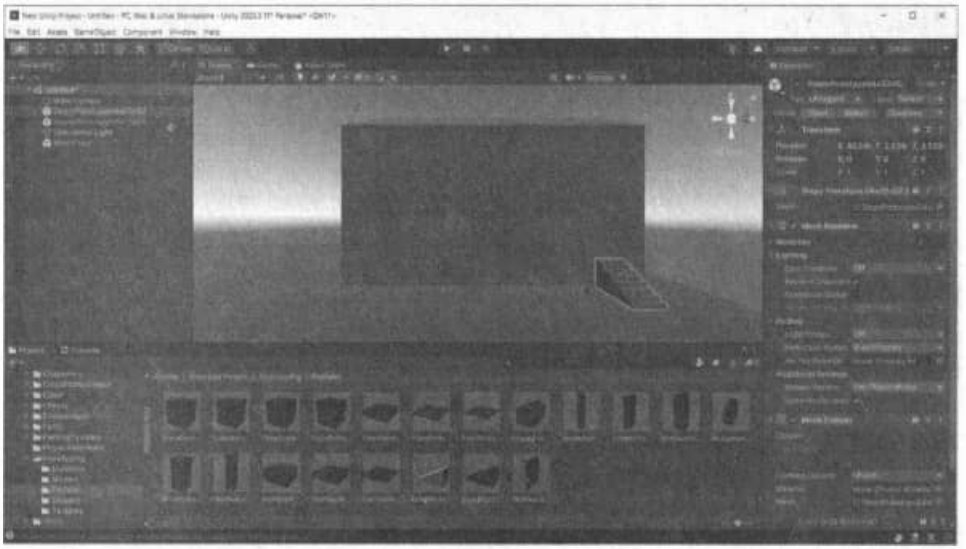


Рис. 1.40. Ступеньки к зданию

На рис. 1.41 показано, что на сцене есть два здания, одно из которых со ступеньками. Чтобы не путаться, сразу присвойте зданиям понятные имена. Посмотрите сверху, как расположены здания.

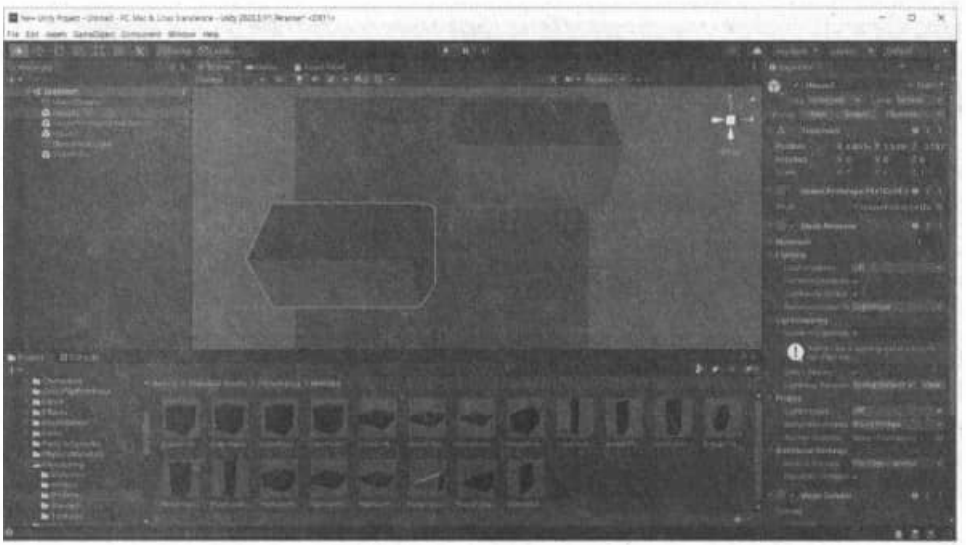


Рис. 1.41. Два здания и ступеньки (вид сверху)

Примечание. Обычно при помещении здания на пол оно сразу же хорошо с ним совмещается. Проверьте, чтобы это было действительно так: профессиональные разработчики не могут полагаться на «обычно». Иначе получится, что некоторые здания втоплены в пол, а некоторые – висят в воздухе.

Теперь вы можете построить полную сцену, используя модели из пакета **Prototyping**. С помощью перемещения, поворота и масштабирования вы можете перемещать, выравнивать и вращать эти объекты. Попрактикуйтесь немного и все у вас получится!

Добавьте еще несколько объектов – здания, кубы и т.д. Сцена не должна выглядеть пустой.

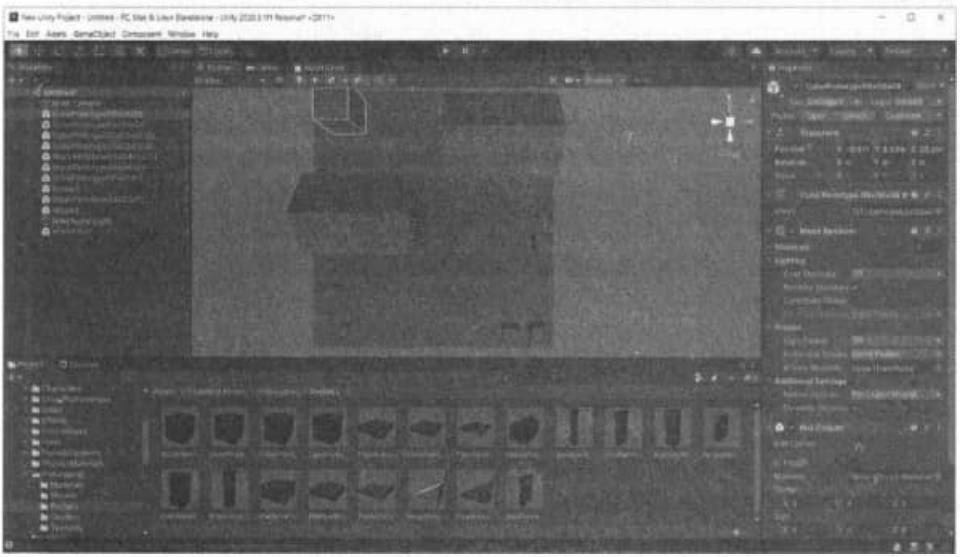


Рис. 1.42. Сцена, дополнительные объекты

Примечание. Любые программы для работы с трехмерными объектами довольно сильно нагружают процессор компьютера. Поэтому настоятельно рекомендую для работы с ними использовать стационарные компьютеры – на них в случае перегрева всегда можно открыть крышку системного корпуса и получить дополнительное охлаждение. С ноутбуками такой трюк не пройдет, а подставки с кулерами, как показывает практика, не особо эффективны. Летом ваш ноутбук может запросто перегреться и выключиться при работе с Unity.

1.5.7. Свет и небо

Мы только что разместили какие-то базовые архитектурные модели. Для этого мы использовали всего лишь несколько ресурсов из стандартного набора и некоторые основные инструменты. Тем не менее, эти инструменты являются мощными и предлагают нам множество комбинаций и опций для создания большого разнообразия и правдоподобия в игровых мирах. Но все же нам не хватает одного важного ингредиента.

Таким ингредиентом является свет. Посмотрите на предыдущие иллюстрации. Все выглядит просто плоским. Нет ни теней, ни светлых или темных областей. Неправдоподобно. Это потому, что свет на сцене еще не настроен для получения лучших результатов.

Включить/выключить отображение неба можно с помощью переключателя Skybox (рис. 1.43). На слабых процессорах можно отключать отображение неба во время проектирования сцены. Как все будет готово, нужно будет приступить к настройке неба и света.

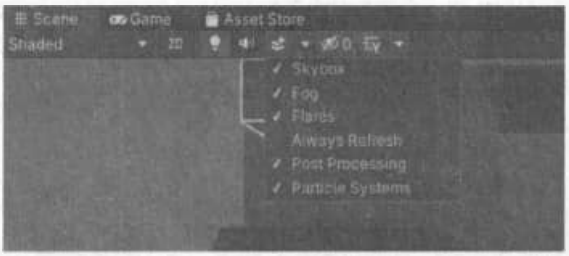


Рис. 1.43. Включение/выключение неба

После включения неба (если оно было у вас выключено) мир вокруг станет лучше, но все же нам не хватает бликов и теней. Используйте кнопку-переключатель с изображением лампочки. Этот параметр предназначен только для отображения. Он влияет только на то, отображаются ли световые эффекты в окне просмотра сцены, а не на свет в финальной игре.

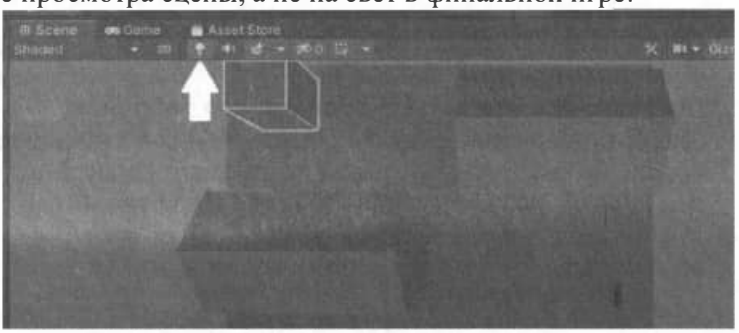


Рис. 1.44. Управление светом и тенями на сцене

Включение подсветки для окна просмотра приведет к некоторым различиям во внешнем виде сцены, и, опять же, сцена должна выглядеть лучше, чем раньше.

Чтобы убедиться, что освещение сцены действует, выберите **Directional Light** на панели иерархии и поверните его. Так вы сможете контролировать время суток, вращая световые циклы между днем и ночью и изменяя интенсивность света и настроение. Это меняет способ визуализации сцены.

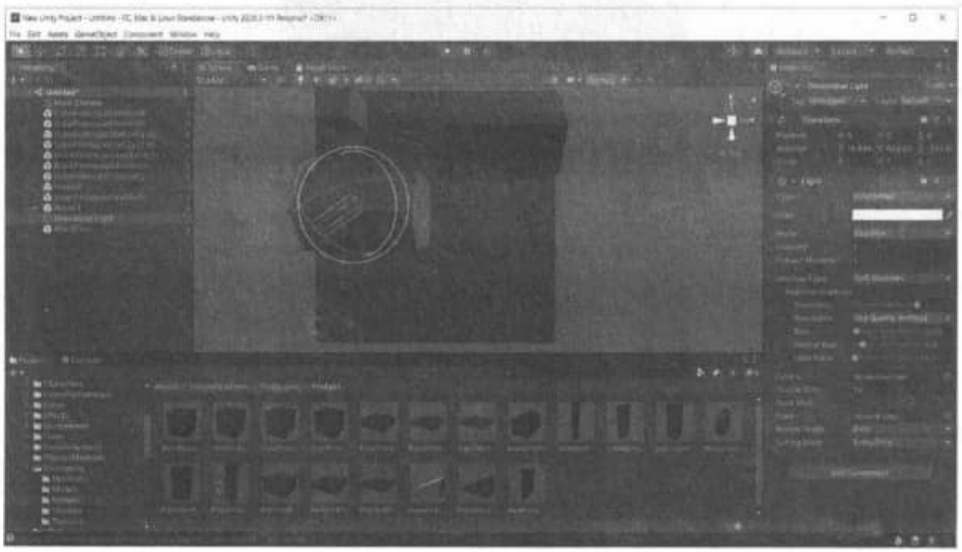


Рис. 1.45. Изменяем направленный свет

Для создания оптимального света все неперемещаемые объекты (здания, деревья и т.д.) должны быть помечены как **Static**. Для этого выделите объект на панели иерархии и на панели инспектора включите переключатель **Static**. Чтобы было быстрее, выделите несколько объектов на панели иерархии при нажатой клавише Shift: так можно выделить несколько объектов сразу. После этого включите переключатель **Static** один раз – все выделенные объекты будут статическими.

Когда вы сделаете неперемещаемые объекты статическими, Unity автоматически вычисляет свет на сцене в фоновом режиме (именно поэтому создается такая нагрузка на процессор). Также она сгенерирует набор данных, называемых **GI Cache**, которые говорят Unity как лучи света должны отражаться и распространяться по сцене для достижения максимального реализма. Однако даже после включения переключателя **Static**, тени у вас не появятся. По

умолчанию свойство **Cast Shadows** отключено для всех объектов. Включите его для всех статических объектов.

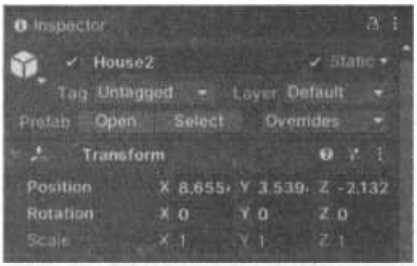


Рис. 1.46. Делаем объект статическим

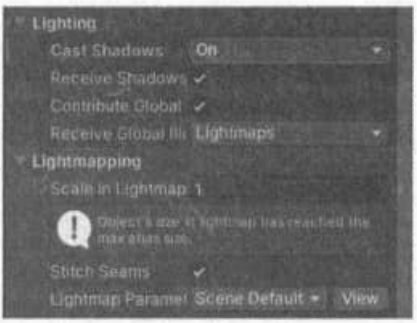


Рис. 1.47. Включение свойства Cast Shadows

Также включите свойство **Receive Shadows**, чтобы объект мог принимать тень от соседнего объекта.

Перейдите к объекту **Main Camera** и воспользуйтесь окном **Camera Preview** – вы увидите, что теперь у наших объектов появились тени.

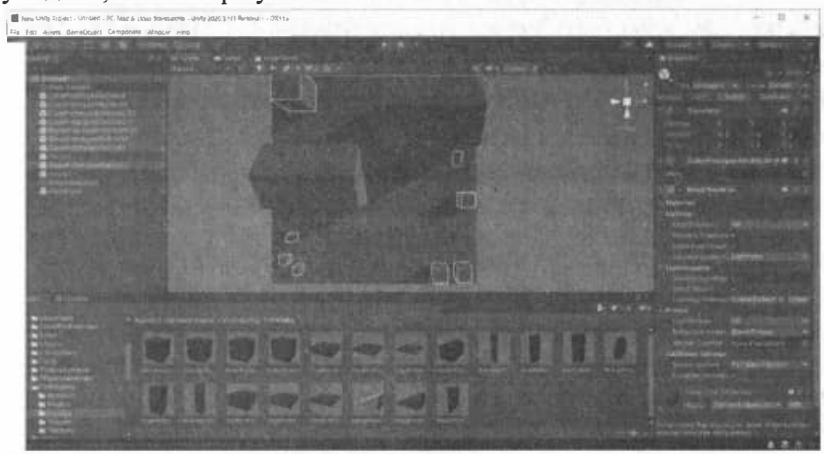


Рис. 1.48. Тени включены

1.5.8. Тестовый запуск

Еще раз посмотрите на сцену. Если все хорошо, самое время ее сохранить: нажмите для этого Ctrl + S или выберите команду меню **File, Save Scene**.

Теперь настало время магии. На данный момент наш уровень не содержит ничего играбельного. Это просто статическое, безжизненное и неинтерактивное 3D-пространство. Давайте сделаем нашу сцену играбельной, чтобы мы могли обойти ее вокруг от первого лица с помощью стандартных клавиш WASD на клавиатуре. Для этого нужно добавить контроллер первого лица (first-person mode).

Откройте папку **Assets, Standard Assets, Characters, FirstPersonCharacter, Prefabs**. Перетащите ресурс FPSController (рис. 1.49).

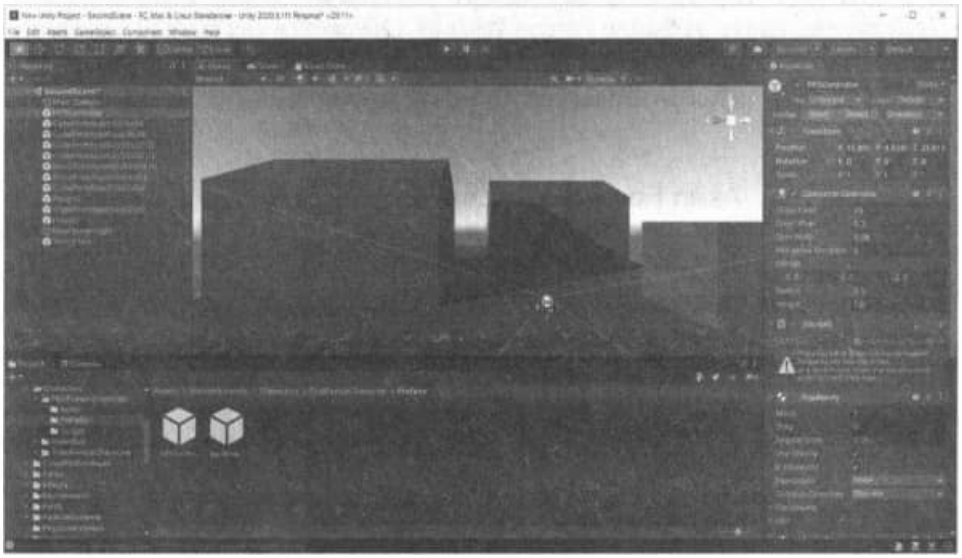


Рис. 1.49. Добавлен ресурс FPSController

> **Внимание!** FPSController нужно установить на пол правильно – чтобы персонаж стоял на «земле», а не висел в воздухе.

После добавления контроллера первого лица нажмите кнопку **Play** на панели инструментов Unity для тестового запуска игры в режиме от первого лица.

Если вы используете версию 2019.3/2020.3, вы увидите на вкладке **Console** сообщения об ошибках. В версии 2018.x все работало нормально. Но в данном случае нужно как-то выходить из ситуации. Ошибка, связанная со стан-

дартным набором ресурсов – их забыли адаптировать под новую версию 2019.3/2020.3. Поправить довольно просто, если вы обладаете навыками программирования на C#, а если нет, то просто следуйте инструкциям далее.

Рис. 1.50. Сообщения об ошибках. Игра не запускается

Перейдите на вкладку **Project**. Перейдите в **Assets, Standard Assets, Utility**. Дважды щелкните на файле ForcedReset.cs. Откроется связанный с расширением .cs текстовый редактор. Возможно, это будет Visual Studio, возможно, другой выбранный вами редактор. Отредактируйте файл так, как показано в лист. 1.1.

Листинг 1.1. Файл ForcedReset.cs

```
using System;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityStandardAssets.CrossPlatformInput;
using UnityEngine.UI;

[RequireComponent(typeof (Image))]
public class ForcedReset : MonoBehaviour
{
    private void Update()
    {
        // if we have forced a reset ...
      if (CrossPlatformInputManager.GetButtonDown("ResetObject"))
        {
         //... reload the scene
         SceneManager.LoadScene(SceneManager.GetSceneAt(0).name);
        }
    }
}
```

Аналогичные изменения необходимо произвести в файле SimpleActivatorMenu.cs (лист. 1.2).

Листинг 1.2. Файл SimpleActivatorMenu.cs

```
using System;
using UnityEngine;
using UnityEngine.UI;

namespace UnityStandardAssets.Utility
{
    public class SimpleActivatorMenu : MonoBehaviour
    {
     // An incredibly simple menu which, when given references
     // to gameobjects in the scene
        public Text camSwitchButton;
        public GameObject[] objects;


        private int m_CurrentActiveObject;


        private void OnEnable()
        {
            // active object starts from first in array
            m_CurrentActiveObject = 0;
            camSwitchButton.text = objects[m_
CurrentActiveObject].name;
        }


        public void NextCamera()
        {
            int nextactiveobject = m_CurrentActiveObject + 1
>= objects.Length ? 0 : m_CurrentActiveObject + 1;

            for (int i = 0; i < objects.Length; i++)
            {
                objects[i].SetActive(i ==
nextactiveobject);
            }

            m_CurrentActiveObject = nextactiveobject;
            camSwitchButton.text = objects[m_
CurrentActiveObject].name;
        }
```

```
    }
}
```

После этого ошибки на вкладке **Console** исчезнут, начнется сборка игры.

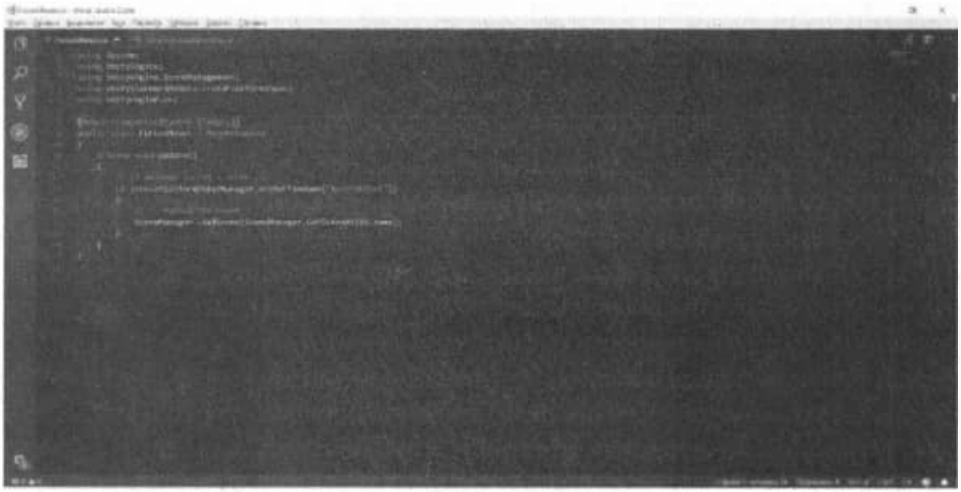


Рис. 1.51. Редактирование исходного кода

В режиме от первого лица (рис. 1.52) сразу заметны все косяки, возможно допущенные при проектировании сцены, если этого вы не заметили ранее. Например, некоторые предметы могут висеть в воздухе, а не находиться на WorldFloor, ступеньки могут не примыкать к зданию и т.д.

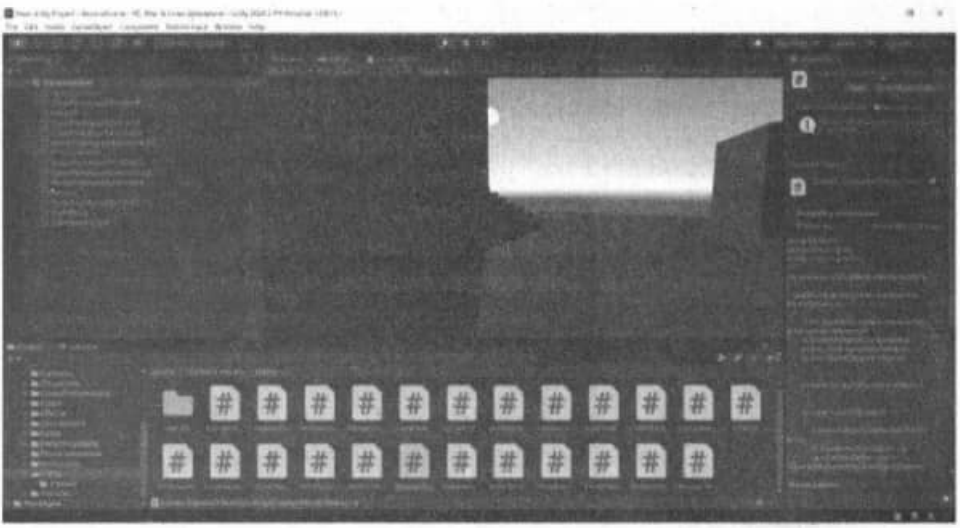


Рис. 1.52. Режим от первого лица

Пока не обращайте внимания на них – прогуляйтесь по созданному миру. Для перемещения игрока используйте стандартные клавиши WASD, а также мышку для поворота головы. Для выхода из режима игры нажмите Ctrl + P.

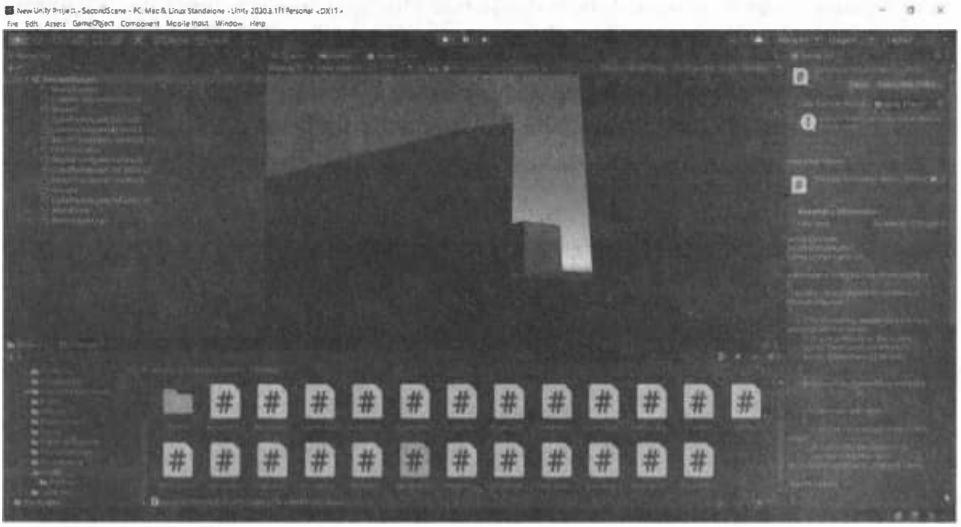


Рис. 1.53. Режим игры: прогулка по миру

Примечание. В режиме игры можно переключиться на вкладку Scene и изменить параметры любого объекта! Другими словами, косяки можно править «в режиме реального времени».

Итак, мы создали небольшой, пусть несовершенный игровой мир, по которому может перемещаться игрок с помощью стандартных средств управления (клавиатура и мышь) и при этом не написали ни одной строчки кода (если не считать вынужденных правок файлов ресурсов)! Просто изумительно!

Панель **Console** (рис. 1.50) выводит сообщения трех типов: информационные, предупреждения и ошибки. Информационные сообщения, как правило, представляют собой способ предоставления рекомендаций и предложений от Unity, основанных на том, как работает ваш проект.

Предупреждения немного серьезнее и представляют проблемы в вашем коде или сцене, которые (если не исправить) могут привести к неожиданному поведению и неоптимальной производительности.

Ошибки описывают области в вашей сцене или коде, которые требуют тщательного и немедленного внимания. Иногда ошибки не позволяют вашей игре работать вообще, а иногда ошибки происходят во время выполнения и могут привести к сбою или зависанию игры.

1.5.9. Добавляем немного воды

На данный момент незадействованное пространство никак не отображается. Оно просто залито однотонным цветом и это, разумеется, нас не устраивает.

Чтобы добавить воду, мы можем использовать уже готовый ресурс. Перейдите в папку **Standard Assets, Environment, Water, Water, Prefabs**. Перетащите ресурс **WaterProDaytime** с панели **Project** на сцену. Вы заметите, что объект меньше, чем нам нужно (рис. 1.54).

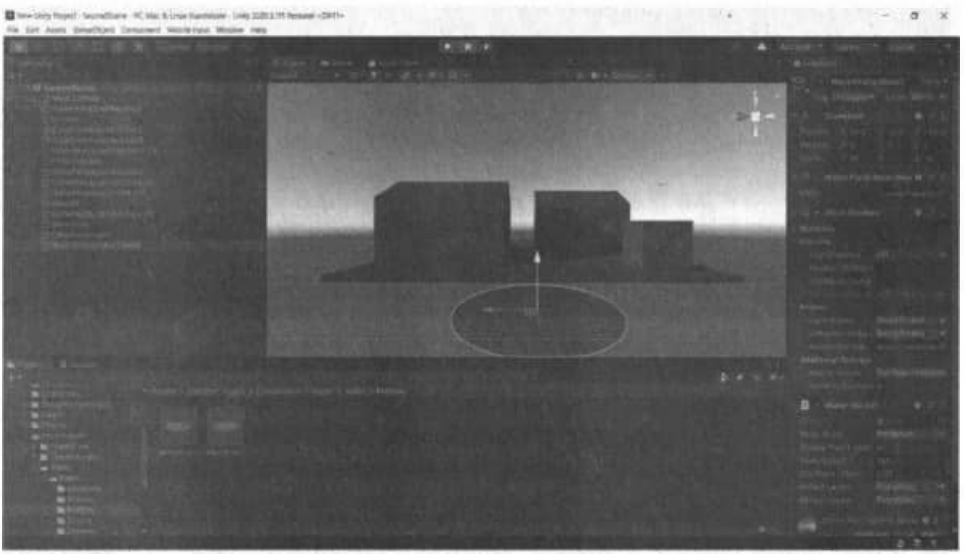


Рис. 1.54. Добавлена вода

После добавления префаба Water поместите его ниже уровня пола и используйте инструмент масштабирования (R) для увеличения его размера (X, Z) так, чтобы заполнить окружающую среду вокруг вплоть до горизонта. Это создает ощущение, что сетки пола – это маленький остров в обширном мире воды. У вас должно получиться примерно так, как показано на рис. 1.55.

Теперь запустите игру. Как видите пространство заполнено вокруг водой, цвет которой зависит от настроек света. Это уже что-то.

Если вы попытаетесь пройти по воде, вы просто провалитесь сквозь нее, опускаясь в бесконечность, как будто воды там никогда не было. Сейчас вода – полностью косметическая особенность, но она делает сцену намного лучше. Но не все сразу – для небольшого интро вполне достаточно даже тех

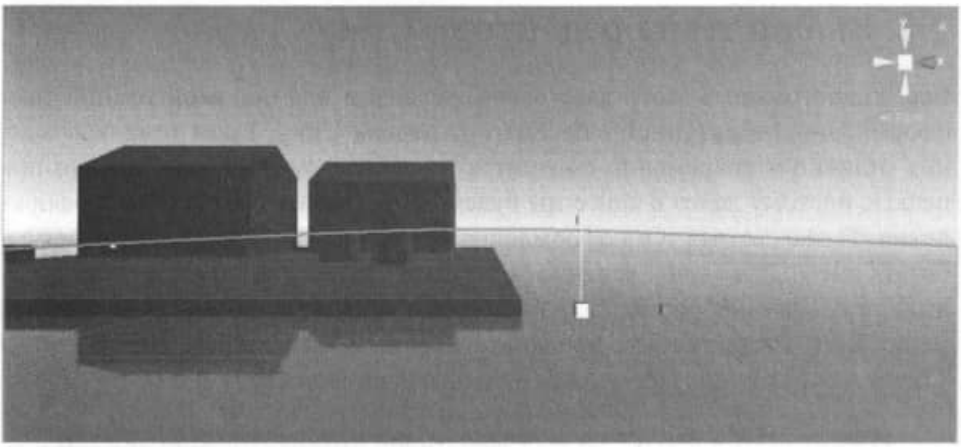


Рис. 1.55. Остров в море

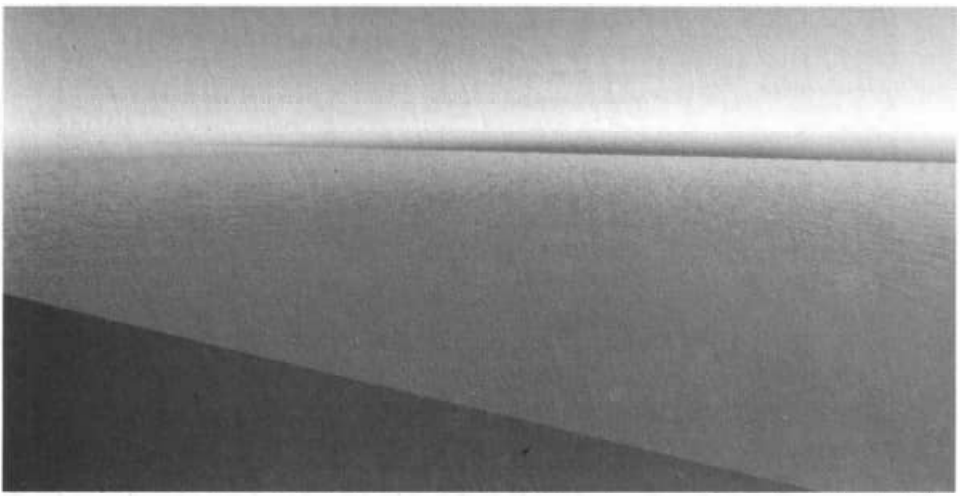


Рис. 1.56. Режим от первого лица. Вода

возможностей Unity, что мы рассмотрели в первой главе. В следующей главе мы поговорим о работе с ассетами. В частности, будет подробно рассмотрен процесс импорта ассетов в проект.

1.6. Выбор темы редактора Unity

Все иллюстрации в этой главе были сделаны в премиальной темной теме оформления, недоступной в бесплатной версии Unity. Такая тема оформления отлично и современно смотрится на экране, но не очень хорошо при печати, поэтому далее в книге мы будем использовать светлую тему оформления. Также светлая тема оформления лучше для тех, кто предпочитает работать днем. Для «сов», конечно же, лучше темная тема оформления. Сменить тему оформления можно в настройках редактора – выполните команду меню **Edit, Preferences** и перейдите в раздел **General**. Параметр **Editor Theme** как раз и отвечает за выбор темы оформления (рис. 1.57).

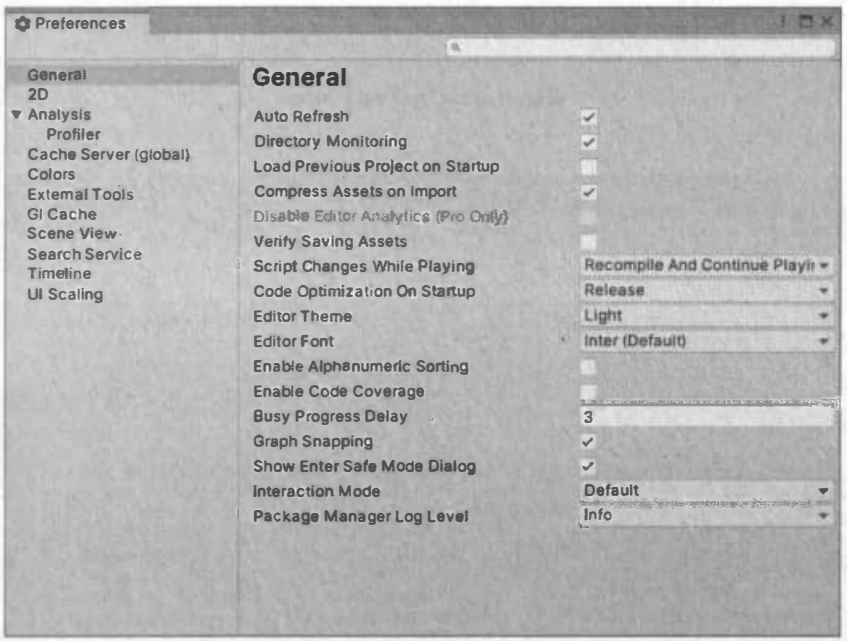
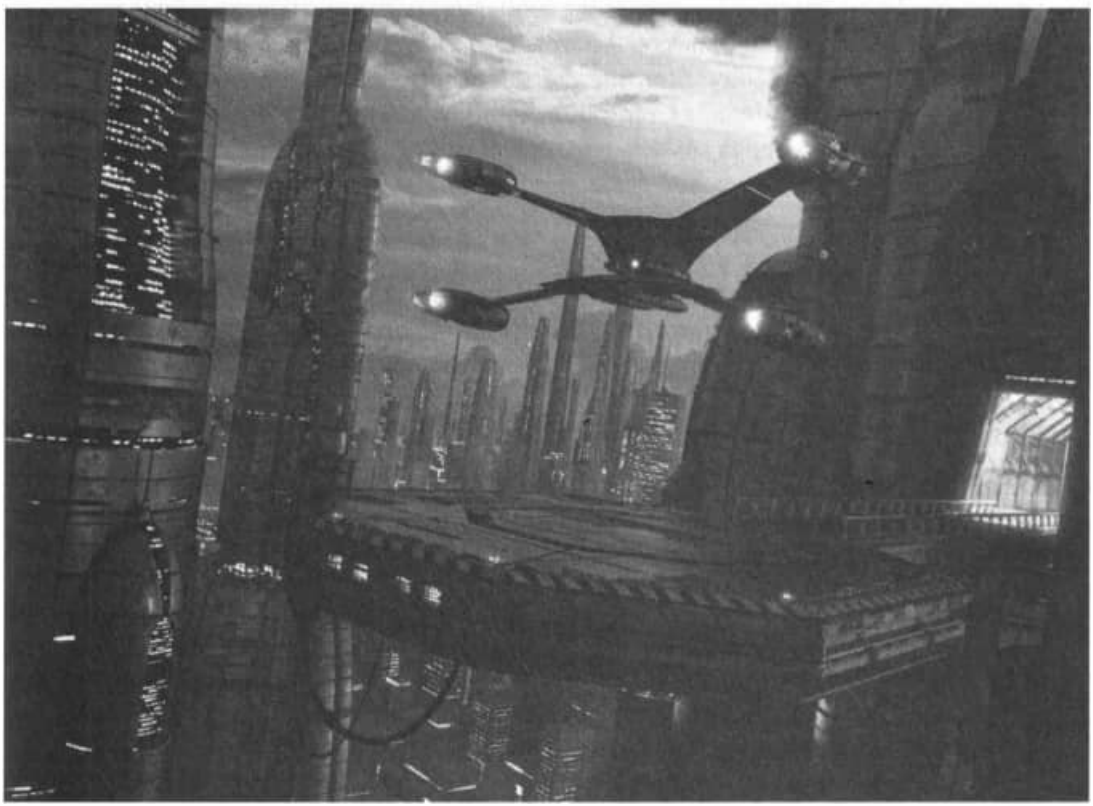


Рис. 1.57. Смена темы редактора Unity

Глава 2.

Работа с ассетами



В этой главе мы поговорим о работе с ассетами[1] (ресурсами) в Unity. Ассетом считается представление любого предмета, который можно использовать в вашей игре или проекте. Ассет может быть получен из файла, созданного за пределами Unity. Конечно, есть и ассеты, которые можно создать в Unity, например, Animator Controller, Audio Mixer, Render Texture. Ассетом может быть трехмерная модель (как правило, распространяются в файлах с расширением .fbx), звуковой файл, шрифт и т.д.

2.1. Графические примитивы

Редактор Unity может работать с 3D моделями любой формы, создаваемыми в приложениях для моделирования. Некоторые примитивные модели можно создать прямо в Unity: Куб (Cube), Сфера (Sphere), Капсула (Capsule), Цилиндр (Cylinder), Плоскость (Plane) и Квад (Quad).

Данные объекты часто применяются как есть (плоскость обычно используется в качестве поверхности земли, например), но также они позволяют быстро создать прототипы сложных объектов в целях тестирования. Любой из примитивов может быть добавлен в сцену с помощью соответствующего пункта в меню **GameObject > 3D Object**.

Не нужно списывать их со счета. Так же плоскость часто используется для создания графического интерфейса пользователя, о чем будет рассказано далее.

1 От анг. asset

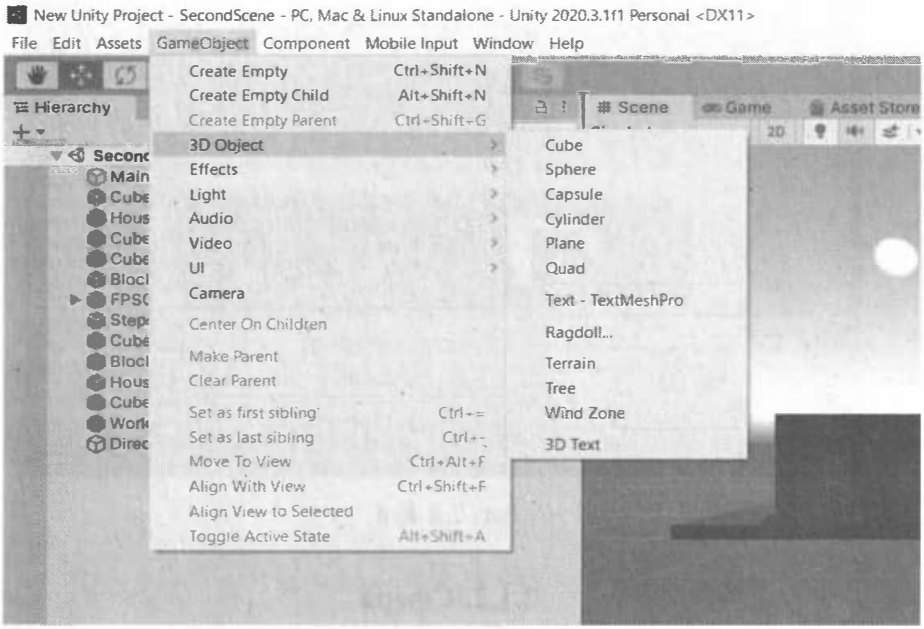


Рис. 2.1. Команда меню GameObject > 3D Object

2.1.1. Куб

Объект **Куб** (Cube) представляет собой куб с длиной ребра в одну единицу измерения. Текстура куба повторяется на каждой из его 6 граней. Непосредственно сам куб используется в играх редко, но его можно текстурировать и использовать в самых разных целях:

- В качестве ящиков
- Столбов
- Ступенек (хотя в стандартных ассетах есть отдельный объект, представляющий ступеньки, о чем было рассказано ранее)

Для нас, как для разработчиков, куб может послужить удобной заменой конечной модели, если она еще не готова. Например, куб, вытянутый до нужного размера, можно использовать вместо модели персонажа. В финальной версии игры это недопустимо, но для тестирования кода управления персонажем – вполне уместно.

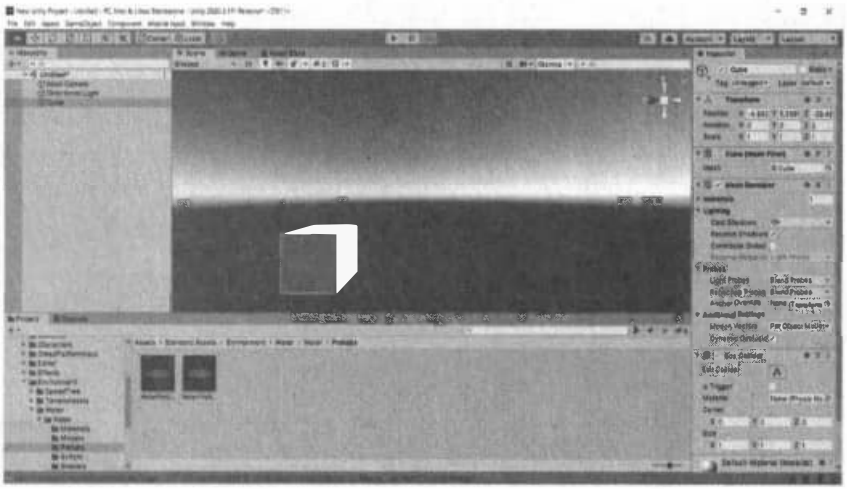


Рис. 2.2. Куб

2.1.2. Сфера

Следующий примитив – **сфера** (Sphere). Она текстурируется так, чтобы изображение огибало сферу один раз, при этом верхняя и нижняя границы картинки будут находиться на полюсах сферы. Очевидно, сферы полезны для создания мячей, планет и снарядов. При создании космической стрелялки, сферы – незаменимый элемент, поскольку очень просто с их помощью создать изображение планет, «натянув» на сферы разные текстуры.

Рис. 2.3. Сфера

2.1.3. Капсула

Капсула (Capsule) – это объект, представляющий собой цилиндр с полусферическими колпаками на концах. Объект имеет диаметр в одну единицу измерения и высоту в две единицы измерения (цилиндр высотой в одну единицу и оба колпака по половине единицы каждый).

Капсула текстурирована так, чтобы изображение огибало ее ровно один раз, с закреплением в вершине каждой из полусфер. В реальном мире и не так много объектов с такой формой, поэтому капсулу можно использовать только для прототипирования. В частности, в некоторых задачах физика закругленного объекта предпочтительнее, чем таковая у прямоугольного. Поэтому капсулу можно с успехом использовать вместо модели персонажа, которая еще не готова.

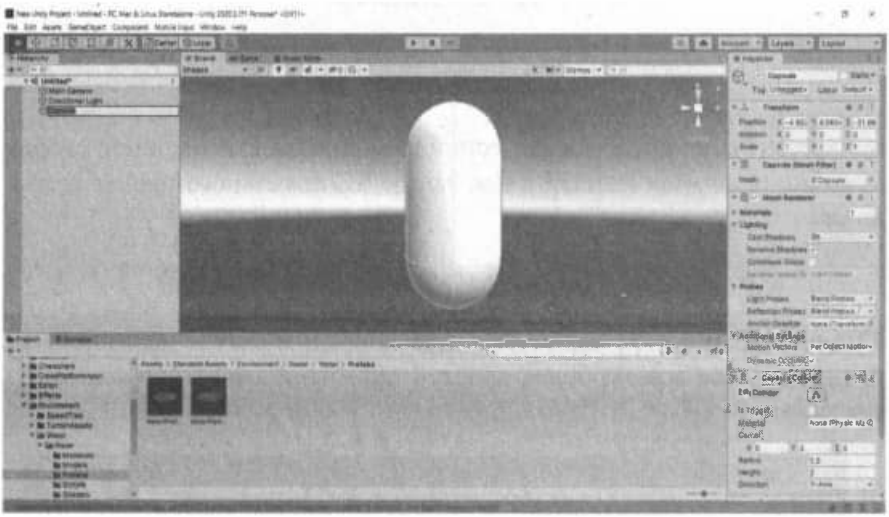


Рис. 2.4. Капсула

2.1.4. Цилиндр

Цилиндр (Cylinder) похож на капсулу, по сути, капсула – этот тот же цилиндр, только с полусферическими колпаками на концах. Цилиндры удобно применять для создания столбов, труб и колес, но вы можете заметить, что форма коллайдера на самом деле – капсула (в Unity нет коллайдера в форме цилиндра). Применять цилиндр с успехом можно для создания различных столбов в игре.

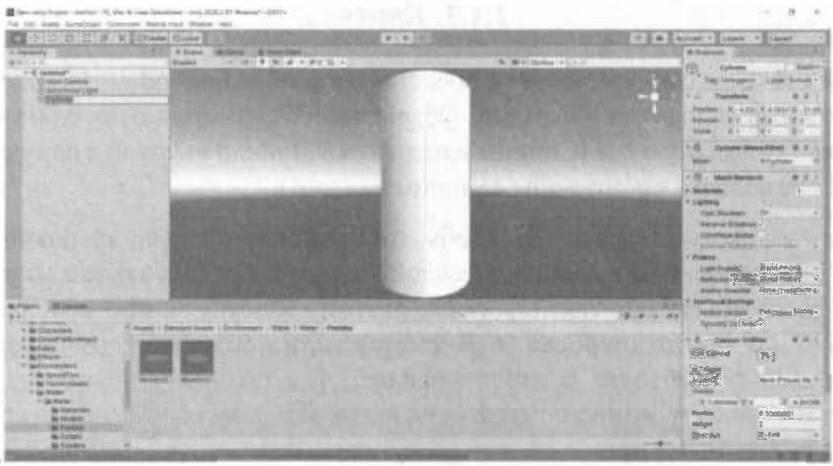


Рис. 2.5. Цилиндр

2.1.5. Плоскость

Плоскость (Plane) представляет собой плоский квадрат с длиной стороны в 10 единиц, ориентированный в плоскости XZ локального пространства координат.

Плоскость текстурирована так, чтобы все изображение точно заполнило квадрат. Данный объект удобен для отображения картинок и видео в GUI, и для различных спецэффектов. Хоть плоскость и может использоваться для описанных выше вещей, более простой примитив квад чаще подходит для таких задач.

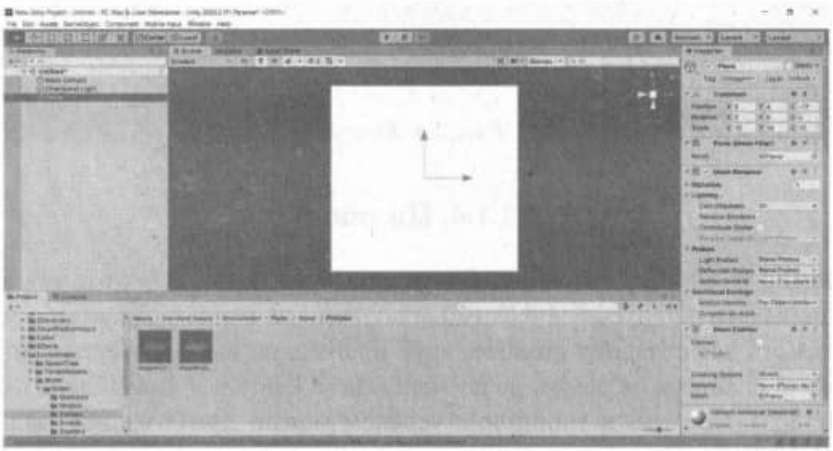


Рис. 2.6. Плоскость

2.1.6. Квад

Объект **квад** (Quad) похож на плоскость, но его стороны имеют длину одной единицы измерения, и поверхность ориентирована по плоскости XY локального пространства координат. Также, квад состоит только из двух треугольников, в то время как плоскость – из двухсот. Квад будет полезен, если в сцене требуется создать экран для вывода изображений или видео. Посредством квада можно создать простые информационные и GUI экраны, а также частицы, спрайты и «обманные» изображения для имитации удаленных больших объектов. Меню игры с кнопками можно легко сконструировать с помощью квадов. Хотя и плоскость отлично подходит для этого. Все зависит от того, какое меню вы хотите видеть – прямоугольное или квадратное.

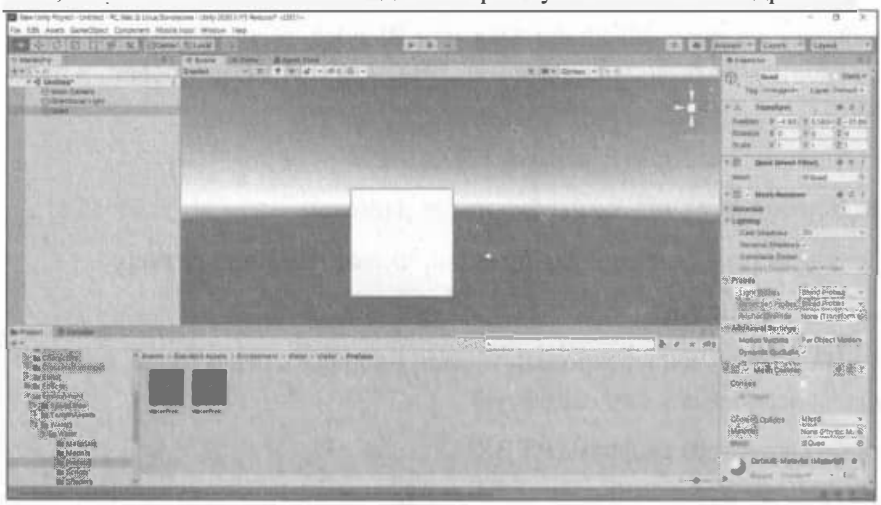


Рис. 2.7. Квад

2.2. Некоторые типы ассетов

2.2.1. Трехмерные модели

Безусловно, самым важным типом ресурсов являются трехмерные модели. Модели можно загрузить из Assets Store или создать самостоятельно в одном из популярных 3D редакторов.

Создать трехмерные модели можно в следующих 3D редакторах:

- Maya
- Cinema 4D

- 3ds Max
- Cheetah3D
- Modo
- Lightwave
- Blender
- Cheetah3D

Перед импортом модели ее нужно экспортировать в форматы .FBX или .OBJ. Также поддерживаются собственные форматы приложений .Max (3D Studio Max) и .Blend (Blender).

Если вы используете 3D Studio Max или Blender, то у вас есть два выбора: или экспортировать модель в формат .FBX или же импортировать в родном формате.

Преимущества экспорта в формат .FBX:

- Экспортируются только необходимые данные
- Модели в этом формате, как правило, имеют меньший размер
- Модульный подход
- В FBX можно конвертировать модели из других редакторов, которые не поддерживаются Unity напрямую

Недостатки экспорта в формат .FBX:

- Процесс прототипирования может быть замедлен
- Легче потерять след между исходной (рабочий файл) и игровой версией данных

К преимуществам использования собственных форматов (.Max, .Blend) можно отнести простоту: ничего не нужно делать, просто скопировать файл в папку Assets.

А вот недостатков гораздо больше:

- На машинах, задействованных в работе над Unity проектом, должны быть установлены лицензионные копии данного программного обеспечения
- Файлы, содержащие ненужные данные могут стать неоправданно большими
- Большие файлы могут замедлить процесс автосохранения

- Меньше проверяется, поэтому труднее устранить ошибки

Файлы моделей, размещенные в папке Assets внутри проекта Unity, автоматически импортируются и сохраняются как ассеты Unity.

Файл модели может содержать 3D модель персонажа, здания или части мебели. Модель импортируется в виде набора ассетов. В окне обозревателя проекта (**Project**) главный импортированный объект представляется в виде **Model Prefab**. Обычно также существует несколько Mesh объектов, на которые ссылается Model Prefab.

2.2.2. Файлы изображений

Редактор Unity поддерживает все распространенные форматы изображений – BMP, TIFF, TGA, JPG и PSD. Сложности могут возникнуть только при импорте PSD с прозрачностью, но все решаемо и об этом говорится в руководстве по Unity:

https://docs.unity3d.com/560/Documentation/Manual/HOWTO-alphamaps.html

(к сожалению, русскоязычная версия этой страницы больше недоступна, поэтому приводится ссылка на оригинал).

2.2.3. Звуковые файлы

Звуковые файлы являются очень важными для организации геймплея. Например, если вы создаете шутер, то вам понадобятся как минимум несколько файлов со звуком стрельбы из разного оружия, если разрабатывается игра в стиле Need For Speed, то понадобится звук моторов разных автомобилей.

В таблице 2.1 приведены аудио-форматы, поддерживаемые Unity.

Таблица 2.1. Аудио-форматы

Формат	**Расширения**
MPEG layer 3	.mp3
Ogg Vorbis	.ogg
Microsoft Wave	.wav
Audio Interchange File Format	.aiff / .aif
Ultimate Soundtracker module	.mod

Impulse Tracker module	.it
Scream Tracker module	.s3m
FastTracker 2 module	.xm

2.3. Импорт ассетов

В прошлой главе было показано, как импортировать стандартные ассеты и как заставить их работать с последней версией Unity 2020.3. Сейчас рассмотрим сам процесс импорта более подробно.

Ресурсы создаются за пределами Unity и могут быть импортированы посредством сохранения файла ресурса в папке **Assets** вашего проекта. Вы можете не использовать встроенные средства импорта, а просто скопировать нужные вам ресурсы в папку **Assets**.

При создании проекта создается папка с именем проекта, которое вы указали при его создании. В этой папке будет подпапка **Assets** (рис. 2.8). Необходимые ресурсы вы можете или сохранять непосредственно в эту папку или же копировать в нее, если они уже готовы.

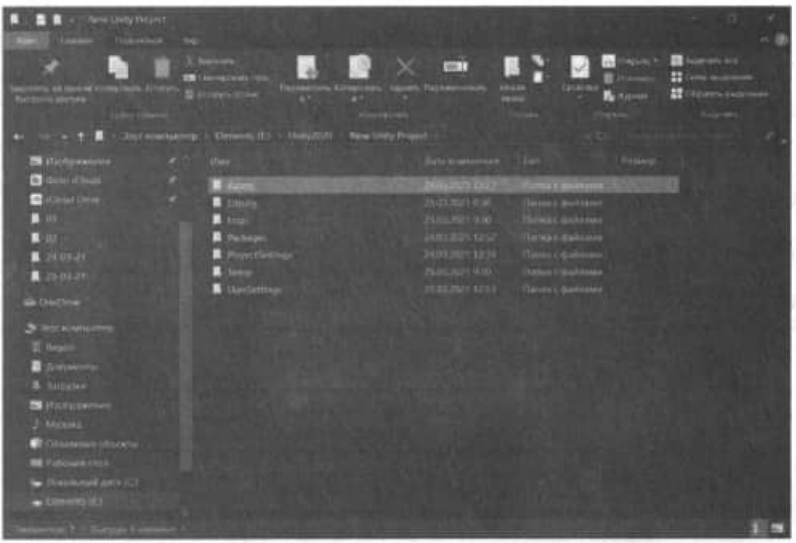


Рис. 2.8. Структура проекта

Содержимое папки Assets показывается в области **Project** главного окна Unity (рис. 2.9). Как только вы скопируйте в эту папку новый ресурс, он сразу же будет доступен в области **Project**. При этом Unity автоматически определит файлы, как только они будут добавлены в папку Assets вашего проекта.

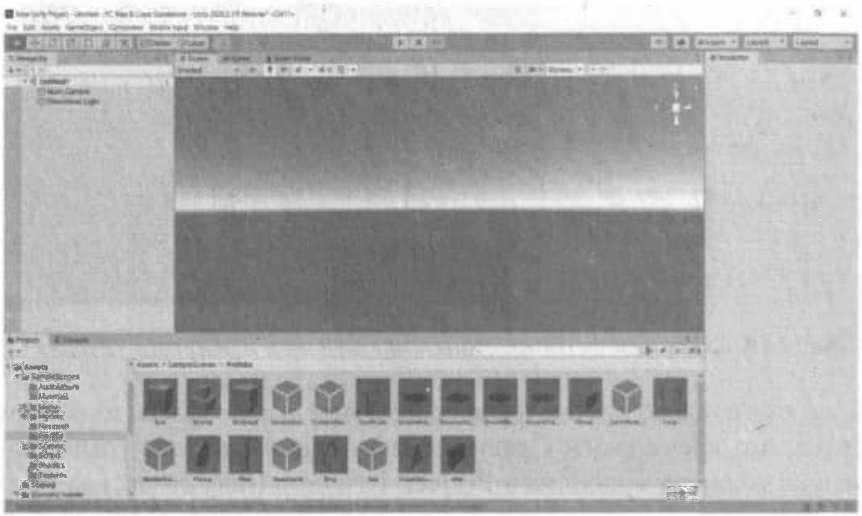


Рис. 2.9. Ресурсы (ассеты) проекта

Если вы перетащите файл в область **Project** с вашего компьютера, точнее с Проводника или любого другого файлового менеджера, то он будет скопирован (не перемещен!) в папку Assets и сразу появится в области **Project**, то есть станет доступен для использования.

Элементы, которые вы видите в области **Project**, в большинстве случаев представляют собой реальные файлы, находящиеся на вашем компьютере. Удалив ресурс, вы удалите и его файл с вашего компьютера. Если вы скопировали файл ресурса в папку Assets, вероятнее всего, у вас еще где-то останется его копия. **Если же файл был сохранен (например, из графического или аудио-редактора) в папку Assets, его удаление приведет к удалению единственной копии, поэтому будьте осторожны с удалением ассетов!**

На рис. 2.10 показана связь между ресурсами в папке Assets и области **Project**. Обратите внимание на названия материалов и файлов в этой самой папке.

Рис. 2.10. Связь между содержимым папки Assets и области "Project"

Если вы внимательно просмотрели содержимое папки Assets, то, наверное, заметили, что кроме файлов ресурсов есть еще и файлы с расширением .meta. Они не видимы в области **Project**. Редактор Unity создает эти файлы для каждого ассета и каждой папки, но по умолчанию они скрыты, поэтому в **Проводнике** вы сможете увидеть их только, если включено отображение скрытых файлов и папок. Данные файлы содержат важную информацию о том, как ресурс используется в проекте. Ни в коем случае не удаляйте, не переименовывайте и не перемещайте эти файлы в **Проводнике**. Если вы переименовали .meta-файл, то соответствующим образом нужно переименовать и файл ресурса и наоборот.

Самый простой способ переименовать ваш ресурс – сделать это в главном окне Unity. В этом случае среда автоматически переименует и соответствующий .meta-файл.

Рассмотрим процесс импорта ассета:

1. Получите модель какого-то объекта. В Сети очень много сайтов, откуда можно скачать уже готовые и бесплатные модели. Примеры сайтов **https://www.turbosquid.com/**, **https://free3d.com/.** В нашем случае в качестве подопытной модели будет выступать модель дракона, доступная по адресу **https://free3d.com/ru/3d-model/black-dragon-rigged-and-game-ready-92023.html**. Если есть выбор, скачайте сразу пакет для Unity (расширение модели будет .unitypackage)

2. Скачайте и распакуйте архив с моделью (для экономии места модели часто сжимаются различными архиваторами)
3. Перетащите файл модели (в нашем случае это файл Dragon Unity 5.unitypackage)
4. Появится окно импорта (рис. 2.11), в котором нужно нажать кнопку **Import**
5. Дождитесь завершения импорта и поместите модель на сцену (рис. 2.12).

Рис. 2.11. Импорт

Не нужно думать, что все качественные модели обязательно платные. На рис. 2.13 изображено две модели – сторожевая башня и дракон. Обе модели бесплатные и загружены с сайта **https://free3d.com**.

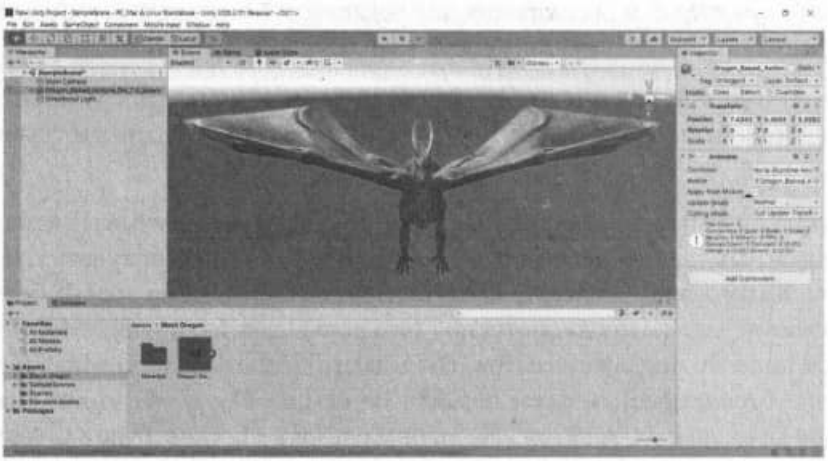


Рис. 2.12. Модель помещена на сцену

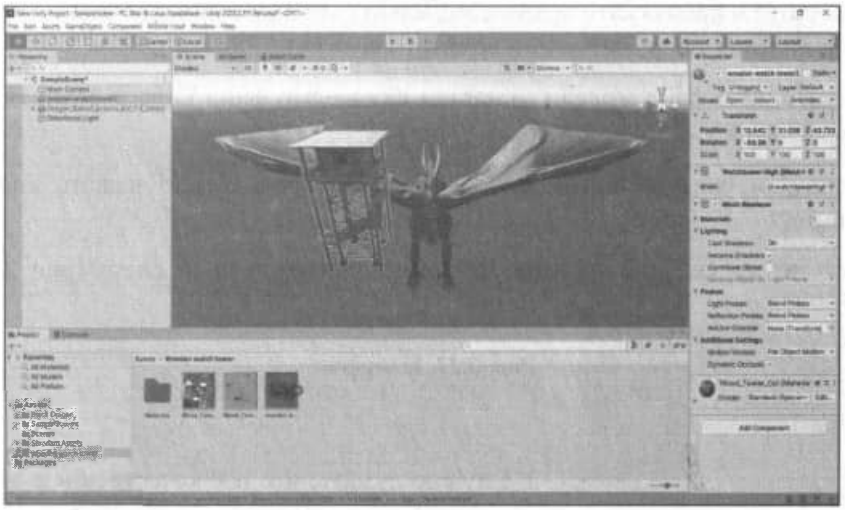


Рис. 2.13. Модели помещены на сцену

2.4. Магазин ассетов

Unity Магазин (Asset Store) – это растущая библиотека, в которой собраны бесплатные и коммерческие ассеты, созданные как компанией-разработчиком – Unity Technologies, так и членами сообщества.

Доступен большой выбор ассетов – текстуры, модели и анимации, примеры проектов, учебники и расширения для редактора. Магазин ассетов доступен по адресу https://assetstore.unity.com/.

Ранее он был встроен в интерфейс самого редактора, но теперь на вкладке Assets Store вы найдете только инструкции по импорту ассетов и ссылку на сам магазин.

Регистрация в магазине бесплатная, при этом не нужно указывать каких-либо платежных данных, если вы не собираетесь пока что покупать платные ассеты. Какого-либо фильтра, предусматривающего выбор только бесплатных ассетов, уже не предусмотрено (хотя раньше он был) – видимо, так пытаются поднять продажи ассетов. Вы можете перейти в категорию *Top free* (лучшие бесплатные), а затем перейти по ссылке *See more*, чтобы увидеть больше бесплатных ассетов. На первых порах ассетов будет более чем достаточно.

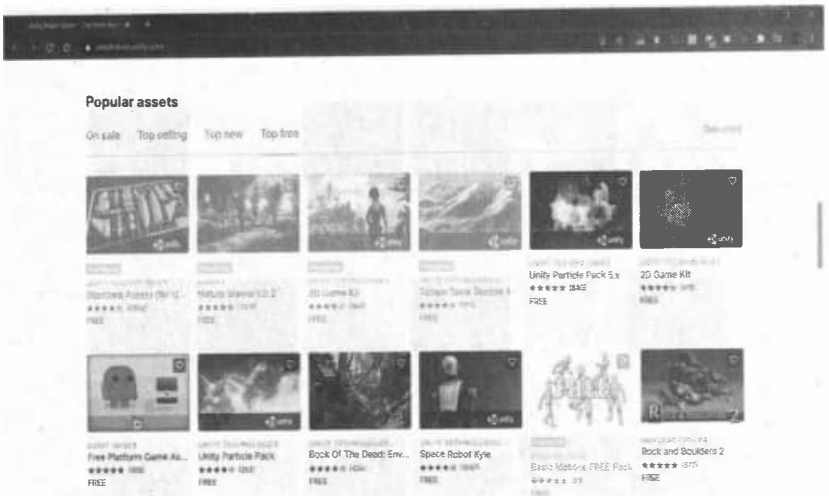


Рис. 2.14. Asset Store

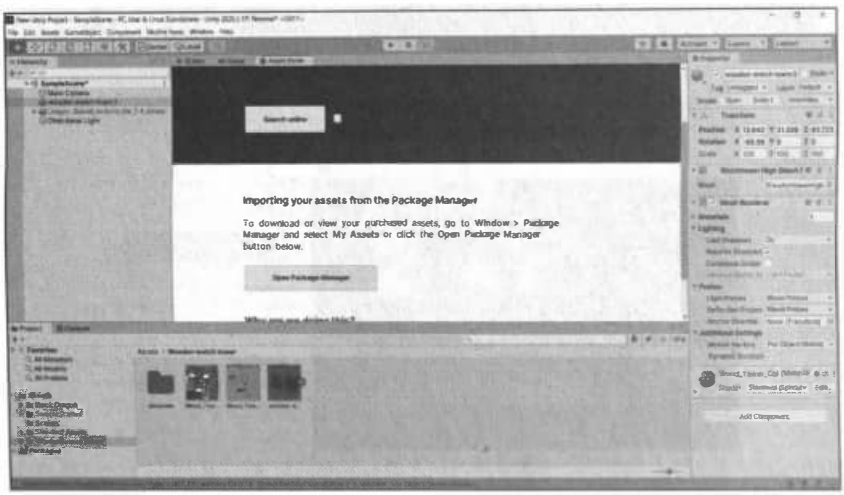


Рис. 2.15. Инструкции по импорту ассетов

Один из неплохих бесплатных пакетов, который можно смело порекомендовать всем новичкам – Nature Starket Kit 2 (рис. 2.17). Пакет содержит три текстуры земли, 4 дерева, 6 кустов и другие модели. Ссылка на этот ассет:

https://assetstore.unity.com/packages/3d/environments/nature-starter-kit-2-52977

Примечание. На прилагающемся к книге диске вы найдете этот и некоторые другие «природные» пакеты.

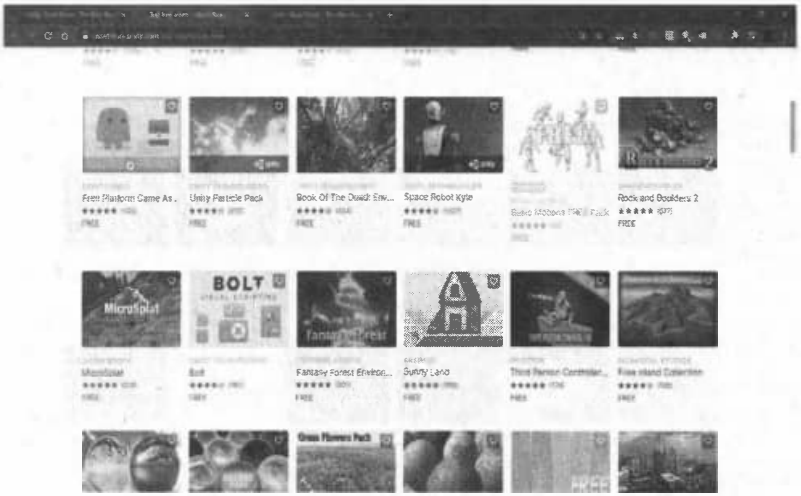


Рис. 2.16. Дополнительные бесплатные ассеты

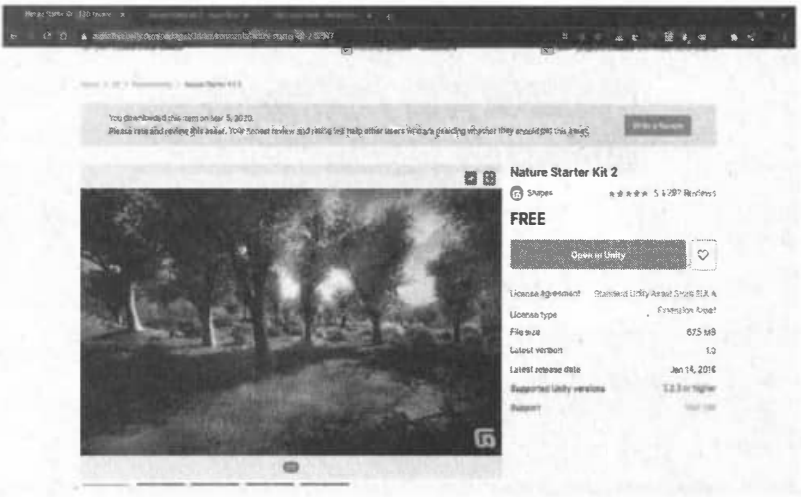


Рис. 2.17. Пакет Nature Starket Kit 2

Для установки выбранного пакета нажмите кнопку **Open in Unity**. При импорте пакета появится уже знакомое окно **Package Manager** – в нем вы можете выбрать только те модели, которые вам нужны в вашем проекте и нажать кнопку **Import** (рис. 2.18). Для изучения состава пакета импортируйте все содержимое.

После этого появится окно **Import Unity Package**, в котором также нужно нажать кнопку Import (рис. 2.19).

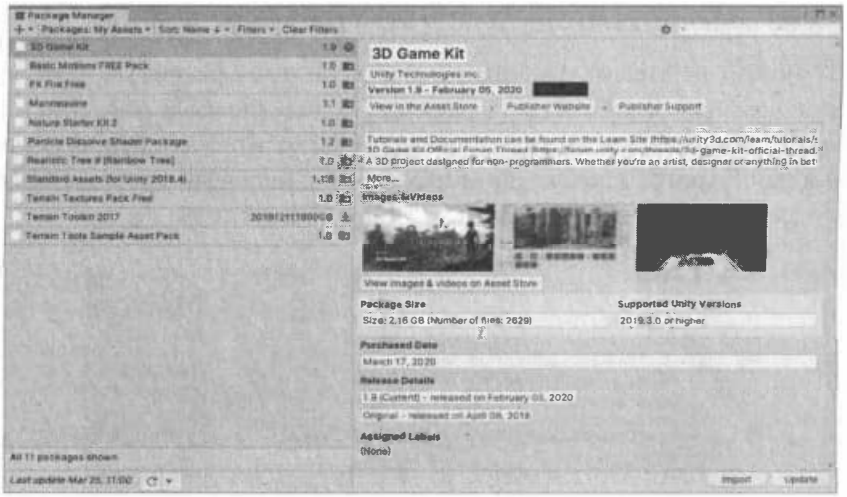


Рис. 2.18. Загрузка пакета

Рис. 2.19. Импорт пакета в проект

После этого вы сможете использовать новые модели в своем проекте.

Примечание. Если вам понадобится доступ (это вообще-то говоря редко когда нужно) к файлам, загруженным через Asset Store, то вы можете их найти в каталоге C:\Users\имя пользователя\AppData\Roaming\Unity\Asset Store.

2.5. Экспорт пакета с ассетами

До этого мы рассматривали только одно направление работы с ассетами – импорт их в проект. Но бывает так, что работая над проектом, вы соберете из разных источников неплохую коллекцию ассетов, которую вы бы хотели использовать в других своих проектах. Часто бывает так, что источник ассета вы не запомнили,

поэтому не помните, где и откуда загружали тот или иной ресурс.

Ассеты можно не только импортировать, но и экспортировать. Для этого выберите команду **Assets, Export package**. В появившемся окне (рис. 2.20) необходимо выбрать, какие именно ресурсы вы хотите экспортировать и нажать кнопку **Export**. Далее введите имя файла с набором ассетов, выберите каталог для его сохранения и нажмите кнопку **Сохранить**.

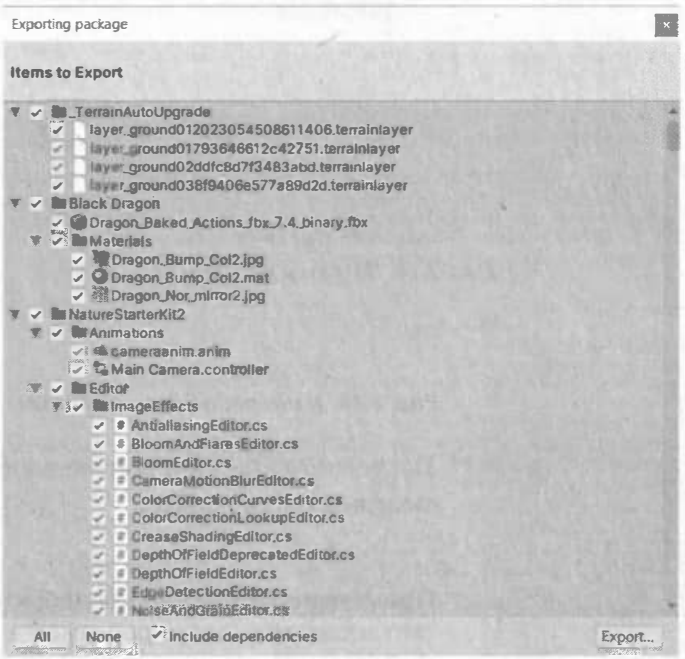


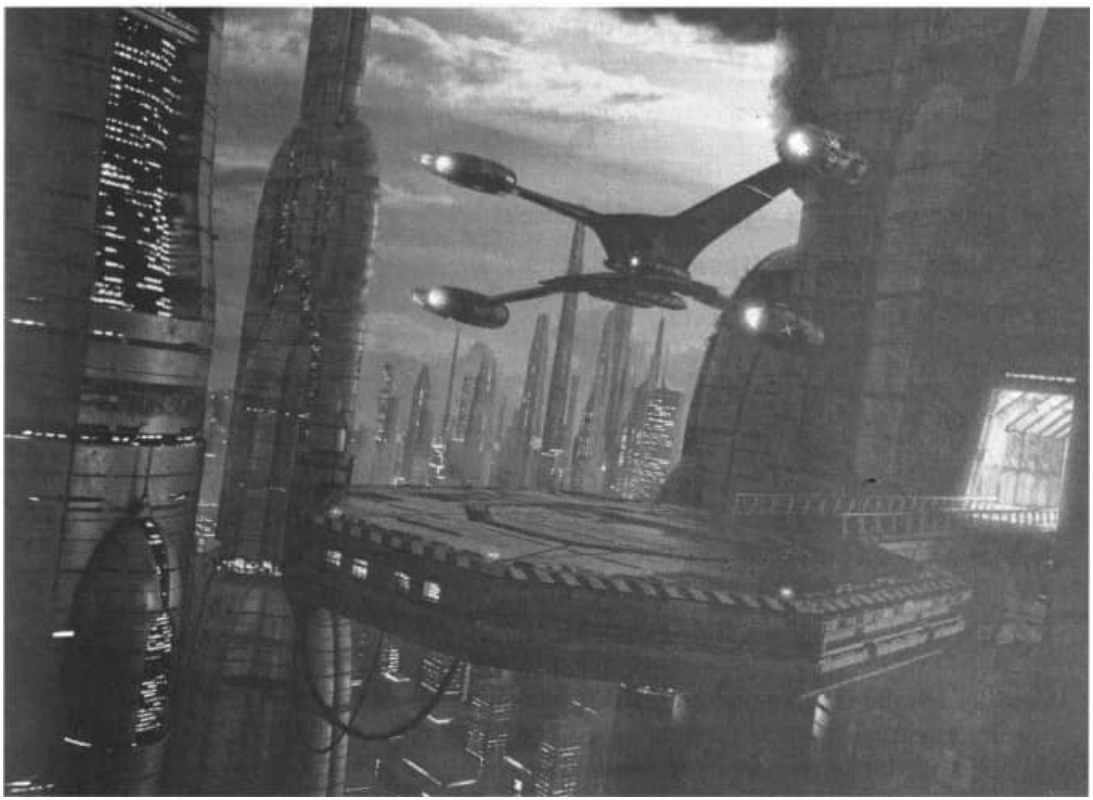
Рис. 2.20. Экспорт ассетов

Внимание! При экспорте ассетов не выключайте зависимости – флажок **Include dependencies**, иначе выбранные вами модели могут не работать в другом проекте.

Если вы хотите с кем-то поделиться своим набором ассетов, то неплохо бы его перед этим сжать. Не все ресурсы хорошо сжимаются – может оказаться так, что после сжатия размер архива особо меньшим не станет, но попытаться стоит.

Глава 3.

Изучаем интерфейс Unity



В предыдущих главах вы вкратце познакомились с интерфейсом пользователя Unity и даже построили первую сцену свою сцену. В этой главе интерфейс Unity будет рассмотрен подробнее.

Основное окно состоит из следующих компонентов:

- Область **Project**
- Основная область (представление **Scene**)
- Область иерархии (**Hierarchy**).
- Окно инспектора (**Inspector**)
- Панель инструментов
- Представление **Game**

Иногда области называются окнами, поскольку их можно закрывать, открывать и перетаскивать внутри основного окна Unity так, как вам удобно.

3.1. Обозреватель проекта

В окне обозревателя проекта (Project) пользователь может работать с ассетами текущего проекта. Левая панель обозревателя отображает структуру папок проекта в виде иерархического списка.

При выборе папки в этом списке с помощью клика мыши в панели справа отобразится ее содержимое. Отдельные ресурсы представлены в виде значков, указывающих их тип (скрипт, материал, подпапка и т.д.). Можно менять размер иконок с помощью слайдера (на рис. 3.1 он выделен стрелкой) в нижней части панели. При перемещении слайдера в крайнее левое положение ассеты будут представлены в виде иерархического списка, а не в виде значков. Левее от слайдера выводится имя выбранного ассета и полный путь до него, если ассет быть найден с помощью поиска.

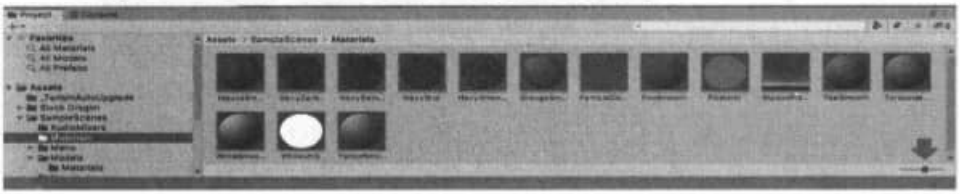


Рис. 3.1. Обозреватель проекта

Над списком структуры проекта есть раздел **Favorites**, содержащий ссылки на часто используемые (избранные) ассеты для быстрого доступа к ним. Чтобы добавить ассет в избранное, просто перетащите его туда.

Над панелью находится навигационная цепочка, отображающая путь до просматриваемой папки. Вы можете щелкать по отдельным элементам цепочки для удобной навигации по иерархии папок.

В верхнем левом углу находится кнопка +. В более старых версиях Unity она называлась кнопкой **Create**. При нажатии на эту кнопку отображается меню **Create**, в котором находятся команды создания новых ассетов и подкаталогов в текущей папке. В версии 2020.3 это меню слишком большое, поэтому на рис. 3.2 оно отображено не полностью.

Справа находится панель поиска ресурсов. Справа от нее – кнопки, управляющие поиском. Можно фильтровать результаты поиска по типу (**Search by type**), по метке (**Search by label**). Можно также сохранять поиск – кнопка с изображением звездочки. Тогда поиск будет добавлен в раздел сохраненных поисков (рис. 3.3). Кнопка с числом и перечеркнутым глазом показывает, сколько элементов было скрыто из результатов поиска. Нажав на нее, можно просмотреть скрытые элементы.

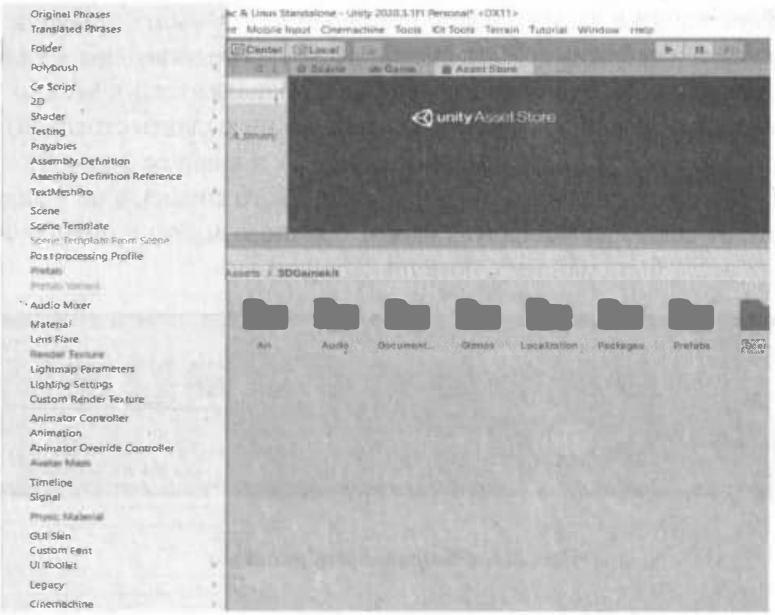


Рис. 3.2. Меню Create

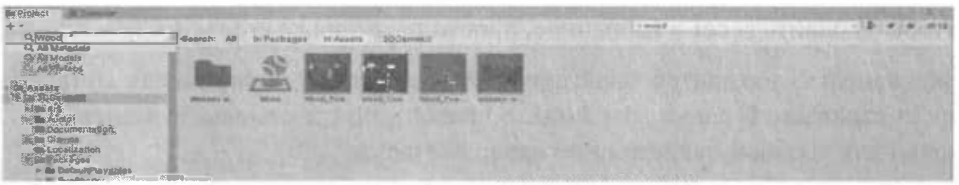


Рис. 3.3. Сохранение поискового запроса

Меню окна (рис. 3.4) позволяет произвести операции с окном, в данном случае с окном **Project**:

- **One Column Layout/Two Column Layout** – переключение между видом на 1 и 2 колонки
- **Lock** – «заморозка» текущего содержимого окна, то есть предотвращает изменение его события, происходящими вне обозревателя. Для быстрой блокировки/разблокировки можно использовать кнопку с изображением замка слева от кнопки вызова этого меню
- **Maximize** – максимизация окна – иногда ресурсов очень много и если развернуть окно, можно быстрее найти нужный
- **Close Tab** – закрывает текущую вкладку

- **Add Tab** – добавляет вкладку. Если вы закрыли какую-то вкладку, то можете открыть ее с помощью команд, находящихся в этом меню

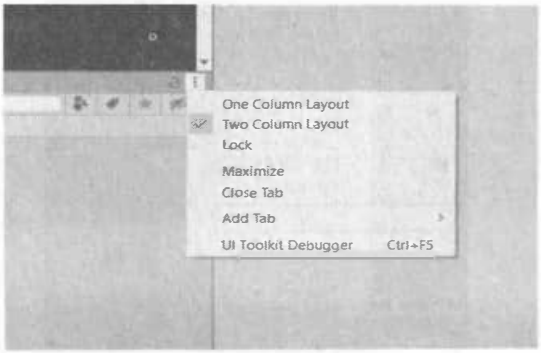


Рис. 3.4. Меню окна

Изюминка обозревателя – мощные возможности поиска. В простых проектах это сложно оценить, но в реальном проекте игры поиск нужного объекта – та еще задача.

Обычный поиск будет фильтровать ассеты согласно тексту, введенному в поле для поиска (рис. 3.5).

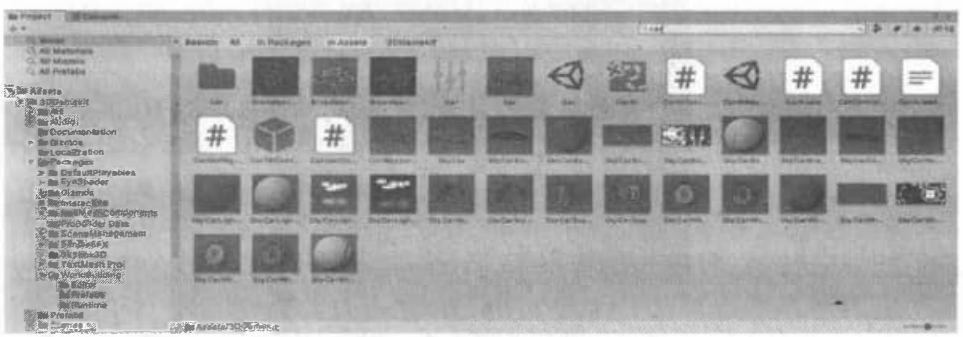


Рис. 3.5. Результаты поиска

Если вы введете более одного слова для поиска, например, *black dragon*, обозреватель найдет ассеты, в именах которых встречаются оба слова «black» и «dragon» (т.е. выражения для поиска логически умножаются оператором AND).

Справа от поля для поиска находятся три кнопки. Первая позволяет дополнительно отфильтровать найденные ассеты по типу, см. рис. 3.6.

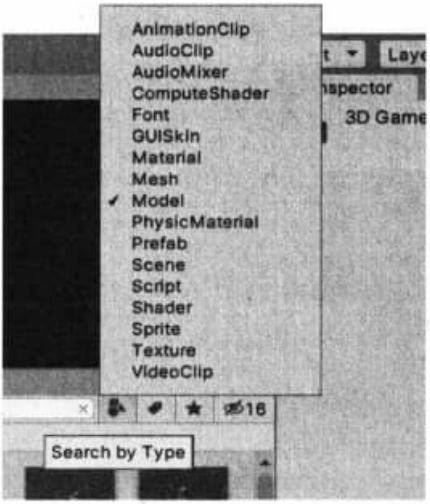


Рис. 3.6. Фильтр ассетов по типу

Фильтр по метке (**Search by Label**) позволяет отфильтровать результаты поиска по метке, которую можно назначить ассету в инспекторе (рис. 3.7).

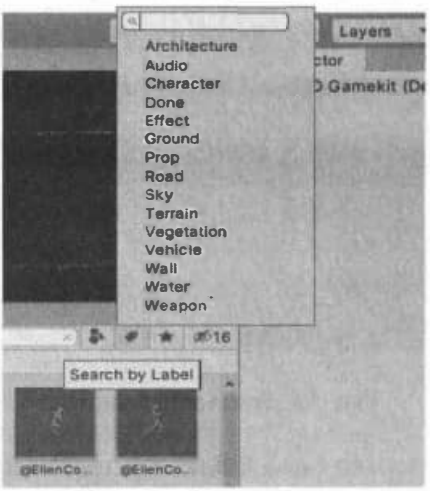


Рис. 3.7. Фильтр результатов поиска по метке

Фильтры работают путем добавления дополнительных выражений к тексту, который вы ищете. Выражение, начинающееся с «t:» фильтрует ассеты по указанному типу, в то время как «l:» фильтрует по метке. Другими словами,

можно не использовать кнопки, а задавать тип фильтрации непосредственно в строке поиска.

Сравните результаты поиска на рис. 3.5 и рис. 3.8. Мы ищем по строке «car», но на рис. 3.8 видно, что задан тип ассета – материал (t:material). В итоге результаты совершенно другие – найдены только материалы, содержащие «car» в названии. На рис. 3.5 мы нашли все ресурсы, в названии которых есть строка «car», а на рис. 3.8 мы уточнили, что нас интересуют только модели.

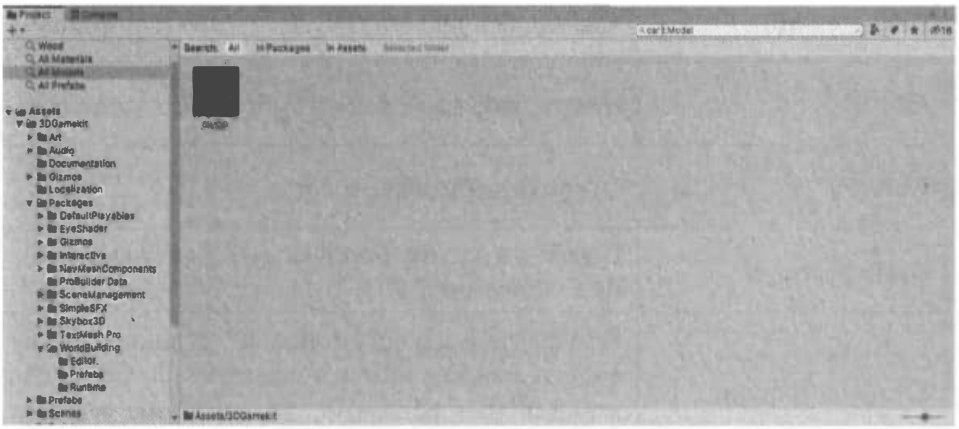


Рис. 3.8. Отфильтрованные результаты поиска

Существенно сократить время при работе с обозревателем можно путем использования специальных комбинаций клавиш (табл. 3.1). Учтите, что некоторые из них работают только в режиме с двумя колонками (он используется по умолчанию, как сменить режим отображения, было показано ранее). Также учитывайте, что комбинации клавиш, указанные в табл. 3.1, будут работать только в Windows. В других операционных системах они могут быть другими.

Таблица 3.1. Комбинации клавиш Обозревателя

Комбинация клавиш	Описание
F	Перейти к избранному
Tab	Перейти от первой ко второй колонке и обратно (в режиме с двумя колонками)
Ctrl/Cmd + F	Перейти к полю для поиска

Ctrl/Cmd + A	Выбрать все видимые элементы в списке
Ctrl/Cmd + D	Дублировать выбранные ассеты
Delete	Удалить с вызовом диалога подтверждения
Delete + Shift	Удалить без диалога подтверждения (осторожно!)
F2	Начать изменение имени выбранного элемента
Enter	Открыть выбранные ассеты
Backspace	Перейти в родительский каталог (в режиме с двумя колонками)
Стрелка вправо	Развернуть выбранный элемент (если он в древовидных списках или в результатах поиска). Если элемент уже развернут, это действие выберет первый из его дочерних элементов
Стрелка влево	Свернуть выбранный элемент (если он в древовидных списках или в результатах поиска). Если элемент уже свернут, это действия выберет его родительский элемент
Alt + стрелка вправо	Развернуть элемент при отображении ассетов в режиме предпросмотра
Alt + стрелка влево	Свернуть элемент при просмотре ассетов в режиме предпросмотра

3.2. Рабочая область

Рабочая область находится по центру главного окна и содержит вкладки:

- Scene;
- Game;
- Asset Store.

Первая вкладка – это ваша песочница. Это представление сцены. Вы используете вкладку **Scene** для выбора и расположения окружений, игрока, камеры, врагов, и всех остальных игровых объектов. Управление и манипулирование объектами с использованием окна **Scene** является одной из наиболее важных функций Unity, поэтому важно разобраться с этим представлением.

Рис. 3.9. Вкладка "Scene"

3.2.1. Навигация по сцене

Область Scene оснащена всем необходимым для того, чтобы ориентироваться в нем быстро и эффективно.

Перемещение по сцене с помощью стрелок и фокусировка на объекте

Для перемещения по сцене можно использовать клавиши со стрелками. Клавиши **влево** и **вправо** поворачивают камеру в стороны, а **вверх** и **вниз** – вперед и назад в том направлении, в которое она смотрит. Вместе с клавишами-стрелками используйте клавишу **Shift** для более ускоренного перемещения.

Если выбрать игровой объект в иерархии, затем поместить мышь над **Scene** и нажать Shift+F, вид отцентрируется на выбранном объекте. Аналогично можно дважды щелкнуть на игровом объекте.

Инструмент Hand

Ключевыми операциями при просмотре сцены являются перемещение, вращение по орбите и масштабирование. Без них невозможен нормальный просмотр сцены.

Нажмите клавишу **Q** для быстрого выбора инструмента **Hand** (рука) или же щелкните на его значке на панели инструментов:

Как только этот инструмент активирован, вам доступны следующие действия:

- **Движение** – перемещайте мышь с зажатой левой клавишей для перемещения камеры
- **Вращение** – удерживая Alt, перемещайте мышь с зажатой левой клавишей – этим вы будете вращать камеру вокруг текущего центра вращения. Данная возможность не поддерживается в 2D режиме
- **Масштабирование** – используйте колесико мыши для масштабирования

> **Примечание.** Для ускорения работы инструмента **Hand** удерживайте нажатой клавишу Shift.

Режим полета

В режиме полета пользователь может летать над сценой в режиме полета от первого лица. Для активации режима полета нажмите и удерживайте правую кнопку мыши. Нажатие должно быть выполнено в окне **Scene**. Для выхода из режима полета просто отпустите правую кнопку мыши.

В режиме полета вы можете использовать клавиши-стрелки для перемещения влево/вправо, вперед/назад. Также можно использовать клавиши **Q** и **E** для перемещения вверх и вниз. Как обычно, Shift ускоряет движение.

Рис. 3.10. В режиме полета заглянули в смотровую башню

3.2.2. Панель управления сценой

Рассмотрим панель управления сценой (рис. 3.11):

1. Выпадающий список, позволяющий выбрать режим отрисовки сцены. Режимов много, экспериментируйте. Основные режимы следующие:
   - **Shaded** – объекты с текстурами – так, как они будут выглядеть в игре
   - **Wireframe** – никаких текстур, только каркас объектов
   - **Shaded Wireframe** – смесь двух режимов – объекты будут с текстурами, но и будет виден каркас
   - **Shadow Cascades** – показывает каскады теней от направленного света
   - **Render Paths** – показывает способ рендеринга для каждого объекта, используя цветовой код: зеленый обозначает отложенный свет (*deferred lighting*), желтый обозначает прямой рендеринг (forward rendering) и красный обозначает освещение вершины (vertex lit)
   - **Alpha** – визуализирует цвета с альфой (прозрачностью)
   - **Overdraw** – визуализирует объекты как прозрачные «силуэты». Прозрачные цвета накапливаются, позволяя легко заметить места, в которых один объект нарисовался поверх другого

- **Mipmaps** – отображает идеальный размер текстур, используя цветовой код: красный обозначает, что текстура больше чем нужно (на текущем расстоянии и разрешении); синий обозначает, что текстура могла бы быть больше. Обычно, идеальный размер текстуры зависит от разрешения, в котором будет запущена игра, и как близко камера может быть расположена от конкретных поверхностей

2. Переключатель 2D режима, позволяющий переключаться между 2D и 3D режимами
3. Включает/выключает освещение сцены
4. Включает/выключает звук
5. Включает/выключает небосвод, туман и другие объекты
6. Количество невидимых объектов на сцене
7. Включает/выключает отображение сетки
8. Включает/выключает отображение панели редактора компонентов
9. Отображает окно настроек камеры сцены
10. Отображает меню Гизмо. Гизмо – это графика, добавляемая на сцену (либо самой Unity, либо ваши собственном скриптом), которая помогает с визуализацией и идентификацией предметов в игровом мире. Например, вы можете добавить значки, которые помогут идентифицировать ваши игровые объекты, и использовать простую каркасную графику, чтобы показывать невидимые элементы. В меню Gismos вы можете просмотреть различные значки, назначенные разным элементам сцены. Некоторые значки уже назначены по умолчанию. Некоторые можно назначить с помощью соответствующих скриптов. О том, как создавать такие скрипты, написано в руководстве по Unity **https://docs.unity3d.com/ru/current/ScriptReference/Gizmos.html**.

Последний элемент на панели управления сценой – это панель поиска, позволяющая быстро найти элемент на сцене по его имени или типу.

3.2.3. Позиционирование объектов на сцене

При разработке сцены вам придется разместить достаточно много разных объектов. Не всегда получается разместить объект правильно. Разберемся,

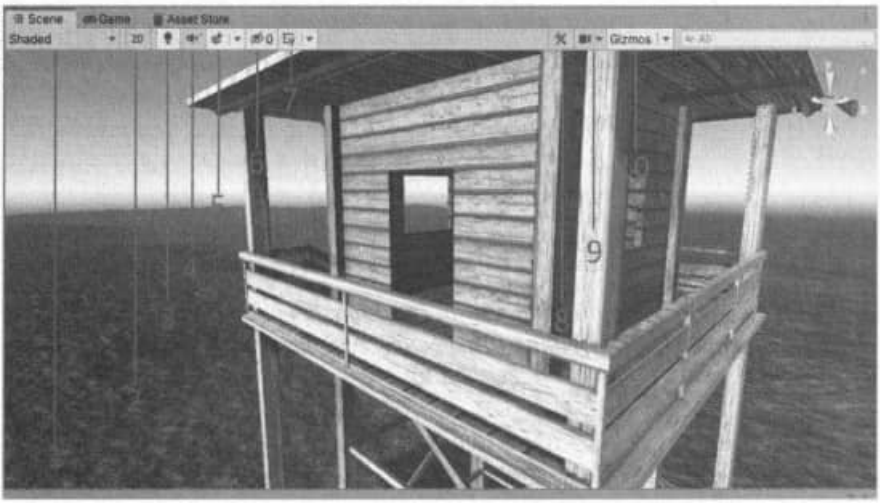
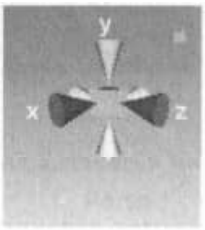


Рис. 3.11. Панель управления сценой

какие средства предоставляет Unity для упрощения позиционирования. Сразу нужно отметить, что в верхнем правом углу находится гизмо сцены. Оно показывает ориентацию камеры сцены и позволяет быстро поменять угол обзора.



Перемещение, вращение и масштабирование игровых объектов

Используйте инструменты трансформаций в панели инструментов для перемещения, вращения и масштабирования отдельных объектов. Каждый инструмент имеет соответствующее гизмо, появляющееся вокруг выделенного игрового объекта в окне **Scene**.

Используйте мышь и манипулируйте любой осью гизмо для изменения компонента Transform игрового объекта, а именно его свойств **Position** и **Rotation**. Также можно ввести значения непосредственно в числовые поля компонента в инспекторе.

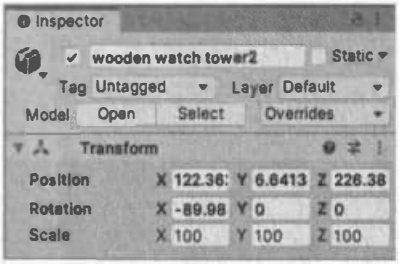


Рис. 3.12. Координаты расположения (position) и вращения (rotation) по осям можно задавать вручную в окне "Inspector" для более точного позиционирования

Каждый из трех режимов – перемещение, вращение, масштабирование – может быть выбран горячими клавишами – **W** для перемещения, **E** для вращения и **R** для масштабирования. На рис. 3.13 показаны соответствующие инструменты.

Рис. 3.13. Инструменты позиционирования игровых объектов

Существуют различные стратегии позиционирования объекта на сцене. Например, можно зажать и перетащить центр гизмо для перемещения объекта сразу по всем осям.

В центре гизмо инструмента перемещения находятся три небольших квадрата, которые можно использовать для перемещения объекта вдоль одной плоскости (т.е. вы можете смещать одновременно две оси, в то время как третья будет неподвижна). При наличии третьей кнопки мыши можно нажать среднюю кнопку и перетащить курсор, чтобы работать с наиболее часто используемой осью (ее стрелка изменится на желтую).

Выбрав инструмент вращения, пользователь может менять поворот объекта, зажав и перетаскивая оси гизмо каркасной сферы, отображаемого вокруг объекта. Так же как и с инструментом перемещения, последняя измененная

ось будет покрашена в желтый цвет и может быть в дальнейшем изменена кликом и перетаскиванием средней кнопкой мыши.

Инструмент масштабирования позволяет вам менять масштаб объекта равномерно по всем осям сразу с помощью нажатия и перетаскивания куба в центре гизмо. При желании можно масштабировать оси по отдельности, но делать нужно это осторожно: если у объекта есть дочерние объекты, то эффект может выглядеть довольно странно. Как обычно, последняя использованная ось будет перекрашена в желтый цвет и может быть использована перемещением средней кнопкой мыши.

> **Внимание!** Учтите, что в режиме 2D у вас нет оси Z (поскольку есть только два измерения), поэтому ось Z не может быть изменена с помощью гизмо.

При работе с инструментами позиционирования запомните действие трех комбинаций клавиш:

- Ctrl – перетаскивайте любую ось гизмо в режиме перемещения с нажатой клавишей Ctrl для изменения значения с использованием привязки, определенной в **Snap Settings**. Другими словами, Ctrl позволяет реализовать привязку к сетке. Изменить шаг привязки можно с помощью команды меню **Edit, Snap Settings**
- Ctrl + Shift – при использовании инструмента вращения и нажатой клавише Shift можно направить взгляд объекта на точку поверхности любого коллайдера, что позволяет легко ориентировать объекты относительно друг друга
- Ctrl + Shift – в режиме перемещения – позволяет выполнить привязку к поверхности. В этом режиме легко точно расставлять объекты

Еще в Unity поддерживается так называемая вершинная привязка. Данной возможности уделяется мало внимания, хотя она является довольно мощным инструментом. Вершинная привязка позволяет вам взять один объект за любую вершину и с помощью мышки расположить его в такое же положение вершины другого объекта, который вы выберите.

Вершинная привязка позволяет очень быстро собирать миры. Вы можете подгонять дороги в гоночной игре с очень высокой точностью.

Использовать вершинную привязку нужно так:

- Выберите объект, которым вы хотите манипулировать и убедитесь, что вы выбрали режим перемещения
- Нажмите и держите клавишу **V** для активации режима вершинной привязки
- Поместите курсор мыши над вершиной, которую вы хотите использовать
- Нажмите и удерживайте левую кнопку мыши, когда курсор находится над нужной вершиной, и перетащите объект к любой другой вершине на другом объекте
- Отпустите кнопку мыши и клавишу **V**, если результат вас устраивает

3.3. Вкладка Game

Вкладка **Game** представляет собой предпросмотр реального игрового режима. Другими словами, вы можете увидеть, как будет выглядеть ваша игра и даже поиграть, запустив игру нажатием кнопки **Play**.

Но обо всем по порядку. В игровом режиме происходит отрисовка сцены из камеры (камер) вашей игры. Для того чтобы что либо увидеть, вам нужна хотя бы одна камера. Добавить камеру на сцену можно командой **GameObject, Camera**. Кстати, на вкладке **Scene** и при активном объекте **Camera** отображается окно предварительного просмотра камеры – **Camera Preview**, что тоже позволяет понять, как будет выглядеть игра (рис. 3.14).

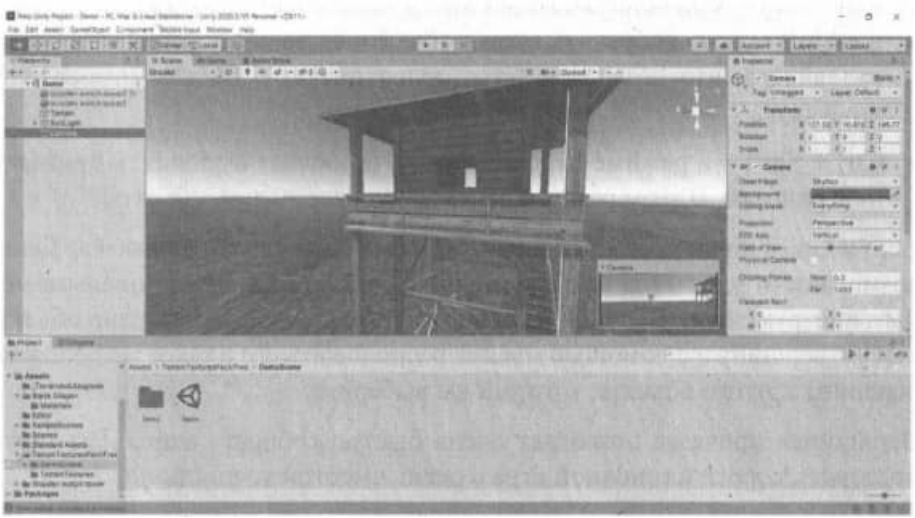


Рис. 3.14. Окно "Camera Preview"

Кнопки на панели инструментов в игровом режиме позволяют запустить игру, поставить игру «на паузу». Пока вы находитесь в режиме *Play*, любые изменения лишь временные, и сбросятся, когда вы выйдете из режима игры. Интерфейс редактора потемнеет, чтобы обратить ваше внимание на это. Для выхода из режима игры (когда вы запустили игру, нажав кнопку *Play*) нажмите комбинацию клавиш Ctrl + P.

> **Примечание.** Не смотря на то, что изменения в режиме запущенной игры временные, они осуществляются в режиме реального времени. Например, вы можете изменить скорость перемещения персонажа, его вес, и посмотреть, как он будет двигаться. После останова игры изменения будут утеряны, но вы их запомните и сможете установить новые значения в области инспектора.

Рассмотрим панель инструментов на вкладке Game (рис. 3.15):

- Список **Display** – позволяет выбрать дисплей. Имеется в виду не реальный дисплей вашего компьютера. Данный список позволяет выбрать список камер, если у вас есть несколько камер на сцене. Вы можете назначить дисплеи камерам в модуле Camera посредством свойства **Target Display**. По умолчанию используется **Display 1**
- Список **Aspect** – позволяет выбрать соотношение сторон монитора, то есть просмотреть, как будет выглядеть ваша игра на мониторах с разным соотношением сторон. По умолчанию используется значение **Free Aspect**
- Слайдер **Scale** – позволяет вам изменить масштаб и рассмотреть игровые объекты во всех деталях. Для управления слайдером вы также можете использовать колесо прокрутки и среднюю кнопку мыши, чтобы сделать это, когда игра остановлена или приостановлена
- Кнопка-переключатель **Maximize on Play** – когда он включен, то игровое окно будет растягиваться на 100% окна редактора после перехода в режим *Play*
- Кнопка-переключатель **Mute audio** – позволяет включать/выключать любые звуки в игровом режиме
- Кнопка-переключатель **Stats** – включает/выключает оверлей **Statistics**, содержащий различную статистику о вашей игре. Полезен при мониторинге производительности в режиме Play
- **Gizmos** – управляет видимостью гизмо

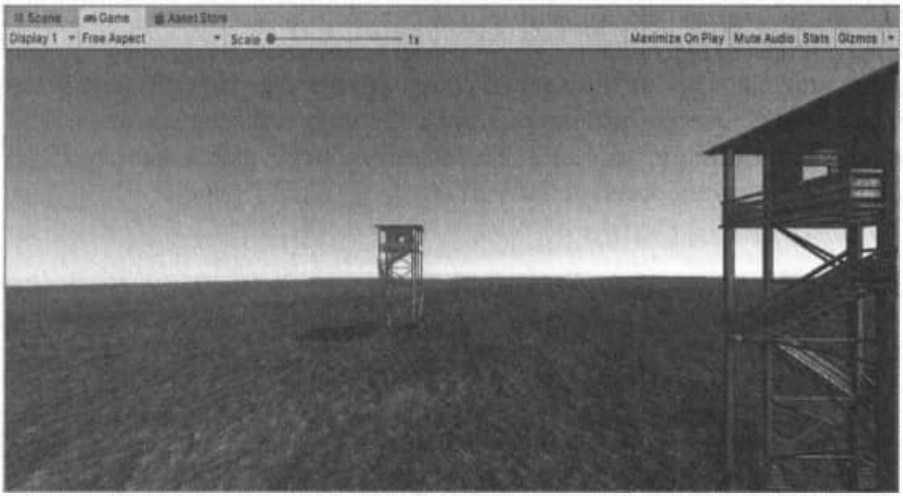


Рис. 3.15. Вкладка **Game**

3.4. Иерархия объектов

Область Иерархия (**Hierarchy**) содержит все игровые объекты (GameObject) в текущей сцене. Некоторые из них являются прямыми экземплярами файлов ассетов, таких как 3D-модели, а другие – экземпляры префабов, пользовательских объектов, из которых состоит большая часть вашей игры.

> **Примечание.** Префаб – это сохраненная коллекция, содержащая один или более полных GameObject'ов с присоединенными компонентами и установленными свойствами. Префабы – это типы ассетов, поэтому они не отображаются в сцене сами по себе. Но они могут быть использованы для создания экземпляров сохраненных объектов в сцене. Каждый экземпляр – это копия оригинального префаба. Например, вы можете использовать префаб для сохранения объекта дерева, а затем создать множество экземпляров этого дерева в сцене леса.

Пользователь может выбрать объекты в иерархии, и перетащить один объект на другой, для создания родительской связи (см. далее). При добавлении и удалении объектов в сцене, они также будут появляться и исчезать из **Иерархии**.

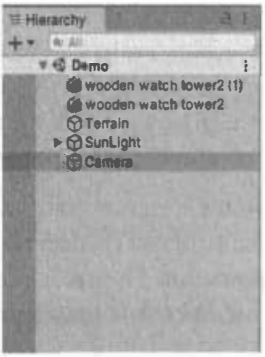
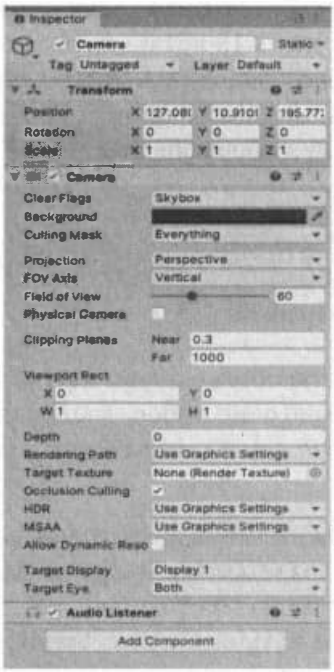


Рис. 3.16. Область "Hierarchy"

Unity использует концепцию управления родительскими/дочерними под названием **Parenting**. Чтобы сделать любой GameObject дочерним для другого, перетащите желаемый дочерний объект на желаемый родительский в иерархии. Дочерний объект будет наследовать перемещение и повороты своего родителя. Вы можете использовать раскрывающую стрелку родительского объекта для того, чтобы, при необходимости, отобразить или скрыть его детей.

3.5. Панель Inspector. Инспектор объектов



3.5.1. Редактирование свойств

Панель **Inspector** используется для просмотра и редактирования свойств объектов разными способами. У каждого игрового объекта имеется множество самых разных свойств. Некоторые из этих свойств можно изменять другими способами, например, с помощью инструментов позиционирования, как было показано ранее, можно изменить значения положения, вращения и масштаба.

Рис. 3.17. Панель "Inspector"

На панели **Inspector** можно задать имя объекта. Рекомендуется приучать себя к порядку на самых ранних этапах – назначайте объектам понятные имена. Во-первых, так будет проще найти нужный объект на стадии проектирования. Во-вторых, рано или поздно вам придется писать код и понятные имена только облегчат процесс программирования.

Параметры и опции компонентов (объектов), которые можно редактировать в инспекторе называются свойствами (*properties*) – по аналогии со свойствами объектов в программировании. При желании свойства объектов можно изменять в процессе выполнения игры программно.

Свойства можно широко разделить на категории ссылки (связи с другими объектами и ассетами) или величины (числа, флажки, цвета и т.д.).

В **Инспекторе** можно создавать ссылки. Для назначения ссылки нужно перетянуть объект или ассет подходящего типа на соответствующий пункт в **Инспекторе**. Например, часто для отрисовки поверхности объекта нужно выбрать материал. Вы можете перетянуть нужный материал на свойство выбора материала (в разных объектах оно может называться по-разному).

Когда ссылка не назначена, в качестве значения свойства вы увидите что-то вроде *None* (тип_объекта), например, None (Mesh). После того, как вы установите значение свойства, в нем будет отображаться реальное название элемента, который вы перетянули (или выбрали посредством кнопки выбора рядом с полем значения) на это свойство. На рис. 3.18 показано, что для свойства **Anchor Override** значение не назначено. Рядом есть кнопка для выбора объекта типа **Transform**. При желании вы можете перетянуть объект из обозревателя проекта, а не использовать кнопку выбора объекта.

Рис. 3.18. Значение свойства не задано

Если вы нажмете кнопку справа от поля ввода, откроется окно выбора объектов соответствующего типа. На рис. 3.19 показано окно выбора материала (**Select Material**). Выбранный материал будет назначен в качестве значения свойства.

Большинство значений редактируются с помощью знакомых текстовых полей, флажков и меню, в зависимости от их типа (для удобства, числовые значения можно увеличивать и уменьшать, двигая мышью вверх и вниз зажав кнопку мыши на названии опции). Тем не менее, есть некоторые значения

более сложных типов, которые имеют свои специфические редакторы. Например, для установки цвета откроется окно **Color**, в котором можно будет выбрать подходящий цвет.

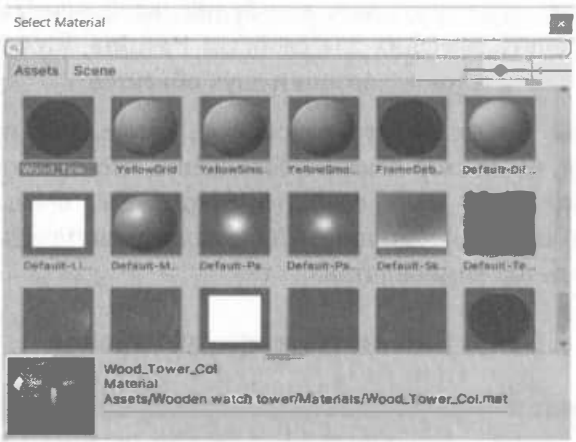


Рис. 3.19. Окно выбора материала

3.5.2. Изменение свойств нескольких объектов одновременно

В панели **Inspector** можно редактировать несколько объектов одновременно. Для этого нужно выбрать более одного объекта и тогда Инспектор позволит изменить их общие свойства: значения, которые вы установите, будут скопированы во все выбранные объекты. На рис. 3.20 показано, что выбрано две сторожевых башни. Инспектор позволит установить их общие свойства.

Рис. 3.20. Редактирование общих свойств нескольких объектов

Когда значение свойств является одинаковым для всех выбранных объектов, то значение будет показано, но в противном случае, оно будет показано в виде знака тире. Только компоненты, общие для всех объектов будут видны в инспекторе. В нашем случае, хотя и выбрано два одинаковых объекта, мы не можем установить значения для свойства **Position**, чтобы не допустить размещения по одной и той же позиции двух объектов.

Если выбранный объект имеет компоненты, которые не присутствуют у других объектов, инспектор покажет сообщение, что некоторые компоненты скрыты. Контекстное меню свойства (открывается правым щелчком по имени свойства) также имеет опции, что позволяет вам установить значение любого из выбранных компонентов.

3.5.3. Библиотеки предустановок

В Unity пользователь может создавать библиотеки предустановок, содержащие данные, созданные пользователем и сохраняющиеся между рабочими сеансами.

Предустановка – это нечто сохраненное пользователем для будущего использования, например: цвет, градиент, кривая. Соответственно, библиотека предустановок – это коллекция созданных пользователем предустановок, сохраненная в один файл.

Библиотеки предустановок могут храниться или в папке пользовательских настроек (когда речь о ваших личных предпочтениях), или в папке **Editor**, расположенной в папке Assets (когда данные относятся к проекту и вся команда должна работать с этими предустановками).

В дальнейшем библиотеки предустановок проекта могут быть легко добавлены в репозиторий системы контроля версий для удобного совместного использования всеми членами команды или для включения в пакеты Asset Store.

Разберемся, как создать предустановку. В качестве примера будем создавать предустановку цвета. Выберите объект **Camera** в области **Hierarchy** и щелкните на цветовом поле свойства **Background**. Откроется окно выбора цвета (рис. 3.21).

Далее выполните следующие действия:

1. Выберите нужный вам цвет
2. В нижней части вы увидите секцию **Swatches**

3. Щелкните на кнопке, находящейся слева от надписи **Click to add new preset**
4. Откройте меню (иконка с тремя точками). Вы можете:
5. переключить режим представления предустановок между списком (**List**) и сеткой (**Grid**) в контекстном меню
6. Выбрать одну из доступных библиотек предустановок (по умолчанию она называется **Default**)
7. Создать новую библиотеку (команда **Create New Library**). Выберите эту команду для создания собственной библиотеки
8. Во всплывающем окне **Create New Library** выберите размещение новой библиотеки – в папке пользовательских настроек (**Preferences Folder**) или в папке проекта **(Project Folder**)
9. Нажмите кнопку **Create**

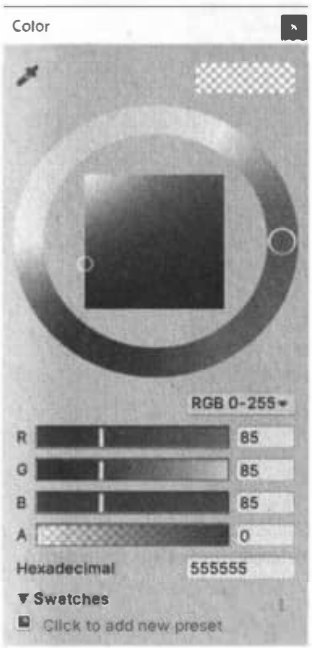


Рис. 3.21. Окно выбора цвета

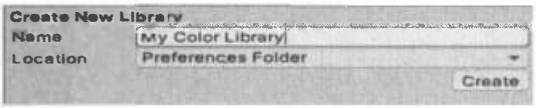


Рис. 3.22. Создание новой библиотеки предустановок

3.5.4. Блокировка инспектора

В Unity вы можете открыть несколько панелей **Inspector**. Обычно инспектор отображает детали текущего выбранного объекта, но иногда бывает полезным, чтобы в инспекторе отображался один объект, пока вы работаете с другими. Чтобы так сделать, у инспектора есть режим блокировки (*Lock*), который вы можете активировать при помощи маленькой иконки замка в верхнем правом угле окна инспектора.

Алгоритм прост:

1. Справа от значка замка есть меню вкладки. Откройте его и выберите команду **Add Tab, Inspector**. У вас появится две вкладки с названием Inspector (рис. 3.23).
2. Перетащите вторую вкладку вниз. Установите размер областей **Inspector** так, чтобы вам было удобно (рис. 3.24).
3. Решите какой из инспекторов вы будете блокировать и свойства какого объекта будете в нем отображать. Пусть вы хотите заблокировать верхний инспектор и отображать в нем свойства камеры. Выделите объект **Camera**. Затем в верхнем инспекторе нажмите кнопку замка.
4. Выберите какой-то другой объект. Нижний инспектор отобразит его свойства. Должно получиться, как показано на рис. 3.25.

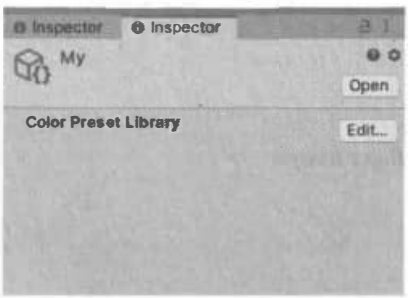


Рис. 3.23. Две вкладки "Inspector"

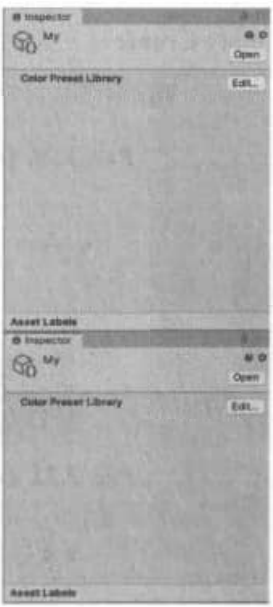


Рис. 3.24. Отображаются два инспектора одновременно

3.6. Панель инструментов

До сих пор нерассмотренной осталась панель инструментов Unity, а ведь она содержит действительно важные возможности:

1. Группа кнопок, управляющая позиционированием и трансформацией объектов. С ней мы уже знакомы

Рис. 3.25. Верхний инспектор всегда отображает свойства объекта "Camera", нижний – свойства выбранного в области "Hierarchy" объекта

2. Кнопки управления позиционированием и вращением при работе с инструментом Hand
3. Кнопки **Play/Pause/Step** – используются в игровом режиме, с их помощью можно запустить игру, приостановить ее выполнение и управлять выполнением (кнопка **Step**)
4. Выпадающее меню **Collab** позволяет управлять вашим облачным хранилищем. С его помощью вы можете опубликовать свой проект в облаке
5. Кнопка с изображением облака также управляет облачными сервисами. Ее нажатие приводит к появлению вкладки **Services**
6. Выпадающее меню **Account** – управляет учетной записью Unity
7. Выпадающее меню **Layers** – позволяет определить, какие объекты будут показаны в окне **Scene**
8. Выпадающее меню **Layout** – контролирует расположение (разметку) всех окон. При желании вы можете выбрать разметку, отличающуюся от используемой по умолчанию

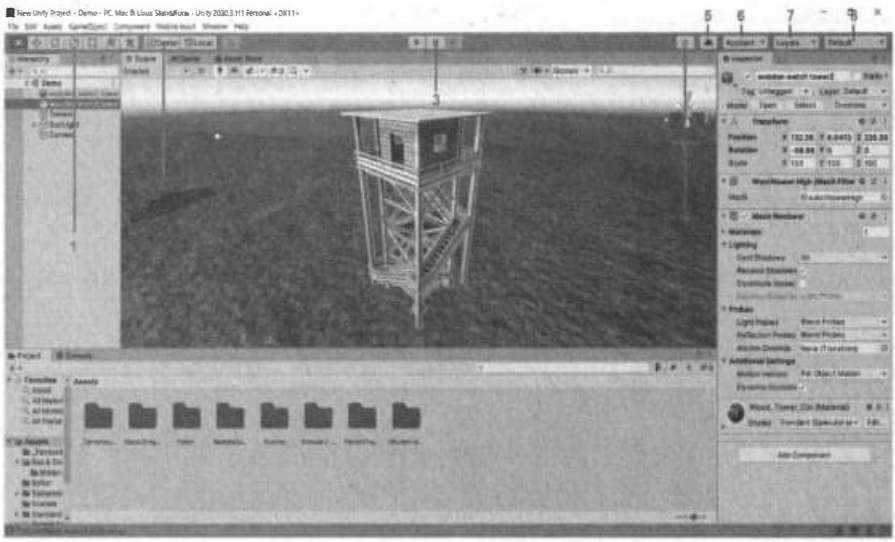


Рис. 3.26. Панель инструментов Unity

3.7. Поиск по сцене

В верхнем правом углу основной рабочей области находится поле поиска объектов. При работе с большими и сложными сценами будет полезен поиск определенных объектов. Используя возможности поиска в Unity, можно отфильтровать объект или группу объектов, которые вы хотите увидеть. Можно искать объекты по имени (**Name**), по типу компонента (**Type**), и, в некоторых случаях, по меткам объектов. Вы можете выбрать режим поиска из выпадающего меню **Search** (рис. 3.27).

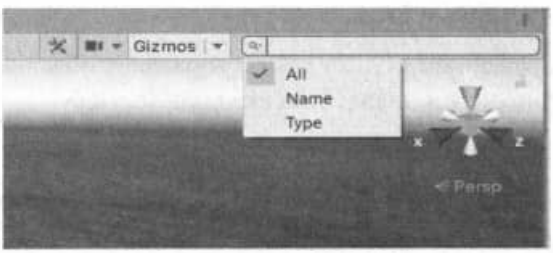


Рис. 3.27. Выбор режима поиска

Обратите внимание, что в отличие от аналогичного поля поиска в Обозревателе проекта (вкладка **Project**), данное поле производит поиск только объектов, находящихся на сцене и отображающихся в окне **Hierarchy**. На рис.

3.28 показан результат поиска по строке «wood». В области **Hierarchy** отображаются только объекты, соответствующие поисковой строке.

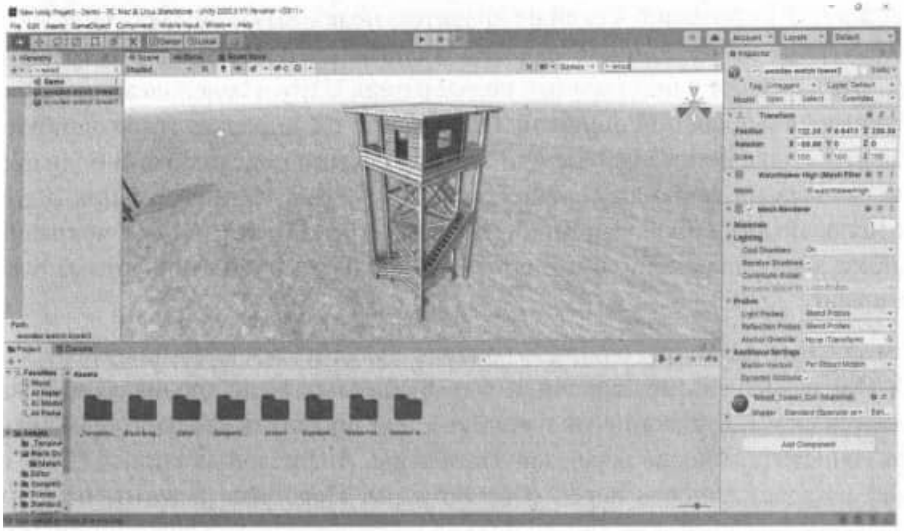


Рис. 3.28. Результат поиска

Примечание. Когда поиск активен, окна **Scene** и **Hierarchy** отличаются: окно **Scene** будет показывать объекты, не прошедшие по фильтру серым, а в окне **Hierarchy** будут отображаться только те объекты, имена которых соответствуют поисковому запросу, что и показано на рис. 3.28.

3.8. Консоль

Консоль показывает различные виды сообщений – информационные, предупреждения и, конечно же, сообщения об ошибках. С ней мы уже сталкивались ранее, когда пытались подружить необновленный пакет стандартных ассетов с самой последней версией Unity.

Рис. 3.29. Консоль

Если ваша игра не запускается, проверьте вкладку **Console**. Наверняка на ней будут ошибки, которые нужно исправить. Ошибки могут быть, как в

ваших скриптах, так и в импортированных наборах ассетов – далеко не все наборы нормально работают с последней версией Unity.

Обычно Unity сообщает, что ей не нравится, подсказывая вам, как исправить ошибку. Если вы не понимаете, в чем заключается ошибка, не расстраивайтесь. Вы далеко не единственный пользователь Unity и наверняка кто-то уже сталкивался с подобной ошибкой. Попробуйте скопировать текст ошибки и вставить в поле поиска Google или любого другого поисковика. В большинстве случаев на форумах сообщества Unity уже будет информация о том, как исправить ошибку. Напомню, что ранее было показано, как исправить ошибку, возникающую при импорте устаревшего пакета стандартных ассетов в Unity 2020.3.

Кнопка **Clear** позволяет очистить консоль и удалить все имеющиеся сообщения. Однако если вы не исправили суть проблемы, то сообщения об ошибках появятся снова. Кнопка-переключатель **Clear on Play** позволяет (если включена) очистить консоль перед запуском игры. Аналогичная кнопка **Clear on Build** очищает консоль перед сборкой игры. Подобные режимы полезны, чтобы вы видели только актуальные, а не устаревшие сообщения. Бывает так, что причина давно устранена, а сообщения об ошибках все равно отображаются и это немного раздражает.

Поле поиска на вкладке **Console** позволяет быстро найти интересующее вас сообщения. Три кнопки с числами справа от поля поиска – это счетчики-фильтры информационных сообщений, предупреждений и ошибок соответственно. По умолчанию консоль отображает все сообщения. Если вы хотите просмотреть только ошибки, тогда выключите отображение информационных сообщений и предупреждений (первые две кнопки).

3.9. Организация рабочего пространства

Редактор Unity обладает очень гибким интерфейсом, позволяющим подстраивать окна так, как будет удобно разработчику. Щелкните по заголовку любой из панелей – далее вы можете перетащить ее так, как вам будет удобно. На рис. 3.30 показано, что панель иерархии откреплена от стандартного места и не закреплена нигде.

Панели также могут быть откреплены от главного окна, и сгруппированы внутри отдельных окон редактора. Отдельные окна могут содержать набор панелей, точно так же, как главное окно Unity. На рис. 3.31 показана совершенно иная разметка. Просматривать сцену с такой разметкой гораздо удобнее, поскольку область сцены больше, чем по умолчанию.

Рис. 3.30. Панель иерархии откреплена

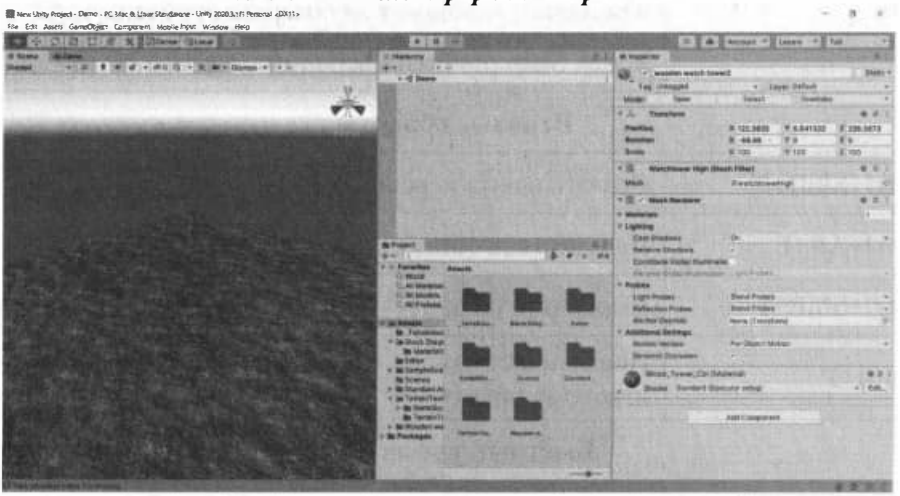


Рис. 3.31. Совершенно иная разметка интерфейса

После того, как вы изменили расположение окон Unity, вы можете сохранить это расположение и затем восстановить его в любой момент. Для этого откройте меню **Window, Layouts** и выберите команду **Save Layout**.... Назовите новое расположение окон и сохраните его, после этого вы сможете восстановить данное расположение окон, просто выбрав его в меню **Layout**.

3.10. Комбинации клавиш

Комбинации клавиш могут сделать процесс разработки игры более эффективным. Таблица 3.2 содержит некоторые полезные комбинации клавиш.

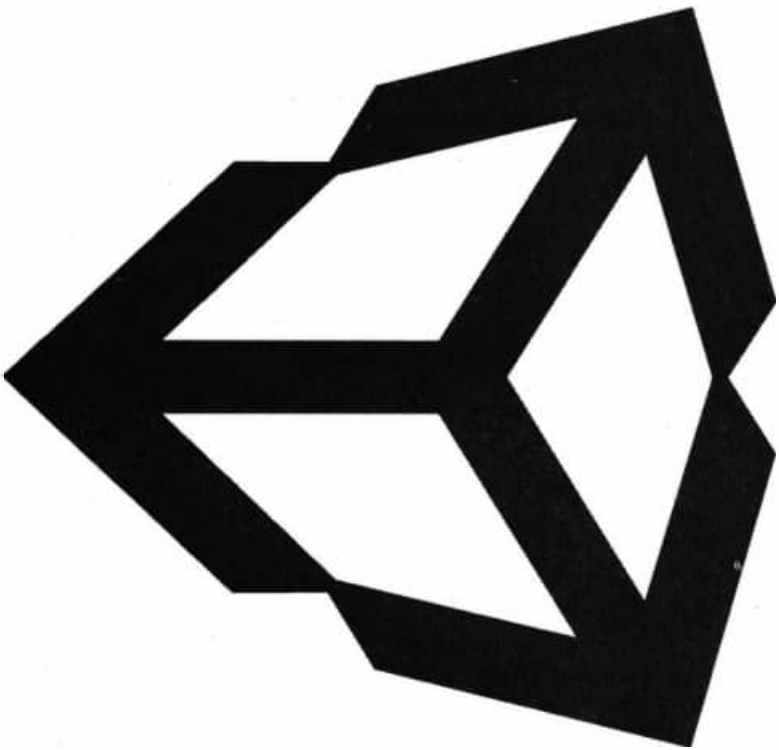
Также не забудьте просмотреть таблицу 3.1, где описаны горячие клавиши Обозревателя проекта.

Таблица 3.2. Некоторые полезные горячие клавиши

Горячая клавиша	Описание
Инструменты	
Q	Выбор инструмента Hand
W	Выбор инструмента Move (перемещение)
E	Инструмент Rotate (вращение)
R	Инструмент Scale (масштабирование)
T	Инструмент Rect Tool(используется в 2D режиме для манипулирования двухмерными объектами)
V	Vertex Snap (привязка вершин)
Игровые объекты	
Ctrl+Shift+N	Создать новый игровой объект
Ctrl+Alt+F	Переместить объект к точке вида
Ctrl+Shift+F	Выровнять по виду
Shift+F	Привязывает камеру окна Scene View к выбранному игровому объекту
Быстрый вызов окон	
Ctrl + 1	Scene (сцена)
Ctrl + 2	Game (режим игры)
Ctrl + 3	Inspector (инспектор свойств)
Ctrl + 4	Hierarchy (иерархия)
Ctrl + 5	Project (обозреватель проекта)
Ctrl + 6	Animation (анимация)

Ctrl + 7	Profiler (профайлер)
Ctrl + 9	Asset Store (магазин ассетов)
Ctrl + 0	Окно системы контроля версий
Ctrl + Shift + C	Консоль
Команды редактирования	
Ctrl + Z	Отменить последнее действие
Ctrl + Y	Повторить последнее действие
Ctrl + X	Вырезать
Ctrl + C	Скопировать
Ctrl + V	Вставить
Ctrl + D	Дублировать
Shift + Del	Удалить без запроса
Del	Удалить с запросом
Ctrl + F	Найти
Ctrl + A	Выбрать все
Ctrl + P	Запустить игру или остановить ее выполнение
Ctrl + Shift + P	Пауза

На этом все. Далее мы поговорим о создании геймплея – о камерах, сценах, игровых объектах, источниках света и т.д.



Глава 4.

Создание геймплея

Unity позволяет создавать игры с минимальными усилиями. Конечно, сил нужно потратить немало, но намного меньше, если бы вы создавали игру без использования Unity. Вам не нужен ни диплом программиста, ни команда художников-профессионалов. Нужно изучить только Unity и создать свою первую игру. Как только вы создадите первую игру, пусть даже самую простую, дальше будет все гораздо проще.

Без использования Unity создание игры, даже относительно простой, требует усилий целой команды – нужен программист, художник и 3D-дизайнер, способный создать трехмерные модели для игры. С помощью Unity вы можете создать игру самостоятельно – по крайней мере, первую и не очень сложную.

Данная глава раскрывает основные понятия, необходимые для создания компьютерных игр. Мы поговорим о сценах, игровых объектах, трансформациях, источников света, камерах.

4.1. Сцены

Сцена – это место, где происходит игра. Можно считать отдельную сцену отдельным уровнем. Если ваша игра предусматривает три уровня, то она будет содержать как минимум три сцены. Как минимум три, поскольку вам понадобятся дополнительные сцены, например, сцена меню, сцена выбора персонажа, сцена инвентаря и т.д.

Когда вы создаете новый проект, создается одна новая сцена. Сцена будет пуста за исключением некоторых важных объектов, которые по умолчанию добавляются на нее – ортографическая камера или камера перспективы с направленным светом – в зависимости от выбранного режима – 2D или 3D.

Основные операции над сценами:

1. Создание сцены – выполните команду меню **File, New Scene** или нажмите Ctrl + N
2. Сохранение сцены – выполните команду **File, Save Scene** или нажмите Ctrl + S
3. Открытие сцены – просто дважды щелкните на файле сцены в обозревателе проекта

После создания сцена будет сохранена в папке **Scene** – вы увидите сцену в обозревателе проекта (рис. 4.1).

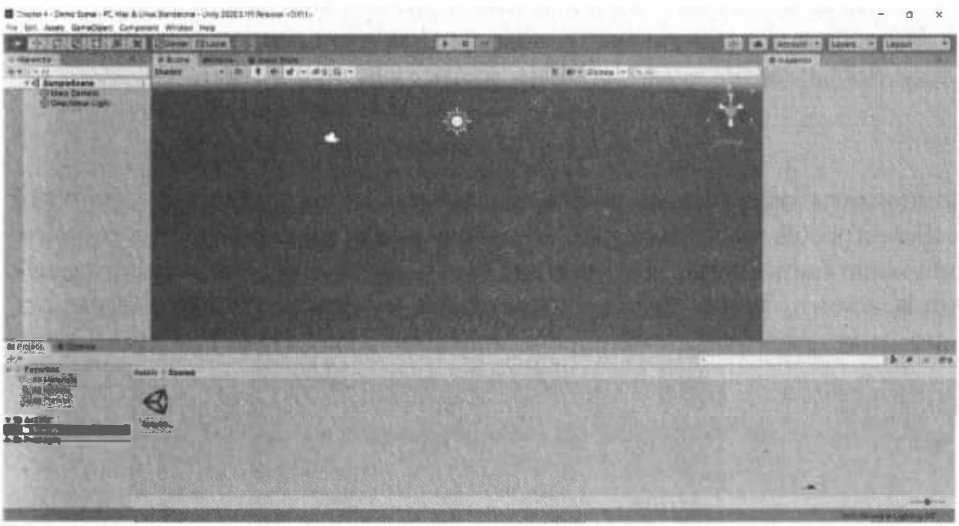


Рис. 4.1. Папка "Scenes": создана одна сцена

4.2. Игровые объекты

Самые важные объекты в Unity – это **игровые объекты** (GameObject). Важно понимать, что такое игровой объект и как его использовать.

По сути, каждый объект в игре — это GameObject. Однако сами по себе игровые объекты ничего не делают. Чтобы они стали персонажами, они требуют дополнительной настройки. Один игровой объект может быть предметом интерьера, другой – персонажем, третий – инструментом, например, оружием. Но как различать все эти игровые объекты? Как мы можем понять, какой из них является персонажем, а какой – предметом интерьера. Та же модель рыцаря может быть и предметом интерьера, и персонажем.

Все игровые объекты являются контейнерами. Чтобы действительно понять игровые объекты, нужно понять их составляющие, которые называются компонентами. В зависимости от того, какой вы хотите создать объект, вы будете добавлять различные комбинации компонентов к объекту. Представляйте GameObject как пустую миску, а компоненты как различные ингредиенты, которые составят ваш рецепт игрового процесса. Вы можете также создать ваши собственные компоненты, используя скрипты. Создание игрового объекта иногда занимает очень мало времени – нашли подходящую модель в Интернете, импортировали ее и добавили на сцену. Пусть это будет модель воина. Но сама по себе модель ничего не делает. Она будет просто стоять. Чтобы добавить логику нашему объекту, нужно написать скрипты. Тогда у объекта будет какой-то алгоритм, которому он будет следовать.

4.2.1. Компоненты

Компоненты заслуживают отдельного разговора. Как уже было отмечено, любой игровой объект содержит компоненты. Добавьте любой игровой объект (меню GameObject, 3D Object) на сцену. Каждый игровой объект содержит компонент **Transform**. Посмотрите вкладку **Inspector** – в нем вы сможете изменять параметры компонента Transform (рис. 4.2).

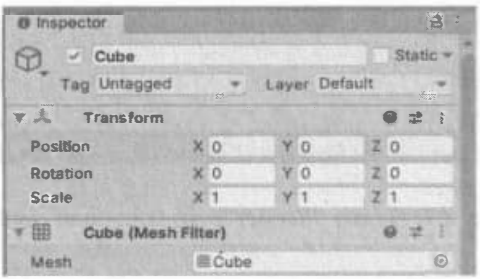


Рис. 4.2. Параметры компонента Transform

Компонент **Transform** определяет **Position** (положение), **Rotation** (вращение), и **Scale** (масштаб) каждого объекта в сцене. Свойства компонента **Transform** приведены в табл. 4.1.

Таблица 4.1. Свойства компонента Transform

Свойство	Описание
Position	Положение объекта в координатах X, Y, и Z
Rotation	Вращение вокруг осей X, Y, и Z, измеряется в градусах
Scale	Масштаб вдоль осей X, Y, и Z. Значение «1» означает исходный размер (размер, с которым был импортирован объект)

Примечание. Значения положения, вращения и масштаба измеряются относительно его родителя. Если у **Transform** нет родителя, тогда все эти свойства измеряются в мировом пространстве.

В Unity невозможно создать игровой объект без компонента **Transform**. Компонент **Transform** – один из самых важных компонентов, так как все свойства игрового объекта, связанные с трансформациями используют этот компонент. Он определяет положение, вращение и масштаб объекта в игровом мире/окне Scene.

Если у игрового объекта не будет компонента **Transform**, он будет не более чем некоторой информацией в памяти компьютера. Он не сможет эффективно существовать в игровом мире – система просто не будет знать, где его отобразить.

Компонент **Transform** также вводит концепцию, называемую *наследование*, которая используется редактором Unity и является критической частью работы с игровыми объектами.

Кроме компонента **Transform** у объекта могут быть и другие компоненты. Все они предоставляют дополнительную функциональность и делают объект именно тем, чем он должен быть. Твердые тела, коллайдеры, частицы, аудио – это все разные компоненты (или комбинации компонентов), которые могут быть добавлены к любому игровому объекту.

Меню **Component** позволяет добавить компоненты к выбранному игровому объекту. Создайте пустой игровой объект – **GameObject, Create Empty**. По умолчанию у пустого объекта есть только компонент Transform.

Сейчас мы добавим компонент **Rigidbody** на пустой игровой объект, который только что создали. Выделите его и выберите в меню **Component,**

Physics, Rigidbody. После этого, вы увидите, что в инспекторе отобразился компонент Rigidbody и его свойства (рис. 4.3).

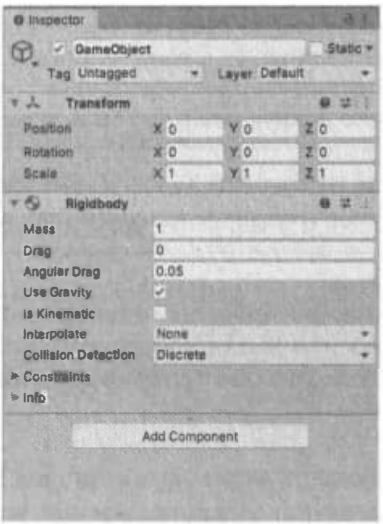


Рис. 4.3. Компонент Rigidbody добавлен к пустому объекту

Если нажать **Play** в то время как пустой игровой объект все еще выбран, вы можете получить небольшой сюрприз (рис. 4.4). Обратите внимание, как компонент Rigidbody добавил функциональность пустому игровому объекту: координата **Y** игрового объекта начинает уменьшаться, потому что физический движок в Unity заставляет игровой объект падать под действием силы тяжести.

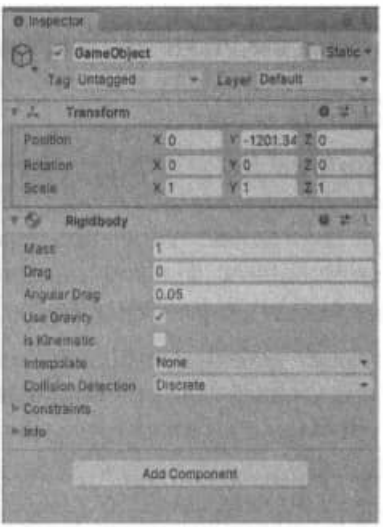


Рис. 4.4. Обратите внимание на изменение координаты Y

Второй способ добавления компонента – использование браузера компонентов. Нажмите кнопку **Add Component** в окне **Inspector** и выберите нужный вам компонент. Браузер обеспечивает удобную навигацию по компонентам с помощью категорий, а также имеет окно поиска, которое можно использовать, чтобы найти компоненты по имени.

Разработчик может прикрепить любое количество или комбинацию компонентов к одному игровому объекту. Некоторые компоненты работают лучше в сочетании с другими. Например, Rigidbody работает с любым коллайдером. Rigidbody контролирует Transform через физический движок Unity, а коллайдер позволяет Rigidbody сталкиваться и взаимодействовать с другими коллайдерами.

Во время разработки вам доступна справка по любому добавляемому компоненту. Нажмите маленький ? на заголовке компонента в инспекторе – будет открыта страница справочника компонентов.

У каждого компонента есть свои свойства. Так, на рис. 4.3 видно, что у RigidBody есть свойства Mass, Drag Angular Drag и т.д. Например, с помощью свойства Mass можно контролировать массу объекта.

Существующие свойства компонента можно изменять, как в редакторе при создании игры, так и во время выполнения игры посредством скриптов. Так, вы можете изменять массу персонажа в зависимости от надетой экипировки.

Как уже отмечалось ранее, есть два типа свойства: значения (*values*) и ссылки (*references*). Первые содержат обычные простые значения, например, значение массы. Вторые – ссылки на другие объекты, например, текстуры. Задать значения для свойства-ссылки очень просто: перетащите файл из обозревателя проекта на свойство-ссылку или же воспользуйтесь выбором объекта.

Компоненты могут включать ссылки на любые другие типы компонентов, игровых объектов или ассетов. Контекстное меню (рис. 4.6) компонента содержит следующие команды:

- **Reset** – восстанавливает значения свойств компонента, которые были до самой последней сессии редактирования
- **Remove Component** – удаляет компонент из игрового объекта. Обратите внимание, что некоторые комбинации компонентов, которые зависят друг от друга, работают только когда Rigidbody также прикреплено; вы увидите предупреждающее сообщение, если вы попробуете удалить компоненты, которые зависят от каких-либо других компонентов

- **Move Up/Down** – некоторые компоненты добавляют определенные визуальные эффекты. Готовый результат зависит от порядка применения таких компонентов к игровому объекту. Посредством этих команд вы можете изменить порядок применения компонентов
- **Copy Component/Paste Component As New/Paste Component Values** – команда **Copy Component** сохраняет тип и текущие настройки свойств компонента. Они могут быть вставлены в другой компонент того же типа с помощью команды **Paste Component Values**. Вы также можете создать компонент в объекте со скопированными свойствами, используя команду **Paste Component As New**
- **Find References In Scene** – позволяет найти используемые компонентом ссылки на сцене

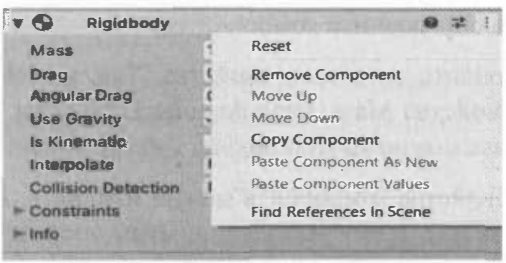


Рис. 4.6. Контекстное меню компонента

Внимание! В режиме игры вы можете изменять свойства компонентов любого игрового объекта так, как вам пожелается в режиме реального времени – в окне **Inspector**. Например, вы можете экспериментировать с разной массой персонажей, разными текстурами и т.д. Когда вы выйдете из режима игры, все значения свойств будут восстановлены – вернутся к значениям, которые были до запуска игры. Подобный рабочий процесс дает вам невероятную мощь для экспериментов, регулировки и совершенствования геймплея без лишних затрат времени.

4.2.2. Включение и отключение объектов

Посмотрите на окно **Inspector**. Рядом с названием объекта (слева от него) есть галка. Данный переключатель позволяет включать/выключать объект. Отключение игрового объекта позволяет временно убрать его со сцены. Вы можете включать/включать объект, как через окно инспектора, так и программно – в скриптах.

Нужно понимать, что произойдет при отключении родительского объекта. Если родительский объект деактивирован, его отключение также перезаписывает опцию *activeSelf* у всех его дочерних элементов, то есть вся иерархия объектов, от родителя до каждого его потомка, становится неактивной.

Данный процесс не меняет значения свойства *activeSelf* у дочерних элементов, они вернутся в первоначальное состояние, как только родитель будет снова активирован. Это важно понимать, иначе бы вам бы пришлось снова активировать все дочерние объекты.

Установить свойство программно в скрипте можно так:

```
obj.activeSelf = true;                    // показываем объект
obj.activeSelf = false;                   // скрываем объект
```

4.2.3. Статические игровые объекты

Многим процессам оптимизации нужно знать, будет ли объект двигаться во время игры. Например, рендеринг может быть оптимизирован путем объединения нескольких статических объектов в один большой объект.

В инспекторе объекта имеется флажок **Static**, позволяющий пометить игровой объект как статический. Статические объекты не будут двигаться во время геймплея.

Отметьте все объекты, которые не должны двигаться, как статические. Деревья, кусты, камни, скалы, дома – все это не должно (как правило) двигаться, поэтому можно оптимизировать игру, пометив соответствующие игровые объекты как статические.

4.2.4. Префабы

Наконец-то мы добрались до одной из ключевых концепций в Unity – **префабов**. Работать в Unity довольно удобно. Как уже было показано ранее, вы можете достаточно просто добавлять новые компоненты, изменять их значения в инспекторе. Но что делать, когда вы работаете над объектом, который должен многократно встречаться на сцене. Первое, что приходит в голову – просто копировать эти объекты, создавая дубликаты, но все равно все они будут редактироваться независимо друг от друга.

Обычно нужно, чтобы все экземпляры отдельно взятого объекта имели одинаковые значения свойств, чтобы при редактировании одного такого объекта

в сцене вам не пришлось повторно вносить те же изменения и во все остальные копии.

Для решения этой проблемы в Unity вы можете создавать префабы. **Префаб** – это особый тип ассетов (ресурсов), позволяющий хранить весь игровой объект со всеми компонентами и значениями свойств. Можно считать префаб шаблоном для создания экземпляров хранимого объекта в сцене. Любые изменения в префабе сразу же отражаются и на всех его экземплярах, при этом вы можете переопределять компоненты и настройки для каждого экземпляра в отдельности.

Важно понимать отличие *ассета* от *префаба*. Когда вы перетаскиваете файловый ассет в сцену, будет создан новый экземпляр такого объекта и все такие экземпляры изменятся при изменении оригинального ассета. Поведение похоже на префаб. Но ассет – это не префаб и у вас не получится добавить к нему компоненты или использовать особенности префабов, о которых мы поговорим далее.

Для создания префаба выберите команду меню **Assets, Create, Prefab Variant**. После этого перетащите объект со сцены в пустой (только что созданный) префаб. После этого можно создавать экземпляры префаба, просто перетаскивая его из окна обозревателя на сцену.

Имена объектов-экземпляров префабы, будут выводиться синим в окне **Hierarchy** (имена обычных объектов имеют черный цвет).

Как уже упоминалось выше, изменения в префабе автоматически применятся ко всем его экземплярам, однако вы можете изменять и отдельные экземпляры. Это полезно, когда вы желаете создать несколько похожих персонажей, но с внешними различиями, чтобы добавить реалистичности. Чтобы было четко видно, что свойство в экземпляре префаба изменено, оно показывается в инспекторе жирным шрифтом (если к экземпляру префаба добавлен совершенно новый компонент, то все его свойства будут написаны жирным шрифтом).

Экземпляры префаба можно создавать и программно – в коде скрипта. Например, вот таким незамысловатым образом можно создать стену программно:

```
for (int y = 0; y < 5; y++) {
   for (int x = 0; x < 5; x++) {
       GameObject cube = GameObject.
CreatePrimitive(PrimitiveType.Cube);
       cube.AddComponent<Rigidbody>();
```

```
        cube.transform.position = new Vector3(x, y, 0);
    }
}
```

Инспектор экземпляра префаба содержит три кнопки, которые у обычных объектов отсутствуют: **Select** (Выделить), **Revert** (Отменить) и **Apply** (Применить).

Кнопка **Select** выделяет файл префаба, из которого был получен данный экземпляр, что позволяет быстро найти и отредактировать оригинальный префаб, применяя изменения ко всем его экземплярам.

Также вы можете сохранить переопределенные свойства из экземпляра в сам оригинальный префаб с помощью кнопки **Apply** (измененные значения положения и вращения трансформации не применяются по очевидным причинам). Это позволяет эффективно редактировать все экземпляры через любой из них, и это быстрый и правильный способ вносить глобальные изменения.

Если вы экспериментируете с переопределением свойств, но в итоге решаете, что изначальные свойства (заданные в префабе) для вас предпочтительнее, то вы можете отменить все сделанные переопределения с помощью кнопки **Revert**, вернув экземпляр в изначальное состояние.

У экземпляров префаба есть кнопка **Open**, позволяющая открыть оригинальный префаб. На рис. 4.7 показаны префабы на сцене. Для них доступны кнопки **Open** и **Select**.

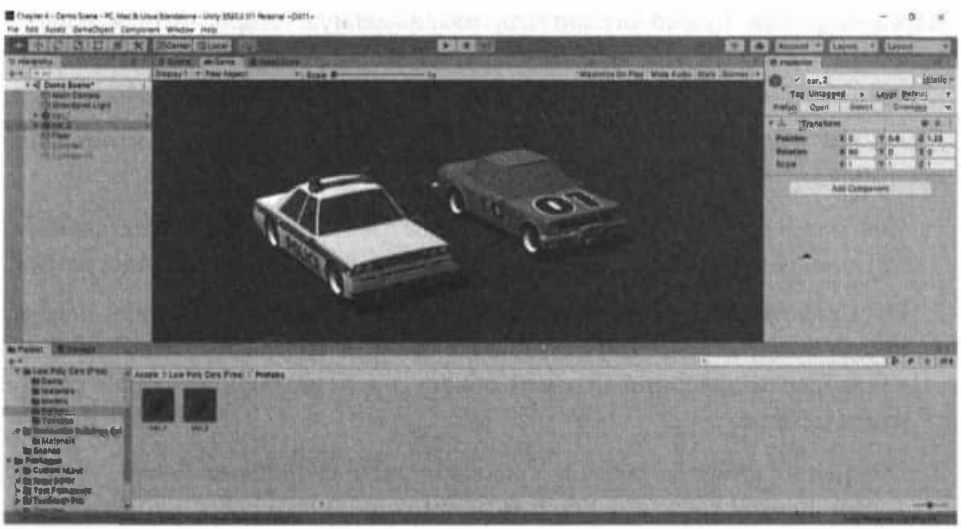


Рис. 4.7. Префабы автомобилей на сцене

4.2.5. Программное создание экземпляров префабов

Префабы – это набор ранее установленных игровых объектов и компонентов, использующихся более одного раза за всю игру. Префабы используются, как правило, при создании экземпляров сложных игровых объектов, например, персонажей игры.

Альтернативой создания экземпляров префабов является программное создание игрового объекта. Создание экземпляра префаба имеет много преимуществ над альтернативным подходом:

- Разработчик может создать экземпляр префаба с полным функционалом при помощи одной строчки кода. Создание эквивалентного GameObject из кода в среднем занимает 5 строк кода, а обычно все же больше. Все зависит от сложности игрового объекта
- Есть возможность легко и быстро настраивать, тестировать и модифицировать префабы в сцене и инспекторе
- Есть возможность изменения префаба, экземпляр которого будет создан, без изменения кода, отвечающего за формирование объекта. Простая ракета может быть превращена в супер-заряженную ракету без изменения кода

Однако все же есть ситуации, когда без программирования не обойтись. Например, изменения свойств игрока во время игры: игрок выбрал другое оружие, другую экипировку, следовательно, масса игрока изменилась. Если масса возросла, нужно ограничить максимальную скорость перемещения персонажа, сделать его медленнее. Аналогично, с гоночной игрой: игрок «протюнинговал» свою машину, значит, она должна ехать быстрее.

Далее мы рассмотрим несколько полезных сценариев программирования префабов:

1. Построение стены. Можно, конечно, построить стену самостоятельно, но это довольно рутинная задача, поэтому проще выстроить ее программно.
2. Пуск ракеты. Мы создаем экземпляр префаба ракеты, когда пользователь жмет кнопку стрельбы. Префаб ракеты будет содержать меш, Rigidbody, коллайдер и дочерний игровой объект, представляющий собой след от пуска ракеты.
3. Попадание ракеты в робота. У нас есть полнофункциональный робот, который должен быть заменен префабом сломанного робота после попада-

ния в него ракеты. Программно вы можете «взорвать» робота одной всего строчкой кода, которая заменяет игровой объект на префаб.

Построение кирпичной стены

В следующих главах мы поговорим о скриптинге более подробно. Пока вам нужно знать, что сценарии для Unity можно было создавать на двух языках – C# и UnityScript, который является собственным диалектом JavaScript. Однако в 2017 году Unity отказалась от UnityScript в пользу C#. Поэтому вам ничего другого не остается, как использовать C#. При использовании версии 2017 возможно все еще создавать скрипты на UnityScript.

Итак, рассмотрим сценарий построения стены (пусть не китайской, а небольшой стены, которая будет состоять из встроенного объекта Cube), листинг 4.1.

Листинг 4.1. Сценарий NewBehaviourScript.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NewBehaviourScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        for (int y = 0; y < 5; y++) {
            for (int x = 0; x < 5; x++) {
                GameObject cube = GameObject.
CreatePrimitive(PrimitiveType.Cube);
                cube.AddComponent<Rigidbody>();
                cube.transform.position = new Vector3(x, y,
0);
            }
        }
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Да, этот код вы уже видели, но сейчас мы его оформили по всем правилам – в виде класса с методом Start().

Создать скрипт можно так:

1. Щелкните правой кнопкой мыши на разделе **Assets** в обозревателе проекта
2. Выберите команду **Create, C# Script**
3. Оставьте имя по умолчанию, оно сейчас не важно
4. Дважды щелкните по скрипту. Откроется редактор, используемый в вашей системе для редактирования .cs-файлов
5. Замените содержимое по умолчанию на содержимое листинга 4.1

> **Совет**. Хотя с Unity и поставляется Visual Studio, я рекомендую установить редактор Visual Studio Code. Это легкий редактор кода от Microsoft и обладает всеми возможностями редактора от IDE Visual Studio, но при этом сама IDE не загружается. Это оптимальный вариант, чтобы не использовать лишние ресурсы системы.

Использовать скрипт можно так:

- Создайте пустой игровой объект с помощью команды меню GameObject, Create Empty
- Перетащите наш скрипт на только что созданный объект в окно инспектора

Вид панели Inspector у вас должен быть таким, как на рис. 4.8. Также обратите внимание на сцену: это сцена по умолчанию из пакета Low Poly Cars, доступного по адресу **https://free3d.com/ru/3d-model/cartoon-vehicles-low-poly-cars-free-874937.html**.

Вы можете использовать пустую сцену, можете открыть любую из созданных ранее вами или же такую как у меня – сцену с машинками. Если будете использовать сцену с машинками, для объекта car2 установите координату **X** в *-2*.

Запустите игру (рис. 4.9). Вы увидите, как первый ряд кубиков выстроен, а второй разбивается о первый и кубики рассыпаются. Все это благодаря физике Unity и компоненту RigidBody, который мы добавили к нашему игрово-

му объекту. Подробно такое поведение объектов мы будем рассматривать в следующей части книги, когда будем говорить о физике.

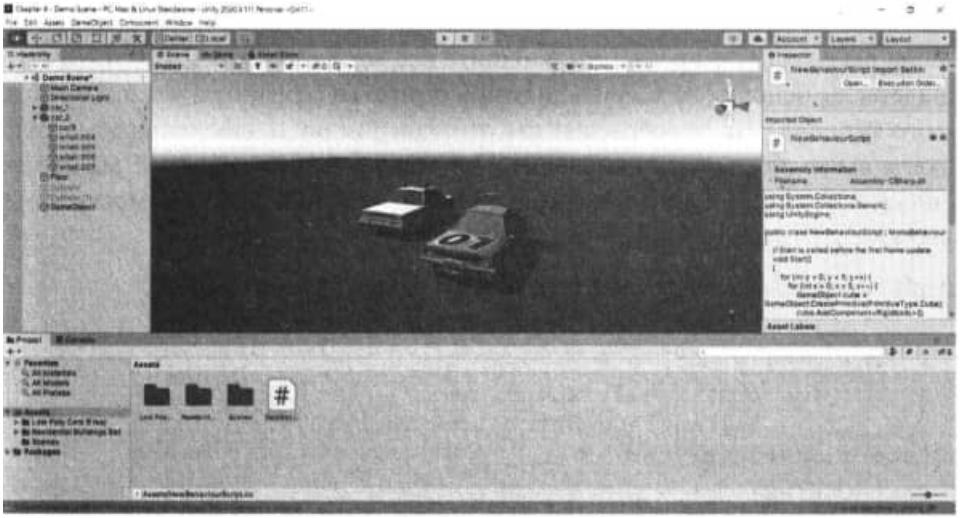


Рис. 4.8. Скрипт присвоен пустому GameObject

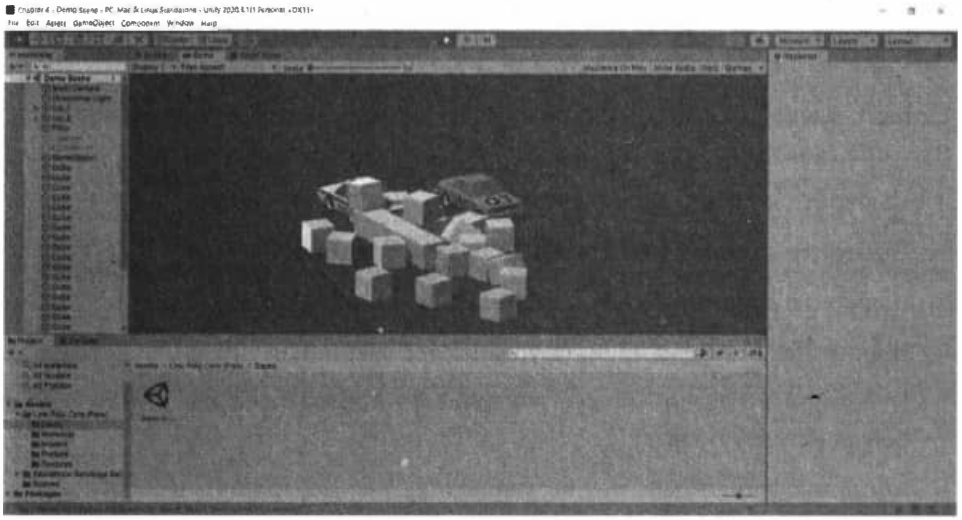


Рис. 4.9. Игра запущена

На отдельный кирпич приходится по 3 строки, отвечающие за функциональность: строки CreatePrimitive(), AddComponent() и строка, задающая позицию куба. Пока что не так и плохо, но все кирпичи у нас без текстур. Каждое

дополнительное действие, которое мы хотим сделать с кирпичом, вроде изменения текстуры, трения или массы Rigidbody, требует дополнительную строку.

Если вы создадите префаб и выполните все установки вручную, то вам понадобится использовать лишь по одной строке на создание и установку каждого кирпича. Это освобождает вас от поддержания и изменения тонн кода, когда вы хотите внести изменения. При использовании префаба вы просто вносите изменения в него и жмете **Play**. Нет никакой необходимости изменять код.

При использовании префаба куба вам понадобится вот такой код:

```
    for (int y = 0; y < 5; y++) {
        for (int x = 0; x < 5; x++) {
            Instantiate(brick, new Vector3(x, y, 0),
Quaternion.identity);
        }
    }
```

Как видите, мы просто создаете кирпич и указываете его координаты. Вам не нужно ни создавать куб, ни добавлять в него Rigidbody, ни задавать текстуры (а это в реальном приложении придется сделать). Все это задано в префабе, вам нужно только использовать его.

Остается дело за малым – нужно создать префаб. Для этого выполните следующие действия:

1. Выберите команду меню **GameObject, 3D Object, Cube**
2. Выберите команду **Component, Physics, Rigidbody**
3. Выберите команду **Assets, Create, Prefab Variant**
4. В окне обозревателя проекта установите имя нового префаба – «Brik»
5. Перетащите созданный куб в иерархию поверх префаба «Brik» в окне обозревателя проекта
6. Когда префаб создан, можно удалить куб из иерархии – выделите его и нажмите **Del**

Мы создали наш префаб кирпича, так что теперь надо присоединить переменную **brick** к нашему скрипту. Когда вы выбираете пустой GameObject, что содержит скрипт, переменная **brick** будет видна в инспекторе.

Теперь перетащите префаб «Brick» из окна Project на переменную **brick** в инспекторе. Нажмите **Play** и вы увидите стену, построенную из префабов.

Пуск ракеты

Префабы идеально подходят для подобного сценария. Орудие создает экземпляр префаба ракеты, когда пользователь жмет кнопку пуска ракеты. Префаб будет содержать меш, Riggidbody, коллайдер и дочерний игровой объект, использующийся для отображения следа от ракеты.

Когда ракета врезается, создается экземпляр префаба взыва. Он содержит систему частиц, источник света, который угасает со временем и скрипт урона, который регистрирует урон окружающим объектам.

Да, вы можете собрать ракеты из кода, вручную добавляя нужные компоненты и устанавливая свойства. Но гораздо проще создать экземпляр префаба. После этого вы сможете создавать ракету одной строчкой кода, независимо от того, насколько сложный префаб ракеты. После создания экземпляра префаба вы можете изменить любые его свойства, например, установить скорость Rigidbody ракеты, изменить направление ее полета и т.д.

Сейчас мы не будем создавать префаб ракеты, поскольку это выходит за рамки этой главы (мы еще мало знакомы с графикой и физикой Unity – эти темы рассматриваются в следующей части), но мы покажем код, позволяющий запустить ракету функцией Fire():

```
// Нужно, чтобы rocket была rigidbody.
// Этим мы не позволим назначить префаб без rigidbody
public Rigidbody rocket;
public float speed = 10f;

void Fire () {
    Rigidbody rocketClone = (Rigidbody) Instantiate(rocket,
transform.position, transform.rotation);
    rocketClone.velocity = transform.forward * speed;

    // Вы также можете получить доступ к другим компонентам ракеты
    rocketClone.GetComponent<MyRocketScript>().DoSomething();
}

// Вызваем метод Fire при нажатии ctrl или мышки
void Update () {
    if (Input.GetButtonDown("Fire1")) {
```

```
            Fire();
        }
    }
```

Замена персонажа другим префабом

При создании шутера обязательно будут два типа персонажей – персонаж игрока и вражеский персонаж. Когда игрок попадает во вражеского персонажа, нужно это как-то отобразить. Например, можно отобразить зомби или летательный аппарат с повреждениями. Поэтому вам понадобятся два префаба для каждого типа вражеского персонажа – один будет использоваться для нормального состояния, второй – для состояния повреждения/смерти.

Без использования префабов, вам, скорее всего, придется позаботиться об удалении нескольких скриптов, добавлении некоторой дополнительной логики, чтобы убедиться, что никто не будет атаковать уже мертвого врага, и о других задачах очистки.

При использовании префабов вам нужно удалить всего персонажа и заменить его другим префабом, например, префабом обломков космического корабля. Это даст вам больше возможностей. Во-первых, вы сможете использовать другой материал для поврежденного/мертвого персонажа. Во-вторых, вы можете присоединить совершенно другие скрипты.

Любой из этих вариантов может быть достигнут разовым вызовом Instantiate(). Вам нужно просто привязать к нему правильный префаб. Важно помнить, что обломки, которые вы Instantiate() (создаете их экземпляр), могут быть сделаны из совершенно отличных от оригинала объектов. Например, если у вас есть самолет, вы можете смоделировать 2 версии. Одна из которых – самолет, состоящий из одного GameObject с Mesh Renderer и скриптами физики самолета. Если эта модель будет единым объектом, то игра будет работать быстрее, поскольку вы сможете сделать модель с меньшим количеством треугольников, а ввиду того, что в результате будет меньше объектов, то и рендер будет проходить быстрее.

Вот как можно создать префаб поврежденного персонажа (не обязательно это может быть самолет, это может быть поврежденный робот или зомби):

1. Смоделируйте объект вражеского персонажа в любом приложении моделирования (3D Max и др.);
2. Создайте пустую сцену;
3. Перетащите модель на созданную сцену;

4. Добавьте компонент Rigidbody всем частям, выделив все части и выбрав команду меню **Component, Physics, Rigidbody**;
5. Добавьте коллайдеры Box Collider всем частям, выделив их и выбрав команду меню **Component, Physics, Box Collider**;
6. Для дополнительного спецэффекта, добавьте системы похожих на дым частиц в виде дочерних объектов для каждой части;
7. Теперь у вас есть объект с множеством раздельных деталей. Они будут падать на землю по законам физики и будут создавать след из частиц, в силу того, что к ним присоединены системы частиц (если это нужно в вашей игре, например, для падающего космического корабля или самолета – это нужно, для распадающегося на части робота – нет). Нажмите **Play** для предварительного просмотра того, как ваша модель будет себя вести, и проведите все необходимые поправки;
8. Выберите **Assets, Create, Prefab Variant**;
9. Перетяните на префаб корневой GameObject, содержащий все части вашего объекта.

Следующий пример покажет, как эти шаги моделируются в коде:

```
public GameObject wreck;        // аварийный объект

// Для примера мы разбиваем наш объект через 3 секунды
IEnumerator Start() {
    yield return new WaitForSeconds(3);
    KillSelf();
}

// Метод вызывается при попадании в объект
void KillSelf () {
    // Создаем игровой объект в том же положении,
    // в котором мы находимся
GameObject wreckClone = (GameObject) Instantiate(wreck,
transform.position, transform.rotation);

    // Иногда нужно перенести некоторые параметры
    // из обычного объекта
    // в аварийный
    wreckClone.GetComponent<MyScript>().someVariable =
GetComponent<MyScript>().someVariable;

    // Убиваем себя
    Destroy(gameObject);
```

}

}

4.3. Ввод

Unity поддерживает самые разные устройства ввода:

- Клавиатуры
- Мыши
- Джойстики
- Геймпады
- Сенсорные экраны мобильных устройств
- Устройства для виртуальной реальности

Кроме того, Unity может воспользоваться микрофоном компьютера и веб-камерой для ввода аудио и видео данных. Но об этом мы поговорим во второй части этой книги.

4.3.1. Традиционный ввод

Unity поддерживает ввод с клавиатуры, джойстика и геймпада. Виртуальные оси и кнопки могут быть созданы в **Input Manager**, а конечные пользователи смогут настраивать ввод с клавиатуры на экране конфигурационного диалогового окна.

Виртуальные оси доступны из скриптов по их именам. При создании каждого проекта по умолчанию создаются следующие оси ввода по умолчанию:

- Horizontal и Vertical привязаны к **w**, **a**, **s**, **d** и клавишам направления
- Fire1, Fire2, Fire3 привязаны к клавишам Control, Option (Alt) и Command (macOS) соответственно
- Mouse X и Mouse Y привязаны к перемещениям мыши
- Window Shake X и Window Shake Y привязаны к перемещению окна

Как правило, в большинстве случаев это достаточно, но вы можете добавить новые виртуальные оси, используя команду меню **Edit, Project Settings, Input Manager**. Вы увидите Input Manager, в котором вы сможете изменить настройки каждой оси (рис. 4.10).

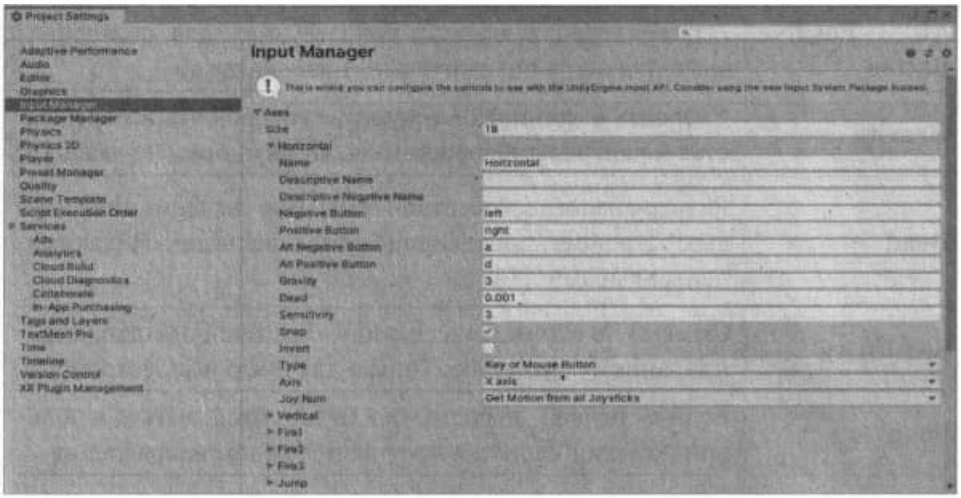


Рис. 4.10. Input Manager

Вы привязываете каждую ось к двум кнопкам на джойстике, мыши или клавиатуре. Таблица 4.2 содержит описание свойств каждой оси.

Таблица 4.2. Свойства виртуальной оси

Свойство	Функция
Name	Имя, используемое для доступа к этой оси из скрипта
Descriptive Name	Имя положительного значения, отображаемое на вкладке Input диалогового окна Configuration в автономных сборках
Descriptive Negative Name	Имя отрицательного значения, отображаемое на вкладке Input диалогового окна Configuration в автономных сборках
Negative Button	Кнопка, используемая для смещения значения оси в отрицательном направлении
Positive Button	Кнопка, используемая для смещения значения оси в положительном направлении
Alt Negative Button	Альтернативная кнопка, используемая для смещения значения оси в отрицательном направлении

Alt Positive Button	Альтернативная кнопка, используемая для смещения значения оси в положительном направлении
Gravity	Скорость в единицах в секунду, с которой ось возвращается в нейтральное положения, когда кнопки не нажаты
Dead	Размер аналоговой мертвой зоны. Все значения аналоговых устройств, попадающие в этот диапазон, считаются нейтральными
Sensitivity	Скорость в единицах в секунду, с которой ось движется к заданному значению. Только для цифровых устройств
Snap	Если включено, значение оси будет сбрасываться в ноль при нажатии кнопки в противоположном направлении
Invert	Если включено, Negative Buttons будут выдавать положительные значения, и наоборот
Type	Тип ввода, который будет управлять осью. Например, **Key or Mouse Button**, означает, что этой осью будет управлять или клавиша клавиатуры или кнопка мыши. Значение **Mouse Movement** означает, что осью будет управлять перемещение мыши, а значение **Joystick Axis** – управление будет осуществляться через джойстик
Axis	Ось подключенного устройства, которая будет управлять этой осью
Joy Num	Подключенный джойстик, который будет управлять этой осью

Примечание. Как со всем этим разобраться? Да довольно просто. Посмотрите на горизонтальную ось. Обычно перемещением игрока управляют клавиши WASD. При этом клавиши **W** и **S** – это вверх и вниз (вертикальная ось), а клавиши **A** и **D** – влево и вправо. Нейтральное положение оси соответствует 0. Соответственно, когда пользователь пойдет влево, то значение оси будет уменьшаться. Смотрим, какая клавиша назначена для **Negative Button** и **Alt Negative Button**. Правильно, это будут клавиши **left** (стрелка влево) и **a** (клавиша A). Когда игрок пойдет вправо, то значение оси будет увеличиваться, следовательно, за это отвечают клавиши **Positive Button** и **Alt Positive Button**. Для горизонтальной оси это будут клавиши **right** и **d**. Все достаточно просто.

Текущее состояние можно запросить из скрипта так:

```
value = Input.GetAxis ("Horizontal");
```

Это в случае ввода с джойстика и клавиатуры. Однако изменения осей Mouse и Window Shake показывают, насколько мышь или окно сдвинулись по сравнению с последним кадром. Это значит, что они могут быть больше, чем 1 или меньше, чем –1, когда пользователь быстро двигает мышь.

Можно создавать несколько осей с одним именем. При получении ввода, будет возвращаться ось с наибольшим абсолютным значением. Это позволяет назначить больше одного устройства ввода на одно имя оси. Например, создайте одну ось для ввода с клавиатуры и одну ось для ввода с джойстика с одинаковым именем. Если пользователь использует джойстик, ввод будет идти с джойстика, иначе ввод будет идти с клавиатуры. Таким образом, вам не нужно учитывать, откуда приходит ввод при написании скриптов, что очень удобно.

Таблица 4.3 содержит названия клавиш (keys), которые вы можете использовать в Unity.

Таблица 4.3. Названия клавиш

Тип клавиши	**Значение**
Обычные клавиши	a, b, c, d и т.д.
Цифровые клавиши	1, 2, 3 и т.д.
Клавиши стрелок	up, down, left, right
Клавиши цифровой клавиатуры	[1], [2], [3], [+], [equals]
Специальные комбинации клавиш (без кавычек)	«right shift», «left shift», «right ctrl», «left ctrl», «right alt», «left alt», «right cmd», «left cmd»
Клавиши мыши	mouse0, mouse1, mouse2

Кнопки джойстика (без кавычек)	«joystick button 0», «joystick button 1», «joystick button 2», …
Кнопки джойстика (от заданного джойстика):	joystick 1 button 0», «joystick 1 button 1», «joystick 2 button 0», …
Специальные клавиши	backspace, tab, escape, return, space, delete, enter, insert, home, end, «page up», «page down»
Функциональные клавиши	f1, f2, f3, …

Названия, используемые для определения кнопок, одни и те же при написании скриптов и в окне **Inspector**. Получить значение клавиши можно так:

```
value = Input.GetKey ("a");
```

Ось может иметь значение от –1 до 1. На нейтральное положение указывает 0

4.3.2. Ввод с мобильного устройства

Класс Input[1] поддерживает также и сенсорные экраны. Он предоставляет доступ к тачскрину, акселерометру и географическим/локационным данным.

Некоторые устройства вроде iPhone, iPad и устройства некоторых других производителей способны отслеживать сразу несколько нажатий на экран одновременно. Вы можете получить статус каждого нажатия на протяжении последнего кадра через массив Input.touches.

Каждое нажатие пальцем представлено в структуре данных Input.Touch (табл. 4.4).

Таблица 4.4. Состав структуры Input.Touch

Свойство	Описание
fingerId	Уникальный индекс для нажатия

1 Подробное описание класса на русском языке доступно по ссылке https://docs.unity3d.com/ru/current/ScriptReference/Input.html

position	Позиция нажатия на экран
deltaPosition	Изменение позиции на экране с последнего кадра
deltaTime	Количество времени, которое прошло с тех пор как изменилось последнее состояние
tapCount	Счетчик нажатий. Позволяет разработчику узнать, сколько раз пользователь нажал на экран без перемещения пальцев в стороны. На устройствах, которые не ведут подсчет нажатий, это значение всегда будет равно 1
phase	Описывает так называемые «фазы» или состояния нажатия. Они помогают вам определить более точно, что происходит на экране: Began – палец только прикоснулся к экране. Moved – палец перемещается по экрану Stationary – палец прикоснулся к экрану, но с последнего кадра не двигался Ended – палец оторвался от экрана, последняя фаза нажатия Canceled – система отказывается отслеживать нажатия, например, пользователь приложил экран к своему лицу или сделал более пяти нажатий одновременно. Финальная стадия нажатия. После отслеживать нажатия нет смысла

Рассмотрим пример кода, который выпускает луч в местах прикосновения пользователя к экрану (лист. 4.2).

Листинг 4.2. Скрипт выпускает луч в местах прикосновения к экрану

```
var particle : GameObject;

function Update () {
    for (var touch : Touch in Input.touches) {
        if (touch.phase == TouchPhase.Began) {
            // Выпускаем луч по координатам начала
прикосновения
```

```
            var ray = Camera.main.ScreenPointToRay (touch.
position);
            if (Physics.Raycast (ray)) {
                // Создаем частицу
                Instantiate (particle, transform.position,
transform.rotation);
            }
        }
    }
}
```

4.3.3. Акселерометр

При разработке игр для мобильных устройств нужно использовать нюансы и технические возможности, характерные для таких гаджетов. Например, большинство мобильных устройств оснащено акселерометром.

При движении мобильных устройств, встроенный акселерометр передает линейное ускорение, которое изменяется вдоль трех основных осей в трехмерном пространстве. Ускорение вдоль каждой оси сообщается непосредственно аппаратным обеспечением как значение G-Force.

Значение 1,0 представляет собой нагрузку около +1g вдоль заданной оси, а величина –1,0 представляет –1g. Если вы держите устройство в вертикальном положении (с кнопкой «домой» внизу) перед собой, ось **X** (положительная) будет по правой стороне, ось **Y** (положительная) будет направлена вверх, а ось **Z** (положительная) будет указывать на вас.

Вы можете получить значение акселерометра, путем доступа к свойству Input.acceleration.

Приведенный ниже пример кода позволяет перемещать объект, используя акселерометр (лист. 4.3.).

Листинг 4.3. Перемещение объекта в зависимости от положения акселерометра

```
var speed = 10.0;
function Update () {
    var dir : Vector3 = Vector3.zero;

    // подразумевается, что устройство удерживается
параллельно земле и
    // кнопка Домой находится в правой руке

    // ремаппинг оси ускорения устройства в координаты игры:
```

```
    // 1) XY плоскость устройства отображается в плоскость XZ
    // 2) вращаем на 90 градусов вокруг Y-оси
    dir.x = -Input.acceleration.y;
    dir.z = Input.acceleration.x;

    // фиксируем вектор ускорения на юните сферы
    if (dir.sqrMagnitude > 1)
        dir.Normalize();

    // Заставляем его двигаться 10 метров в секунду вместо 10
метров в кадр
    dir *= Time.deltaTime;

    // Двигаем объект
    transform.Translate (dir * speed);
}
```

С акселерометром случают проблемы. Дело в том, что показания акселерометра могут быть прерывистыми и с шумом. Применив низкочастотную фильтрацию на сигнал, вы сгладите его и избавитесь от высокочастотного шума.

Приведенный ниже скрипт демонстрирует, как применить низкочастотную фильтрацию на показания акселерометра:

```
var AccelerometerUpdateInterval : float = 1.0 / 60.0;
var LowPassKernelWidthInSeconds : float = 1.0;

private var LowPassFilterFactor : float =
AccelerometerUpdateInterval / LowPassKernelWidthInSeconds;

private var lowPassValue : Vector3 = Vector3.zero;

function Start () {
    lowPassValue = Input.acceleration;
}

function LowPassFilterAccelerometer() : Vector3 {
    lowPassValue = Vector3.Lerp(lowPassValue, Input.
acceleration, LowPassFilterFactor);
    return lowPassValue;
}
```

Чем больше значение LowPassKernelWidthInSeconds, тем медленнее фильтруется значение, которое будет приближаться к значению входного образца (и наоборот).

Также учтите, что Unity замеряет результат при частоте 60 Гц и сохраняет его в переменную. Другими словами, система может сделать два замера за один кадр и один замер за следующий кадр. Вы можете получить доступ ко всем замерам, выполненным акселерометром в текущем кадре. Следующий код иллюстрирует простое среднее всех событий акселерометра, которые были собраны в течение последнего кадра:

```
var period : float = 0.0;
var acc : Vector3 = Vector3.zero;
for (var evnt : iPhoneAccelerationEvent in iPhoneInput.
accelerationEvents) {
    acc += evnt.acceleration * evnt.deltaTime;
    period += evnt.deltaTime;
}
if (period > 0)
    acc *= 1.0/period;
return acc;
```

4.4. Трансформации объектов. Компонент Transform

4.4.1. Изменение трансформации объекта

Вы уже немного знакомы с компонентом **Transform**. Настало время познакомиться с ним подробнее. Компонент **Transform** используется для хранения позиции, вращения, размеров и состояния наследования игрового объекта, поэтому он очень важен.

Компонент **Transform** является обязательным – он всегда добавлен к игровому объекту. Невозможно его удалить или создать игровой объект без него.

Компонент **Transform** управляются в 3D пространстве по осям X, Y, и Z, или в 2D пространстве просто по **X** и **Y**. В Unity эти оси представлены красным, зеленым и синим цветами соответственно.

Параметры Transform можно изменять следующими способами:

- В окне **Scene** – можете изменять Transform, используя инструменты **Translate, Rotate** и **Scale** (двигать, вращать и масштабировать). Эти инструменты расположены в верхнем левом углу редактора Unity
- В окне **Inspector** – путем ввода значений в соответствующие поля

- Программно – используя код C#, как было показано ранее

Инструменты перемещения, вращения и масштабирования можно применять к любому объекту на сцене. Когда вы щелкаете на объекте, появляется гизмо инструмента. Вид гизмо зависит от выбранного инструмента (рис. 4.11).

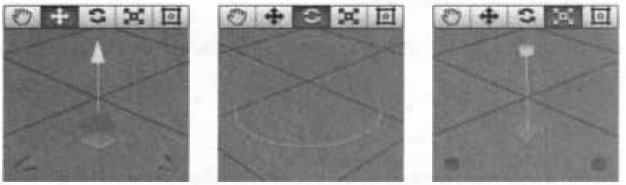


Рис. 4.11. Вид гизмо в зависимости от выбранного инструмента

Если вы нажмете на одну из трех осей гизмо и потянете, ее цвет изменится на *желтый*. По мере движения мышки вы увидите, как объект будет двигаться, вращаться или изменять размер соответственно выбранной оси. Как только вы отпустите кнопку мыши, ось останется выделенной. Если вы будете двигать мышь с зажатым колесиком, то будет использоваться последняя выбранная ось, независимо от позиции индикатора мыши.

В режиме перемещения есть дополнительная опция – перемещение объекта в отдельной плоскости – вы можете перемещать объект в двух осях сразу, не затрагивая третью. Три маленьких квадрата вокруг центра гизмо перемещения активируют фиксацию для каждой из плоскостей – цвета соответствуют оси, которая будет зафиксирована, если кликнуть на квадрате (например, красный квадрат фиксирует ось X).

4.4.2. Наследование

Одна из самых важных концепций в Unity – **наследование**. Если игровой объект является родительским для другого объекта, дочерний GameObject будет двигаться, вращаться и менять размер в той же степени, что и родительский объект. Чтобы было проще, можете представить наследование как связь между вашими руками/ногами и вашим телом. Когда ваше тело движется, то руки/ноги двигаются вместе с ним. Дочерние объекты также могут иметь свои дочерние объекты (например, ладони могут считать дочерними объектами ваших рук, а ступни – дочерними объектами ног). Уровень вложенности не ограничен (пальцы – это дочерние объекты ваших ладоней). Любой объект может иметь несколько "детей", но только одного родителя.

Эти многоуровневые связи родители-дети формируют иерархию трансформаций. Объект на самом верху иерархии (т.е. единственный объект, у которого нет родителя) известен как root (корень).

Для создания родительского объекта просто перетащите один игровой объект на другой в окне иерархии. Это создаст связь родительский-дочерний между двумя игровыми объектами.

Посмотрите на рис. 4.12. У нашего объекта car_2 есть 5 дочерних объектов – кузов и 4 колеса.

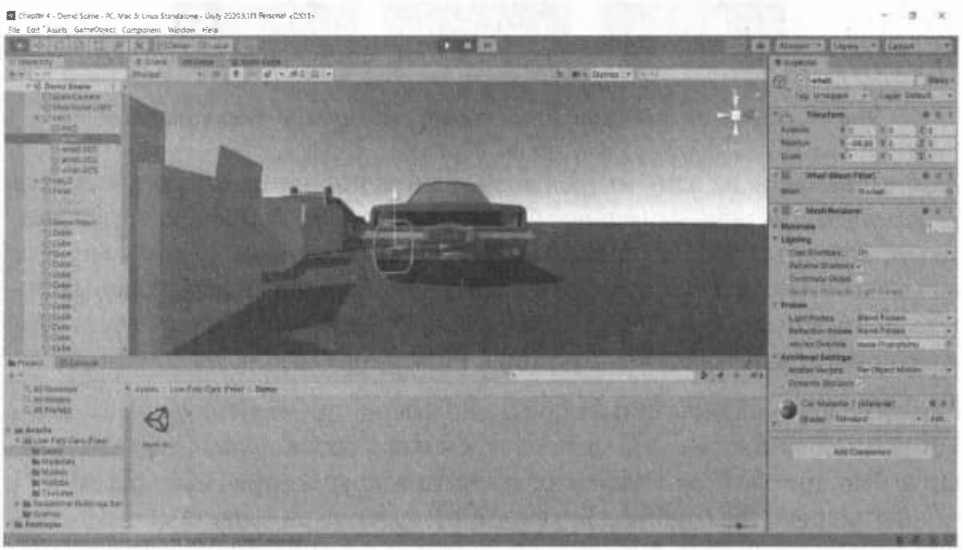


Рис. 4.12. Пример связи родительский-дочерний между игровыми объектами

Помните, что значения **Transform** в инспекторе для любого дочернего объекта показаны относительно значений Transform родительского объекта. Эти значения известны как локальные координаты. Возвращаясь к аналогии тела и рук, положение вашего тела может изменяться по мере ходьбы, но ваши руки будут присоединены в одном и том же месте относительно тела.

Для построения сцены обычно достаточно работать с локальными координатами для дочерних объектов, но во время игрового процесса зачастую полезно найти их точное положение в мировом пространстве или их мировые координаты. API для компонента Transform имеет отдельные настройки для локальных и мировых координат.

4.4.3. Масштаб

Масштаб определяет разницу между размером меша[2] в приложении для моделирования и размером этого же меша в Unity. Размер меша в Unity важен во время физических симуляций. По умолчанию физический движок полагает, что одна единица меры в мировом пространстве соответствует одному метру.

Если объект очень большой, может получиться, что он будет падать с эффектом «slow motion». Симуляция на самом деле правильная, т.к. по сути, вы смотрите с большого расстояния на то, как падает очень большой объект.

Обычно вам не нужно настраивать параметр **Scale** в компоненте **Transform**. Создавайте модели реалистичного размера, и вам не придется изменять масштаб. Но при желании масштаб можно изменить в компоненте **Transform**. Также на масштаб объекта влияет коэффициент **Mesh Scale Factor**, устанавливаемый в настройках импорта объекта (окно **Import Settings**). Некоторые оптимизации производятся на основе размера при импорте, и создание экземпляра объекта с изменённым значением масштаба может снизить производительность.

4.5. Источники света

Неотъемлемой частью каждой сцены являются источники света. Меши и текстуры определяют форму и внешний вид сцены, а источники света создают атмосферу вашего 3D окружения. Скорее всего, вам придется работать более чем с одним источником света в каждой сцене. Заставить их работать вместе – задача не простая, но оно того стоит. Результаты, которые дает применение нескольких источников, просто потрясающие.

Источники света могут быть добавлены в вашу сцену посредством меню **GameObject, Light, <Тип источника света>**. После того, как источник света будет добавлен в сцену, вы можете управлять им как любым другим игровым объектом. Также разработчик может добавить компонент **Light** к любому выделенному объекту, используя **Component, Rendering, Light**.

Объект **Directional Light** имеет много различных свойств, которые можно менять в инспекторе (Inspector), см. рис. 4.13.

2 Меш - набор вершин и многоугольников, определяющих форму трехмерного объекта.

Рис. 4.13. Свойства объекта Directional Light

Просто **Color** меняя свойство (цвет) источника света, вы можете придать сцене совсем другое настроение. Установив другой цвет, можно создать определенную атмосферу, например, на рис. 4.14 установлен цвет E2A214. Попробуйте установить его для своей сцены – создастся эффект инопланетного освещения.

Рис. 4.14. Другой цвет для источника света

Поэкспериментируйте с цветом источника света, и вы сможете создать подходящую для вашей сцены атмосферу.

4.6. Камеры

Камеры в Unity используются для показа игроку созданного вами игрового мира. По крайней мере, у вас будет хотя бы одна камера, которая обычно называется **Main Camera**. Посредством нескольких камер можно разделить экран для двух игроков или создать определенные визуальные эффекты (например, с помощью второй камеры можно имитировать зеркало заднего вида в гоночной игре). Камеры управляются и анимируются по законам физики. Вы можете создавать различные камеры и настраивать их способом, который больше подходит стилю вашей игры.

Камеры являются устройствами, захватывающими и отображающими мир игроку. Путем настройки и манипулирования камерами, вы можете сделать презентацию своей игры поистине уникальной. Вы можете иметь неограниченное количество камер в сцене. Вы можете настроить рендеринг камерами в любом порядке, на любом месте экрана, либо только в определенных частях экрана. Таблица 4.5 содержит описание основных свойств камеры.

Рис. 4.15. Свойства камеры

Таблица 4.5. Свойства камеры

Свойство	Описание
Clear Flags	Позволяет задать части экрана, которые будут очищены. Это удобно при использовании нескольких камер для прорисовки разных элементов игры
Background	Цвет, применяемый для фона после прорисовки всех элементов, в случае отсутствия скайбокса
Culling mask	Включение или исключение слоев объектов на рендер этой камерой. Назначение слоев объектам производится через **Inspector**

Projection	Возможны значения: Perspective (перспектива) – используется для 3D-режим, Ortographic (ортографическая камера) – для 2D-режима
Size (когда выбран ортографический режим)	Размер зоны видимости камеры для ортографического режима
Field of view (режим перспективы)	Ширина угла обзора камеры, измеряется в градусах по локальной оси Y
Clip Planes	Дистанция, на которой камера начинает и заканчивает рендеринг
Near	Ближайшая точка относительно камеры, которая будет рисоваться
Far	Дальняя точка относительно камеры, которая будет рисоваться
View Port Rect	Четыре значения, отражающие то, в какой области экрана будет выведено изображение с камеры, в экранных координатах (от 0 до 1)
X	Начальная позиция области по горизонтали вида камеры, который будет рисоваться
Y	Начальная позиция области по вертикали, где вид камеры будет рисоваться
W (Ширина)	Ширина вида камеры на экране
H (Высота)	Высота вида камеры на экране
Depth	Позиция камеры в очереди отрисовки. Камеры с большим значением будут нарисованы поверх камер с меньшим значением
Target Texture	Ссылка на Render Texture, которая будет содержать результат рендеринга камеры. Назначение этой ссылки отключает способность камеры рендерить на экран
HDR	Включение технологии High Dynamic Range

Target Display	Определяет, какой внешний дисплей будет использоваться для рендеринга (значения от 1 до 8). Другими словами, к системе может быть подключено от 1 до 8 мониторов и эта опция позволяет определить монитор, на который будет производиться рендеринг

Некоторые из этих опций нуждаются в дополнительных комментариях.

4.6.1. Глубина (Depth)

Разработчик может создать несколько камер и назначить каждой свою глубину. Камеры будут прорисовываться от низшей глубины до высшей глубины. Другими словами, камера с Depth 2 будет прорисована поверх камеры с Depth 1. Вы можете настроить значение свойства **View Port Rectangle** для изменения позиции и размера изображения с камеры на экране, например для создания нескольких экранов в одном, или для создания зеркала заднего вида.

Каждая камера хранит информацию о цвете и глубине, когда рендерит свой вид. По умолчанию, незаполненные части экрана, будут показаны в виде скайбокса. Если используется несколько камер, каждая камера будет иметь свои буферы цвета и глубины, заполняемые при каждом рендеринге. Каждая камера будет рендерить то, что видно с ее ракурса, а путем изменения настройки **Clear Flags** можно выбрать набор буферов, которые будут обновлены (очищены), во время рендеринга.

Поясню, как еще можно использовать **Clear Flags**. Если нужно прорисовать оружие игрока, не подвергая его обрезке объектами окружения, внутри которых оно находится, создайте одну камеру, рисующей окружение, с Depth 0, и еще одну камеру, рисующую оружие — Depth 1. Для камеры оружия установите для свойства **Clear Flags** значение **Depth only**. Это позволит отобразить окружение на экране, но проигнорировать всю информацию о взаимном положении предметов в пространстве. Когда оружие будет прорисовано, непрозрачные части будут полностью прорисованы на экране поверх ранее находившегося там изображения, независимо от того, насколько близко оружие находится к стене.

Если для свойства **Clear Flags** установлено значение **Don't clear**, то не очищаются ни цвет, ни буфер глубины. В результате каждый кадр рисуется поверх другого, из-за чего получится эффект размытия. Это обычно

не используется в играх, и лучше использовать вместе с пользовательским шейдером.

4.6.2. Режим камеры

Переключив камеру в ортографичесий (Orthographic) режим, вы устраните всю перспективу из прорисовываемого ей изображения. Это полезно для создания двумерных и изометрических игр. Посмотрим, в чем разница. На рис. 4.16 и 4.17 – одна и та же сцена, но на рис. 4.16 камера находится в режиме перспективы, а на рис. 4.17 – в ортографическом режиме. В ортографическом режиме изображение объектов не уменьшается при увеличении дистанции.

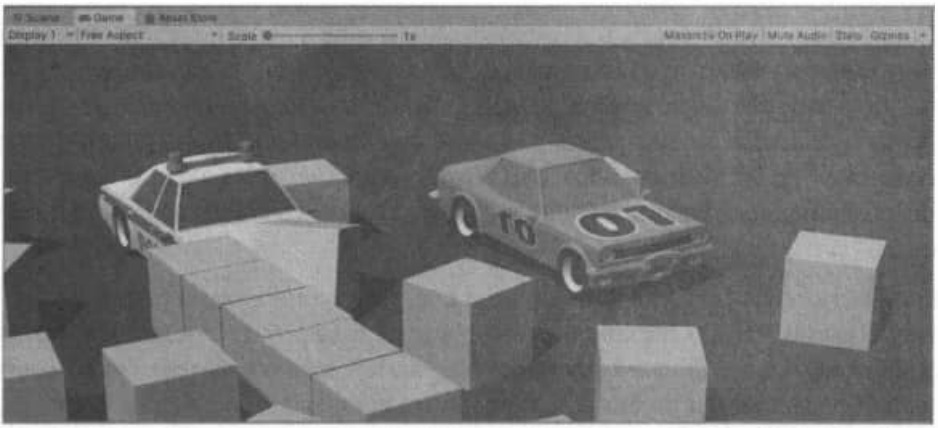


Рис. 4.16. Режим перспективы

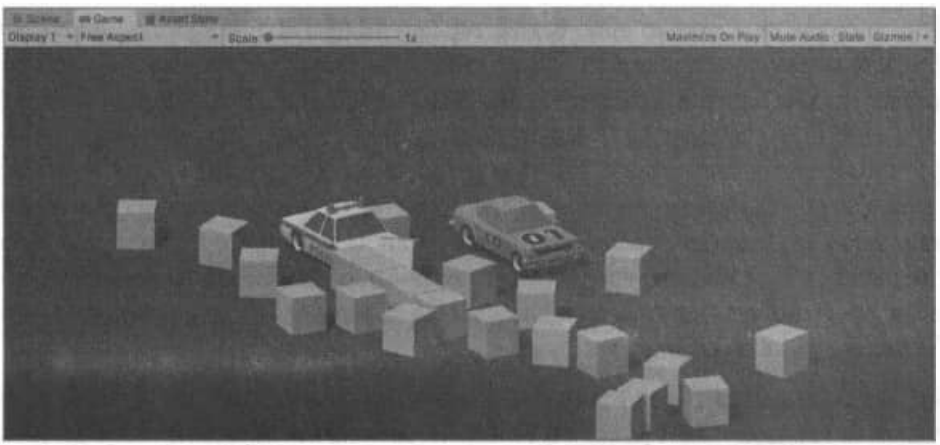


Рис. 4.17. Ортографический режим

4.7. Немного практики

А сейчас немного практики. Вы уже узнали достаточно информации, чтобы «расшевелить» машинки, которые вы уже видели ранее в этой главе. Создайте новый проект и импортируйте в него пакет с двумя моделями автомобилей, который бесплатно можно скачать по адресу

https://free3d.com/ru/3d-model/cartoon-vehicles-low-poly-cars-free-874937.html

Чтобы сэкономить время, откройте сцену **Demo**, доступную по адресу **Assets, Low Poly Cars (Free), Demo**. Вы увидите две машины – гоночную и полицейскую.

Переименуйте префаб любой из машинок (ту, которую вы хотите двигать) в Player. Это и будет машина игрока. После этого создайте скрипт PlayerController.cs и перетащите его на префаб Player. Первый вариант нашего скрипта приведен в лист. 4.4.

Листинг 4.4. Скрипт PlayerController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour
{

     // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        var direction = new Vector3(Input.GetAxis("Vertical"),
Input.GetAxis("Horizontal"), 0);
        GetComponent<Rigidbody>().AddForce(direction);
    }
}
```

Функция **Update** вызывается в каждом кадре, что очень удобно для реализации движения, внутри нее мы и будем размещать код. Нажатия кнопок игроком можно получить с помощью класса **Input**. В комплекте с Unity есть заме-

чательные настройки ввода, достаточно написать Input.GetAxis(«Horizontal») и мы уже знаем, нажал ли игрок на клавиатуре стрелку вправо или влево. Если у игрока подключен геймпад, то он может управлять и с него, нам даже не надо писать лишний код. Обратите внимание: учитывая особенности нашей сцены, мы хотим, чтобы наша машина ездила вперед и назад с помощью клавиш **W** и **S**, а не **A** и **D**, поэтому при формировании направления мы меняем оси местами – сначала указываем вертикальную, а потом – горизонтальную.

Всего одной строчкой мы получаем информацию о действиях пользователя и создаем вектор движения. Чтобы вектор куда-нибудь приложить, нам понадобится Rigidbody. Выделяем префаб Player и через меню **Component, Physics, Rigidbody** добавляем нужный компонент. Теперь мы можем на него ссылаться в нашем скрипте, что мы и делаем во второй строчке нашего метода Update().

Запускаем и видим.... что наша машинка улетела в никуда вместо того, чтобы двигаться вперед и назад. Нам нужно запретить вращение и передвижение по оси **Z** и **Y** (нам не нужно, чтобы машинка двигалась вверх и вниз, а также улетала со сцены). Выделяем префаб, смотрим на компонент Rigidbody и видим раздел *Constraints*. Оставляем неотмеченными только первую галочку **X**, остальные четыре включаем. Чуть выше снимаем галочку **Use Gravity** и прописываем *Drag* равный четырем (см. рис. 4.18).

Рис. 4.18. Параметры Rigidbody

Запускаем игру еще раз. Удача нам улыбнулась: оно шевелится! Но делает это очень медленно. Изменим немного наш скрипт (лист. 4.5).

Листинг 4.5. Измененный вариант PlayerController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class PlayerController : MonoBehaviour
{

    public int acceleration;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        var direction = new Vector3(Input.GetAxis("Vertical"),
Input.GetAxis("Horizontal"), 0);
        GetComponent<Rigidbody>().AddForce(direction *
acceleration);
    }
}
```

Обратите внимание (рис. 4.18): у нашего игрока появился параметр Acceleration. Установите его значение равным 40 (можете установить любое, главное, чтобы больше 0). Запустите игру снова: машинка двигается быстрее (рис. 4.19).

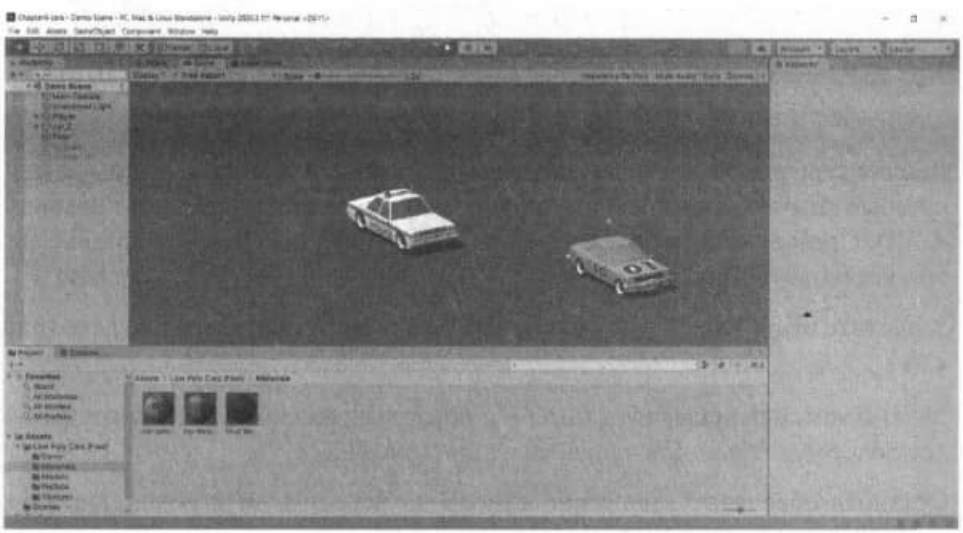


Рис. 4.19. Машинка перемещается по сцене

На данный момент наша машинка двигается только вперед и назад, что не очень хорошо. Давайте заставим ее двигаться влево и вправо. Первым делом

разрешим движение по оси **Z** – именно перемещение по ней в нашем случае будет означать перемещение влево и вправо.

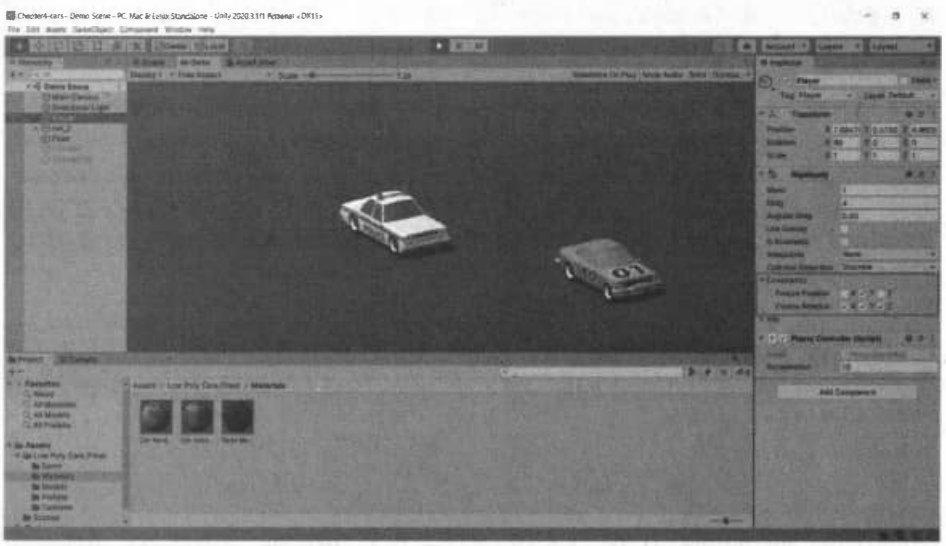


Рис. 4.20. Разрешаем движение влево и вправо

Затем нам нужно переписать наш скрипт, заменив в нем одну строчку:

```
var direction = new Vector3(Input.GetAxis("Vertical"), 0,
Input.GetAxis("Horizontal"));
```

Данная строчка обеспечивает перемещение по оси **Z** посредством клавиш, которые обычно закреплены за горизонтальным положением – это клавиши **A** и **D**. Сделаем еще небольшое изменение – для нашей камеры (Main Camera) установите **Field of View** равным 35. Так угол обзора будет больше.

Запустите игру. Теперь наша машинка сможет двигаться влево и вправо (рис. 4.21).

Можно также переключить камеру в ортографический режим, который будет более уместным для нашей ситуации (рис. 4.22).

Осталось обеспечить движение камеры за персонажем игрока. Для этого создайте скрипт CameraController.cs и перетащите его на MainCamera. Код скрипта приведен в лист. 4.6.

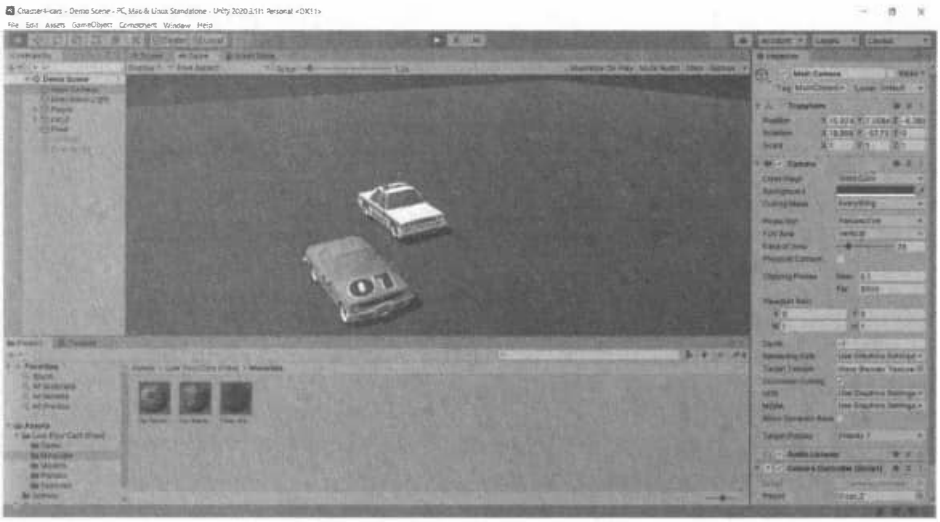


Рис. 4.21. Перемещение машинки по сцене

Рис. 4.22. Камера в ортографическом режиме

Листинг 4.6. Сценарий CameraController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```
public class CameraController : MonoBehaviour
{

    public GameObject Player;

    private Vector3 offset;
    // Start вызывается перед первым изменением кадра
    void Start()
    {
       // вычисляем смещение
        offset = transform.position - Player.transform.
position;
    }

    // Вызывается каждый кадр
    void Update()
    {
       // позиция камеры равна позиции игрока + смещение
        transform.position = Player.transform.position +
offset;

    }
}
```

После присоединения данного сценария к камере в свойствах камеры нужно будет задать машинку с именем Player. Именно за ней будет двигаться машинка.

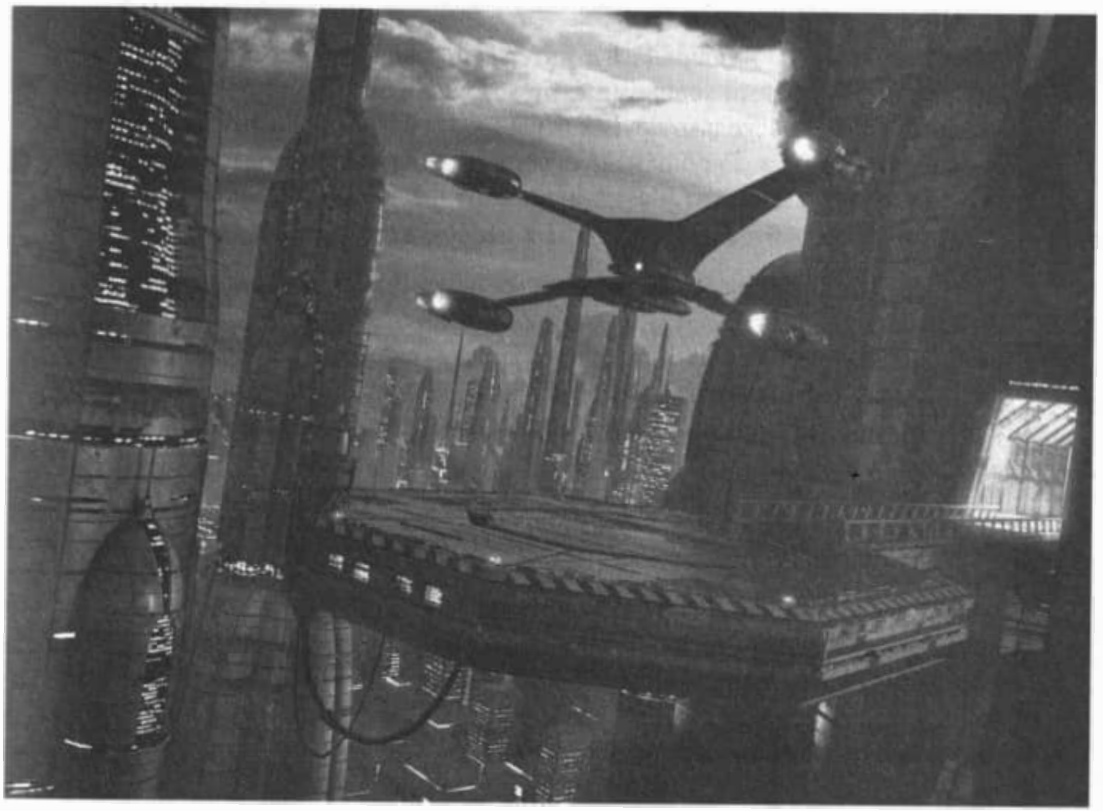
> **Примечание.** Созданный нами проект называется Ch4-cars и доступен в материалах, прилагаемых к этой книге.

Дополнительная информация о камерах будет приведена в других главах этой книги, а на этом мы завершаем первую часть. Мы ознакомились с основами использования Unity, во второй части мы будем говорить о более сложных материях, в том числе о скриптинге.

ЧАСТЬ II. ОСНОВЫ РАЗРАБОТКИ ИГРЫ

Глава 5.

Графические возможности в Unity



В первой части мы разобрались с интерфейсом Unity и нахватались кое-каких основ. Но с этими знаниями игру все еще не создашь. Во второй части мы поговорим о несколько других вещах – о графике, о физике, рассмотрим основы скриптинга. В общем, обо всем, что нам понадобится в процессе разработки игры. Все эти знания мы применим на практике в третьей части этой книги, когда будем разрабатывать первую вашу игру.

Unity предлагает разработчикам удивительную визуальную точность, мощный рендеринг и атмосферу. С помощью Unity ваша игра будет выглядеть именно так, как вы ее представляете. Понимание графики является ключевым моментом, который поможет добавить элемент погружения в вашу игру. В этой главе рассматриваются графические возможности Unity, такие как освещение и рендеринг.

5.1. Освещение

Чтобы рассчитать затенение трехмерного объекта, Unity необходимо знать интенсивность, направление и цвет падающего на него света. Посмотрите на рис. 5.1. На нем изображен трехмерный объект и источник света. Свет отражается от объекта в местах, где лучи света попадают на объект. На противоположной стороне объекта возникают тени.

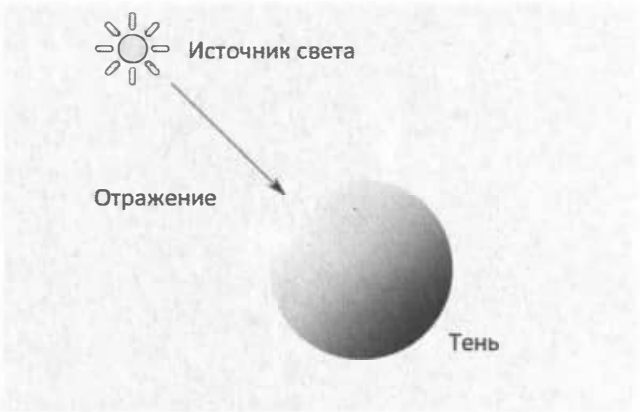


Рис. 5.1. Освещение трехмерного объекта

Источники света представлены объектами **Light**. Базовый цвет и интенсивность устанавливаются одинаково для всех источников света, но направление зависит от того, какой тип освещения вы используете. Кроме того, свет может уменьшаться с расстоянием от источника.

5.1.1. Типы источников света

В Unity доступно четыре типа источников света:

- **Directional Light** – направленный свет
- **Point Light** – точечный свет
- **Spotlight** – прожектор, еще одна разновидность точечного света
- **Area Light** – используется для освещения площади

Для начала разберемся с точечным светом, поскольку в Unity есть два объекта точечного освещения. Первый, Point Light, находится в точке в пространстве и направляет свет во всех направлениях одинаково. Направление света, попадающего на поверхность, – это линия от точки соприкосновения к центру светового объекта. Интенсивность уменьшается с удалением от света, достигая нуля в указанном диапазоне. Посмотрите на рис. 5.2 – на нем показан эффект от источника света типа Point Light.

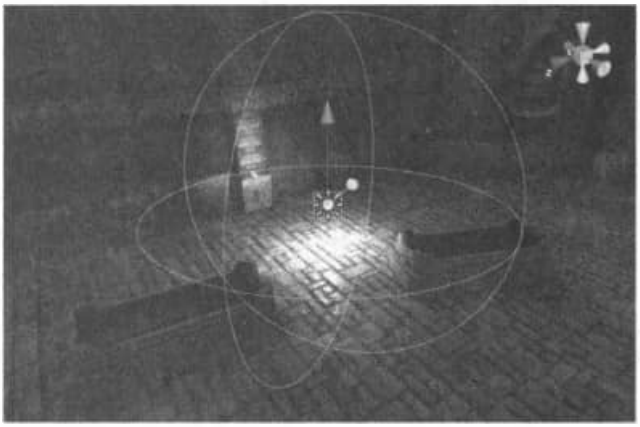


Рис. 5.2. Источник света типа Point Light

Такие источники света полезны для моделирования ламп и других локальных источников света в сцене. Вы также можете использовать их, чтобы искра или взрыв осветили окружающую обстановку убедительным образом.

Как и в случае с Point Light, Spotlight имеет определенное местоположение и диапазон, в котором свет падает. Тем не менее, Spotlight ограничен углом, что приводит к конусообразной области освещения. Эффект от такого объекта показан на рис. 5.3.

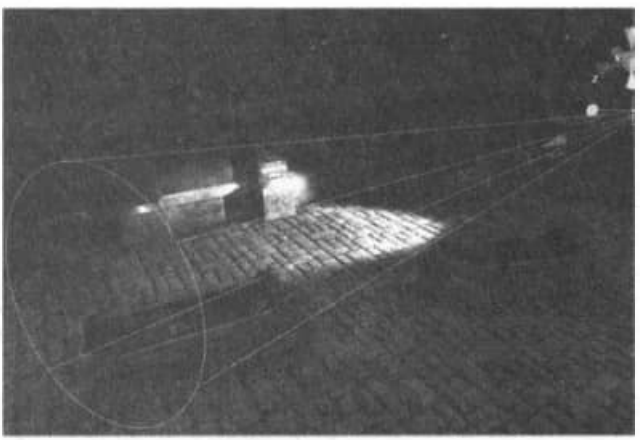


Рис. 5.3. Эффект от источника света типа Spotlight

С направленным светом (Directional Light) мы уже сталкивались, поскольку именно он используется на сценах Unity по умолчанию, и такой источник света добавляется в каждый новый проект. Направленный свет не имеет

определенного положения источника, поэтому световой объект может быть размещен в любом месте сцены. Все объекты на сцене освещаются так, будто свет всегда идет в одном направлении. Расстояние света от целевого объекта не определено, и поэтому свет не уменьшается.

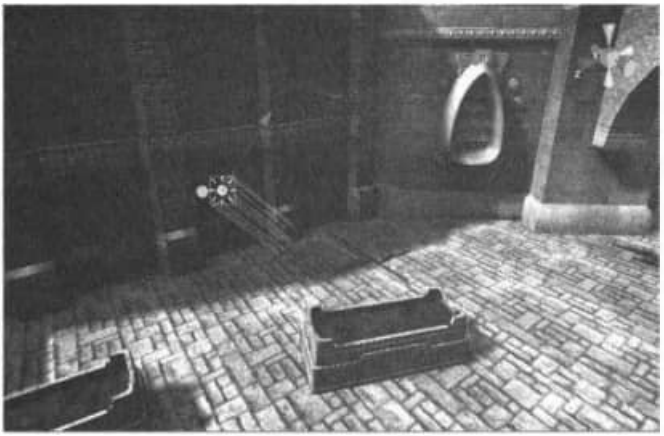


Рис. 5.4. Направленный свет

Направленные источники света, как правило, представляют большие, удаленные источники, находящиеся за пределами игрового мира. В игре их можно использовать для имитации солнца или луны. Они полезны для добавления убедительных оттенков к объектам без точного указания, откуда исходит свет. При проверке объекта в виде сцены (например, чтобы посмотреть, как выглядят его сетка, шейдер и материал), направленный свет часто является самым быстрым способом получить представление о том, как будет выглядеть его затенение.

Вспомните источники света, которыми освещаются стадионы. Для имитации таких источников предназначен тип *Area Light*. Свет излучается во всех направлениях, но только с одной стороны прямоугольника. Свет падает в указанном диапазоне. Поскольку расчет освещения довольно интенсивно использует процессор, локальные источники света недоступны во время выполнения и могут быть включены только в карты освещения.

Поскольку область освещения освещает объект сразу с нескольких разных направлений, затенение имеет тенденцию быть более мягким и тонким, чем у других типов освещения. Вы можете использовать его, чтобы создать реалистичный уличный фонарь или группу огней рядом с игроком. Освещение небольшой площади может имитировать более мелкие источники света (например, внутреннее освещение дома), но с более реалистичным эффектом, чем точечный свет.

5.1.2. Использование освещения

Использовать источники света достаточно просто: вам нужно создать источник света нужного типа (меню **GameObject, Light, <тип источника>**) и поместить его в нужное место на сцене. Если вы включите освещение в виде сцены (кнопка «лампочка» на панели инструментов), вы сможете увидеть предварительный просмотр освещения при перемещении световых объектов и задании их параметров.

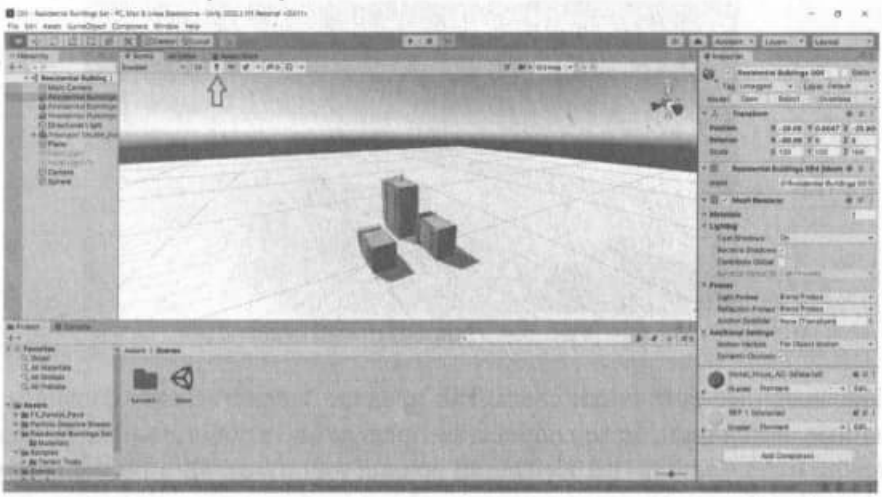


Рис. 5.5. Освещение выключено

Направленный источник света, как правило, может быть размещен в любом месте сцены, при этом прямая ось / Z указывает направление. Точечный светильник также имеет направление, но поскольку он имеет ограниченный диапазон, его положение имеет значение. Параметры формы точечных, точечных и площадных источников света можно настроить из инспектора или с помощью гизмо источников света непосредственно в виде сцены.

На рис. 5.6 помещено два точечных (*Point Light*) источника света. Один ближе к стене двухэтажного дома, второй дальше от домов, при этом он обладает большей интенсивностью (свойство *Intensity*). Посмотрите на получившийся эффект.

5.1.3. Рекомендации по размещению освещения

Направленный свет часто представляет солнце и оказывает значительное влияние на внешний вид сцены. Направление света должно быть слегка направлено вниз, но вы, как правило, хотите убедиться, что он также имеет

небольшой угол с основными объектами в сцене. Например, примерно кубический объект будет более интересно заштрихован и будет казаться, что он «выскакивает» в 3D гораздо больше, если свет не падает прямо на одну из граней.

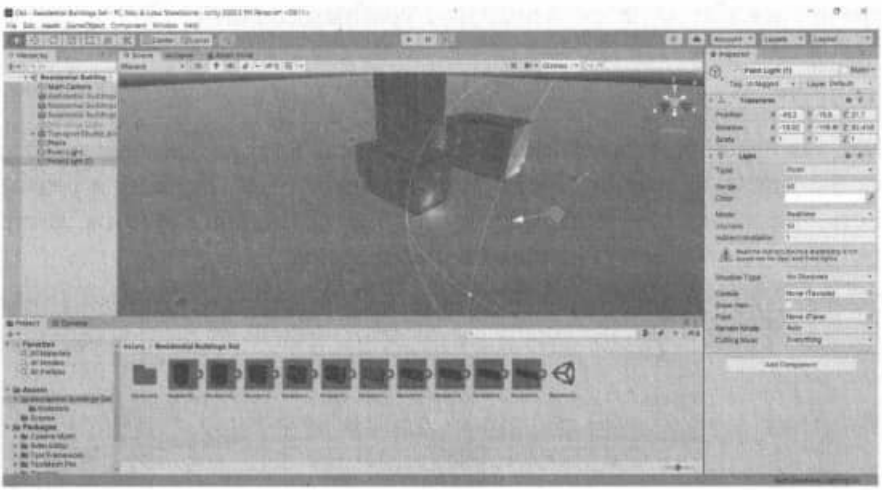


Рис. 5.6. Два точечных источника света

Прожекторы (*spot light*) и точечные светильники (*point light*) обычно представляют собой источники искусственного света, поэтому их положение обычно определяется объектами сцены. Одна из распространенных ловушек с этими источниками света заключается в том, что они, кажется, не имеют никакого эффекта при первом добавлении их в сцену. Это происходит, когда вы настраиваете диапазон света, чтобы он аккуратно вписывался в сцену. **Дальность света** – это предел, при котором яркость света уменьшается до нуля. Если вы установите, скажем, точечный источник света, чтобы основание конуса аккуратно приземлилось на пол, то этот свет будет иметь небольшой эффект или вообще не будет действовать, если другой объект не пройдет под ним. Если вы хотите, чтобы геометрия уровня была освещена, вам следует расширить точечные и точечные источники света, чтобы они проходили через стены и полы.

Цвет и **интенсивность света** (яркость) – это свойства, которые вы можете установить из инспектора. Интенсивность по умолчанию и белый цвет подходят для «обычного» освещения, которое вы используете для наложения затенения на объекты, но вам может потребоваться изменить свойства для получения специальных эффектов. Например, светящееся зеленое силовое поле может быть достаточно ярким, чтобы омывать окружающие объекты

интенсивным зеленым светом; автомобильные фары (особенно на старых автомобилях) обычно имеют слегка желтый цвет, а не ярко-белый. Эти эффекты чаще всего используются с точечными и точечными источниками света, но вы можете изменить цвет направленного света, если, скажем, ваша игра происходит на далекой планете с красным солнцем.

5.1.4. Тени

Свет, как правило, отбрасывает тени (если речь идет не о вампирах, которые, как мы знаем, не имеют теней). Тени придают сцене глубину и реалистичность, поскольку они выделяют масштаб и положение объектов, которые в противном случае могут выглядеть «плоскими».

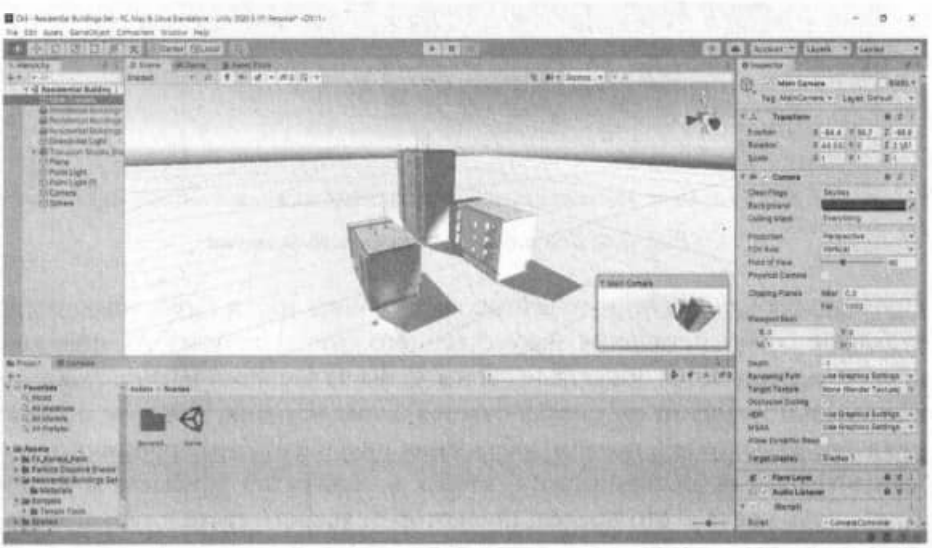


Рис. 5.7. Сцена с объектами, отбрасывающими тени

Рассмотрим простейший случай сцены с одним источником света. Лучи света распространяются от этого источника по прямым линиям и могут в конечном итоге попасть в объекты на сцене. Как только луч достиг объекта, он не может двигаться дальше, чтобы осветить что-либо еще (т.е. он «отскакивает» от первого объекта и не проходит сквозь него). Тени, отбрасываемые объектом, – это просто области, которые не освещены, потому что свет не может достичь их.

Вы можете включить тени для отдельного источника света с помощью свойства «Shadow Type» в инспекторе (рис. 5.9).

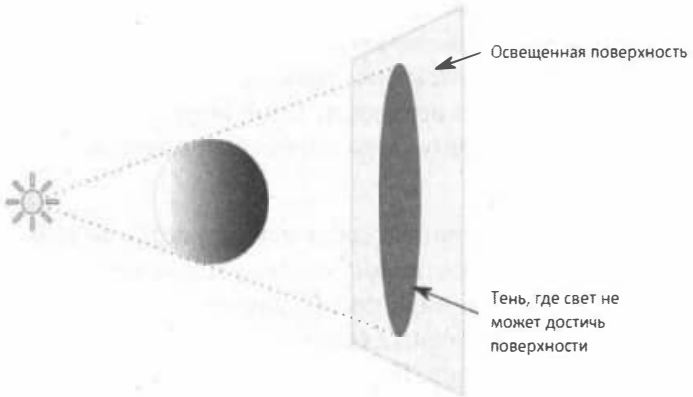
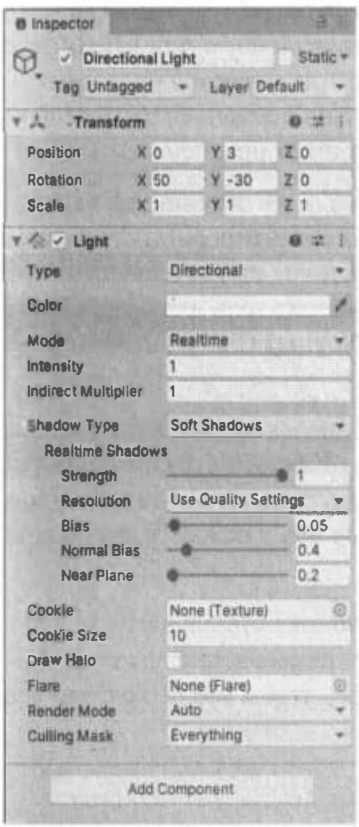


Рис. 5.8. Природа теней

Рис. 5.9. Свойства источника света



Значение **Hard Shadows** создает тени с острым краем. Жесткие тени редко возникают в повседневной жизни (если вы не астронавт), но они требуют меньше затрат на обработку, чем более реалистичные мягкие тени (**Soft Shadows**).

Кроме того, мягкие тени имеют тенденцию уменьшать «блочный» эффект наложения на карте теней. Параметр **Strength** определяет, насколько темны тени; в общем, некоторое количество света будет рассеиваться атмосферой и отражаться от других объектов, поэтому обычно вы не хотите, чтобы тени были настроены на максимальную силу. Свойство **Resolution** устанавливает разрешение рендеринга для карты теней «камеры», упомянутой выше. Если вы обнаружите, что ваши тени имеют очень видимые края, вы можете увеличить это значение. Свойство **Near Plane** позволяет вам выбрать значение для ближней плоскости при рендеринге теней. Любые объекты ближе, чем это расстояние к свету, не будут отбрасывать тени.

Свойство **Cast Shadows** имеет простые опции **On** и **Off** , чтобы включить или отключить отбрасывание теней для меша. Существует также опция **Two Sided**, позволяющая отбрасывать тени по обе стороны от поверхности (т. е. отбраковка задней поверхности игнорируется при отбрасывании теней), тогда как **Shadows Only** позволяет отбрасывать тени другим невидимым объектом.

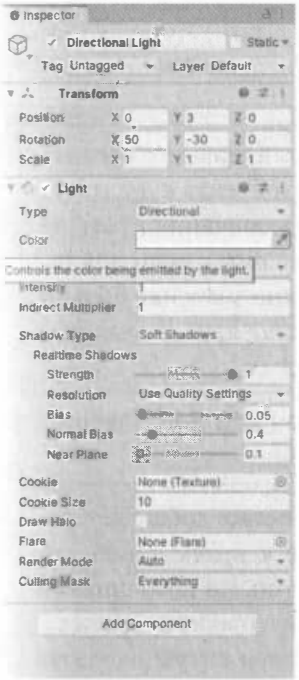
Тени для конкретного источника света определяются во время финального рендеринга сцены. Когда сцена визуализируется в камере основного вида, каждая позиция пикселя в виде преобразуется в систему координат света. Расстояние пикселя от источника света затем сравнивается с соответствующим пикселем на карте теней. Если пиксель находится дальше, чем пиксель карты теней, то он предположительно скрыт от света другим объектом и не получит никакого освещения.

Иногда поверхность, освещаемая светом, может частично находиться в тени. Это связано с тем, что пиксели, которые должны находиться точно на расстоянии, указанном в карте теней, иногда считаются более отдаленными (следствие использования изображения с низким разрешением для карты теней или использования фильтрации теней). Результатом являются произвольные образцы пикселей в тени, когда они действительно должны быть освещены, создавая визуальный эффект, известный как «тени от прыщей».

Чтобы предотвратить появление прыщей в тени, к расстоянию на карте теней можно добавить значение смещения, чтобы гарантировать, что пиксели на границе будут точно соответствовать сравнению; или при рендеринге в карту теней объекты могут быть вставлены немного вдоль их нормалей. Эти значения устанавливаются свойствами **Bias** и **Normal Bias** в инспекторе источника света, когда для него включены тени.

Однако не устанавливайте слишком большое значение смещения, так как области тени рядом с объектом, из которого он отбрасывается, иногда могут быть неправильно освещены. Этот эффект известен как «Питер Пэннинг» (т.е. отсоединенная тень заставляет объект выглядеть так, как будто он летит над землей, как Питер Пэн), см. рис. 5.10.

Аналогично, установка слишком высокого значения нормального смещения сделает тень слишком узкой для объекта. Значения смещения для света (рис. 5.11), возможно, потребуется немного настроить, чтобы убедиться, что тени отображаются нормально.



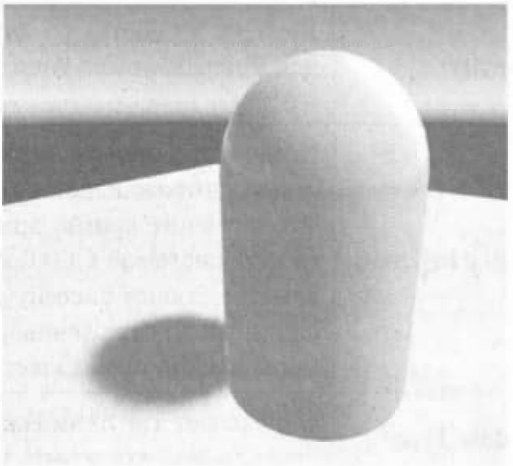


Рис. 5.10. Эффект «Питер Пэна»: слишком высокое значение смещения заставляет тень казаться «отсоединенной» от объекта

Рис. 5.11. Свойства Bias и Normal Bias для источника света

5.1.5. Основные свойства источника света

Мы уже рассмотрели некоторые свойства источника света. Описание остальных опций источника света приведено в таблице 5.1.

Таблица 5.1. Свойства источника Свет

Свойство	Функция
Type	Текущий тип освещения. Возможные значения: Directional, Point, Spot и Area (ранее мы их рассматривали)
Range	Дальность излучения света из центра объекта. Только для источников света типа Point и Spot
Spot Angle	Задает угол конуса в градусах. Используется только для источника света типа Spot
Color	Цвет излучаемого света

Intensity	Яркость света. Значение по умолчанию для Point, Spot или Area – 1, для направленного света – 0.5
Bounce Intensity	Позволяет изменять интенсивность непрямого света (т. е. света, отражающегося от одного объекта к другому). Это значение кратно яркости по умолчанию, рассчитанной системой GI (Global Illumination); если для параметра Bounce Intensity установлено значение больше единицы, то отраженный свет станет ярче, а значение меньше единицы сделает его тусклым
Shadow Type	Определяет тип тени Hard Shadows или Soft Shadows. Здесь можно отключить тени вообще
Baked Shadow Radius	Если тени включены, то это свойство добавляет некоторое искусственное смягчение к краям теней, отбрасываемых точечными или точечными источниками света (в теории свет, исходящий из точки, отбрасывает совершенно резкие тени, но такая ситуация редко встречается в природе)
Baked Shadow Angle	Если тени включены, то это свойство добавляет некоторое искусственное смягчение к теням, отбрасываемым направленным светом (теоретически, параллельные лучи света, исходящие от действительно «направленного» источника, создают совершенно резкие тени, но естественные источники света не ведут себя строго так)
Draw Halo	Если это свойство включено, будет нарисован сферический ореол света с радиусом Range
Flare	Опциональная ссылка на блик, который будет рендериться в позиции источника света
Render Mode	Режим рендеринга. Значение этой опции может повлиять на производительность. Значение Auto автоматически выбирает подходящий режим рендеринга, значение Important следует выбирать для наиболее заметных визуальных эффектов, например, фар автомобиля игрока. Значение Not important увеличивает производительность за счет менее подробной прорисовки деталей

Culling Mask	Используется для выборочного освещения групп объектов

5.1.6. Направленные светлые тени

Направленный свет обычно имитирует солнечный свет, и один источник света может освещать всю сцену. Это означает, что карта теней часто покрывает большую часть сцены сразу и это делает тени восприимчивыми к проблеме, называемой «сглаживание перспективы». Проще говоря, сглаживание перспективы означает, что пиксели карты теней, видимые близко к камере, выглядят увеличенными и «короткими» по сравнению с теми, которые находятся дальше. Данная проблема хорошо видна на рис. 5.12.

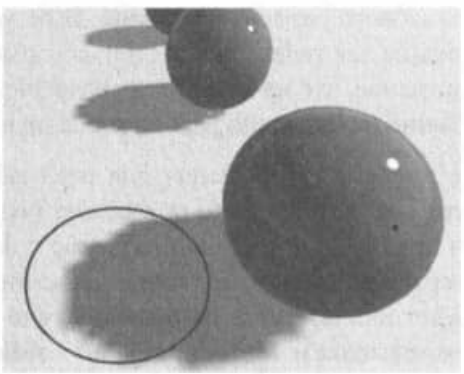


Рис. 5.12. Сглаживание перспективы

Сглаживание перспективы менее заметны при использовании мягких теней и высокого разрешения для карты теней. Однако использование этих функций повысит системные требования, в частности, к видеокарте, поэтому может пострадать частота кадров.

Причина, по которой возникает наложение перспективы, заключается в том, что различные области карты теней пропорционально масштабируются с точки зрения камеры. Карта теней от источника света должна покрывать только ту часть сцены, которая видна камере, что определяется усечением обзора камеры. Если вы представите простой случай, когда направленный свет исходит непосредственно сверху, вы можете увидеть связь между усеченным конусом и картой теней (рис. 5.13).

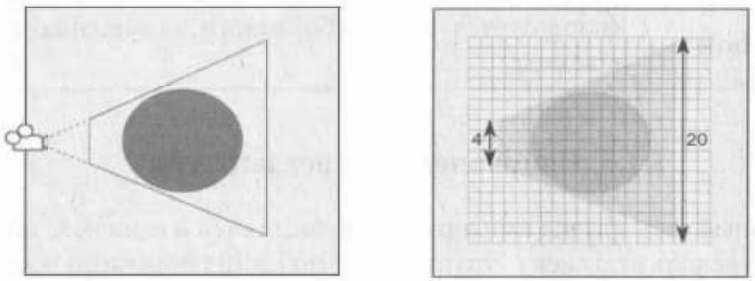


Рис. 5.13. Направленный свет и карта теней. Причина «ступенчатых» теней

Посмотрите на рис. 5.13. Дальний конец усеченного контура покрыт 20 пикселями карты теней, а ближний – только 4 пикселями. Тем не менее, оба конца отображаются одинакового размера на экране. В результате разрешение карты значительно меньше для теневых областей, которые находятся близко к камере. Обратите внимание, что на самом деле разрешение намного выше, чем 20x20, и карта обычно не идеально расположена прямо перед камерой.

Использование более высокого разрешения для всей карты может уменьшить эффект «коротких» областей, но это потребляет больше памяти и пропускной способности при рендеринге. По диаграмме понятно, что большая часть карты теней «теряется» в ближнем конце усеченного конуса, потому что она никогда не будет видна; также разрешение тени далеко от камеры, вероятно, будет слишком высоким. Можно разделить область усеченного конуса на две зоны в зависимости от расстояния до камеры. Зона на ближнем конце может использовать отдельную карту теней с уменьшенным размером (но с тем же разрешением), чтобы количество пикселей было несколько выровнено.

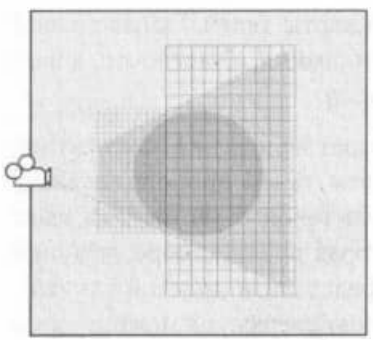


Рис. 5.14. Использование более высокого разрешения

Эти поэтапные сокращения размера карты теней известны как каскадные карты теней. В настройках качества (**Edit, Project Settings, Quality**) вы можете установить ноль, два или четыре каскада для данного уровня качества (рис. 5.15) – параметр **Shadow Cascades**.

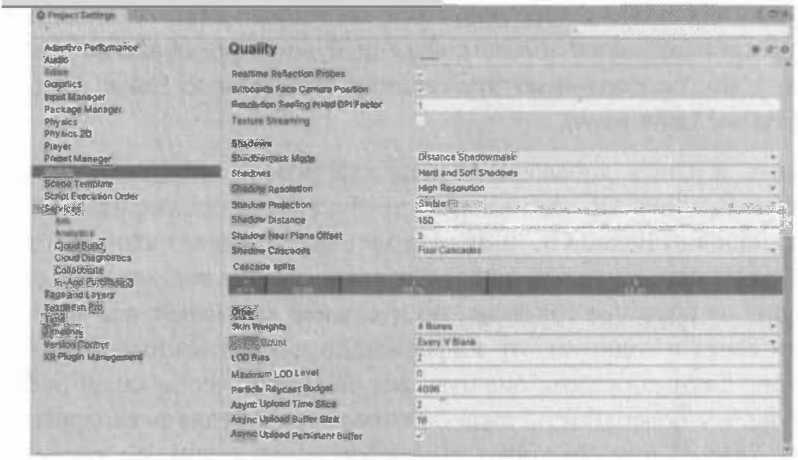


Рис. 5.15. Настройки качества для проекта

5.2. Глобальное освещение

Глобальное освещение (GI, Global Illumination) – это система, моделирующая, как свет отражается от поверхностей на другие поверхности (непрямой свет), а не ограничивается только светом, который падает на поверхность непосредственно от источника света (прямой свет).

Моделирование непрямого освещения реализует эффекты, которые делают виртуальный мир более реалистичным и взаимосвязанным, поскольку объекты влияют на внешний вид друг друга. Одним из классических примеров является «цветное кровотечение», когда, например, солнечный свет, попадающий на красный диван, вызывает отражение красного света на стене позади него. Другой случай, когда солнечный свет падает на пол при открытии пещеры и отскакивает внутри, так что внутренние части пещеры также освещаются.

Традиционно, видеоигры и другие графические приложения реального времени ограничивались прямым освещением, в то время как вычисления, необходимые для непрямого освещения, были слишком медленными, поэтому их можно было использовать только в ситуациях, не связанных с реальным временем, таких как анимационные фильмы CG[1].

1 Computer Graphics

Способ обойти это ограничение в играх заключается в том, чтобы рассчитывать непрямой свет только для объектов и поверхностей, о которых известно, что они не перемещаются заранее (которые являются статичными). Таким образом, медленное вычисление может быть выполнено заблаговременно, но поскольку объекты не перемещаются, косвенный свет, который предварительно рассчитан таким образом, все еще будет корректным во время выполнения. Unity поддерживает эту технику, называемую Baked GI (другое название Baked Lightmaps).

Кроме того, в Unity добавлена поддержка нового метода, называемого Precomputed Realtime GI. Она все еще требует фазы предварительного вычисления, подобно Baked GI, и она все еще ограничена статическими объектами. Однако она не только предварительно вычисляет, как свет отражается в сцене во время ее создания, но и заранее вычисляет все возможные источники света и кодирует эту информацию для использования во время выполнения. Таким образом, она отвечает на вопрос «если какой-либо свет попадает на эту поверхность, куда он отражается?» (для всех статических объектов). Затем Unity сохраняет эту информацию о том, по каким путям свет может распространяться для дальнейшего использования. Окончательное освещение выполняется во время выполнения путем подачи фактических источников света на эти ранее рассчитанные пути распространения света. Это означает, что количество и тип источников света, их положение, направление и другие свойства могут быть изменены, и непрямое освещение будет обновляться соответствующим образом. Точно так же возможно также изменить свойства материала объектов, такие как их цвет, сколько света они поглощают или сколько света они излучают сами.

В то время как предварительно вычисленное GI в реальном времени также приводит к мягким теням, они будут более грубыми, чем те, которые можно получить с помощью Baked GI, если только сцена не очень мала.

Также обратите внимание, что, хотя Precomputed Realtime GI реализует окончательное освещение во время выполнения, оно делает это итеративно в течение нескольких кадров, поэтому, если в освещении будут сделаны большие изменения, потребуется больше кадров, чтобы оно полностью вступило в силу. Другими словами, если целевая платформа имеет очень ограниченные ресурсы, может быть лучше использовать Baked GI для лучшей производительности во время выполнения.

Примеры эффектов GI:

- Изменение направления и цвета направленного света для имитации эффекта движения солнца по небу. Изменяя скайбокс вместе с направлен-

ным светом, можно создать реалистичный эффект времени дня, который обновляется во время выполнения

- В течение дня солнечный свет, проникающий через окно, движется по полу, и этот свет реально отражается по комнате и потолку. Когда солнечный свет достигает красного дивана, красный свет отражается на стене позади него. Изменение цвета дивана с красного на зеленый приведет к тому, что цвет на стене за ним тоже изменится с *красного* на *зеленый*
- Анимация излучающей способности материала неонового знака, чтобы он начал светиться на свое окружение при включении

5.2.1. Окно Lighting

Окно **Lighting** (команда меню **Window, Rendering, Lighting Settings**) является главной контрольной точкой для функций глобального освещения Unity (GI). Нужно отметить, что в большинстве случаев настройки по умолчанию вполне приемлемые (рис. 5.16), но свойства окна позволяют настраивать многие аспекты процесса GI для настройки сцены или оптимизации качества, скорости и места для хранения по мере необходимости. Это окно также содержит настройки освещения, которые были доступны в старых версиях Unity.

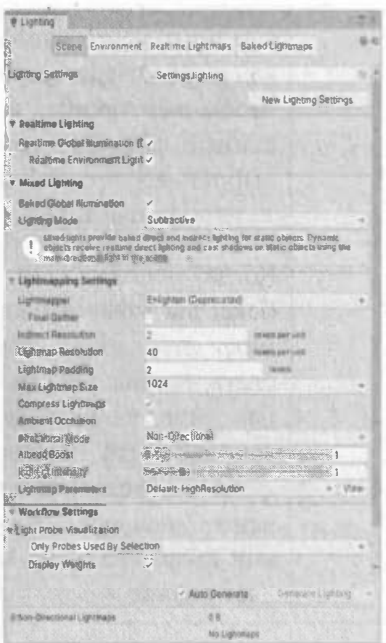


Рис. 5.16. Окно Lighting

Элементы управления в окне **Lighting** разделены на три вкладки:

- **Scene** – здесь содержатся настройки, которые применяются к сцене в целом, а не к отдельным объектам. Эти настройки управляют световыми эффектами, а также параметрами оптимизации
- **Environment** – настройки освещения окружающей среды. В предыдущей версии часть этих настроек была на вкладке **Scene**, а затем была перенесена на отдельную вкладку. Здесь, например, хранятся настройки небосвода (можно выбрать материал небосвода и установить другие параметры), включить туман и т.д.
- **Realtime Lightmaps** – отображаются файлы ресурсов карты освещения, созданные процессом GI для Precomputed Realtime GI
- **Backed Lightmaps** – отображаются файлы ресурсов карты освещения, созданные процессом GI для Backed GI

Параметры, которые можно изменить в окне **Lighting** (без разделения на вкладки), представлены в табл. 5.2.

Таблица 5.2. Параметры освещения

Параметр	Описание
Skybox Material	Скайбокс – это изображение, которое появляется позади всего остального в сцене, чтобы имитировать небо или другой отдаленный фон. Это свойство позволяет вам выбрать ассет скайбокса, который вы хотите использовать для сцены
Sun Source	Когда используется процедурный скайбокс, вы можете использовать этот параметр для выбора объекта направленного света, чтобы указать направление «солнца» (или любого другого большого источника света, освещающего вашу сцену). Если для этого параметра установлено значение None, предполагается, что самый яркий направленный свет в сцене представляет солнце
Environment Lighting	Параметры освещения окружающей среды

Source	Позволяет выбрать, что будет выступать в качестве освещения окружающего мира – Skybox (небосвод), Color (цвет), Gradient (градиент)
Intensity Multiplier	Позволяет изменять яркость скайбокса на сцене
Ambient Mode	Позволяет выбрать режим глобального освещения (GI), который будет использоваться для прорисовки окружающего света – Backed или Realtime
Environment Reflections	Управление отражениями
Source	Позволяет указать, следует ли использовать скайбокс для эффектов отражения (по умолчанию) или же выбрать кубическую карту (значение **Custom**) для использования вместо него. Если в качестве источника выбран скайбокс, то предоставляется дополнительная опция для установки разрешения скайбокса для целей отражения
Resolution	Задает разрешение скайбокса
Compression	Определяет, как Unity будет сжимать кубические карты. Доступны значения: Auto (автоматически), Compressed (используется сжатие), Not compressed (без сжатия)
Intensity Multiplier	Множитель интенсивности для отражений на сцене
Bounces	Определяет, сколько раз отражение будет включать другие отражения
Realtime Lighting	Параметры Realtime Lighting
Realtime Global Illumination	Включает/выключает Realtime GI
Backed Global Illumination	Включает/выключает Backed GI

Lighting Mode	Режим освещения
Realtime Shadow Color	Цвет тени в режиме реального времени, если используется режим освещения Subtractive
Lightmapper	Позволяет выбрать систему, которая будет использоваться для создания освещения
Indirect Resolution	Разрешение для расчетов непрямого освещения. Эквивалентно разрешению в реальном времени при использовании предварительно вычисленного GI в реальном времени (доступно только в том случае, если Precomputed Realtime GI в реальном времени отключен)
Compress Lightmaps	Будут ли сжиматься лайтмапы. Сжатие сокращает размер дискового пространства, но негативно влияет на производительность
Ambient Occlusion	Относительная яркость поверхностей в окружающей окклюзии (то есть частичная блокировка внешнего освещения во внутренних углах). Более высокие значения указывают на больший контраст между окклюзированными и полностью освещенными участками. Это относится только к непрямому освещению, рассчитанному системой GI

Directional Mode	Карта освещения может быть настроена для хранения информации о доминирующем входящем свете в каждой точке на поверхностях объектов. В режиме Directional генерируется вторая карта освещения для сохранения доминирующего направления входящего света. Это позволяет рассеивать материалы с нормальным отображением для работы с GI. Режим Non-Directional отключает вторую карту. Направленный режим требует примерно вдвое больше места для хранения дополнительных данных карты освещения
Indirect Intensity	Значение, которое масштабирует яркость непрямого света, видимого в окончательной карте освещения (т. е. окружающий свет или свет, отраженный и испущенный объектами). Установка этого параметра в 1.0 использует масштабирование по умолчанию; значения меньше 1,0 уменьшают интенсивность, а значения больше 1,0 увеличивают ее
Lightmap Parameters	Позволяет выбрать пресет для лайт-мапа. В большинстве случаев достаточно **Default-Medium**, но вы можете выбрать **Default-HighResolution** для наивысшего качества. Кнопка **View** позволяет просмотреть настройки пресета. Если из этого списка выбрать **Create New**, то это позволит вам создать собственный пресет настроек
Fog	Включает/выключает туман. Очень интересный параметр, после включения которого у вас будет возможность настроить отображение тумана – выбрать его цвет, плотность, режим. Включите туман и посмотрите, как преобразится ваша сцена (рис. 5.17)

Halo Textture	Текстура, используемая для рисования ореола вокруг источников света
Halo Strength	Видимость ореолов вокруг огней
Flare Fade Speed	Время, в течение которого блики объектива исчезают из поля зрения после их появления
Flare Strength	Видимость бликов от фонарей
Spot Cookie	Cookie текстура используется для местной подсветки

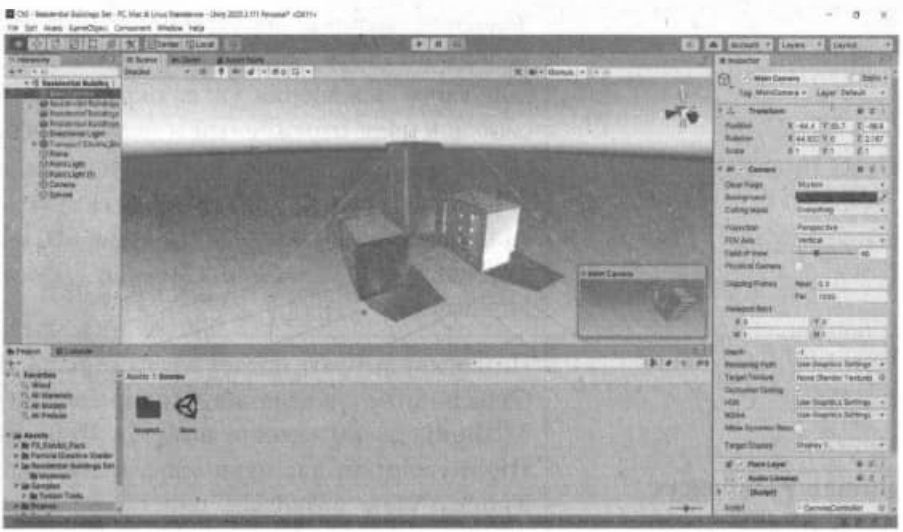


Рис. 5.17. Сцена при включенном тумане

5.2.2. Кэш GI

Механизм кэширования GI Cache используется системой Global Illumination для хранения промежуточных файлов при создании световых карт, световых зондов и зондов отражения. Кэш GI является общим кэшем, поэтому проекты с одинаковым содержимым могут совместно использовать некоторые файлы для ускорения сборки.

Настройки GI Cache можно найти в **Edit, Preferences, GI Cache**. Здесь вы можете установить размер кэша, очистить текущие кэшированные файлы (кнопка **Clean Cache**), а также установить собственное местоположение кэша (рис. 5.18).

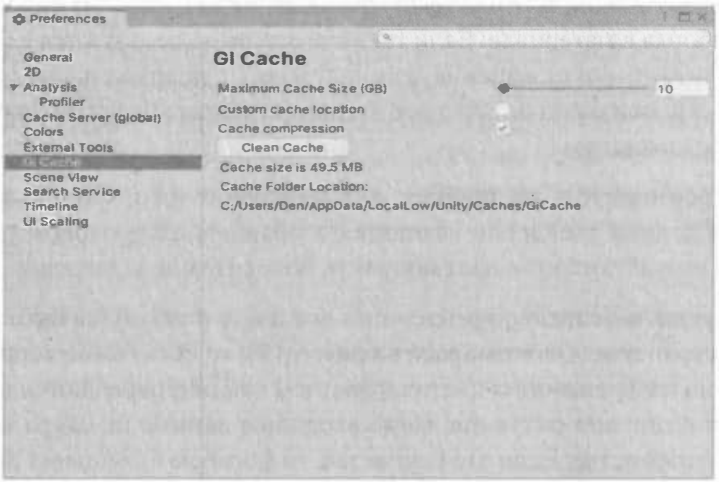


Рис. 5.18. Параметры кэша GI

Если ничего в сцене не изменилось, то при перезагрузке вашей сцены данные освещения должны загружаться из кэша за очень короткое время.

Когда размер кэша становится больше, чем размер, указанный в настройках, Unity создает задание для обрезки давно использованных файлов. Если все файлы в кэше используются, потому что сцена очень большая или размер кэша слишком мал, вам нужно увеличить размер кэша в настройках, иначе Unity будет подтормаживать.

Удалять каталог GI Cache во время работы Unity небезопасно. Папка GI Cache создается при запуске редактора, и редактор поддерживает коллекцию хэшей, которая используется для поиска файлов в папке GI Cache. Если файл или каталог внезапно исчезнет, система не всегда может восстановиться после сбоя, и в консоли вы получите ошибку.

Для удаления кэша всегда используйте только кнопку **Clean Cache** (рис. 5.18), а не очищайте каталог кэша вручную.

Можно совместно использовать папку GI Cache на разных компьютерах, что сделает ребилд ассета Lighting Data быстрее.

5.3. Использование линейного освещения

Линейное освещение – это процесс освещения сцены при полностью линейных входящих данных. Обычно текстуры существуют с заранее примененной к ним гаммой, потому текстуры не линейны, если по ним осуществляется выборка в материале. Если такие текстуры используются в формулах расчета стандартного освещения, это приводит к неправильному результату формул, т.к. они ожидают, что все входящие данные будут линеаризованы перед использованием.

Линейное освещение – это процесс подтверждения того, что и входящие, и исходящие данные шейдеров находятся в правильном цветовом пространстве, что в результате позволяет получить более точное освещение.

В существующем конвейере рендеринга все цвета и текстуры выбираются в гамма пространстве (Gamma Space на рис. 5.19), то есть гамма коррекция не убирается из изображений и цветов перед тем как они передаются в шейдер. Из-за этого возникает ситуация, когда входящие данные шейдера находятся в гамма-пространстве. Если это случается, то формула освещения воспринимает такие данные, как если бы они были в линейном пространстве (Linear Space на рис. 5.19), и в итоге к окончательному пикселю не применяется гамма коррекция. В большинстве случаев это выглядит приемлемо, т.к. две ошибки в итоге друг друга нивелируют. Но это не правильно.

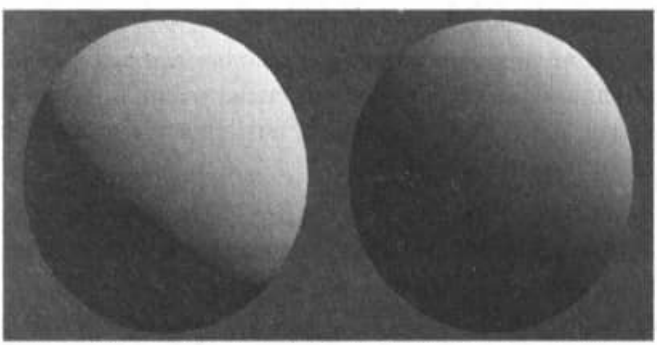


Рис. 5.19. Гамма-коррекция и линейная коррекция

Если линейное освещение включено, тогда входящие данные шейдерной программы поставляются вместе с удаленной из них гамма коррекцией. Для цветов это преобразование применяется автоматически, если вы в линейном пространстве. Текстуры выбираются с использованием аппаратных sRGB

считываний; исходная текстура поставляется в гамма-пространстве и затем при выборке в графическом оборудовании результат автоматически конвертируется. Затем эти входящие данные поставляются шейдеру и освещение рассчитывается так, как оно рассчиталось бы в обычной ситуации. Затем результат записывается в кадровый буфер (*framebuffer*). Затем это значение либо подвергнется гамма коррекции и снова будет записан в кадровый буфер, либо оно останется в линейном пространстве для того, чтобы позже подвергнуться гамма-коррекции; это зависит от текущей конфигурации рендеринга (позже будет показано, как ее изменить).

При использовании линейного освещения, входящие значения уравнений расчета света не такие, как в гамма-пространстве. Другими словами, при столкновении с поверхностями, пучки света будут иметь иную ответную кривую, чем при текущем конвейере рендеринга Unity.

Посмотрите на рис. 5.19. Спад от освещения, основанного на расстоянии (distance-based) или нормалях (normal-based), меняется двумя способами. Во-первых, при рендеринге в линейном режиме, дополнительная производимая гамма-коррекция заставит радиус источника света казаться больше. Во-вторых, края освещения станут более крутыми. Это правильнее моделирует спад интенсивности света на поверхностях.

Когда вы используете освещения в гамма-пространстве, к цветам и текстурам, передаваемым в шейдер, применена гамма коррекция. Когда они используются в шейдере, цвета высокой яркости на самом деле ярче, чем они должны быть при линейном освещении. То есть яркость поверхности будет расти нелинейно при увеличении интенсивности света. Это приводит к слишком яркому освещению во многих местах, и также может привести к ощущению, что все модели и сцены вымыты. Когда вы используете линейное освещение, интенсивность подсвечивания поверхности остается линейной при увеличении интенсивность источника света. В результате вы получит более реалистичное затенение поверхности и намного более красивую цветовую чувствительность поверхности.

Посмотрите на рис. 5.20 – пример линейного (сверху) и гамма (снизу) пространства при разных уровнях интенсивности.

Теперь самое главное, как выбрать тот или иной способ коррекции. Для этого выберите команду меню **Edit, Project Settings, Player, Other Settings**. Параметр **Color Space** позволяет выбрать нужный тип цветового пространства (рис. 5.21).

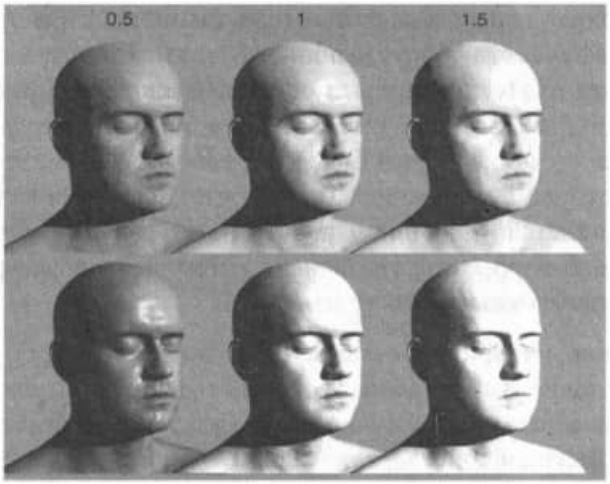


Рис. 5.20. Сравнение линейного и гамма пространств[2]

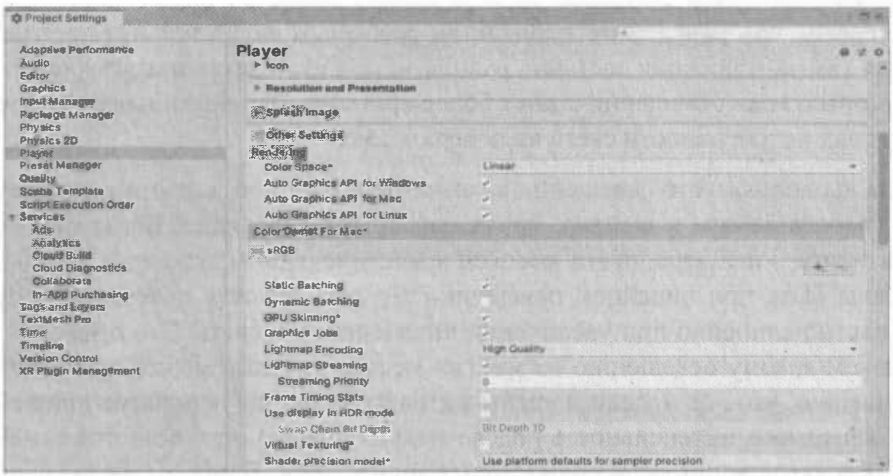


Рис. 5.21. Выбираем тип цветовой коррекции при освещении

Выберите прямо сейчас значение **Linear** и посмотрите, как изменится ваша сцена.

5.4. Камеры

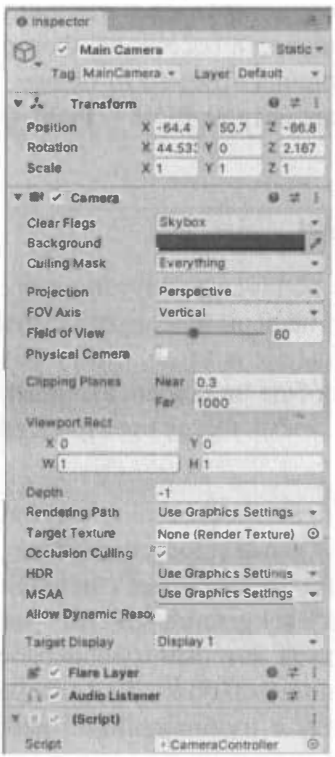
Сцена Unity создается путем размещения и перемещения объектов в трехмерном пространстве. Поскольку экран зрителя является двумерным, дол-

2 Изображение создано Lee Perry-Smith и распространяется по лицензии Creative Commons Attribution 3.0, источник http://www.ir-ltd.net/

жен быть способ захвата изображения и «выравнивания» его для отображения. Это достигается с помощью камер.

Камера – это объект, определяющий вид в пространстве сцены. Положение объекта определяет точку обзора, в то время как прямая (Z) и восходящая (Y) оси объекта определяют направление обзора и верхнюю часть экрана соответственно. Компонент Camera также определяет размер и форму области, которая попадает в область просмотра. После установки этих параметров камера может отображать то, что она «видит» на экране. Когда объект камеры перемещается и вращается, отображаемый вид также будет соответственно перемещаться и вращаться.

5.4.1. Перспективные и ортографические камеры



Камера в реальном мире, или даже человеческий глаз, видит мир так, что объекты выглядят меньше, чем они с точки зрения. Этот известный эффект перспективы широко используется в искусстве и компьютерной графике и важен для создания реалистичной сцены. Естественно, Unity поддерживает перспективные камеры, но в некоторых целях вы хотите визуализировать вид без этого эффекта. Например, вы можете создать карту или информационный дисплей, который не должен выглядеть точно так же, как реальный объект. Камера, которая не уменьшает размер объектов с расстоянием, называется орфографической, и камеры Unity также имеют опцию для этого. Выбор типа камеры осуществляется с помощью свойства **Projection** объекта камеры (рис. 5.22).

Рис. 5.22. Свойства камеры

Как перспективные, так и орфографические камеры имеют ограничение на то, как далеко они могут «видеть» с их текущего положения. Предел определяется плоскостью, которая перпендикулярна направлению камеры (Z). Это известно как *дальняя плоскость отсечения*, поскольку объекты, находящиеся на большем расстоянии от камеры, «обрезаются» (т.е. исключаются из

рендеринга). Кроме того, рядом с камерой имеется соответствующая *ближняя плоскость отсечения* – видимый диапазон расстояний – это расстояние между двумя плоскостями. Соответствующие параметры регулируются свойствами **Clipping Planes Far** и **Clipping Planes Near** в окне инспектора (см. рис. 5.22).

Без перспективы объекты отображаются одинакового размера независимо от их расстояния. Это означает, что объем просмотра орфографической камеры определяется прямоугольной рамкой, проходящей между двумя плоскостями отсечения.

При использовании перспективы объекты уменьшаются в размерах с увеличением расстояния от камеры. Это означает, что ширина и высота видимой части сцены увеличивается с увеличением расстояния. Таким образом, объем обзора перспективной камеры – это не коробка, а пирамида с вершиной в положении камеры и основанием в дальней плоскости отсечения. Форма, однако, не совсем пирамидальная, потому что вершина отрезана от ближней плоскости отсечения. Усечение определяется параметром FOV – **Field Of View**.

5.4.2. Фон для камеры

Для внутренних сцен камера всегда может быть полностью внутри какого-либо объекта, представляющего интерьер здания, пещеры или другой структуры. Однако когда действие происходит на открытом воздухе, между объектами будет много пустых областей, которые вообще не заполнены ничем; эти фоновые области обычно представляют небо, космическое пространство или темные глубины подводной сцены.

Камера не может оставить фон полностью неопределенным, поэтому она должна чем-то заполнить пустое пространство. Самый простой вариант – очистить фон до ровного цвета перед рендерингом сцены поверх него. Вы можете установить этот цвет, используя свойство **Background** камеры – или из инспектора или из скрипта. Более сложный подход, который хорошо работает с наружными сценами, состоит в использовании скайбокса. Как следует из названия, скайбокс ведет себя как «коробочка» с изображениями неба. Камера располагается в центре этой «коробчки» и может видеть небо со всех сторон. Камера видит другую область неба, когда она вращается, но она никогда не сдвигается от центра (поэтому камера не может «приблизиться» к небу). Скайбокс отображается за всеми объектами в сцене, и поэтому он представляет вид на бесконечном расстоянии. Наиболее распространенное использование – это изображение неба в стандартной наружной сцене, но

коробка фактически полностью окружает камеру. Это означает, что вы можете использовать скайбокс для представления частей сцены (например, холмистых равнин, которые простираются за горизонт) или кругового обзора сцены в космосе или под водой.

5.4.3. Использование более одной камеры. Практический пример переключения камер

По умолчанию проект создается с одной камерой на сцене – MainCamera. Однако вы можете создать несколько камер – столько, сколько вам нужно. Например, вы можете использовать две камеры – одну для вида от первого лица, а другую – для вида сверху.

По умолчанию камера отображает вид на весь экран, поэтому за один раз можно увидеть только один вид камеры (видимая камера имеет наибольшее значение для своего свойства Depth). Отключив одну камеру и включив другую из сценария, вы можете получить разные виды сцены. Например, так можно делать, чтобы переключиться между видом карты сверху и видом от первого лица. Рассмотрим соответствующий сценарий (лист. 5.1) на C#.

Листинг 5.1. Переключение камер. Абстрактный пример

```
using UnityEngine;
using System.Collections;

public class ExampleScript : MonoBehaviour {
    public Camera firstPersonCamera;
    public Camera overheadCamera;

    public void ShowOverheadView() {
        firstPersonCamera.enabled = false;  // выключаем камеру
от первого лица
        overheadCamera.enabled = true;
    }

    public void ShowFirstPersonView() {
        firstPersonCamera.enabled = true; // включаем камеру от
первого лица
        overheadCamera.enabled = false;
    }
}
```

Обычно требуется, чтобы хотя бы один вид камеры покрывал весь экран (настройка по умолчанию), но часто полезно показывать другой вид внутри

небольшой области экрана. Например, вы можете показать зеркало заднего вида в гоночной игре. Вы можете установить размер экранного прямоугольника камеры, используя его свойство **Viewport Rect**.

Координаты прямоугольника области просмотра «нормализованы» относительно экрана. Нижний и левый края имеют координату 0.0, а верхний и правый края – 1.0. Значение координаты 0,5 – половина. Также значение свойства **Depth** для основной камеры должно быть меньше, чем для камеры с меньшим видом. Точное значение не имеет значения, но правило состоит в том, что камера с более высоким значением **Depth** отображается *поверх* камеры с более низким значением.

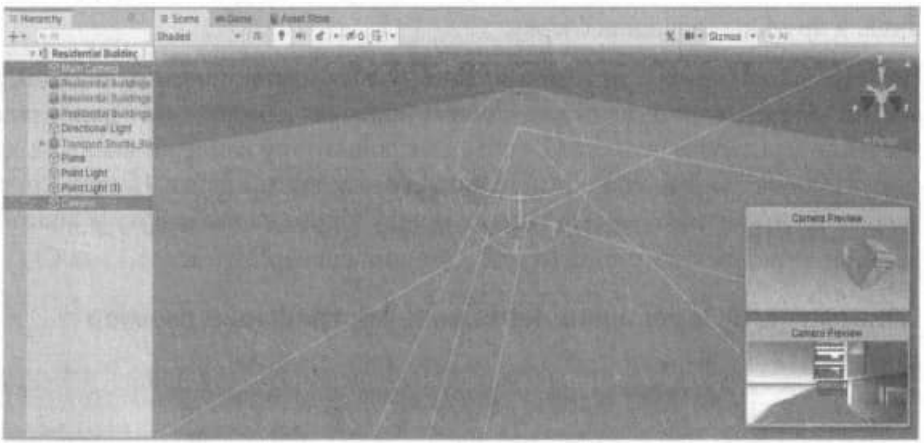


Рис. 5.23. Две камеры с разными видами

Теория – это, конечно, хорошо, но, наверное, вам хочется узнать, как организовать переключение камер на практике. У нас уже есть демо-сцена из пакета **Residential Buildings Set** (https://free3d.com/ru/3d-model/array-house-example-3033.html). Расположите на сцене несколько домов по своему усмотрению. Пусть в качестве игрока у нас будет выступать сфера – поместите ее на сцену.

К сфере нужно добавить два компонента:

- **Rigidbody** – для него выключите параметр **Use Gravity** и установите ограничения (**Constraints**) так, как показано на рис. 5.24. Мы не будем перемещаться по оси Y, но будем перемещаться по осям X и Z. В принципе все равно, по каким осям будем «летать» ваша сфера, главное продемонстрировать переключение камер
- **Character Controller** – контроллер игрока

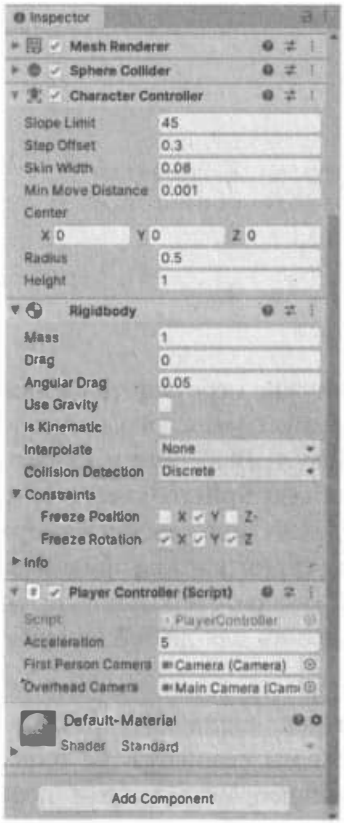


Рис. 5.24. Свойства сферы

Также для сферы установите значение **Player** для **Tag** – чтобы система и не сомневалась, кто будет здесь игроком.

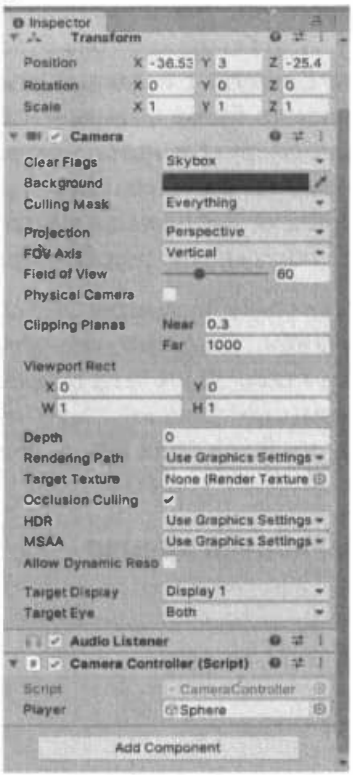
Поместите на сцену еще одну камеру (одна, Main Camera) у вас уже есть. Используя инструменты перемещения и вращения, переместите **Main Camera** так, чтобы она показывала сцену сверху. Другую камеру (она будет называться просто **Camera**) с помощью этих инструментов направьте ближе к земле – чтобы она показывала сферу. Свойства камеры установите по своему усмотрению. Пусть она смотрит с некоторым отдалением на нашего игрока – чтобы получить вид от третьего лица.

Теперь нужно заставить наши камеры следить за нашим персонажем. Создайте сценарий CameraController.cs (лист. 5.2). С этим сценарием вы уже знакомы по прошлой главе, но, чтобы вы лишний раз не листали книгу, приведу его еще раз.

Листинг 5.2. Сценарий CameraController. Рабочий пример

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CameraController : MonoBehaviour
{

    public GameObject Player;

    private Vector3 offset;
    // Start вызывается перед первым изменением кадра
    void Start()
    {
       // вычисляем смещение
        offset = transform.position - Player.transform.
position;
```

```
    }

    // Вызывается каждый кадр
    void Update()
    {
        // позиция камеры равна позиции игрока + смещение
         transform.position = Player.transform.position +
offset;

    }
}
```



Данный сценарий нужно перетащить **на каждую камеру**. После этого у каждой камеры появится свойство Player. Выберите в качестве его значения наш объект **Sphere**. Заметьте: это действие нужно сделать **для каждой камеры** (рис. 5.25). В самом скрипте мы не привязываемся к конкретной камере, следовательно, мы можем его использовать для любой камеры в проекте.

Теперь реализуем наш сценарий управления персонажем. В нем мы реализуем не только управление с помощью WASD, но и переключение камер. Камеры будут переключаться так: при нажатии клавиши Fire1 (ей соответствует клавиша Ctrl на клавиатуре) будет включаться верхняя камера, при нажатии клавиши **Jump** (это Пробел) – основная камера.

Рис. 5.25. Свойства камеры

Листинг 5.3. Управление персонажем

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour
{
```

```
    public int acceleration;
    public Camera firstPersonCamera;
    public Camera overheadCamera;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        var direction = new Vector3(Input.
GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
        GetComponent<Rigidbody>().AddForce(direction *
acceleration);

        if(Input.GetButtonDown("Fire1"))
     {
firstPersonCamera.enabled = false;              overheadCamera.
enabled = true;
     }
        if(Input.GetButtonDown("Jump"))
        {
firstPersonCamera.enabled = true;            overheadCamera.
enabled = false;

        }

    }
}
```

После сохранения этого сценария перетащите его на сферу. У сферы появится два новых свойства (рис. 5.25) – **First Person Camera** и **Overhead Camera**. В качестве значения первого свойства установите созданную вами камеру (объект Camera), в качестве значения второго свойства – камеру Main Camera (или наоборот, если вы использовали объект Camera для вида сверху). Для параметра **acceleration** установите значение, отличное от 0, иначе сфера никуда не покатится. Не устанавливайте слишком большое значение, иначе будет слишком быстро. Значения от 5 до 10 вполне уместны.

> **Примечание.** При открытии некоторых устаревших ассетов вроде Particle Dissolve Shader by Moonflower Carnivore в новой версии 2020.3 вы можете столкнуться с ошибками вида:

```
'GUIText' is obsolete: 'GUIText has been removed. Use
UI.Text instead.

'GUITexture' is obsolete: 'GUITexture has been removed.
Use UI.Image instead.'
```

В этом случае в самом начале файла с ошибкой в конец деклараций using нужно добавить строчку:

```
using UnityEngine.UI;
```

Далее все вхождения GUIText нужно заменить на Text, а все вхождения GUITexture – на Image. После сохранения файла ошибка будет исправлена.

Запустите игру, посмотрим, что получилось. На рис. 5.26 показана наша сфера, катящаяся вдоль сцены. Нею можно управлять с помощью клавиш WASD.

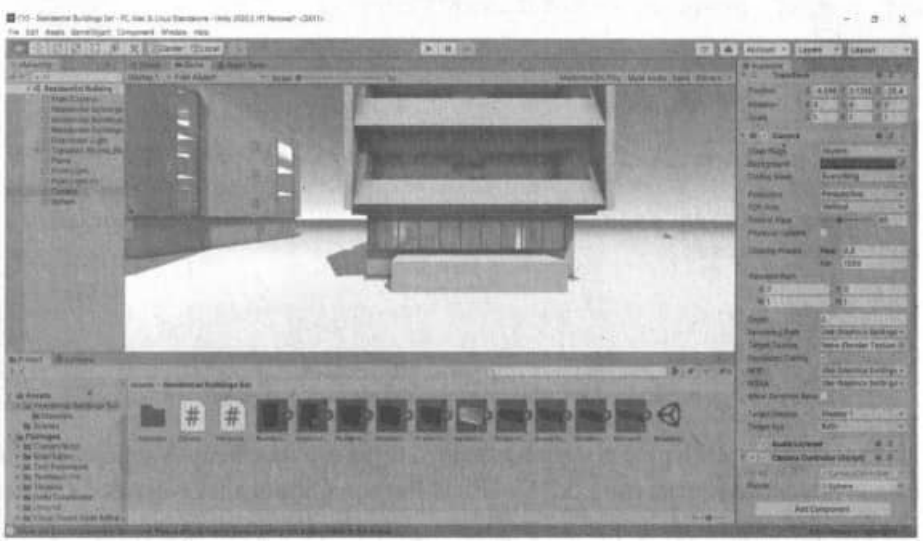


Рис. 5.26. Сфера катится вдоль сцены. Нормальный вид

Нажмите клавишу Ctrl. Будет отображен вид сверху (рис. 5.27). Нажатие пробела вернет обычный вид. На рис. 5.28 показано, каким будет вид, если в настройках освещения (**Window, Rendering, Lighting Settings**) отключить туман (параметр **Fog**).

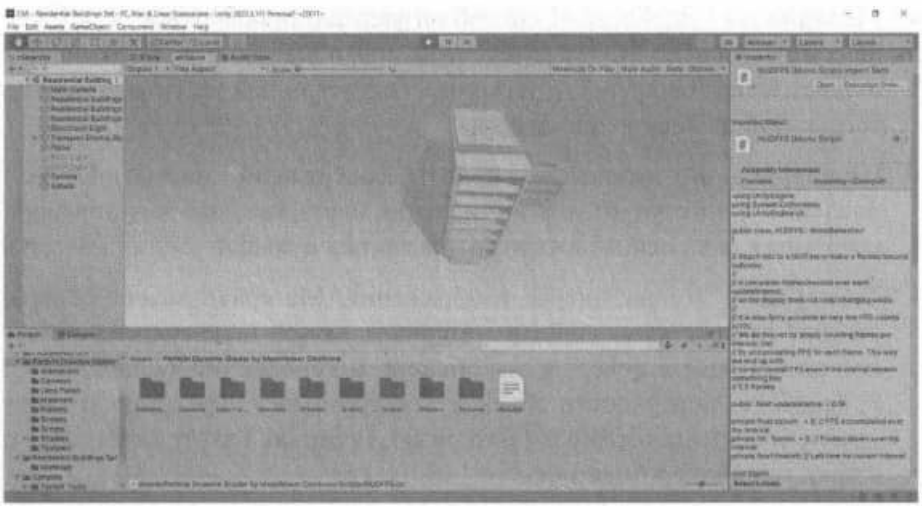


Рис. 5.27. Вид сверху

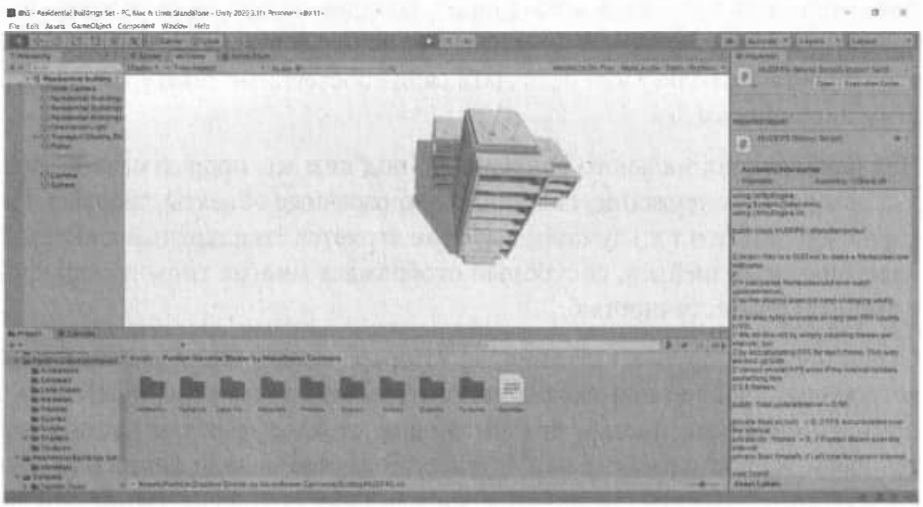


Рис. 5.28. Вид сверху без тумана

Итак, вы только что получили два практических навыка:

- Вы научились переключать камеры
- Узнали, как устанавливать реакции на нажатия клавиш Fire1 и Jump

5.5. Материалы, текстуры и шейдеры

Рендеринг в Unity осуществляется с помощью материалов, текстур и шейдеров. Это три ключевых понятия, относящиеся к рендерингу. Разберемся, что они означают:

- **Материалы** – определяют способ визуализации поверхности, в том числе ссылки на используемые текстуры, информацию о мозаике, цветовые оттенки и многое другое. Доступные параметры для материала зависят от того, какой шейдер используется
- **Шейдеры** – это небольшие скрипты, содержащие математические вычисления и алгоритмы для вычисления цвета каждого визуализированного пикселя на основе входного освещения и конфигурации материала
- **Текстуры** – это растровые изображения. Материал может содержать ссылки на текстуры, шейдер материала может использовать текстуры при расчете цвета поверхности объекта. В дополнение к основному цвету (альбедо) поверхности объекта, текстуры могут представлять многие другие аспекты поверхности материала, такие как его отражательная способность или шероховатость

В материале указывается один конкретный шейдер, а он уже определяет, какие параметры доступны в материале. Шейдер задает одну или несколько переменных текстур, которые он ожидает использовать, а инспектор материалов в Unity позволяет вам назначать свои собственные текстурные ресурсы этим переменным.

Для наиболее нормального рендеринга (под ним мы подразумеваем символы, декорации, окружение, сплошные и прозрачные объекты, твердые и мягкие поверхности и т.д.) лучшим выбором является стандартный шейдер. Это настраиваемый шейдер, способный отображать многие типы поверхностей с высокой реалистичностью.

Существуют и другие ситуации, в которых может быть уместен другой встроенный шейдер или даже пользовательский письменный шейдер – например, жидкости, листва, преломляющее стекло, эффекты частиц, мультфильмы, иллюстративные или другие художественные эффекты или другие специальные эффекты, такие как ночное видение, тепловое зрение или рентгеновское зрение и т.д.

5.5.1. Создание и использование материалов

Самый экономный по времени способ создания материалов – импортировать их вместе с каким-то пакетом ассетов. Как правило, в нем будут все необходимые тематические материалы. Но при желании материал можно создать самому. Для этого выберите команду меню **Assets, Create, Material**. В папке **Materials** появится новый ассет – **New Material**, а область инспектора будет отображать доступные для редактирования свойства (рис. 5.29). По

умолчанию новым материалам назначается стандартный шейдер со всеми пустыми свойствами карты.

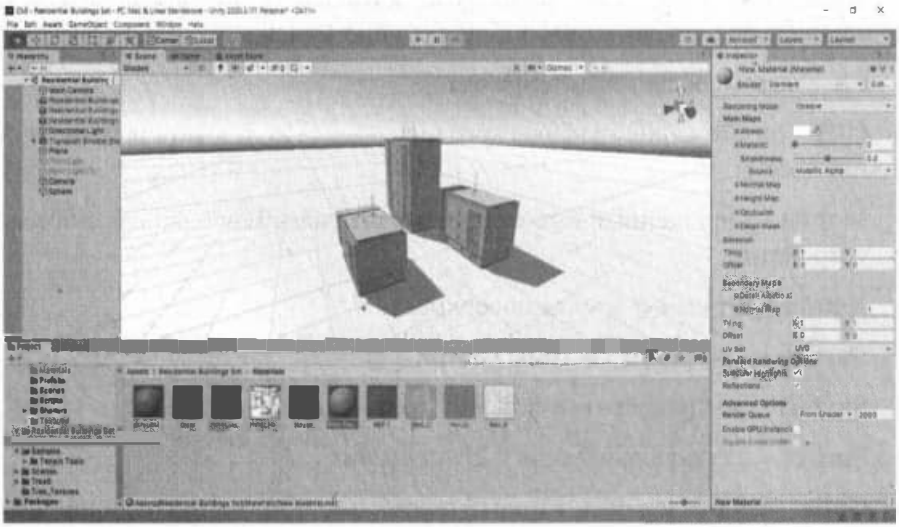


Рис. 5.29. Создание нового материала

Вы можете выбрать, какой именно шейдер будет использоваться в конкретном материале. Для этого просто раскройте выпадающее меню **Shader** в инспекторе и выберите подходящий вам шейдер (находится сразу после названия материала в самом верху окна **Inspector**).

Выбранный вами шейдер будет задавать доступные свойства. Свойствами могут быть цвета, текстуры, числа или векторы. Если вы применили материал к активному объекту в сцене, вы увидите, как в реальном времени будут применяться изменяемые вами свойства.

Существует два способа присваивания текстуры свойству:

1. С помощью перетаскивания текстуры из окна **Project** на квадратную область свойства **Texture**

2. Кнопки выбора и выбора из появившегося выпадающего списка

Созданный материал можно потом использовать в качестве свойства **Material** для объектов, способных принимать такие свойства. Кстати, после создания собственного материала не забудьте щелкнуть на нем (в окне обозревателя) правой кнопкой мыши и выбрать команду **Rename** для его переименования. Приучайтесь к порядку и назначайте значимые имена для всех создаваемых вами ассетов.

5.5.2. Встроенные шейдеры

В дополнение к основным шейдерам, применяемым к игровым объектам, существует ряд других категорий для особых целей:

- **FX** – водные и световые эффекты
- **GUI** – отображение графического интерфейса пользователя (Graphic User Interface[3])
- **Mobile** – упрощенный высокопроизводительный шейдер для мобильных устройств
- **Nature** – деревья и земная поверхность
- **Particles** – эффекты системы частиц
- **Skybox** – для рендеринга фоновой среды (скайбокса)
- **Sprites** – для использования в 2D-системах
- **Unlit** – для рендеринга, который полностью обходит свет и тени
- **Unity5 (Legacy Shaders)** – большой набор шейдеров, которые использовались в ранних версиях Unity и которые сейчас заменены стандартным шейдером

Шейдер – это скрипт, который содержит математические вычисления и алгоритмы того, как должны выглядеть пиксели на поверхности модели. Стандартный шейдер выполняет сложные и реалистичные расчеты освещения. Другие шейдеры могут использовать более простые или разные вычисления, чтобы показать разные результаты. Внутри любого данного шейдера есть ряд свойств, которым может быть присвоено значение материалом, использующим этот шейдер. Этими свойствами могут быть числа, определения цветов или текстуры, которые отображаются в инспекторе при просмотре материала. Затем материалы используются компонентами рендерера, прикрепленными к игровым объектам, для визуализации меша каждого игрового объекта.

Возможно, и, часто желательно, иметь несколько разных материалов, которые могут ссылаться на одинаковые текстуры. В этих материалах также могут использоваться одни и те же или разные шейдеры, в зависимости от требований.

Чтобы было понятно, кто за что отвечает, просмотрим таблицу 5.3, в которой в левой колонке показано, что определяет шейдер, а в правой – материал.

3 О разработке графического интерфейса пользователя мы поговорим в других главах

Таблица 5.3. Шейдер vs Материал

Шейдер	Материал
Метод визуализации объекта – кодовые и математические вычисления, которые могут включать в себя углы источников света, угол обзора и любые другие соответствующие вычисления. Шейдеры также могут указывать разные методы в зависимости от графического оборудования конечного пользователя	Какие текстуры использовать для отрисовки
Параметры, которые можно настроить в инспекторе материалов, такие как карты текстур, цвета и числовые значения	Конкретные значения параметров шейдера – например, какие карты текстур, цветовые и числовые значения использовать

Созданием шейдеров занимаются особые программисты – 3D-программисты. Шейдеры создаются на языке ShaderLab, рассмотрение которого выходит за рамки этой книги. Хотя язык и довольно простой, но заставить шейдер работать правильно на всем многообразии имеющихся видеокарт – задача очень сложная и требует специальных знаний и понимания того, как работают видеокарты.

5.5.3. Стандартный шейдер

Стандартный шейдер Unity (шейдер Standard) – это встроенный шейдер с полным набором функций. Он может использоваться для визуализации объектов «реального мира», таких как камень, дерево, стекло, пластик и металл, и поддерживает широкий спектр типов и комбинаций шейдеров. Его функции можно включить или отключить, просто используя или не используя различные слоты текстуры и параметры в редакторе материалов.

Сейчас в Unity для большинства потребностей используется шейдер Standard. Он предназначен для замены большого количества старых шейдеров, использовавшихся в предыдущих версиях Unity – наиболее существенно он заменяет все шейдеры, которые использовались для рендеринга объектов «реального мира», таких как камень, дерево, стекло, пластик, металл, и т.п.

Потребности использовать **Legacy Shaders** (устаревшие) шейдеры нет, хотя такая возможность есть (но оставлена она из соображений обратной совместимости).

Стандартный шейдер также включает в себя усовершенствованную модель освещения, которая называется затенением на физическом уровне. Физически обоснованное затенение (PBS) имитирует взаимодействие материалов и света таким образом, чтобы имитировать реальность. PBS относительно недавно стал возможен в графике в реальном времени. Наилучшим образом он работает в ситуациях, когда освещение и материалы должны существовать вместе интуитивно и реалистично.

Идея нашего PBS-шейдера состоит в том, чтобы создать удобный для пользователя способ достижения последовательного, правдоподобного вида при различных условиях освещения. Он моделирует, как свет ведет себя в реальности, без использования нескольких специальных моделей, которые могут работать или не работать. Для этого он следует принципам физики, включая сохранение энергии (это означает, что объекты никогда не отражают больше света, чем они получают), отражения Френеля (все поверхности становятся более отражающими при углах скольжения) и то, как поверхности перекрывают себя.

С помощью стандартного шейдера широкий спектр типов шейдеров (таких как Diffuse, Specular, Bumped Specular, Reflective) объединяются в один шейдер, предназначенный для использования со всеми типами материалов. Преимущество этого заключается в том, что одни и те же расчеты освещения используются во всех областях вашей сцены, что дает реалистичное, согласованное и правдоподобное распределение света и тени по всем моделям, использующим шейдер.

Стандартный шейдер предоставляет вам список параметров материала. Эти параметры немного различаются в зависимости от выбранного режима рендеринга. Большинство параметров одинаковы для всех режимов и представлены в таблице 5.4.

Таблица 5.4. Параметры материала (стандартный шейдер)

Параметр	Описание

Rendering Mode	Определяет режим рендеринга. Позволяет вам выбрать, использует ли объект прозрачность, и если да, то какой тип режима наложения использовать. **Opaque** (по умолчанию) – по умолчанию. Подходит для обычных твердых объектов без прозрачных областей. **Cutout** – позволяет создать прозрачный эффект с жесткими краями между непрозрачными и прозрачными областями. В этом режиме нет полупрозрачных областей, текстура либо на 100% непрозрачна, либо невидима. Это полезно при использовании прозрачности для создания формы таких материалов, как листья или ткань с отверстиями и лохмотьями. **Transparent** – подходит для рендеринга реалистичных прозрачных материалов, таких как прозрачный пластик или стекло. В этом режиме сам материал будет принимать значения прозрачности (на основе альфа-канала текстуры и альфа-цвета оттенка), однако отражения и блики освещения будут оставаться видимыми с полной ясностью, как в случае с настоящими прозрачными материалами. **Fade** – позволяет значениям прозрачности полностью затухать объект, включая любые зеркальные блики или отражения, которые он может иметь. Данный режим полезен, если вы хотите анимировать исчезающий или исчезающий объект. Он не подходит для рендеринга реалистичных прозрачных материалов, таких как прозрачный пластик или стекло, поскольку отражения и блики также будут размыты. Подходит для прорисовки голограмм – отображения непрозрачного объекта, который частично исчезает
Albedo	Управляет основным цветом поверхности. Иногда полезно указать один цвет для значения Albedo, но гораздо чаще назначается карта текстуры для параметра Albedo. Это должно представлять цвета поверхности объекта. Важно отметить, что текстура Albedo не должна содержать никакого освещения, так как освещение будет добавлено к нему в зависимости от контекста, в котором виден объект

Metallic	Регулирует, как отражательная способность и световой отклик поверхности изменяются в зависимости от уровня металла и уровня гладкости. Если простыми словами, то он определяет, насколько металлоподобно поверхность. Когда поверхность более металлическая, она больше отражает окружающую среду, а ее цвет альбедо становится менее заметным. На полностью металлическом уровне цвет поверхности полностью зависит от отражений от окружающей среды. Когда поверхность менее металлическая, ее цвет альбедо становится более четким, и любые поверхностные отражения видны поверх цвета поверхности, а не скрывают его. По умолчанию без назначения текстуры параметры **Metallic** и **Smoothness** (гладкость) контролируются ползунком каждый. Этого достаточно для некоторых материалов. Если на поверхности вашей модели есть области со смесью типов поверхности в текстуре альбедо, вы можете использовать карту текстуры, чтобы контролировать, как металлические и гладкие уровни изменяются по поверхности материала. Например, если ваша текстура содержит одежду персонажа, включая металлические пряжки и молнии. Вы бы хотели, чтобы пряжки и молнии имели более высокую металлическую ценность, чем ткань одежды. Чтобы достичь этого, вместо использования одного значения ползунка можно назначить карту текстуры, которая содержит более светлые пиксельные цвета в областях пряжек и молний и более темные значения для ткани. Как только вы назначите текстуру параметру Metallic, ползунки Metallic и Smoothness исчезнут. Вместо этого уровни металла для материала контролируются значениями в канале текстуры красного цвета, а уровни гладкости материала контролируются альфа-каналом текстуры

Normal Map	Задает карту нормалей. Карты нормалей представляют собой особый вид текстуры, который позволяет добавлять детали поверхности, такие как неровности, бороздки и царапины, к модели, которая улавливает свет, как будто они представлены реальной геометрией. Например, вы можете захотеть показать поверхность, на которой есть углубления и винты или заклепки, например корпус самолета. Один из способов сделать это – смоделировать эти детали как геометрию (рис. 5.30). В современных арт-конвейерах для разработки игр художники будут использовать свои приложения для трехмерного моделирования для создания карт нормалей на основе исходных моделей очень высокого разрешения. Карты нормалей затем сопоставляются с готовой к игре версией модели с более низким разрешением, чтобы исходные детали высокого разрешения отображались с использованием карты нормалей
Height Map	Задает карту высот. Карта высот схожа по концепции с картой нормалей. Карты высот обычно используются в сочетании с картами нормалей, и часто они используются для придания дополнительного определения поверхностям, где карты текстур отвечают за рендеринг больших выпуклостей и выступов. Карта нормалей изменяет освещение по всей поверхности текстуры, отображение высоты параллакса делает шаг вперед и фактически смещает области видимой текстуры поверхности вокруг, чтобы достичь своего рода эффекта окклюзии на уровне поверхности. Карта высот должна быть изображением в оттенках серого, с белыми областями, представляющими высокие области вашей текстуры, и черными, представляющими низкие области. Назначение и карты высот, и карты нормалей делает изображение наиболее реалистичным, но это довольно дорогостоящий способ, как стоимости затрат материальных, так и затрат системных ресурсов. Обычно вполне достаточно назначения карты нормалей

Occlusion	Карта окклюзии используется для предоставления информации о том, какие области модели должны получать высокое или низкое непрямое освещение. Непрямое освещение исходит от окружающего освещения и отражений, и поэтому крутые вогнутые части вашей модели, такие как трещина или складка, не будут реально получать много непрямого света. Карты текстуры окклюзии обычно рассчитываются 3D-приложениями напрямую из 3D-модели с использованием программы моделирования или стороннего программного обеспечения. Карта окклюзии – это изображение в градациях серого, с белым цветом обозначены области, которые должны получать полное непрямое освещение, и черным цветом – отсутствие косвенного освещения. Иногда это так же просто, как карта высот в оттенках серого, для простых поверхностей
Emission	Контролирует цвет и интенсивность света, излучаемого с поверхности. Когда в вашей сцене используется излучающий материал, он кажется видимым источником света. Объект будет выглядеть «самосветящимся». Материалы эмиссии обычно используются на объектах, где какая-то часть должна казаться освещенной изнутри, таких как экран монитора, дисковые тормоза автомобиля, тормозящие на высокой скорости, светящиеся кнопки на панели управления или глаза монстра, которые видны в темноте
Secondary Maps	Параметры этого раздела позволяют задавать дополнительные карты. Они позволяют накладывать второй набор текстур поверх основных текстур, перечисленных выше. Вы можете применить вторую карту цветов Альбедо и вторую карту нормалей

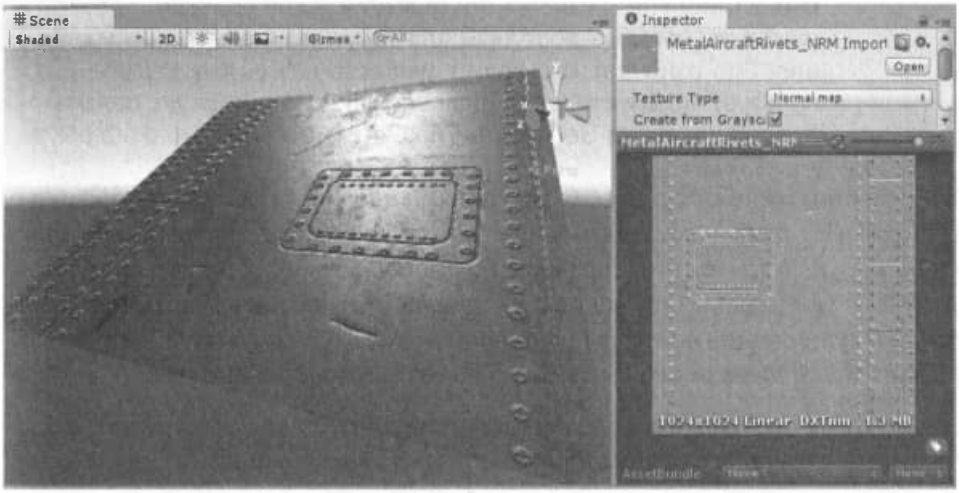


Рис. 5.30. Использование карты нормалей в качестве типа текстуры

5.5.4. Изменение материала через скрипт

Все параметры материала, которые вы видите в инспекторе при просмотре материала, доступны через скрипт, что дает вам возможность изменять или анимировать работу материала во время выполнения.

Это позволяет вам изменять числовые значения на материале, изменять цвета и динамически менять текстуры во время игры. Вот некоторые из наиболее часто используемых методов:

- SetTexture() – устанавливает новую текстуру материала
- SetColor() – устанавливает новый цвет материала
- SetInt() – устанавливает целое значение в материале
- GetInt() – получает установленное ранее целое значение в материале
- SetFloat() – устанавливает вещественное значение в материале
- GetFloat() – получает установленное ранее float-значение в материале
- HasProperty() – возвращает true, если материал содержит свойство с заданным именем (имя передается в качестве строкового параметра)

Это, конечно, не все функции класса Material, но это самые часто используемые на практике.

Внимание! Помните, что эти функции устанавливают только те свойства, которые доступны для текущего шейдера на материале. Это означает, что если у вас есть шейдер, который не использует никаких текстур, или если у вас нет шейдера, связанного вообще, вызов SetTexture() не будет иметь никакого эффекта.

Эти функции работают для всех простых шейдеров, таких как устаревшие шейдеры (legacy), и встроенных шейдеров, кроме стандартного шейдера (например, шейдер частиц, спрайтов, пользовательского интерфейса и неосвещенных шейдеров). Однако для материала, использующего стандартный шейдер, существуют некоторые дополнительные требования, о которых вы должны знать, прежде чем сможете полностью изменить материал.

Стандартный шейдер имеет некоторые дополнительные требования, если вы хотите модифицировать материалы во время выполнения, потому что – за кулисами – это фактически много разных шейдеров, объединенных в один.

Эти различные типы шейдеров называются вариантами шейдеров и могут рассматриваться как все различные возможные комбинации функций шейдера, когда они активированы или не активированы.

Например, если вы решили назначить карту нормалей вашему материалу, вы активируете тот вариант шейдера, который поддерживает отображение нормалей. Если впоследствии вы также назначите карту высот, вы активируете вариант шейдера, который поддерживает нормальное отображение и отображение высоты.

Это хорошая система, потому что это означает, что если вы используете Стандартный шейдер, но не используете карту нормалей в определенном материале, вы не несете затрат производительности при выполнении кода шейдера карты нормалей.

Если вы используете сценарии для изменения материала, который заставит его использовать другой вариант стандартного шейдера, вы должны включить этот вариант с помощью функции EnableKeyword. Другой вариант потребуется, если вы начнете использовать функцию шейдера, которая изначально не использовалась материалом. Например, назначение карты нормалей для материала, у которого его нет, или установка уровня эмиссии на значение больше нуля, когда оно ранее было нулевым.

Конкретные ключевые слова, необходимые для включения функций стандартного шейдера, приведены в табл. 5.5.

Таблица 5.5. Ключевые слова для включений функций стандартного шейдера

Ключевое слово	Особенность
_NORMALMAP	Карта нормалей
_ALPHATEST_ON	Режим рендеринга прозрачности Cutout
_ALPHABLEND_ON	Режим рендеринга прозрачности Fade
_ALPHAPREMULTIPLY_ON	Режим рендеринга прозрачности Transparent
_EMISSION	Цвет эмиссии
_PARALLAXMAP	Карта высот
_DETAIL_MULX2	Дополнительные «карты (карта альбедо и нормалей)
_METALLICGLOSSMAP	Включение параметра Metallic

Дополнительная информация о скриптинге материалов доступна в руководстве Unity https://docs.unity3d.com/ScriptReference/Material.html[4].

5.6. Создание ландшафта

Система ландшафта Unity позволяет вам добавлять обширные ландшафты в ваши игры. Во время выполнения рендеринг ландшафта сильно оптимизирован для эффективности рендеринга, а в редакторе доступен выбор инструментов, позволяющих легко и быстро создавать ландшафты.

5.6.1. Создание и редактирование местности

Для создания местности выберите команду меню **GameObject, 3D Object, Terrain**. Будет создана плоская и ничем не примечательная поверхность (рис. 5.31).

4 Когда будете пользоваться руководством, обращайте на номер версии Unity, руководство по которой вы читаете, обычно номер версии можно изменить в верхнем левом углу сайта

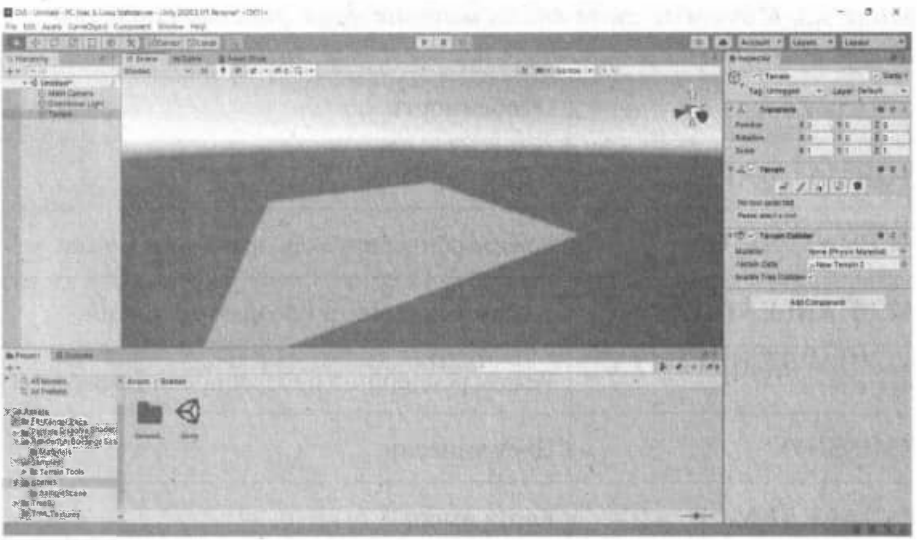


Рис. 5.31. Новая местность

Попробуем придать поверхности рельеф. Для этого выделите объект поверхности в иерархии и перейдите в область инспектора. В разделе **Terrain** нажмите вторую кнопку – с изображением кисти. Данная кнопка позволит редактировать поверхность с помощью кистей. Далее все достаточно интересно:

1. Выберите кисть подходящей формы.
2. Щелкните на поверхности, где нужно организовать выпуклость. Если хотите организовать впадину, то нужно щелкнуть левой кнопкой мыши при нажатой клавише **Shift** (рис. 5.32).
3. Область **Brushes** позволяет сменить форму кисти, при этом параметр **Brush Size** устанавливает размер кисти, а параметр **Opacity** – непрозрачность.
4. Проявите фантазию. За пару кликов мышки можно создать что-то вроде изображенного на рис. 5.33 – эдакую земляную крепость, которые были популярны много веков назад.

Когда вы наиграетесь с кистями, обратите внимание на выпадающий список, в котором по умолчанию выбрано значение **Raise or Lower Terrain**. Это означает, что выбран инструмент, позволяющий нарисовать на поверхности разные возвышенности и впадины. Причем впадины можно нарисовать

только на нарисованных ранее возвышенностях – это такая себе особенность этого инструмента.

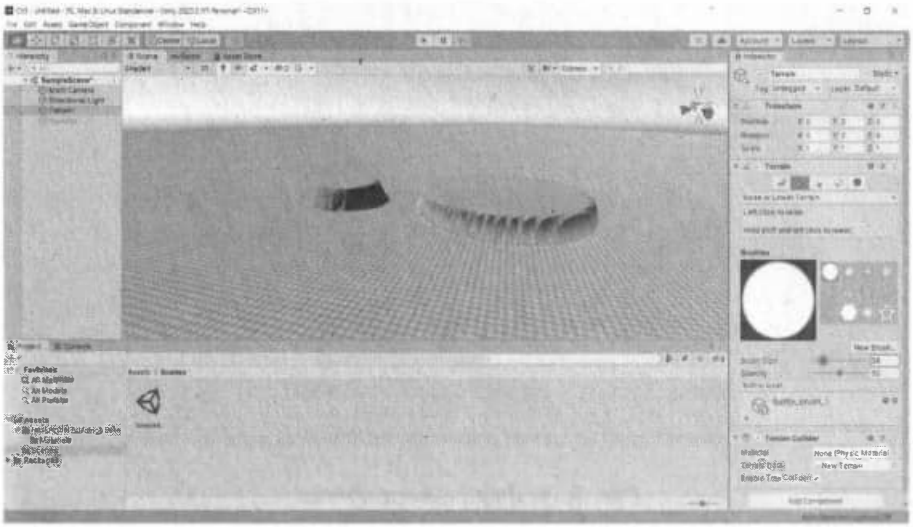


Рис. 5.32. Создаем неповторимый рельеф

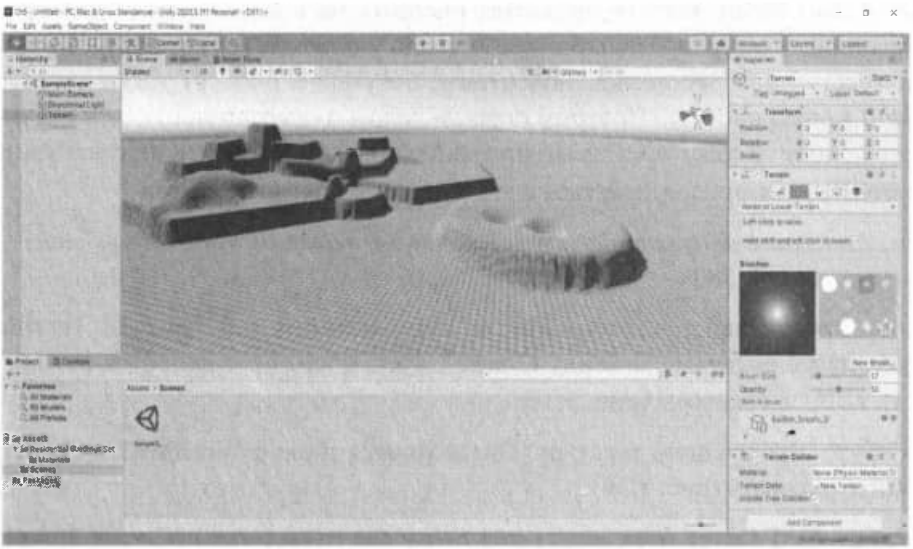


Рис. 5.33. На создание такой местности ушло не более 30 секунд

Инструмент **Paint Holes** рисует «дыры» на поверхности. Обратите внимание, что это не впадины, а именно дыры, через которые игрок может «провалиться» под поверхность и собственно улететь в никуда. Нужно вам это или нет, зависит от игры, но на рис. 5.34 изображена типичная «дыра».

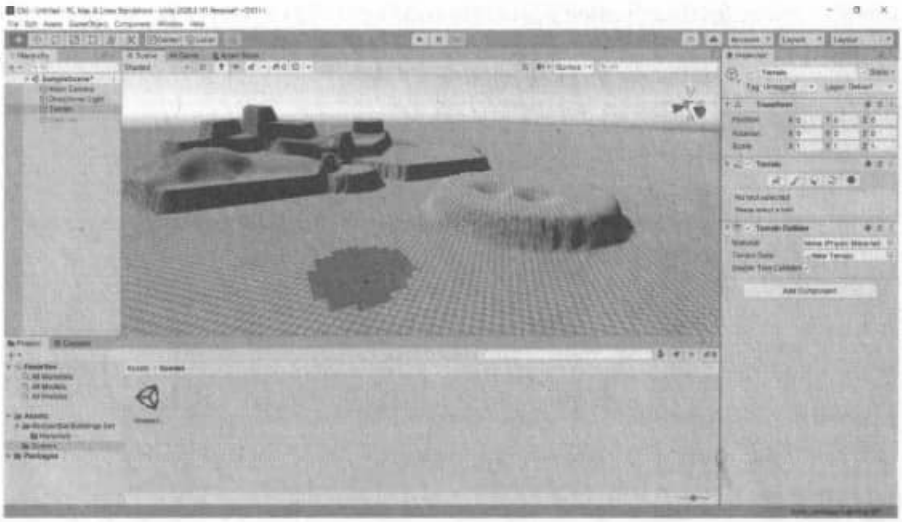


Рис. 5.34. Дыра в местности

Инструмент **Paint Texture** позволяет рисовать на поверхности с помощью выбранной текстуры. Однако по умолчанию у объекта Terrain нет назначенных текстур для рисования. Более того, по умолчанию нет таких текстур даже в вашем проекте. Поэтому первым делом нужно позаботиться об импорте подобных текстур. Самый простой вариант – загрузить их с магазина ассетов, можно воспользоваться следующим бесплатным паком:

https://assetstore.unity.com/packages/2d/textures-materials/terrain-tools-sample-asset-pack-145808

Импортируйте набор текстур. После этого нажмите кнопку **Edit Terrain Layer**, далее выберите команду **Create Layer**, после чего появится возможность выбора текстуры (рис. 5.35).

Выберите подходящую текстуру. Она появится в окне **Terrains Layer**. Начинайте рисовать (рис. 5.36).

Команда **Add Layer** (она находится в том же меню, что и **Create Layer**) позволяет выбрать (добавить) предопределенный уровень местности. В зависимости от выбранного уровня выбирается автоматически и текстура. В импортированном в этом разделе пакете предопределены уровни, показанные на рис. 5.37 – rock (скала), sand (песок), snow (снег) и т.д.

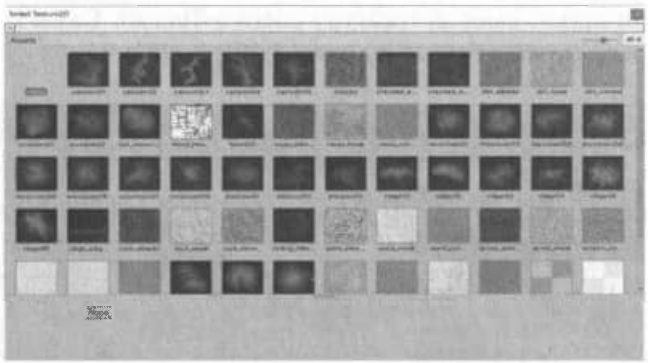


Рис. 5.35. Выбор текстуры

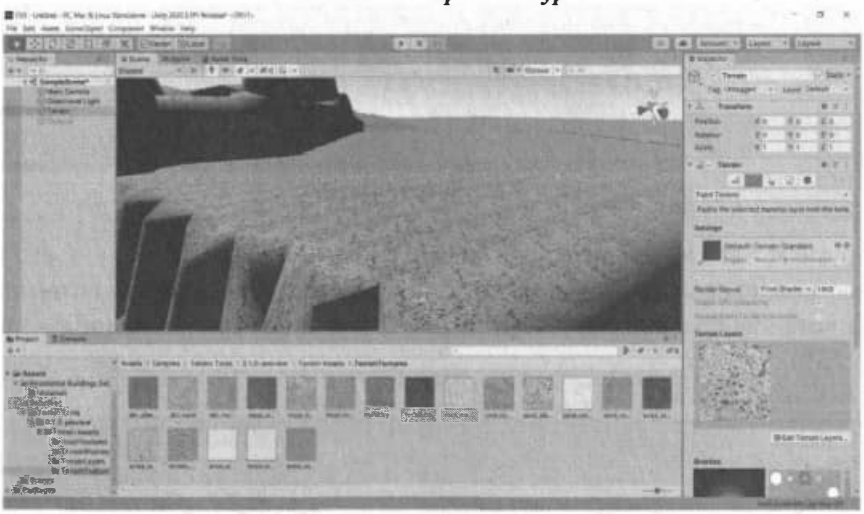


Рис. 5.36. Применение текстуры к Terrain

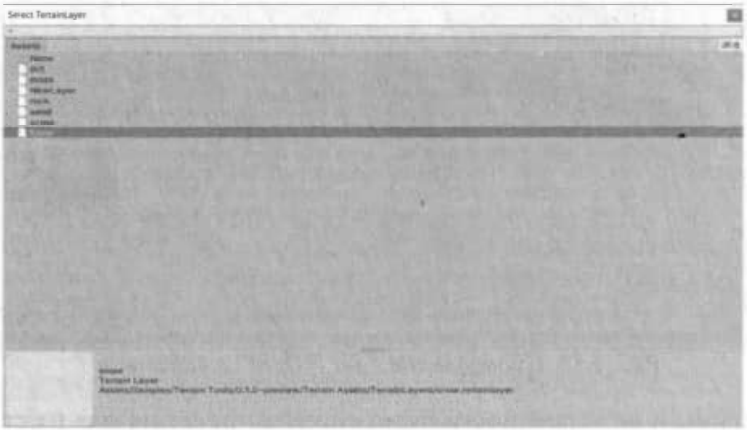


Рис. 5.37. Уровни местности

Далее принцип прост: выбрав текстуру снега в **Terrain Layers**, разукрасьте места, в которых должны быть снег (рис. 5.38).

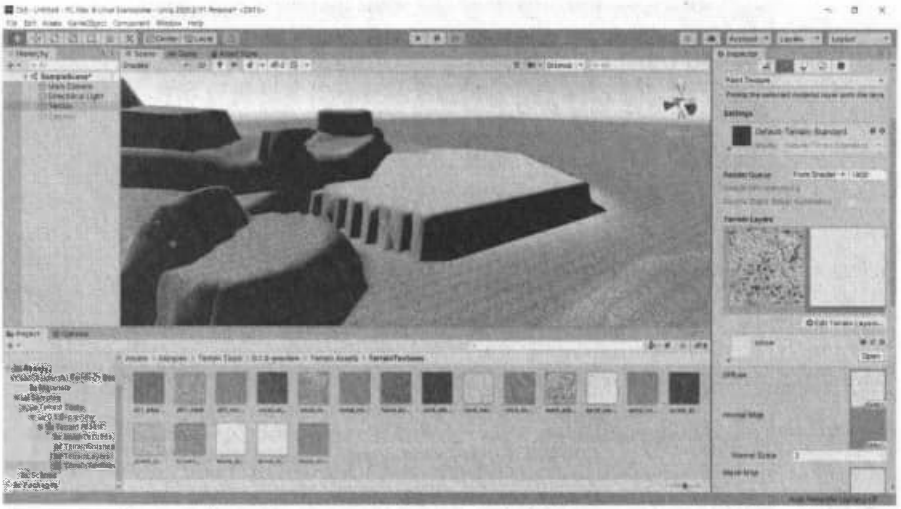


Рис. 5.38. Добавили снег

Инструмент **Set Height** позволяет редактировать высоты. Именно с его помощью можно создать возвышенности и впадины. Для создания возвышенности установите свойство **Height** выше 0, а для создания впадины – ниже (рис. 5.39).

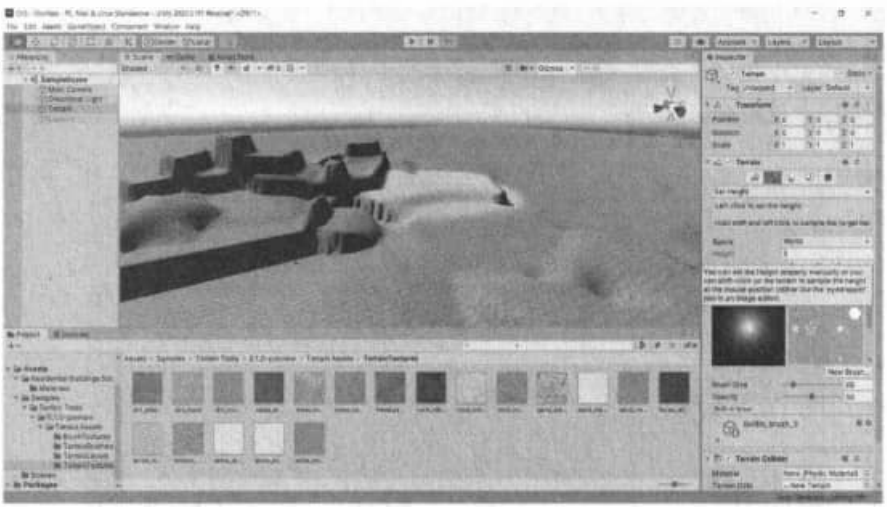


Рис. 5.39. Инструмент "Set Height" в действии

Если впадины и возвышенности будут слишком резкими, для их сглаживания предназначен инструмент **Smooth Height**.

Пики, изображенные на рис. 5.40 можно создать с помощью инструмента **Stamp Terrain**.

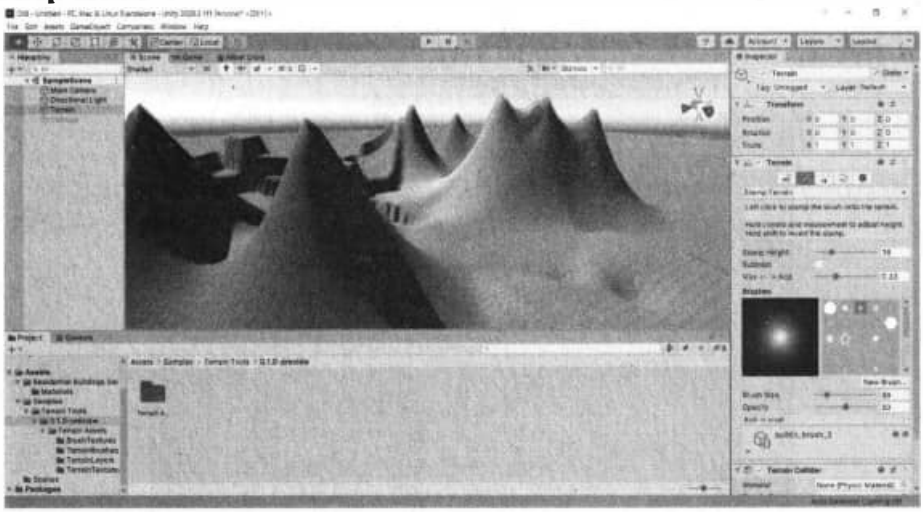


Рис. 5.40. Инструмент "Stamp Terrain"

5.6.2. Деревья

Импорт ассетов деревьев

Местность в Unity может быть дополнена деревьями. Островки с деревьями можно рисовать прямо на местности, аналогично рисованию карт высот и текстур, но деревья – полноценные 3D объекты, которые растут из поверхности. Unity использует оптимизации (например, превращает удаленные деревья в спрайты) для сохранения хорошей производительности рендеринга, так что вы можете иметь плотно засаженные леса с тысячами деревьев, и сохранять при этом приемлемую частоту кадров.

Кнопка с деревом на панели инструментов (в окне инспектора) активирует рисование деревьями. Изначально, у местности не будет доступных деревьев, но если вы нажмете на кнопку **Edit Trees** и в выпавшем меню выберите **Add Tree**, вы увидите окно, в котором можно выбрать ассет дерева из вашего проекта.

Чтобы деревья появились в вашем проекте, необходимо откуда-то импортировать модели деревьев. Пока вы учитесь, могу порекомендовать неплохой бесплатный пакет Reslistic Tree 9, доступный из магазина ассетов по ссылке:

https://assetstore.unity.com/packages/3d/vegetation/trees/realistic-tree-9-rainbow-tree-54622

Загрузите и импортируйте данный пакет. Затем выделите объект **Terrain**, нажмите кнопку с изображением дерева в окне инспектора, щелкните на кнопке **Edit Trees**, из появившегося меню выберите команду **Add Tree**. В окне **Add Tree** назначьте объект свойству **Tree Prefab** (рис. 5.41).

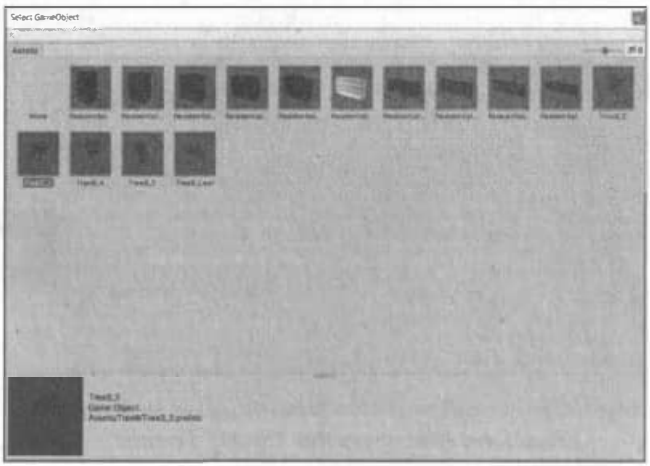


Рис. 5.41. Выбор и назначение объекта дерева

После этого «нарисуйте» деревья на сцене. Просто щелкайте в том месте, в котором должны расти деревья (рис. 5.42).

Рис. 5.42. Деревья «посажены»

После выбора дерева, вы можете рисовать на ландшафте аналогично тому, как вы рисуете текстуры или карты высот. Вы можете удалять любые деревья, зажав клавишу Shift во время рисования и удалить только выбранный в данный момент тип деревьев зажав клавишу CTRL. При рисовании деревьев доступна знакомая вам опция **Brush Size**, но свойство Opacity заменено свойством **Tree Density**, которое управляет средним количеством рисуемых деревьев в заданной единице площади. Также существует возможность управлять свойством **Color Variation** (вариация цвета) и вариациями высоты и ширины деревьев. Вариативные опции помогают создать впечатления неоднородности, натуральности леса, вместо искусственных плантаций одинаковых деревьев.

Имеется ползунок **Tree Height** для управления минимальной высотой и максимальной высотой дерева. Флажок **Random** (если включен) означает, что высота деревьев будет устанавливаться случайным образом. Если вы отключите **Random**, вы можете указать значение для высоты всего дерева. По умолчанию ширина дерева привязана к высоте, поэтому деревья всегда масштабируются равномерно. Однако вы можете отключить опцию **Lock Width to Height** и указать ширину отдельно.

Существует также элемент управления для случайного поворота дерева – **Random Tree Rotatic**. Он помогает создать впечатление случайного, естественно выглядящего леса, а не искусственной плантации идентичных деревьев.

Кнопка **Mass Place Trees** (массово разместить деревья) очень полезна для полного покрывания terrain'a деревьями без рисования по ландшафту. После массового размещения вы все еще можете использовать инструменты рисования для добавления или удаления деревьев, чтобы создать более плотные или редкие области.

Создание собственных деревьев

В Unity есть свой инструмент для создания деревьев (Tree creator), который вы можете использовать для создания новых ассетов деревьев (чуть позже мы его рассмотрим подробно), но вы также можете использовать и стандартные приложения для 3D моделирования для этой задачи. Меш дерева должен иметь менее 2000 треугольников (для повышения производительности) и центр вращения должен быть расположен прямо в основании дерева, там, где оно стыкуется с землей. Меш всегда должен иметь ровно два материала, один для ствола и веток, другой для листвы.

Деревья должны использовать шейдеры Nature/Soft Occlusion Leaves и Nature/Soft Occlusion Bark. Чтобы использовать эти шейдеры, нужно поместить дерево в специальную папку, содержащую в имени «Ambient-Occlusion». Когда вы поместите модель в эту папку и повторно импортируете ее, Unity просчитает мягкое рассеянное затенение специально созданным для деревьев способом. Шейдеры «Nature/Soft Occlusion» опираются на соглашение об именовании папок, и дерево не прорисуется корректно, если вы не будете ему следовать.

После сохранения ассета дерева из пакета для 3D моделирования, вам потребуется нажать на кнопку **Refresh** (отображается в инспекторе, когда выбран инструмент рисования деревьев) для того, чтобы увидеть обновленные деревья на вашей местности.

Коллайдер дерева

Вы можете добавить **Capsule Collider** к новому ассету дерева, создав его экземпляр в сцене, добавив компонент коллайдера (меню **Component, Physics, Capsule Collider**) и создав новый префаб для измененного объекта дерева. Когда вы добавляете дерево в terrain для рисования, убедитесь, что вы выбрали префаб дерева с коллайдером, а не оригинальный объект. Вам также следует включить флажок **Create Tree Colliders** в инспекторе компонента **Terrain Collider** на вашей местности, чтобы разрешить коллайдерам работать.

Рис. 5.43. Игровой объект WindZone: жаль иллюстрация не передает должного эффекта, но вы все увидите своими глазами, когда откроете проект ch5

Реакция деревьев на ветер

Если вы запустите игру и пройдетесь по созданной сцене, вы поймете, что деревья выглядят неестественно: деревья не реагируют на ветер. Да и вообще ветра нет как такового. Сейчас мы это исправим. Добавьте объект **GameObject, 3D Object, Wind Zone**.

Вы можете заметить, что ваши деревья движутся довольно бурно. Чтобы деревья не слишком трепетались на ветру, нужно снизить значение **Turbulence** до 0,1-0,3 и все будет выглядеть естественно.

5.7. Редактор деревьев

Для создания ассета дерева, выберите **GameObject, 3D Object, Tree**. Вы увидите новый ассет дерева, который был создан в окне **Project** и его экземпляр, созданный в текущей открытой сцене. Это новое дерево очень простое и содержит только ствол, так что давайте добавим ему немного листьев (рис. 5.44).

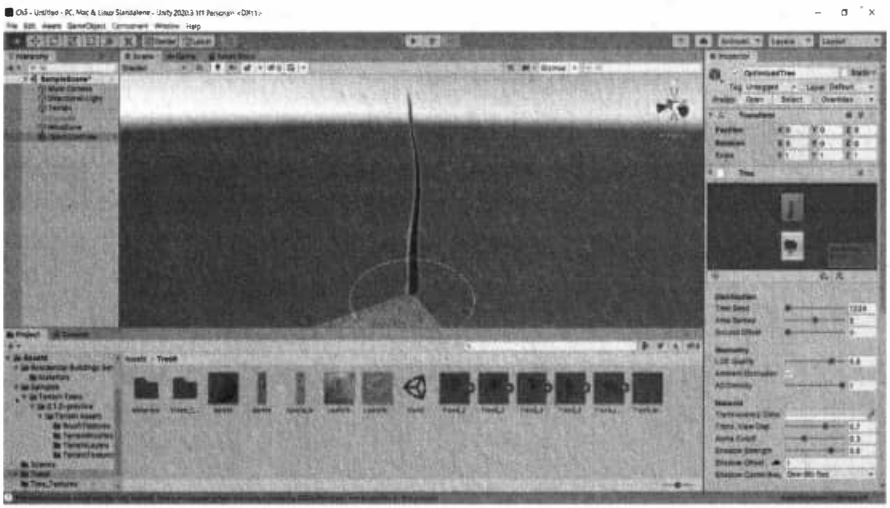


Рис. 5.44. Созданное дерево

Выберите дерево в иерархии для отображения инструмента **Tree Creator** в инспекторе. Этот инструмент предоставляет все необходимые инструменты для придания формы и коррекции фигуры ваших деревьев. Вы увидите иерархию дерева (**Tree**) с двумя узлами: узел **Tree Root** (корень дерева) и один узел **Branch Group** (группа ветвей), который мы будем называть стволом дерева.

5.7.1. Добавляем ветки

В иерархии дерева (**Tree**), выберите **Branch Group**, которая представляет собой ствол дерева. Нажмите на кнопку **Add Branch Group** (кнопка с изображением веток и +) и вы увидите, как появилась новая **Branch Group** (группа веток) соединившись с основным узлом. Теперь вы можете поиграться с настройками ниже, чтобы посмотреть как поведут себя присоединенные к стволу ветки. Например, параметр **Frequency** задает частоту (не точное количество!) веток, **Distribution** – как будут ветки расположены. Группа параметров **Shape** определяет форму веток. Параметр **Length** задает длину веток, **Radius** – толщину.

Теперь, после создания присоединенных к стволу ветвей, мы можем добавить маленькие веточки к созданным ветвям с помощью присоединения к ним других узлов **Branch Group**. Выберите вторичную **Branch Group** и снова нажмите на кнопку **Add Branch Group**. Настройте значения этой группы для создания дополнительных веток, присоединенных ко вторичным веткам.

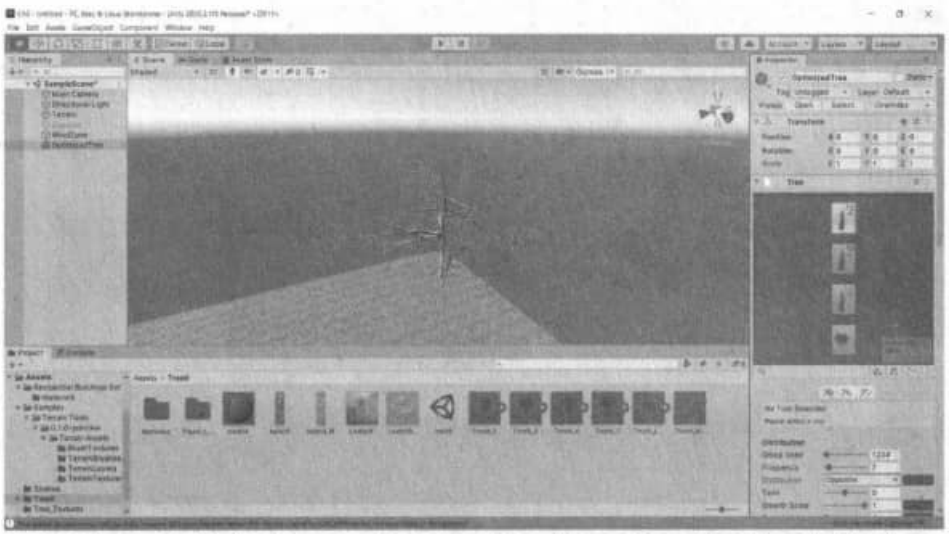


Рис. 5.45. Дерево с ветками

5.7.2. Добавляем листья

Теперь структура веток дерева готова. Наша игра не происходит в зимнее время, так что нам стоит добавить немного листьев (**Leaves**) к различным веткам.

Выберите узел с вашей вторичной группой веток (**Branch Group**) и нажмите на кнопку **Add Leaf Group** (кнопка с изображением листьев и +). Если вы хотите пуститься во все тяжкие, вы также можете добавить другую группу листьев к самым тонким ветвям дерева.

Теперь у нас есть листья, которые прорисовываются как непрозрачные плоскости (рис. 5.46). Это происходит для того, чтобы мы могли настроить значения листьев (размер, положение, вращение и т.д.) перед тем, как добавлять к ним материал. Изменяйте значения листьев (**Leaf Group**) до тех пор, пока вы не подберете устраивающие вас настройки.

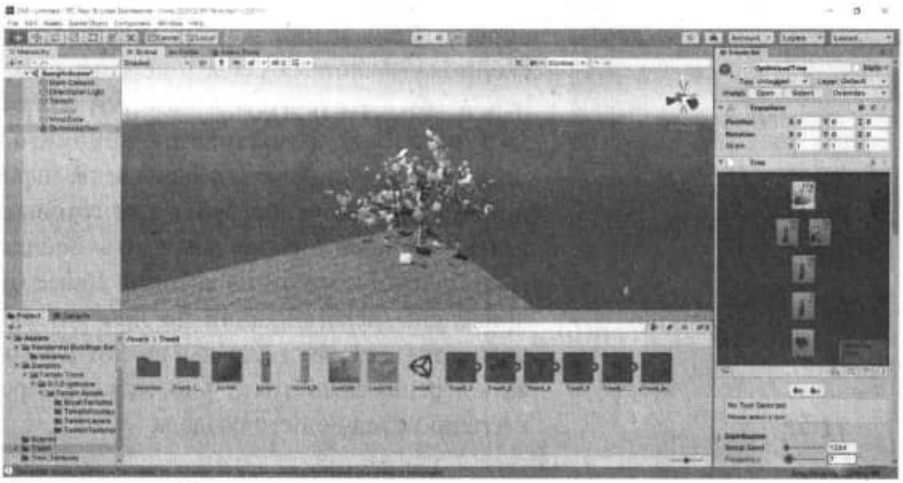


Рис. 5.46. Дерево с листьями

Чтобы наше дерево выглядело реалистично, нам нужно применить материалы (**Materials**) к веткам и листьям. Создайте новый материал в вашем проекте с помощью **Assets, Create, Material**. Переименуйте его в «Tree Bark» (кора дерева) и выберите **Nature, Tree Creator Bark** из выпадающего списка с шейдерами. Теперь вы можете назначить текстуры на свойства материала коры **Base, Normalmap и Glass**. Текстуры можно или разработать самостоятельно или позаимствовать из пакета, который мы недавно импортировали для «посадки» деревьев.

Теперь мы таким же образом создадим материал для листьев. Создайте новый материал и поменяйте его шейдер на **Nature, Tree Creator Leaves**. Присвойте текстурным слотам текстуры листьев из пакета **Tree9**.

Когда оба материала готовы, мы применим их к различным узлам групп дерева. Выберите ваше дерево и кликните по любому узлу ветви или листьев,

затем откройте секцию **Geometry**. Вы увидите слот для применения материала выбранному вами типу слота. Примените соответствующий созданный вами материал и посмотрите на результаты.

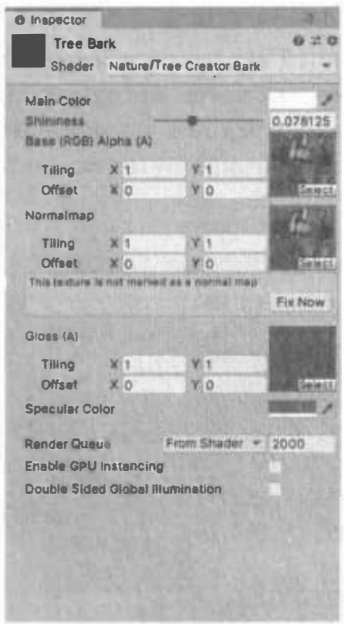


Рис. 5.47. Создание нового материала

Для завершения дерева, примените ваши материалы ко всем Branch и Leaf Group узлам в дереве. Теперь ваше дерево можно использовать в игре. Пусть с первого раза получилось и не самое красивое дерево, но помните, что первый блин всегда комом. Тренируйтесь, и скоро вы начнете создавать деревья не хуже, чем в том пакете Tree 9, который мы ранее загрузили. Если же вы хотите сэкономить время и сконцентрироваться больше на игровом процессе, тогда используйте уже готовые деревья, которые можно загрузить бесплатно (или платно) в магазине ассетов. Далее будут приведены рекомендации по использованию редактора дерева. Если вы не хотите создавать деревья самостоятельно, можете перейти к чтению следующего раздела.

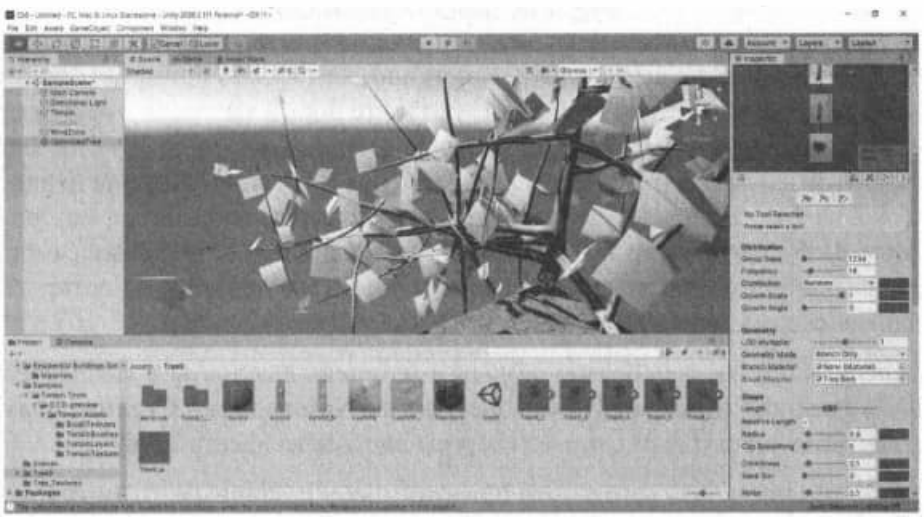


Рис. 5.48. Применен материал коры дерева

5.7.3. Эффективное использование редактора дерева

Итак, после создания объекта **Tree** (точнее в инспекторе он называется OptimizedTree) у вас будет только корень и ствол дерева. Никаких веток или листьев. В инспекторе объектов посредством редактирования свойств вы можете создать дерево своей мечты.

В верхней части инспектора компонента **Tree** находится редактор структуры дерева, в котором указывается простое расположение ветвей и листьев (рис. 5.49).

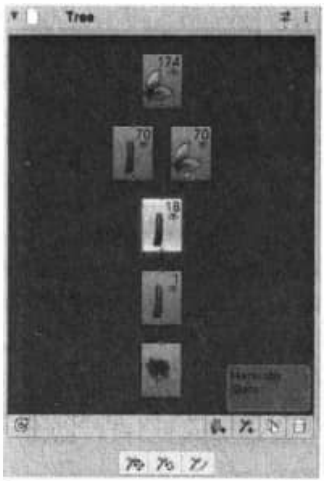


Рис. 5.49. Редактор структуры дерева

Вам нужно понимать концепцию уровней дерева при работе с редактором. Ствол имеет ветки, которые, в свою очередь, имеют дочерние ветки; этот процесс ветвления продолжается до тех пор, пока не создастся завершающая небольшая веточка. Ствол считается первым уровнем дерева, а ветки, растущие из ствола, представляют собой второй уровень. Любые ветки, растущие из веток второго уровня, формируют третий уровень и т.д.

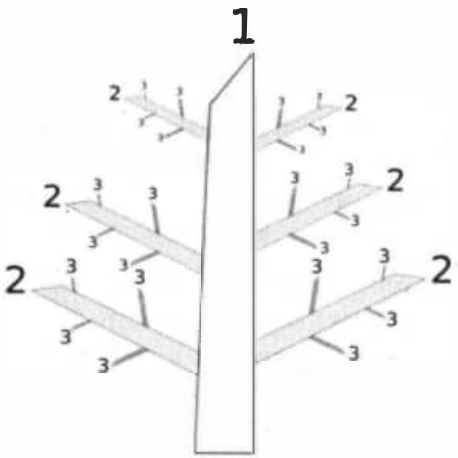


Рис. 5.50. Структура дерева

Посмотрите на рис. 5.49. Иконки соединены линиями для отображения уровней ветвления дерева. Иконка в самом низу (с изображением дерева) обозначает «корень» дерева. После выбора этой иконки, ниже, в панели инспектора, будут показаны настройки для всего дерева. Этот корень продлевают первый и второй уровни ветвления. Иконки отображают некоторую часть информации, а именно количество элементов данного типа.

Главное изображение показывает тип элемента. Число в правом верхнем углу – это количество веток, существующих на этом уровне дерева, соответствующее свойству инспектора **Frequency**. После выбора такой иконки, изменение значения **Frequency** изменит количество ветвей на этом уровне. Изображение глаза сразу под цифрой означает видимость ветвей в окне Scene; вы можете переключить видимость нажатием на эту иконку глаза.

Расположение групп ветвей может быть отредактировано с помощью элементов управления, расположенных в правой нижней части редактора. Слева направо:

- Первый инструмент добавляет к дереву группы листьев (Leaf Group). Листья располагаются на уровнях так же, как и ветки, но в отличие от веток, листья не могут подразделяться на большее количество уровней
- Второй инструмент добавляет новую группу ветвей (Branch Group) на текущем уровне (т.е., он создает новый дочерний элемент для выбранной иконки группы веток)
- Третий инструмент дублирует выбранную группу
- Четвертый удаляет выбранную группу из дерева. Можно иметь несколько групп на каждом уровне дерева

Щелкните на объекте OptimizedTree дважды. Вы перейдете в режим редактирования структуры дерева (рис. 5.51). Затем щелкните, например, на стволе дерева, а затем выберите один из трех инструментов, находящихся под структурой дерева. Вы увидите, как на стволе дерева появятся квадратики.

Квадраты являются контрольными точками вдоль всей ветви (т.е., центр линии ветви проходит через все квадраты, но также и плавно искривляется между ними). Вы можете кликнуть и потащить любой из квадратов для сдвига контрольных точек и изменения формы ветви. Это достигается с помощью первого инструмента.

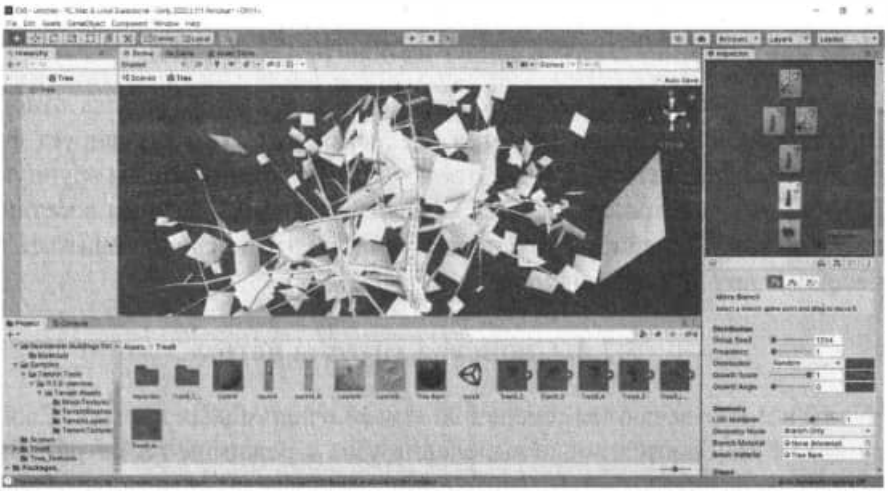


Рис. 5.51. Режим редактирования структуры дерева

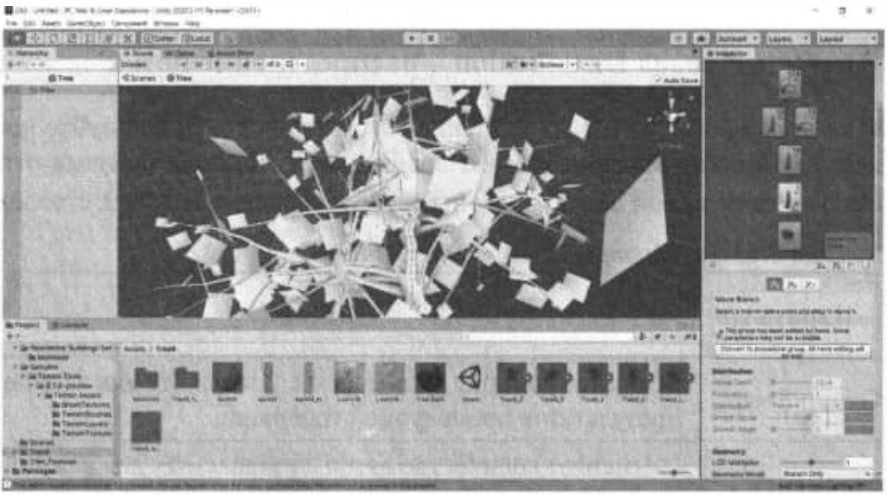


Рис. 5.52. Ствол дерева искривлен

Сдвиг контрольных точек – на самом деле одна из трех возможных опций на панели инструментов ручного редактирования.

Второй инструмент позволяет сгибать ветвь, вращая ее в указанной контрольной точке. Третий инструмент позволяет мышкой от руки рисовать ветвь, начинающуюся из выбранной контрольной точки. Ветвление все еще будет контролироваться из панели структуры дерева, только формы ветвей могут быть нарисованы. Если в структуре выбрана группа листьев, тогда

отобразится соответствующая панель инструментов с опциями для перемещения или вращения листьев вокруг их родительской ветки.

Учтите, что некоторые свойства в инспекторе редактора дерева относятся к процедурной генерации деревьев (т.е., компьютер сам генерирует форму случайным образом) и они будет отключены после того, как вы вручную попробуете отредактировать дерево. Существует кнопка, которая восстановит дерево в статус генерируемого процедурно, но это также отменит все ваши внесенные вручную изменения.

5.7.4. Свойства группы веток

Группы веток отвечают за генерацию ветвей и пальмовых листьев. Свойства группы показываются после выделения узла с режимом геометрии Branch Only (только ветка), Fronds Only (только пальмовые листья) или Branch + Fronds (ветка + пальмовые листья). Свойства группы веток разбиты на несколько разделов.

Раздел Distribution – распределение веток

Изменяет количество и расположение веток в группе. Используйте кривые для точной настройки положения, вращения и масштаба. Кривые относительны родительской ветки или зоны распространения в случае ствола.

Таблица 5.6. Параметры группы Distribution

Параметр	Описание
Group Seed	Образец данной группы веток. Изменяйте для изменения результатов процедурной генерации
Frequency	Изменяет количество веток, создаваемых для каждой родительской ветки
Distribution	Тип распределения веток вдоль своих родителей
Twirl	Закручивание вокруг родительской ветки
Whorled Step	Определяет количество узлов в каждом мутовчатом шаге при использовании типа распределения Whorled (мутовка). У настоящих растений это обычно число Фибоначчи

Growth Scale	Определяет масштаб узлов вдоль родительского узла. Используйте кривую для тонкой настройки и слайдер для изменения степени проявления эффекта
Growth Angle	Определяет начальный угол роста относительно родителя. Используйте кривую для тонкой настройки и слайдер для изменения степени проявления эффекта

Раздел Geometry – геометрия веток

Выберите тип генерируемой геометрии и применяемые материалы для этой группы веток. LOD Multiplier позволяет вам настроить качество этой группы относительно LOD Quality дерева.

Таблица 5.7. Параметры группы Geometry

Параметр	Описание
LOD Multiplier	Настраивает качество этой группы относительно LOD Quality дерева, позволяя сделать группу более качественной или менее качественной, чем все остальное дерево
Geometry Mode	Тип геометрии для этой группы веток (Branch Group): Branch Only (только ветки), Branch + Fronds (ветки + пальмовые листья), Fronds Only (только пальмовые листья)
Branch Material	Основной материал для веток
Break Material	Материал на торцах сломанных ветвей
Frond Material	Материал для пальмовых листьев

Раздел Shape. Форма веток

Настраивает форму и рост ветвей. Используйте кривые для точно настройки формы, все кривые относительны самому стволу.

Таблица 5.8. Раздел Shape

Параметр	Описание
Length	Настраивает длину ветвей
Relative Length	Определят, будет ли длина ветви влиять на ее радиус
Radius	Настраивает радиус ветвей, используйте кривую для точной настройки радиуса на протяжении всей длины ветвей
Cap Smoothing	Определяет округлость торцов / кончиков ветвей. Полезно для кактусов
Crinkliness	Настраивает степень извилистости / изломанности ветвей, используйте кривую для точной настройки
Seek Sun	Используйте кривую для настройки изгиба ветвей вверх/вниз и используйте слайдер для изменения масштаба эффекта
Noise	Общий коэффициент поверхностного шума, используйте кривую для тонкой настройки
Noise Scale U	Масштаб шума вокруг ветви, чем ниже значения, тем более волнистым будет вид, чем выше значения – тем более случайным будет вид
Noise Scale V	Масштаб шума вдоль ветви, чем ниже значения, тем более волнистым будет вид, чем выше значения – тем более случайным будет вид
Break Chance	Шанс облома ветви, т.е. 0 = ветви не будут сломаны, 0.5 = половина ветвей сломаны, 1.0 = все ветви сломаны
Break Location	Это значение указывает место облома ветвей. Относительно всей длины ветви
Weld Length	Определяет то, как далеко вверх по росту ветви начинается шов срастания

Spread Top	Коэффициент распространения срастания на верхней стороне ветви, относительно ее родительской ветви. Ноль означает отсутствие распространения
Spread Bottom	Коэффициент распространения срастания на нижней стороне ветви, относительно ее родительской ветви. Ноль означает отсутствие распространения

Раздел Wind. Параметры ветра

Настраивает параметры, используемые для анимации этой группы ветвей. Зоны ветра активны только в режиме воспроизведения (Play Mode):

- **Main Wind** – основной эффект ветра. Создает мягкое движение покачивания и обычно единственный параметр, необходимый первоначальным ветвям
- **Edge Turbulence** – турбулентность вдоль кромки пальмовых листьев. Полезно для папоротников, пальм и т.д.

5.7.5. Параметры группы листьев

Группы листьев генерируют геометрию листьев, используются или примитивы или созданные пользователем меши. Параметров не так много, поэтому все они (со всех разделов) объединены в табл. 5.9.

Таблица 5.9. Параметры группы листьев

Параметр	Описание
Group Seed	Образец данной группы листьев. Изменяйте для изменения результатов процедурной генерации
Frequency	Изменяет количество листьев, создаваемых для каждой родительской ветки
Distribution	Тип распределения листьев вдоль своих родителей
Twirl	Закручивание вокруг родительской ветки

Whorled Step	Определяет количество узлов в каждом мутовчатом шаге при использовании типа распределения Whorled (мутовка). У настоящих растений это обычно число Фибоначчи
Growth Scale	Определяет масштаб узлов вдоль родительского узла. Используйте кривую для тонкой настройки и слайдер для изменения степени проявления эффекта
Growth Angle	Определяет начальный угол роста относительно родителя. Используйте кривую для тонкой настройки и слайдер для изменения степени проявления эффекта
Geometry Mode	Тип создаваемой геометрии. Вы можете использовать свой меш, выбрав опцию Mesh, идеальную для цветов, фруктов и т.д.
Material	Материал, используемый для листьев
Mesh	Меш, используемый для листьев
ize	Настраивает размер листьев, используется диапазон для определения минимального и максимального размера
Perpendicular Align	Настраивает перпендикулярность листьев к родительской ветке
Horizontal Align	Настраивает горизонтальное выравнивание листьев
Main Wind	Основной эффект ветра. Обычно стоит использовать небольшое значение, чтобы избежать отставания листьев от родительской ветки
Main Turbulence	Второстепенный эффект турбулентности. Для листьев следует использовать небольшое значение

Edge Turbulence	Определяет количество турбулентности ветра вдоль краев листьев

5.7.6. Зона ветров

Наши деревья (как уже готовые, так и те, которые создавали мы сами) умеют реагировать на ветер. Однако зону ветров еще нужно создать. С помощью объекта GameObject, 3D Object, Wind Zone вы сможете контролировать силу ветра и его направление.

Создайте объект WindZone. Изменим его параметры.

Параметр Mode задает режим зоны ветров:

- **Spherical** – зона ветров действует только в рамках указанного радиуса и ветер дует из центра к поверхности сферы
- **Directional** – зона ветров действует на всю сцену в определенном направлении

Параметр **Radius** задает радиус сферической зоны ветра, разумеется, он доступен только если используется сферический режим зоны ветров. Параметр **Main** задает основную силу ветра, а вот параметр **Turbulence** – определяет силу турбулентности, что приводит к резкому изменению давления ветра.

Параметр **Pulse Magnitude** определяет то, как сильно меняется ветер со временем, а параметр **Pulse Frequency** определяет частоту изменений ветра.

Пользоваться зоной ветров очень просто. Поместите объект WindZone на сцену и настройте его параметры. Если зона ветра сферическая (Spherical), тогда вам следует разместить ее так, чтобы деревья, которые вы хотите обдувать с ее помощью, были внутри радиуса сферы. Если зона ветра направленная (*directional*), тогда неважно, в каком месте сцены вы ее разместите.

Есть несколько советов относительно использования данного объекта:

- Если нужен эффект плавного меняющегося ветра:
  - » Создайте направленную зону ветра
  - » Установите Wind Main в 1.0 или меньшее значение, в зависимости от того, насколько мощным должен быть ветер
  - » Установите Turbulence в 0.1
  - » Установите Pulse Magnitude в 1.0 или большее значение

  - » Установите Pulse Frequency в 0.25
- Эффект от пролетающего вертолета:
  - » Создайте сферическую зону ветра
  - » Установите Radius в значение, соответствующее размеру вашего вертолета
  - » Установите Wind Main в 3.0
  - » Установите Turbulence в 5.0
  - » Установите Pulse Magnitude в 0.1
  - » Установите Pulse Frequency в 1.0
  - » Добавьте зону ветра на GameObject, представляющий ваш вертолет
- Создаем эффект взрыва:
  - » Проделайте всё то же самое, что и для вертолёта, только после взрыва надо быстро и плавно обнулить Wind Main и Turbulence, чтобы эффект пропал

5.8. Системы частиц

В трехмерной игре большинство контента (персонажи, окружение, объекты) представлены в виде мешей, в то время как в 2D игре для этого используются спрайты. Меши и спрайты идеально подходят для отображения целых объектов с четко определённой формой. Но, как мы все видели, в играх встречаются и другие элементы, которые по своей природе не имеют четкой формы и изменяются в реальном времени, потому их трудно отобразить с помощью мешей и спрайтов. Для эффектов в виде текущих жидкостей, дыма, облаков, пламени и магических заклинаний следует применять другой подход, известный как системы частиц.

5.8.1. Введение в системы частиц

Частицы – это небольшие простые изображения или сетки, которые отображаются и перемещаются в большом количестве системой частиц. Каждая частица представляет собой небольшую часть жидкого или аморфного объекта, и влияние всех частиц вместе создает впечатление целостного объекта. Используя облако дыма в качестве примера, каждая частица будет иметь небольшую текстуру дыма, напоминающую крошечное облако само по себе.

Когда многие из этих мини-облаков скомпонованы в области сцены, общий эффект представляет собой большое облако, заполняющее объем.

Каждая частица имеет определенное время жизни (обычно несколько секунд), в течение которого частица может претерпеть различные изменения. Частица начинает свое существование, когда она создана, или испущена своей системой частиц. Система испускает частицы в случайных точках в пространстве, ограниченном регионом в форме сферы, полусферы, конусом, прямоугольным параллелепипедом (для простоты такую форму будем называть просто "коробкой") или мешем произвольной формы. Частица отображается до тех пор, пока не закончится время ее жизни, затем она удаляется из системы. Частота испускания частиц системы жестко определяет количество испускаемых частиц в секунду, хотя точное время испускания содержит небольшой фактор случайности. Частота испускания в совокупности со средним временем жизни частицы определяют количество "стабильных" частиц (то есть, тех, чье испускание и уничтожение происходят с одинаковой частотой) и время, необходимое системе для достижения такого состояния.

Настройки испускания и времени жизни влияют на общее поведение системы, однако отдельные частицы тоже могут изменяться со временем. Каждая частица имеет вектор скорости, определяющий направление и расстояние, которое проходит частица за один кадр. Скорость может быть изменена с помощью сил и гравитации, применяемых самой системой или когда частицы сдуваются в зонах ветра над земной поверхностью (Terrain). Цвет, размер и вращение каждой частицы также могут изменяться в течение времени жизни или пропорционально ее текущей скорости движения. Цвет включает альфа-канал (для прозрачности), так что частица может плавно исчезать и появляться вместо резкого изменения видимости в таких случаях.

При использовании в совокупности, динамика частиц определенно может применяться для симуляции многих типов текучих эффектов. Например, водопад может быть симулирован с помощью узкой области испускания, из которой падают частицы воды под действием гравитации, ускоряясь по ходу движения. Дым от огня обычно поднимается вверх, расширяется и в конце рассеивается, так что система должна применять восходящую силу к частицам дыма, увеличивать их в размере и увеличивать прозрачность с течением их времени жизни.

5.8.2. Использование систем частиц в Unity

Создать систему частиц можно двумя способами – или добавить в виде самостоятельного объекта (**GameObject, Effect, Particle System**) или же путем

добавления компонента (**Components, Effects, Particle System**) к уже существующему объекту (рис. 5.53).

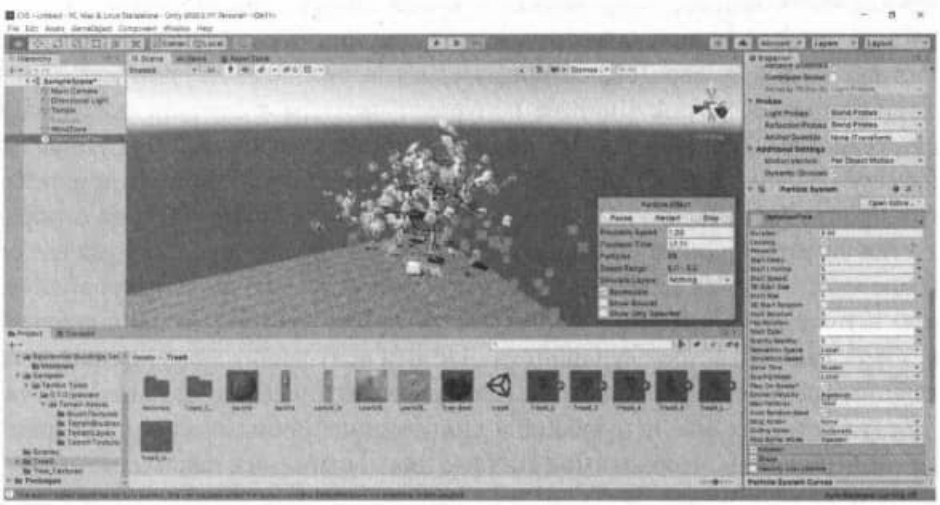


Рис. 5.53. Вот что будет, если добавить систему частиц к объекту дерева

Компонент довольно сложен, поэтому инспектор разделен на несколько разборных подразделов или модулей, каждый из которых содержит группу связанных свойств. Кроме того, вы можете редактировать одну или несколько систем одновременно, используя отдельное окно редактора, доступное через кнопку **Open Editor** в Инспекторе (см. рис. 5.53).

При выборе объекта с системой частиц, в окне **Scene** появится небольшая панель **Particle Effect** с простыми элементами управления для удобной визуализации изменений, вносимых в настройки системы, что также показано на рис. 5.53.

Многие числовые свойства частиц или даже всей системы могут изменяться с течением времени. Unity предоставляет несколько различных способов указания, как будет происходить изменение:

- **Constant** – значение свойства не меняется со временем
- **Curve** – значение меняется по кривой / графу
- **Random between two constants** – два постоянных значения определяют верхнюю и нижнюю границы для итогового значения, которое выбирается случайно из всех значений, попадающих в эти рамки

- **Random between two curves** – две кривые определяют верхнюю и нижнюю границы диапазона значений в определённый момент времени; текущее значение случайным образом выбирается из этого диапазона

Для цветовых свойств, таких как **Color over Lifetime**, есть два отдельных параметра:

- **Gradient** – значение цвета выбирается из градиента
- **Random between two gradients** – два градиента определяют верхнюю и нижнюю границы значения цвета в определённый момент времени; текущее значение является случайным средневзвешенным из цветов на границах

5.8.3. Использование системы частиц для создания взрыва

Визуализация взрыва – типичный пример использования системы частиц в Unity. Динамика взрыва, возможно, немного сложнее, чем кажется на первый взгляд. По своей сути взрыв – это просто выброс частиц наружу, но есть несколько простых модификаций, которые можно применить, чтобы он выглядел намного реалистичнее.

Простой взрыв порождает шар пламени, который быстро расширяется во всех направлениях. Первоначальный взрыв имеет много энергии и поэтому очень горячий (то есть яркий) и движется очень быстро. Эта энергия быстро рассеивается, что приводит к замедлению распространения пламени и его охлаждению (т.е. становится менее ярким). Наконец, когда все топливо сгорело, пламя угаснет и вскоре полностью исчезнет.

Взрывная частица обычно имеет короткий срок службы, и вы можете изменить несколько различных свойств в течение этого срока для моделирования эффекта. Частица начнет двигаться очень быстро, но затем ее скорость должна сильно уменьшиться, поскольку она удаляется от центра взрыва. Кроме того, цвет должен начинаться ярким, а затем темнеть и в конечном итоге исчезать до прозрачности. Наконец, уменьшение размера частиц в течение всего срока службы даст эффект рассеивания пламени по мере использования топлива.

Все это хорошо, но давайте рассмотрим практическую реализацию. Создайте систему частиц как отдельный объект. Перейдите к модулю **Shape** и установите форму эмиттера для маленькой сферы, скажем, около 0,5 единиц в радиусе.

Далее нам понадобится материал взрывов/огня. Вы можете раздобыть свой материал, а можете использовать бесплатный пакет FX Fire Free

(*https://assetstore.unity.com/packages/vfx/particles/fire-explosions/fx-fire-free-21587*) из магазина ассетов.

Вы можете установить этот материал для системы с помощью модуля **Renderer**. Разверните модуль Renderer и назначьте материал. Также вы должны также отключить **Cast Shadows** и **Receive Shadows**, так как пламя взрыва должны выдавать свет, а не получить его.

На этом этапе система выглядит как множество маленьких огненных шаров, выбрасываемых из центральной точки (рис. 5.54).

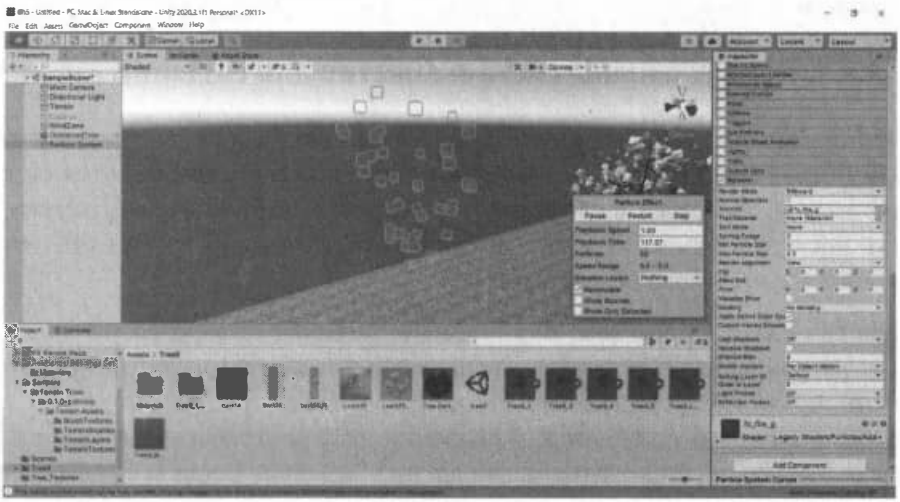


Рис. 5.54. Система частиц в процессе создания

Взрыв должен, конечно, создать взрыв с большим количеством частиц одновременно. В модуле **Emission** установите значение **Rate** на ноль и добавьте один **Burst** с нулевым временем. Теперь наша система частиц напоминает взрыв.

Количество частиц во взрыве будет зависеть от размера и интенсивности проектируемого взрыва, но хорошая отправная точка составляет около пятидесяти частиц (параметр Count в области **Bursts**).

В модуле Particle System, установите значение 2 для параметров **Duration** и **Start Lifetime**.

В настоящее время взрыв принимает форму взрыва, но выглядит так, как будто он происходит в космосе. Частицы выбрасываются и перемещаются на

большие расстояния с постоянной скоростью, прежде чем исчезнуть. Если ваша игра происходит в космосе, то это может быть именно тот эффект, который вы хотите. Однако взрыв, происходящий в атмосфере, будет замедлен и ослаблен окружающим воздухом. Включите модуль **Limit Velocity Over Lifetime** и установите **Speed** около 3,0, а параметр **Dampen** – 0,4, и вы увидите, что взрыв теряет небольшую силу по мере его развития.

Рис. 5.55. Уже что-то похоже на взрыв

Сэкономить время можно с помощью бесплатного пакета Particle Dissolve Shaper Package:

https://assetstore.unity.com/packages/vfx/particles/fire-explosions/particle-dissolve-shader-package-33631.

В его состав входят уже готовые ассеты взрывов и все, что нужно – перетащить на сцену нужный взрыв (рис. 5.56) и использовать его в своей игре.

Во время тестирования полезно включить свойство **Looping**, чтобы вы могли неоднократно видеть взрыв, но в законченной игре вы должны отключить его, чтобы взрыв произошел только один раз. Когда взрыв предназначен для объекта, который потенциально может взорваться (скажем, топливный бак), вы можете добавить компонент системы частиц к объекту с отключенным свойством **Play On Awake**. Затем вы можете при необходимости запустить взрыв из сценария так:

```
void Explode() {
        var exp = GetComponent<ParticleSystem>();
```

```
        exp.Play();
        Destroy(gameObject, exp.duration);
}
```

Рис. 5.56. Готовый взрыв

Взрывы также могут происходить в точках удара. Если взрыв происходит от объекта (например, от гранаты), вы можете вызвать функцию Explode(), описанную выше, через некоторое время или когда она вступает в контакт с целью.

```
// Порождаем взрыв с задержкой
public float fuseTime; // задержка

void Start() {
    Invoke("Explode", fuseTime);
}


// Взрыв
void OnCollisionEnter(Collision coll) {
    Explode();
}
```

5.9. Отражающие зонды

5.9.1. Зачем нужны отражающие зонды

Фильмы и анимация CG обычно имеют очень реалистичные отражения, которые важны для придания ощущения «связности» между объектами сцены. Однако точность этих отражений сопряжена с высокой стоимостью процессорного времени, и, хотя это не проблема для фильмов, это серьезно ограничивает использование объектов в играх реального времени.

Традиционно игры использовали технику, называемую отображением отражений, для имитации отражений от объектов, сохраняя при этом производительность на приемлемом уровне. Этот метод предполагает, что все отражающие объекты в сцене могут «видеть» (и, следовательно, отражать) одно и то же окружение. Это хорошо работает для космического корабля, если он находится в открытом космосе, но неубедительно, если персонаж попадает в другое окружение: это выглядит странно, если обычный автомобиль заедет в туннель, но небо все еще заметно отражается в его окнах.

Улучшить отражение объектов можно с помощью отражающих зондов, которые позволяют визуализировать визуальную среду в стратегических точках сцены. Как правило, вы должны размещать их в каждой точке, где внешний вид отражающего объекта будет заметно меняться (например, туннели, области возле зданий и места, где меняется цвет грунта). Когда отражающий объект проходит рядом с зондом, отражение, отобранное зондом, может использоваться для карты отражения объекта. Кроме того, когда несколько зондов находятся рядом, Unity может интерполировать между ними, чтобы учесть постепенные изменения в отражениях. Таким образом, использование зондов отражения может создать достаточно убедительные отражения с приемлемыми накладными расходами на обработку.

Визуальная среда для точки на сцене может быть представлена кубической картой (см. рис. 5.57). Концептуально это похоже на коробку с плоскими изображениями вида с шестью направлениями (вверх, вниз, влево, вправо, вперед и назад), нарисованных на его внутренних поверхностях.

Чтобы объект отображал отражения, его шейдер должен иметь доступ к изображениям, представляющим кубическую карту. Каждая точка поверхности объекта может «видеть» небольшую область кубической карты в направлении поверхности (то есть в направлении вектора нормали поверхности). На этом этапе шейдер использует цвет кубической карты при расчете того, каким должен быть цвет поверхности объекта; зеркальный материал может

точно отражать цвет, в то время как блестящий автомобиль может выцветать и слегка его подкрашивать.

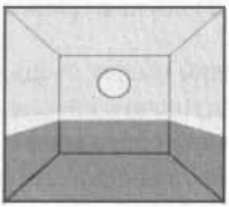


Рис. 5.57. Кубическая карта

Как упомянуто выше, традиционное отображение отражения использует только одну кубическую карту для представления окружения для всей сцены. Кубическая карта может быть нарисована художником, или ее можно получить, сделав шесть «снимков» из точки сцены, по одному снимку для каждой грани куба. Зонды отражения улучшают это, позволяя вам установить много предопределенных точек в сцене, где могут быть сделаны снимки кубической карты. Поэтому вы можете записывать окружающий вид в любой точке сцены, где отражения заметно различаются.

5.9.2. Типы зондов

Тип зонда выбирается в инспекторе объектов. Существует три типа зондов:

- **Backed** – хранит карту отражений, сгенерированную в редакторе
- **Realtime** – создают кубическую карту во время выполнения. Это означает, что отражения не ограничиваются статическими объектами и могут обновляться в режиме реального времени для отображения изменений в сцене
- **Custom** – пользовательские зонды

5.9.3. Использование зондов

Компонент **Reflection Probe** может быть добавлен к любому объекту в сцене, но обычно зонды добавляются в пустой GameObject. Пример рабочего процесса:

1. Создайте новый пустой **GameObject** (команда меню **GameObject, Create Empty**).

2. Добавьте в него компонент **Reflection Probe** (команда меню **Component, Rendering, Reflection Probe**). В качестве альтернативы, если у вас уже есть зонд в сцене, вам, вероятно, будет проще дублировать его (команда меню **Edit, Duplcate**).
3. Поместите новый зонд в нужное место и установите его исходную точку (свойство **Origin**) и размер зоны воздействия.
4. При желании установите другие свойства на зонд, чтобы настроить его поведение.
5. Продолжайте добавлять зонды, пока не будут назначены все необходимые места.

Чтобы увидеть отражения, вам также понадобится как минимум один отражающий объект на сцене. Простой тестовый объект может быть создан следующим образом:

1. Добавьте к сцене примитивный объект, например, сферу (**GameObject, 3D Object, Sphere**).
2. Создайте новый материал (**Assets, Create, Material**) и не изменяйте стандартный шейдер.
3. Сделайте материал отражающим, установив свойства **Metallic** и **Smoothness** на 1.0.
4. Перетащите только что созданный материал на объект сферы, чтобы назначить его.
5. Сфера теперь может отображать отражения, полученные от зондов. Простого расположения с одним зондом достаточно, чтобы увидеть основной эффект отражений.

Наконец, зонды должны быть подготовлены до того, как отражения станут видимыми. Если в окне **Lighting** включена опция *Auto* (это настройка по умолчанию), отражения будут обновляться по мере размещения или изменения объектов в сцене, хотя реакция не мгновенная. Если вы отключите автоматику, то вы должны нажать кнопку **Bake** в инспекторе **Reflection Probe**, чтобы обновить зонды. Основная причина отключения автоматики состоит в том, что процесс подготовки отражений может занять довольно много времени для сложной сцены со многими зондами.

5.10. Оптимизация производительности графики

Хорошая производительность важна для многих игр. В этом разделе мы рассмотрим основные концепции поддержания производительности на уровне в вашей игре.

Графика нагружает в первую очередь две системы компьютера: графический процессор (GPU) и центральный процессор (CPU). Чем мощнее видеокарта и основной процессор, тем быстрее будет выполняться игра.

Если у вашей игры есть проблемы с производительностью, нужно понять, где узкое место, то есть где возникает проблема, поскольку стратегия оптимизации для GPU и CPU имеет существенные различия.

Типичные узкие места и их проверка:

- GPU часто ограничен филлрейтом (fillrate[5]) или пропускной способностью памяти
- Запуск игры с более низким разрешением экрана увеличивает производительность? Тогда, скорее всего, вы ограничены филлрейтом GPU
- CPU часто ограничен количеством вещей, которые должны быть отрисованы, также известно, как «draw calls»
- Проверьте показатель «draw calls» в окне Rendering Statistics; если он составляет больше нескольких тысяч (для PC) или нескольких сотен (для мобильных устройств), то вам может потребоваться оптимизация количества объектов. Другими словами, число объектов на сцене придется сократить

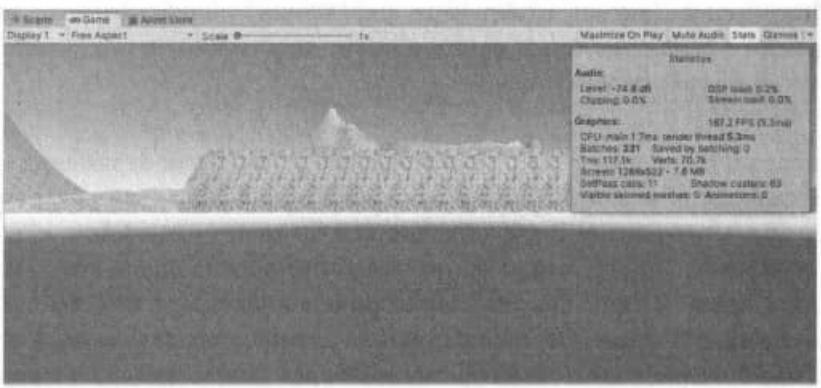


Рис. 5.58. Окно "Rendering Statistics"

5 Fill Rate (филлрейт) — скорость заполнения пикселами, одна из важнейших характеристик видеокарты. Обозначает число пикселей, для которых видеокарта может просчитать освещение и все необходимые операции (выполнить фрагментные шейдеры, Z-тест, операции со стенсилом, альфа-тест, антиалиасинг и пр.)

Менее распространенные узкие места:

- GPU обрабатывает слишком много вершин. Какое количество вершин является нормальным, определяется GPU и набором вертексных шейдеров. Можно посоветовать использовать не более 100 тысяч для мобильных устройств и не более нескольких миллионов для PC
- Процессор имеет слишком много вершин для обработки. Это может быть сетчатые сетки, моделирование ткани, частицы или другие игровые объекты и сетки. Как указано выше, обычно рекомендуется сохранять это число как можно ниже без ущерба для качества игры
- Рендеринг не создает проблем ни для GPU, ни для CPU. Проблема может быть, к примеру, в скриптах или физике. Используйте профайлер для поиска источника проблемы

5.10.1. Оптимизация работы центрального процессора

В процессе визуализации любого объекта на экране CPU должен немного потрудиться: выяснить, какие источники света влияют на объект, настроить шейдер и его параметры, отправить команды прорисовки графическому драйверу, который подготовит команды для отправки на видеокарту. Все эти операции могут быть очень ресурсоемкими, если у вас есть много видимых объектов.

Все это требует значительных ресурсов, поэтому, если у вас много видимых объектов, оно может сложиться. Например, если у вас есть тысяча треугольников, на процессоре гораздо проще, если они все находятся в одной сетке, а не в одной сетке на треугольник (добавляя до 1000 сеток). Стоимость обоих сценариев на графическом процессоре очень похожа, но работа, выполняемая процессором для рендеринга тысячи объектов (вместо одного), значительно выше.

Чтобы уменьшить объем работы, которую должен выполнить процессор:

- Уменьшите количество видимых объектов:
  - » Объединяйте близко расположенные объекты: вручную или используя инструмент draw call batching[6] в Unity
  - » Используйте меньше материалов, объединяйте текстуры в большие текстурные атласы

6 См. https://docs.unity3d.com/ru/current/Manual/DrawCallBatching.html

  - » Используйте меньше объектов, которые должны визуализироваться несколько раз (отражения, тени, попиксельные источники света и т. п., смотрите ниже)

- Объединяйте объекты так, чтобы каждый меш содержал хотя бы несколько сотен треугольников и использовать только один материал. Важно понимать, что объединение двух объектов, использующих разные материалы, не даст увеличения производительности. Основная причина, по которой два меша используют разные материалы, состоит в том, что они используют разные текстуры. Для оптимизации производительности CPU нужно убедиться, что объекты, которые вы объединяете, используют одну текстуру

Когда вы используете много пиксельных источников света при **Forward rendering path**, бывают ситуации, в которых не имеет смысла объединять объекты, это более подробно описано ниже.

5.10.2. Оптимизация GPU

Существует два основных правила оптимизации геометрии модели:

- Не используйте треугольников больше, чем необходимо
- Старайтесь держать количество швов на UV-карте и количество жёстких рёбер (удваивающих вершины) как можно более низким

Следует отметить, что количество вершин, которое обрабатывает видеокарта, обычно не совпадает с количеством, показываемым 3D-приложением. Приложения для моделирования обычно показывают геометрическое количество вершин, то есть, количество угловых точек, составляющих модель. Для видеокарты некоторые геометрические вершины необходимо разбить на несколько логических вершин для корректной визуализации. Вершина может быть разбита на несколько, если она имеет несколько нормалей, UV-координат или вертексных цветов. Следовательно, количество вершин в Unity неизменно выше, чем количество вершин в 3D-приложении.

В то время как количество геометрии в моделях оказывает влияние в первую очередь на GPU, некоторые функции Unity предполагают обработку моделей и на CPU, например mesh skinning.

5.10.3. Оптимизация освещения

Самое быстрое освещение — это то, которое не рассчитывается. Используйте карты освещения для статичного освещения вместо расчета освещения в каждом кадре. Процесс создания карт освещения требует много времени, чем простое размещения источников света в сцене, но:

- Это намного быстрее работает (в 2–3 раза по сравнению с 2 пиксельными источниками света)
- Это выглядит лучше, так как вы можете использовать Backed GI и с более высоким качеством

Во многих случаях можно заменить размещение источников света правильной настройкой шейдеров и контента. Для примера, вместо размещения источника света прямо перед камерой для получения эффекта «подсветка краев модели» (rim lighting), проще добавить расчет этого эффекта прямо в шейдере.

5.11. Слои

Слои (Layers) в основном используются камерами для рендера частей сцены и источниками света для освещения частей сцены. Но слои можно использовать и для избирательного игнорирования коллайдеров при рейкастинге или для определения столкновений.

Первый шаг – создать новый слой, который, впоследствии, можно будет назначить игровому объекту. Для создания нового слоя, откройте меню **Edit** и выберите там пункт **Project Settings, Tags & Layers**.

На изображении 5.59 мы создаем новый слой в пустом слоте **User Layer 8**. Мы назвали его My Player.

Теперь, когда вы создали новый слой, его следует назначить одному из игровых объектов. Из списка **Layers** в инспекторе объекта выберите ваш слой (рис. 5.60).

Теперь вы можете избирательно прорисовывать объекты, находящиеся в определенном слое с помощью свойства **Culling mask** камеры (эта функция позволяет выбирать отображаемые камерой слои). Для этого выберите камеру, которая должна избирательно прорисовывать объекты.

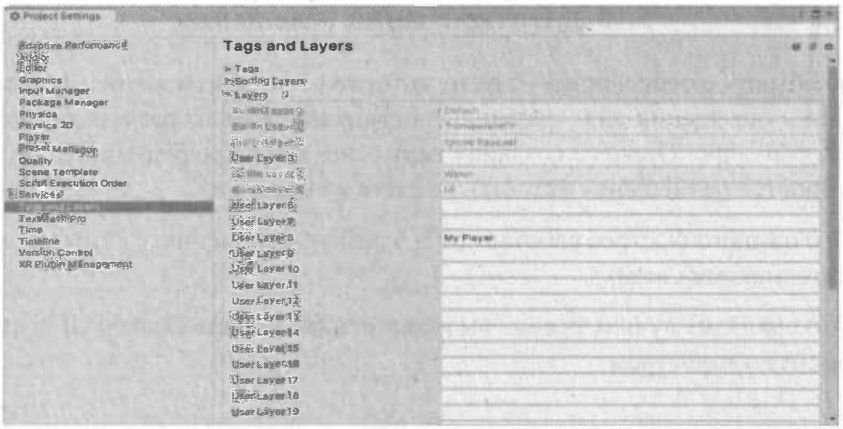


Рис. 5.59. Создание нового слоя

Рис. 5.60. Выбор пользовательского слоя

Измените свойство **Culling mask** с помощью включения или отключения слоев в выпадающем меню (рис. 5.61).

Используя слои, можно кастить (испускать) лучи, игнорируя коллайдеры на определенных слоях. Например, вы можете пожелать выпустить луч только в слое с главным персонажем, игнорируя при этом все остальные коллайдеры.

Метод Physics.Raycast принимает битовую маску, в которой каждый бит определяет – будет ли игнорироваться слой, или нет. Мы будем сталкивать луч со всеми коллайдерами, если в layerMask включены все биты. Если layerMask = 0, то луч никогда не столкнется ни с одним из коллайдеров.

Рис. 5.61. Изменение свойства "Culling mask" главной камеры

```
int layerMask = 1 << 8;

// Пересекает ли луч какие-либо объекты, которые находятся в
слое игрока.
if (Physics.Raycast(transform.position, Vector3.forward,
Mathf.Infinity, layerMask))
    Debug.Log("Луч попал в игрока");
```

Однако в реальной игре обычно требуется обратное, а именно: выпустить луч для столкновения со всеми коллайдерами, кроме тех, которые находятся в слое My Player:

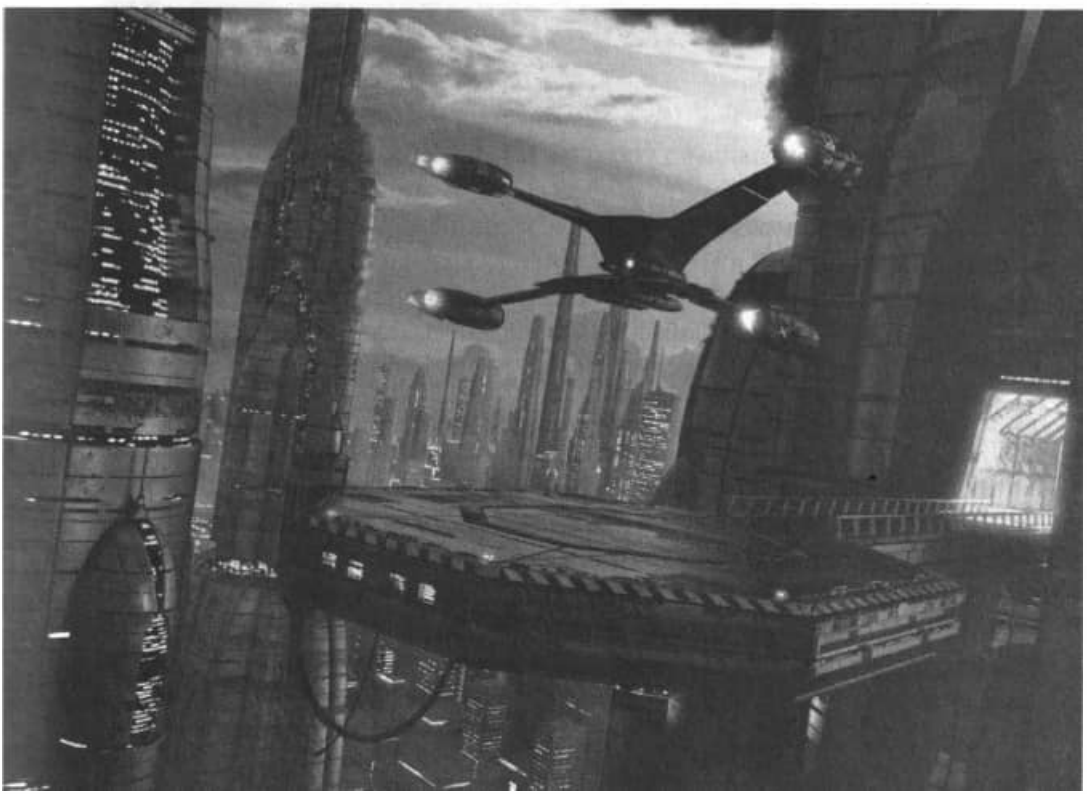
```
void Update () {
    // Битовый сдвиг индекса слоя (8) для получения битовой
маски
    int layerMask = 1 << 8;

    // Мы хотим столкнуться со всем, кроме слоя 8.
    // Оператор ~ делает это, он инвертирует битовую маску.
    layerMask = ~layerMask;

    RaycastHit hit;
    // Пересекает ли луч какие-либо объекты, кроме слоя игрока
    if (Physics.Raycast(transform.position, transform.
TransformDirection (Vector3.forward), out hit, Mathf.Infinity,
layerMask)) {
        Debug.DrawRay(transform.position, transform.
TransformDirection (Vector3.forward) * hit.distance, Color.
yellow);
        Debug.Log("Да");
    } else {
        Debug.DrawRay(transform.position, transform.
TransformDirection (Vector3.forward) *1000, Color.white);
        Debug.Log("Нет");
    }
}
```

Если вы не передаете layerMask в метод Raycast, то будут игнорироваться только те коллайдеры, что находятся на слое IgnoreRaycast. Это самый простой способ игнорировать некоторые коллайдеры при рейкастинге.

В следующей главе мы рассмотрим другую фундаментальную тему – мы поговорим о физике объектов. Без нее мы не сможем сделать игру реалистичной.

Глава 6.

Физика в Unity



Данная глава очень важна. Игра должна быть реалистичной, а для этого объект в игре нужно правильно ускорить и задействовать столкновения, гравитацию и другие силы. Будет смотреться очень комично, если главный игрок сможет проходить сквозь стены – если он не призрак, то такого быть не должно. Встроенный в Unity физический движок обеспечивает разработчика всеми необходимыми для симуляции физики компонентами. С помощью настройки всего нескольких параметров, можно создать объекты, которые ведут себя пассивно реалистично (т.е., они будут перемещены в результате столкновений и падений, но не начнут двигаться сами по себе). Управляя физикой из скриптов, вы можете придать объекту динамику автомобиля, машины или даже подвижного куска ткани. В этой главе мы поговорим об основах физики в Unity.

Если быть предельно точным, то на самом деле в Unity есть два отдельных физических движка, один для 3D физики и один для 2D физики. Основные понятия идентичны в обоих движках (за исключением дополнительного измерения в 3D), но они реализованы с разными компонентами. Так, например, существует компонент Rigidbody для 3D физики и аналогичный Rigidbody 2D для 2D физики.

6.1. Основные понятия

6.1.1. Твердые тела

Rigidbody – это основной компонент, реализующий физическое поведение для объекта. Как только вы добавите Rigidbody, объект немедленно начнет реагировать на гравитацию. Если добавлен один или несколько компонентов Collider, то при коллизиях (столкновениях) объект будет передвигаться.

Так как компонент Rigidbody управляет перемещением объекта, к которому он прикреплен, вам не следует пытаться воздействовать на объект из кода с помощью изменения таких свойств **Transform**, как *Position* и *Rotation*. Вместо этого вам следует применять силы для того, чтобы толкать объект и позволить физическому движку рассчитать результаты.

Существуют случаи, при которых вы можете захотеть, чтобы у объекта был Rigidbody, но физический движок не управлял его движением. Например, вам нужно управлять своим персонажем напрямую из кода, но при этом вы хотите, чтобы он взаимодействовал с триггерами. Этот, производимый с помощью кода, тип движения называется кинематическим движением. У компонента Rigidbody есть свойство **Is Kinematic**, которое может исключить объект из-под контроля физического движка, и позволить перемещать его кинематически из скрипта. Значение **Is Kinematic** можно менять из кода, чтобы включать или отключать физику для объекта, но эта возможность требует дополнительных ресурсов и должна использоваться как можно реже.

Когда твердое тело перемещается со скоростью, меньшей определенного минимального порога, физический движок предполагает, что оно остановилось и находится в покое. При этом объект не будет вновь двигаться до тех пор, пока с ним не произойдет столкновение или пока к нему не применят силу, так что он уходит в «спящий» режим. Эта оптимизация означает, что на объект не будут расходоваться ресурсы CPU, пока его вновь не «разбудят» (т.е. не вновь не приведут в движение). По многим причинам засыпание и пробуждение твердых тел происходит прозрачно. Однако иногда объект не удается разбудить, если в него или от него переместится статичный коллайдер (тот, что без твердого тела) изменяя положение трансформации. Это может привести, скажем, к висящему в воздухе твердому телу, когда пол под ним сдвинулся вниз. В таких случаях объект можно разбудить принудительно, с помощью функции WakeUp().

6.1.2. Коллайдеры

Основная информация о коллайдерах

Компоненты коллайдера определяют форму объекта для целей физических столкновений. Невидимый коллайдер необязательно должен иметь ту же форму, что и сетка объекта, и на самом деле грубое приближение часто более эффективно и неразличимо в игровом процессе.

Простейшими (и наименее ресурсоемкими) коллайдерами являются так называемые *примитивные типы коллайдеров*. В 3D это Box Collider, Sphere Collider и Capsule Collider. В 2D вы можете использовать Box Collider 2D и Circle Collider 2D. Вы можете добавить любое количество примитивных коллайдеров к объекту и получить тем самым *составной коллайдер*.

При аккуратном позиционировании и определении размеров, составные коллайдеры часто могут достаточно хорошо соответствовать форме объекта, сохраняя при этом низкую нагрузку на процессор. Дополнительную гибкость можно получить, если создать дополнительные коллайдеры на дочерних объектах (например, ящики можно поворачивать относительно локальных осей родительского объекта).

Обратите внимание, что примитивные коллайдеры работают некорректно со сдвиговыми преобразованиями – это означает, что если вы используете комбинацию вращений и неоднородных масштабов в иерархии преобразования, так что результирующая фигура больше не будет соответствовать примитивной форме, примитивный коллайдер будет работать неправильно.

Однако в некоторых случаях даже составные коллайдеры недостаточно точны. В 3D вы можете использовать *Mesh-коллайдеры*, чтобы точно соответствовать форме меша объекта. В 2D, как правило, Polygon Collider 2D не будет идеально соответствовать форме спрайтовой графики, но вы можете уточнить форму до любого уровня детализации, который вам нравится.

Однако Mesh-коллайдеры интенсивно используют CPU, поэтому используйте их экономно, чтобы поддерживать производительность на высоком уровне. Также вы должны знать, что Mesh-коллайдер обычно не может столкнуться с другим Mesh-коллайдером (то есть, когда они вступают в контакт, ничего не произойдет). В некоторых случаях вы можете обойти это, пометив такой коллайдер как **Convex** в инспекторе. Преимущество этого решения заключается в том, что выпуклый Mesh-коллайдер может сталкиваться с другими Mesh-коллайдерами, поэтому вы можете использовать эту возможность, когда у вас есть движущийся персонаж с подходящей формой.

Тем не менее, хорошее общее правило – использовать Mesh-коллайдеры для геометрии сцены, а для движущихся объектов использовать примитивные составные объекты.

Коллайдеры могут быть добавлены к объекту без компонента Rigidbody для создания стен, пола и других недвижущихся элементов сцены. Такие коллайдеры называются *статическими*. Вы не должны менять положение статических коллайдеров, изменяя положение **Transform**, так как это сильно повлияет на производительность физического движка. Коллайдеры на объекте с **Rigidbody** называются *динамическими*. Статические коллайдеры могут взаимодействовать с динамическими, но поскольку у них нет твердого тела, они не будут двигаться в ответ на столкновения.

Физические материалы

С коллайдерами тесно связаны физические материалы. Когда коллайдеры взаимодействуют, их поверхности должны имитировать свойства материала, который они должны представлять. Например, слой льда будет скользким, в то время как резиновый шарик будет сильно тереться. Хотя форма коллайдеров не деформируется во время столкновений, их трение и отскок можно настроить с помощью физических материалов. Правильное определение параметров может потребовать небольшого количества проб и ошибок, но, например, ледяной материал будет иметь нулевое (или очень низкое) трение и малую прочность, а резиновый материал будет иметь высокое трение и почти идеальную прочность.

Триггеры и callback-функции

Система сценариев может обнаруживать столкновения и инициировать действия с помощью функции OnCollisionEnter. Тем не менее, вы также можете использовать физический движок просто для определения, когда один коллайдер входит в пространство другого, не создавая столкновения. Коллайдер, настроенный как триггер (используя свойство **Is Trigger**), не ведет себя как сплошной объект и просто пропускает другие коллайдеры. Когда коллайдер входит в свое пространство, триггер вызовет функцию OnTriggerEnter в сценариях объекта триггера.

В случае коллизий, в которых задействованы объекты с особыми именами в скриптах. Вы можете установить любой код для реакции на событие. Когда автомобиль врезается в препятствие.

При столкновении двух коллайдеров вызывается функция OnCollisionEnter. Во время обновлений, где поддерживается контакт (столкновение уже было ранее, но на данный момент оба коллайдера еще находятся в состоянии коллизии), вызывается функция OnCollisionStay и, наконец, функция OnCollisionExit указывает, что контакт был разорван. Trigger-коллайдеры вызывают аналогичные функции – OnTriggerEnter, OnTriggerStay и OnTriggerExit. Обратите внимание, что для 2D физики, есть эквивалентные функции, например, OnCollisionEnter2D (к названию функции добавляется 2D). Подробное описание этих функций представлено в классе

https://docs.unity3d.com/ru/current/ScriptReference/MonoBehaviour.html

У обычных не триггерных коллизий есть еще одна дополнительная деталь: как минимум один из вовлеченных в коллизию объектов должен обладать не кинематическим твердым телом (т.е. IsKinematic должен быть выключен). Если оба объекта являются кинематическими, то тогда не будут вызываться функции, вроде OnCollisionEnter и т.д. С триггерными столкновениями это условие не применяется, так что и кинематические и не кинематические Rigidbody будут незамедлительно вызывать OnTriggerEnter при пересечении триггерного коллайдера.

Взаимодействие коллайдеров

Коллайдеры взаимодействуют друг с другом по-разному, в зависимости от того, как настроены их компоненты Rigidbody. Тремя важными конфигурациями являются статичный коллайдер (Static Collider) (т.е. компонент Rigidbody отсутствует вообще), Rigidbody-коллайдер (Rigidbody Collider), и кинематический Rigidbody-коллайдер (Kinematic Rigidbody Collider).

Начнем со **статического коллайдера**. Как мы уже знаем, статический коллайдер – это игровой объект, у которого есть коллайдер, но нет Rigidbody. Статичные коллайдеры используются для объектов, которые всегда стоят на месте и совсем не двигаются. Встречные Rigidbody-объекты будут врезаться в статичный коллайдер, но не сдвинут его.

В физический движок заложено предположение, что статичные коллайдеры никогда не двигаются или меняются, и, на основе этого предположения, движок делает полезные оптимизации. Следовательно, статичные коллайдеры нельзя включать/выключать, двигать или масштабировать во время игрового процесса. Если вы измените статичный коллайдер, то в результате физическим движком будет вызван дополнительный внутренний перерасчет, который будет сопровождаться большим падением производительности. Хуже

того, изменения иногда могут оставить коллайдер в неопределенном состоянии, в результате чего будут производиться ошибочные физические расчеты. Например, рейкаст к измененному статичному коллайдеру может не обнаружить коллайдера или обнаружить его в случайном месте в пространстве. По этим причинам, следует изменять только коллайдеры с Rigidbody. Если вы хотите, чтобы на коллайдер объекта не влияли встречные Rigidbody, но чтобы его можно было двигать при помощи скрипта, то вам следует прикрепить кинематический Rigidbody к нему, нежели вообще не добавлять Rigidbody.

Теперь поговорим о **Rigidbody-коллайдерах**. Это игровой объект, к которому прикреплен коллайдер и нормальный, не кинематический, Rigidbody. Rigidbody-коллайдеры полностью симулируются физическим движком и могут реагировать на коллизии и силы, приложенные из скрипта. Они могут сталкиваться с другими объектами (включая статичные коллайдеры) и являются самой распространенной конфигурацией коллайдера в играх.

Осталось рассмотреть последний тип коллайдера – **кинематический** Rigidbody-коллайдер. Он похож на предыдущий тип коллайдера, но свойство IsKinematic у Rigidbody включено. Изменяя компонент **Transform**, вы можете перемещать объект с кинематическим Rigidbody, но он не будет реагировать на коллизии и приложенные силы так же, как и не кинематические Rigidbody. Кинематические Rigidbody должны использоваться для коллайдеров, которые могут двигаться или периодически выключаться/включаться, иначе они будут вести себя как статичные коллайдеры. Примером этого является скользящая дверь, которая обычно является недвижимым физическим препятствием, но по надобности может открываться. В отличие от статичного коллайдера, движущийся кинематический Rigidbody будет применять трение к другим объектам и, в случае контакта, будет «будить» другие Rigidbody.

Даже когда они неподвижны, кинематические Rigidbody-коллайдеры ведут себя иначе, в отличие от статичных коллайдеров. Например, если коллайдер настроен как триггер, то вам также понадобится добавить к нему Rigidbody, чтобы в вашем скрипте можно было принимать события триггера. Если вы не хотите, чтобы триггер падал под действием силы гравитации или подвергался влиянию физики, то тогда вы можете включить свойство IsKinematic.

Компонент Rigidbody можно переключать между нормальным и кинематическим поведением в любое время с помощью свойства **IsKinematic**.

Типичным примером этого является эффект «тряпичной куклы», когда персонаж обычно движется под анимацией, но на него может повлиять взрыв или сильное столкновение. Каждому члену персонажа может быть присво-

ен собственный компонент Rigidbody с включенным IsKinematic по умолчанию. Конечности будут нормально двигаться с помощью анимации, пока IsKinematic не будет отключен для всех из них, и они сразу же будут вести себя как физические объекты. В этот момент сила столкновения или взрыва отправит персонажа, летящего с убедительно брошенными конечностями.

Таблицы 6.1 и 6.2 представляют собой матрицу действий коллизии. В первой таблице показано матрица определения столкновений, при обнаружении которых посылаются сообщения. Во второй – когда отправляются сообщения триггера. Вы можете не запомнить эти таблицы, но вы должны помнить, что законы физики не применяются к объектам, у которых нет Rigidbody.

Таблица 6.1. Определение столкновений и отправка сообщений при их возникновении

	Статичный коллайдер	Rigidbody-коллайдер	Кинематический Rigidbody-колайдер	Статичный коллайдер-триггер	Rigidbody-коллайдер триггер	Кинематический Rigidbody коллайдер-триг.
Статичный коллайдер		+				
Rigidbody-коллайдер	+	+	+			
Кинематический Rigidbody-колайдер		+				
Статичный коллайдер-триггер						
Rigidbody-коллайдер триггер						
Кинематический Rigidbody коллайдер-триг.						

Таблица 6.2. При коллизиях отправляются сообщения триггера

	Статичный коллайдер	Rigidbody-коллайдер	Кинематический Rigidbody-колайдер	Статичный коллайдер-триггер	Rigidbody-коллайдер триггер	Кинематический Rigidbody коллайдер-триг.
Статичный коллайдер					+	+
Rigidbody-коллайдер				+	+	+
Кинематический Rigidbody-колайдер				+	+	+
Статичный коллайдер-триггер		+	+		+	+
Rigidbody-коллайдер триггер	+	+	+	+	+	+
Кинематический Rigidbody коллайдер-триг.	+	+	+	+	+	+

6.1.3. Сочленения

С помощью компонентов **Joint**, вы можете добавить один Rigidbody-объект к другому или к фиксированной точке в пространстве. Обычно нужно, чтобы у сочленения была хоть какая-то свобода движения, так что Unity предоставляет различные Joint-компоненты, реализующие разные ограничения. Например, *Hinge Joint* (*шарнирное соединение*) позволяет вращать вокруг определенной точки и оси, в то время как *Spring Joint* (*пружинное соединение*) удерживает объекты друг от друга, но позволяет слегка растянуть расстояние между ними. Как обычно, названия 2D-компонентов оканчиваются на 2D, например, Hinge Joint 2D.

У сочленений также есть другие опции, которые можно включить для различных эффектов. Например, вы можете настроить сочленение так, чтобы оно разрывалось при применении к нему силы больше заданного порога. Некоторые сочленения также позволяют возникать движущей силе (drive

force) между соединенными объектами для автоматического приведения их в движение.

6.1.4. Контроллеры персонажа

Персонажу в игре от первого или третьего лица часто требуется некоторая физика, основанная на столкновениях, чтобы он не падал сквозь пол или не ходил сквозь стены. Обычно, ускорение и движение персонажа физически нереалистичны, поэтому он может ускоряться, тормозить и менять направление мгновенно, не подвергаясь влиянию импульса.

В 3D физике для создания реалистичного движения персонажа используется Character Controller. Этот компонент дает персонажу простой коллайдер в форме капсулы, который всегда находится в вертикальном положении. Данный компонент обладает своими особыми функциями для назначения скорости и направления объекта, но, в отличие от настоящих коллайдеров, он не требует Rigidbody.

Character Controller не может проходить сквозь статические коллайдеры в сцене, поэтому он будет двигаться по полу и блокироваться стенами. Во время движения он может отталкивать Rigidbody-объекты, но входящие столкновения не будут влиять на его ускорение. Это значит, что вы можете использовать стандартные 3D-коллайдеры, чтобы создать сцену, по которой будет ходить Controller, но вы не ограничены реалистичным, с точки зрения физики, поведением самого персонажа.

Вы узнали основные сведения о физике в Unity. Настало время рассмотреть упомянутые объекты подробнее.

6.2. Коллайдеры разных форм

В игре мало создать коллайдер. Важно, чтобы он хотя бы приблизительно соответствовал форме объекта. Именно поэтому в Unity существуют коллайдеры разных форм:

- **Box Collider** – коллайдер в виде куба (коробки)
- **Capsule Collider** – коллайдер в виде капсулы
- **Sphere Collider** – коллайдер сферической формы
- **Wheel Collider** – коллайдер в виде колеса, специально разработанный для транспортных средств
- **Terrain Collider** – создает коллайдер на базе поверхности (terrain)

Каждый коллайдер обладает своими свойствами. Некоторые из свойств общие для всех коллайдеров, некоторые – уникальны для конкретного вида коллайдера. Для геометрических коллайдеров (Box Collider, Capsule Collider, Sphere Collider) общими являются следующие свойства:

- **Is Trigger** – если включено, коллайдер используется для запуска событий и игнорируется физическим движком
- **Material** – ссылка на физический материал, определяющий, как этот коллайдер взаимодействует с другими
- **Center** – позиция коллайдера в локальном пространстве объекта, то есть задает центр объекта

У **Box Collider** есть уникальное свойство Size, определяющее размеры коллайдера в направлениях X, Y, Z. Очевидно, «коробочные» коллайдеры используются для любых объектов в общих чертах формы параллелепипеда – ящик, сундук. Также такой коллайдер можно использовать в роли пола, стены, рампы.

Капсульный коллайдер (**Capsule Collider**) состоит из двух полусфер, соединенных между собой цилиндром. У него такая же форма, как и у примитива капсулы (Capsule). Сферический коллайдер (**Sphere Collider**) принимает форму сферы. У капсульного и сферического коллайдеров есть общее свойство *Radius*, задающее размер коллайдера (радиус капсулы или сферы).

Размер сферического коллайдера может быть изменен только при помощи свойства *Radius*, но он не может быть масштабирован по одной из осей отдельно (то есть, вы сможете сплющить сферу в эллипс). Как можно понять, используется он в основном для сферических объектов, вроде теннисных мячиков и т.п. Сфера также хорошо подходит для падающих камней и других объект, которые должны катиться и падать.

Регулировок для капсульного коллайдера больше, поскольку вы можете задать не только радиус, но и высоту и направление капсуля:

- **Height** – общая высота коллайдера
- **Direction** – ось, вдоль которой расположен коллайдер в локальном пространстве объекта

Вы можете настроить радиус (Radius) и высоту (Height) коллайдера независимо друг от друга. Такой тип коллайдера обычно используется в компоненте Character Controller и неплохо работает для продолговатых предметов, а

также можно комбинировать с другими коллайдерами для необычных форм. На рис. 6.1 показано, чем отличаются свойства Radius и Height.

Рис. 6.1. Свойства Radius и Height капсульного коллайдера

Коллайдеры Wheel Collider и Terrain Collider особенные, поэтому стоят немного в стороне. Первый коллайдер был разработан специально для транспортных средств, стоящих на земле. Он имеет встроенную систему обнаружения столкновений, физику колес и модель трения шин на основе проскальзывания. Его можно использовать для других предметов, кроме колес, но он специально разработан для автомобилей с колесами. Таблица 6.3 содержит свойства этого коллайдера.

Таблица 6.3. Свойства коллайдера Wheel Collider

Свойство	Описание
Mass	Масса колеса
Radius	Радиус колеса
Wheel Damping Rate	Значение демпфирования, применяемое к колесу
Suspension Distance	Максимальное расстояние расширения подвески колеса, измеряется в локальном пространстве. Подвеска всегда расширяется вниз по локальной оси Y
Force App Point Distance	Этот параметр определяет точку, где будут приложены силы колеса. Ожидается, что это будет в метрах от основания колеса в положении покоя вдоль направления движения подвески. Когда forceAppPointDistance = 0 силы будут приложены к колесной базе в состоянии покоя
Center	Центр колеса в локальном пространстве объекта

Suspension Spring	Подвеска пытается достичь целевого положения путем добавления пружины и демпфирующих сил
Spring	Сила пружины пытается достичь **целевого положения** (Target Position). Чем больше значение, тем быстрее подвеска достигает целевого положения
Damper	Уменьшает скорость подвески. Большее значение замедляет ход подвесной пружины (Suspension Spring)
Target Posision	Расстояние покоя подвески по расстоянию подвески. 1 соответствует полностью расширенной приостановке, а 0 – полностью сжатой приостановке. Значение по умолчанию – 0.5, что соответствует поведению подвески обычного автомобиля
Forward/Sideways Friction	Свойства трения шины, когда колесо катится вперед и в сторону

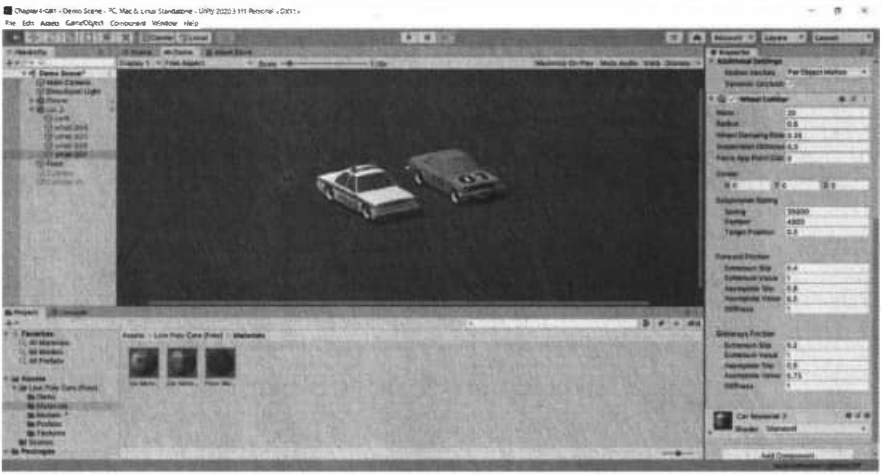


Рис. 6.2. Свойства "Wheel Collider"

Wheel Collider рассчитывает трение отдельно от остального физического движка, используя модель трения, основанную на скольжении шин. Это позволяет реализовать более реалистичное поведение, и при этом, такие коллайдеры игнорируют стандартные настройки физического материала.

Не поворачивайте и не вращайте объекты с Wheel Collider для управления машиной – объекты с Wheel Collider всегда должны оставаться в фиксированном положении относительно машины. Однако вы можете поворачивать

и вращать визуальные представления колес. Лучший способ это сделать – разделить объекты с Wheel Collider'ами и видимыми колесами (рис. 6.3).

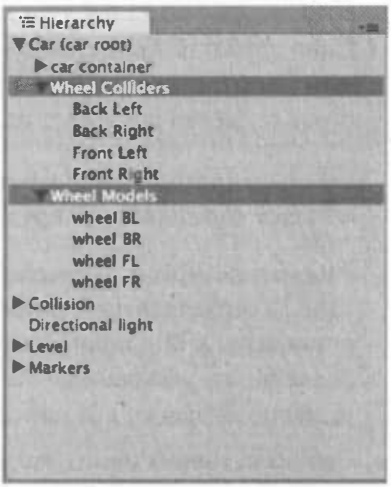


Рис. 6.3. Настройка колес

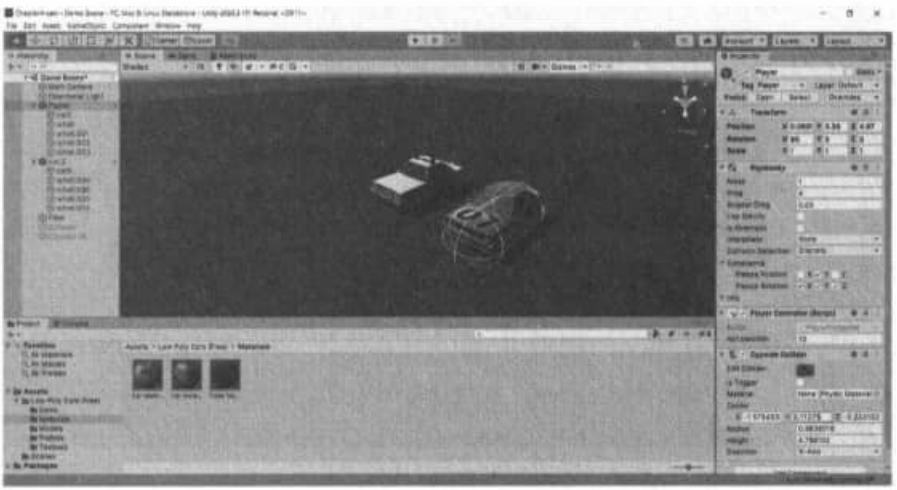


Рис. 6.4. Процесс настройки коллайдера для автомобиля

Поскольку машины могут развивать большую скорость, очень важен правильный подход к геометрии гоночного трека, используемой для определения столкновений. Например, меш-коллайдер не должен иметь выпуклостей или вмятин, которые служат для украшения видимой модели (например, столбы ограждения). Обычно меш-коллайдер для гоночного трека создают отдельно от видимого меша, делая его как можно плавнее. В нем также не

должно быть тонких объектов – если у вас тонкий забор трека, сделайте его шире в меше-коллайдере (или полностью удалите внешнюю часть, если машина туда никогда не сможет добраться).

Terrain Collider намного проще. Он берет поверхность (объект Terrain) и делает на его основе коллайдер. Свойство **Material** задает физический материал, определяющий как коллайдер взаимодействует с другими. А свойство **Enable Tree Colliders** включает коллайдеры деревьев, если они созданы на местности.

6.3. Mesh-коллайдер

Остался только нерассмотренный **Mesh Collider**. Поскольку он заслуживает отдельного разговора, то он вынесен в отдельный раздел.

Mesh Collider использует меш-ассет для создания под него коллайдера на основе этого меша. Это гораздо удобнее, чем использовать примитивы для определения столкновений сложных мешей. Меш-коллайдеры, которые помечены как Convex могут взаимодействовать (сталкиваться) с другими меш-коллайдерами.

Свойств, не смотря на важность этого типа коллайдера, немного:

- **Is Trigger**, **Material** – аналогичны таким же свойствам у других коллайдеров
- **Mesh** – ссылка на меш, используемый для столкновений
- **Convex** – если включено, текущий меш коллайдер будет взаимодействовать с другими меш коллайдерами. Выпуклые (Convex) меш коллайдеры ограничены 255 треугольниками

Mesh-коллайдер создает свое представление столкновения из меша, присоединенного к GameObject, и считывает свойства присоединенного **Transform**, чтобы правильно установить его положение и масштаб. Преимущество этого подхода заключается в том, что форма коллайдера может быть точно такой же, как форма видимой сетки для объекта, что приводит к более точным и достоверным столкновениям.

Тем не менее, эта точность достигается с более высокими накладными расходами обработки, чем столкновения с участием примитивных коллайдеров (сфера, коробка, капсула), и поэтому лучше всего использовать Mesh Colliders экономно.

Меши, служащие для расчета столкновений, используют отсечение невидимых поверхностей. Если объект столкнется с таким мешем визуально, то он также столкнется с ним и физически.

Есть кое-какие ограничения в использовании меш коллайдеров. Обычно два меш коллайдера не могут столкнуться друг с другом. Все меш коллайдеры могут сталкиваться с примитивными коллайдерами. Если ваш меш помечен как Convex, то он сможет сталкиваться с другими меш коллайдерами.

6.4. Свойства контроллера персонажа

Ранее мы разобрались, зачем используется Character Controller (контроллер персонажа). Теперь этот компонент мы рассмотрим подробнее.

Компонент Character Controller в основном используется для управления от третьего или первого лица, где не требуется физика **Rigidbody**.

Разберемся, зачем нужен этот компонент. Традиционное управление от первого лица, если вы, например, хотите сделать шутер в стиле Counter Strike, мало вас устроит. Ну не может персонаж двигаться со скоростью 100 км/ч, моментально разворачиваться и проходить сквозь стены. Использование только Rigidbody тоже не дает реалистичного эффекта. Именно для решения подобных задач и был разработан компонент Character Controller. Он просто создает коллайдер (Collider) в форме капсулы, который можно двигать в некотором направлении при помощи скрипта. Контроллер позаботится о движении, но столкновения его сдерживают. Он будет скользить вдоль стен, подниматься по лестницам (если ступеньки ниже, чем свойство Step Offset) и ходить по наклонам, учитывая свойство **Slope Limit**.

Контроллер персонажа не реагирует на силы сам по себе и не отталкивает Rigidbody объекты автоматически. Если вы хотите толкать Rigidbody-тела или объекты, использующие Character Controller, вы можете применить силы к любому объекту, в которого он врезается, через скриптинг при помощи функции OnControllerColliderHit().

С другой стороны, если вы хотите, чтобы персонаж игрока подвергался влиянию физики, то, возможно, вам было бы лучше использовать Rigidbody вместо Character Controller.

Вы можете менять значения свойств **Height** и **Radius** для соответствия коллайдера мешу персонажа. Для человекоподобных персонажей рекомендуется использовать высоту около 2-х метров. Вы также можете изменить центр

(свойство **Center**) капсулы в случае, если точка вращения (pivot) не соответствует точному центру персонажа.

Значение свойства **Step Offset** тоже важно при настройке, убедитесь, что это значение находится в диапазоне между 0.1 и 0.4 для 2-х метровых гуманоидов.

Значение свойства **Slope Limit** тоже должно быть не слишком маленькое. Зачастую использование значения в 90 градусов подходит лучше всего. Character Controller не сможет ползать по стенам в виду формы капсулы.

Свойство **Skin Width** является одним из самых критических свойств для верной настройки Character Controller. Если вам персонаж застрял, то, скорее всего, ваше значение **Skin Width** слишком мало. Параметр **Skin Width** позволяет объектам слегка пересекать контроллер, но уменьшает тряску и защищает от застревания.

Достаточно неплохой практикой является сохранение значения **Skin Width** как минимум больше чем 0.01 и и больше чем 10% радиуса коллайдера (свойство *Radius*).

Значение свойства **Min Move Distance** рекомендуется установить равным 0. Остальные свойства компонента с некоторыми комментариями приведены в табл. 6.4.

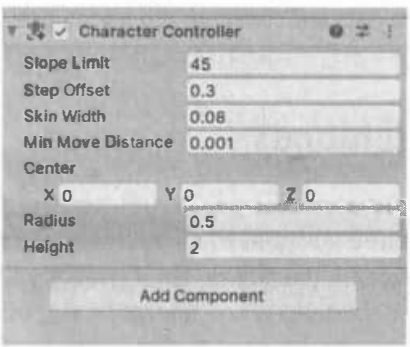


Рис. 6.5. Свойства компонента "Character Controller"

Таблица 6.4. Свойства компонента Character Controller

Свойство	Описание
Slope Limit	Ограничивает возможность коллайдера взбираться по склонам – только по склонам равным или меньше чем указанное значение

Step Offset	Персонаж переступит на поверхность, только если она ближе к земле, чем заданное значение
Skin width	2 коллайдера могут пересечься друг с другом на глубину, равную значению Skin Width. Большее значение уменьшит степень тряски. Слишком низкое значение может привести к тому, что персонаж застрянет. Наилучшим вариантом является установление этого значения равным 10% от радиуса
Min Move Distance	Если персонаж попробует сдвинуться ниже указанной величины, то он не сдвинется. Это может быть использовано, чтобы уменьшить тряску. В большинстве ситуаций это значение стоит оставить равным нулю (0)
Center	Сдвиг коллайдера в мировом пространстве без влияния на то, как вращается персонаж
Radius	Значение радиуса коллайдера. По сути дела, это ширина коллайдера
Height	Высота коллайдера Capsule Collider персонажа. Изменение ее растянет коллайдер вдоль оси X в оба направления

6.5. Снова о Rigidbody

Компонент Rigidbody позволяет игровым объектам действовать под управлением физики. Назначение компонента – получать силу и крутящий момент, чтобы ваши объекты могли двигаться. На Rigidbody влияют также силы гравитации (если иное не указано в его настройках), объекты с Rigidbody попадают под действие дополнительных сил посредством сценариев, а также могут взаимодействовать с другими объектами посредством физического движка Unity.

Свойства компонента Rigidbody приведены в табл. 6.5.

Таблица 6.5. Компонент Rigidbody

Свойство	Описание
Mass	Масса объекта (в килограммах)

Drag	Какое воздушное сопротивление оказывается на объект, пока он перемещается под воздействием сил. 0 означает отсутствие сопротивления, а бесконечность (infinity) тут же прекращает перемещение объекта
Angular Drag	Какое воздушное сопротивление оказывается на объект, пока он вращается под воздействием силы вращения. 0 означает отсутствие сопротивления. Вы **не можете** остановить вращение объекта путем установки его углового сопротивления (Angular Drag) в бесконечное (infinity) положение
Use Gravity	При включении на объект действует гравитация
Is Kinematic	Если этот параметр включен, объект не будет управляться физическим движком, а может управляться только его компонентом Transform. Это полезно для перемещения платформ или если вы хотите анимировать Rigidbody, к которому прикреплен HingeJoint
Interpolate	Попробуйте одну из опций, если вы замечаете тряску в перемещении своего твердого тела
- None	Не применяется никакой интерполяции
- Interpolate	Сглаживание трансформации на основании трансформации из предыдущего кадра
- Extrapolate	Сглаживание трансформации на основании приблизительной трансформации следующего кадра
Collision Detection	Используется для предотвращения проникновения быстро перемещающихся объектов сквозь другие объекты без определения столкновений
- Discrete	На всех остальных коллайдерах сцены используйте осторожное (Discreet) обнаружение столкновений. Другие коллайдеры будут использовать осторожное обнаружение столкновений при проверке на столкновения против них. Используется для нормальных столкновений (стандартное значение)

- Continuous	Используется осторожное обнаружение столкновений против динамических коллайдеров (с твердыми телами) и непрерывное обнаружение столкновений против статичных меш коллайдеров (без твердых тел). Твердые тела установленные как непрерывные динамические (Continuous Dynamic) будут использовать непрерывное обнаружение столкновений при тестировании на столкновения против этого твердого тела. Другие твердые тела будут использовать осторожное обнаружение столкновений. Используется для объектов, с которыми нужно столкнуться при помощи динамического обнаружения столкновений. Все это оказывает большое влияние на производительность, поэтому оставьте данное значение установленным в Discrete, если не испытываете проблем со столкновением быстро движущихся объектов
- Continuous Dynamic	Используется непрерывное обнаружение столкновений против объектов настроенных на непрерывное и непрерывное динамическое столкновение. Также будет использоваться непрерывное обнаружение столкновений со статичными меш-коллайдерами (без твердых тел). Для всех остальных коллайдеров будет использоваться осторожное обнаружение столкновений. Используется для быстро движущихся объектов
Constraints	Ограничения движения твердого тела
-Freeze Position	Выборочно останавливает перемещение твердого тела по осям X, Y и Z
- Freeze Rotation	Выборочно останавливает вращение твердого тела по осям X, Y и Z

Вот, что нужно знать о Rigidbody:

- Компонент Rigidbody должен быть явно добавлен в объект. Добавить этот компонент можно посредством команды меню **Components, Physics, Rigidbody**. После этого ваш объект будет находиться под влиянием физического движка.

- Когда объект находится под управлением физики, он перемещается частично независимо от своих родителей. Если вы переместите одного из родителей, они потянут за собой Rigidbody потомков. Однако твердые тела также будут падать вниз под воздействием силы тяжести и реагировать на события столкновений. Пока вы не настроите коллайдеры поверхности (Terrain Colliders), ваши объекты с Rigidbody будут падать в бездну. Временно можете выключить свойство Use Gravity, чтобы объект на провалился «под сцену».

- Для управления объектом с Rigidbody, вам нужно использовать сценарии для добавления силы или крутящего момента. Для этого используйте методы AddForce() и AddTorque() для Rigidbody объекта. Помните, что вы не должны напрямую изменять Transform объекта, когда вы используете физику.

- Коллайдеры – это другой тип компонентов, которые должны быть добавлены наряду с твердыми телами, чтобы задействовать столкновения. Если два твердых тела врезаются друг в друга, физический движок не будет просчитывать столкновение, пока к обоим объектам не будет назначен коллайдер. Твердые тела, не имеющие коллайдеров, будут просто проходить сквозь друг друга при просчете столкновений.

- Размер вашего игрового объекта играет большую роль по сравнению с массой Rigidbody. Если вы заметили, что объект ведет себя не так, как вы изначально задумывали – медленно двигается, колеблется, или некорректно сталкивается – попробуйте настроить масштаб своего меш-ассета. Стандартная единица измерения в Unity 1 юнит = 1 метр, например, разрушающийся на части небоскреб выглядит иначе, чем башня, собранная из игрушечных блоков, поэтому объекты разного размера должны быть смоделированы в масштабе. При проектировании человека проверьте, чтобы его рост был около 2 метров. Можете сравнить высоту модели человека с высотой модели куба. Высота куба по умолчанию – 1 метр. Следовательно, модель человека не должна быть выше двух кубов, поставленных друг на друга.

- Не забывайте о наличии поля Drag. Установка более высокой массы не сделает тело более быстрым в свободном падении. Для этого принято использовать свойство Drag. Низкое значение Drag сделает объект более тяжелым, высокое – легким. Типичные значения Drag – от 0,001 (что-то очень тяжелое) до 10 (перо).

6.6. Физический материал

Физический материал (Physics Material) используется для настройки эффектов трения и отскакивания объектов при столкновениях.

> **Примечание.** Следует отметить, что по историческим причинам, компонент называется Physic Material, а не Physics Material – так в свое время получилось, а вот 2D-версия материала называется как нужно - Physics Material 2D.

Для создания физического материала выберите из меню **Assets, Create, Physic Material**. Затем перетащите физический материал из окна **Project View** в **Collider**, находящийся в сцене.

Свойства физического материала представлены в табл. 6.6.

Таблица 6.6. Свойства физического материала

Свойство	Описание
Dynamic Friction	Трение, использующееся во время движения. Обычно значения бывают в диапазоне от 0 до 1. Значение, равное 1, соответствует трению как на льду, в то время как значение, равное 1, означает слабое трение, в результате которого объекту будет сложно двигаться без воздействия внешних сил
Static Friction	Трение, использующееся когда объект лежит на поверхности. Обычно значения бывают в диапазоне от 0 до 1. Значение, равное 0, означает отсутствие трения, в то время как значение равное 1 будет означать абсолютное трение (т.е. объектам по такой поверхности будет сложно передвигаться)
Bounciness	Насколько упругой является поверхность? Значение равное 0 означает неупругую поверхность. Значение равное 1 приведет к упругости, при которой объект не будет терять изначальную энергию
Friction Combine	Как комбинируется между собой трение двух объектов

Average	Значения двух трений усредняются
Minimum	Из двух значений выбирается меньше
Maximum	Из двух значений выбирается большее
Multiply	Значения трений умножаются друг на друга
Bounce Combine	Как комбинируется упругость двух сталкивающихся объектов. Она поддерживает те же режимы, что и Friction Combine режим

Трение – это величина, отвечающее за предотвращение трения поверхностей друг о друга. Это величина важна, если вы пытаетесь создать нагромождение из каких-нибудь объектов. Трение бывает двух видов, динамичным и статичным. Свойство **Static friction** используется, когда объект находится в неподвижном состоянии. Оно не даст такому объекту сдвинуться с места без воздействия внешних сил. И в этот момент в действие вступает свойство **Dynamic Friction**. При использовании **Dynamic Friction** объекты будут замедляться при столкновении друг с другом.

При контакте двух объектов, к каждому агенту отскакивание и трение применяется индивидуально. Другими словами, когда тело **A** находится в режиме Average, а тело **B** в режиме Multiply, поведение тела **A** будет зависеть от усредненных параметров, а тела **B** от произведения этих параметров.

Обратите внимание, что модель трения, используемая физическим движком Unity, настроена на производительность и стабильность симуляции и не обязательно представляет собой близкое приближение к реальной физике. В частности, контактные поверхности, которые больше, чем одна точка (например, две коробки, опирающиеся друг на друга), будут рассчитаны как имеющие две точки контакта, и будут иметь силы трения в два раза больше, чем в физике реального мира. Возможно, вы захотите умножить ваши коэффициенты трения на 0.5, чтобы получить более реалистичные результаты в таком случае.

6.7. Постоянная сила

Постоянная сила (Constant Force) – это простое средство для добавления сил к Rigidbody-объектам. Она отлично работает для одноразовых объектов, вроде ракеты, если вы хотите не начинать с большой скорости, а ускорять-

ся постепенно. Добавить компонент Constant Force можно так: **Component, Physics, Constant Force**.

Рис. 6.6. Постоянная сила

Таблица 6.7. Свойства компонента Constant Force

Свойство	Описание
Force	Применяемый вектор силы в мировом пространстве
Relative Force	Применяемый вектор силы в локальном пространстве объекта
Torque	Применяемый вектор крутящего момента в мировом пространстве. Объект начнет вращаться вокруг этого вектора. Чем длиннее вектор, тем быстрее вращение
Relative Torque	Применяемый вектор крутящего момента в локальном пространстве. Объект начнет вращаться вокруг этого вектора. Чем длиннее вектор, тем быстрее вращение

Чтобы ракета ускорялась вперед, установите Relative Force вдоль положительной оси Z. Затем используйте свойство **Drag** компонента Rigidbody, чтобы не была превышена максимальная скорость (чем выше значение **Drag**, тем ниже будет максимальная скорость). Также отключите в Rigidbody гравитацию (свойство **Use Gravity**), чтобы ракета не сходила со своего пути.

Два совета:

1. Чтобы объект поднимался вверх, добавьте компонент **Constant Force** с положительным значением Y свойства **Force**
2. Чтобы объект летел вперед, добавьте компонент **Constant Force** с положительным значением Z в свойстве **Relative Force**

6.8. Сочленения разных типов

Ранее было отмечено, что с помощью компонентов Joint, вы можете добавить один Rigidbody-объект к другому или к фиксированной точке в пространстве. Настало время рассмотреть данные компоненты.

6.8.1. Компонент Character Joint

Компоненты Character Joints в основном используются для эффектов тряпичной куклы. Они представляют собой шароподобное соединение, позволяющее вам ограничивать соединения на каждой оси. Свойства данного компонента представлены в табл. 6.8.

Таблица 6.8. Свойства компонента Character Joints

Свойство	Описание
Connected Body	Возможная ссылка на Rigidbody, от которого зависит соединение. Если не установить, то соединение будет связано с миром
Anchor	Точка в локальном пространстве игрового объекта (GameObject), вокруг которой соединение вращается
Axis	Оси закручивания. Визуализировано при помощи оранжевого конуса-гизмо
Auto Configure Connected Anchor	Если оно включено, то позиция Connected Anchor будет автоматически рассчитана для соответствия глобальной позиции свойства привязки. Это поведение по умолчанию. Если свойство отключено, вы можете настроить положение подключенного якоря вручную
Connected Anchor	Ручная конфигурация позиции Connected Anchor
Swing Axis	Оси качения. Визуализировано при помощи зеленого конуса-гизмо

Low Twist Limit	Нижний лимит соединения
High Twist Limit	Верхний лимит соединения
Swing 1 Limit	Нижний лимит относительно заданной оси качения Swing Axis
Swing 2 Limit	Верхний лимит относительно заданной оси качения Swing Axis
Break Force	Сила, которую надо применить к соединению, чтобы сломать его
Break Torque	Крутящий момент, который надо применить к соединению, чтобы сломать его
Enable Collision	Если включено, то включаются коллизии между телами связанными соединением
Enable Preprocessing	Отключение предварительной обработки помогает стабилизировать невозможные для выполнения конфигурации

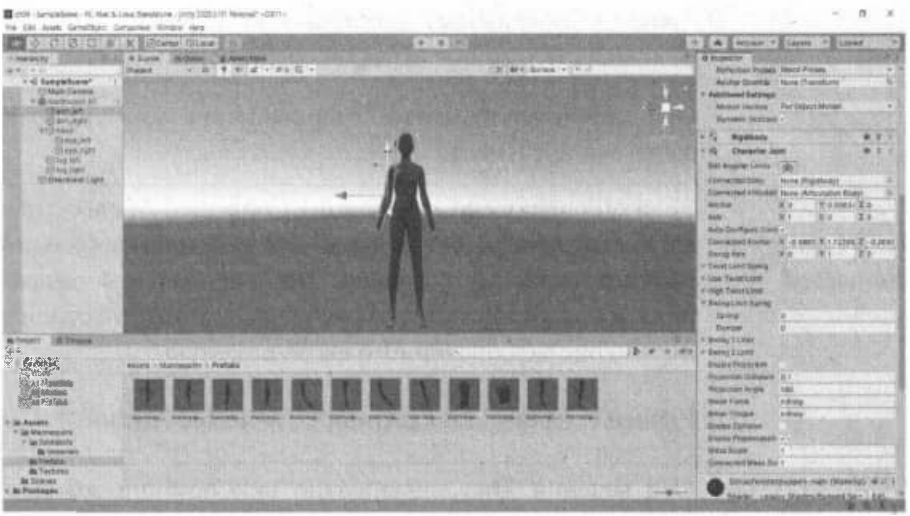


Рис. 6.7. Свойства компонента "Character joints"

Компоненты **Character joint** дают вам кучу возможностей для управления движением конечностей.

Ось закручивания (визуализировано в виде оранжевого гизмо) больше других осей позволяет вам управлять лимитами, т.к. вы можете указать и нижний и верхний лимиты в градусах (лимит угла измеряется относительно стартовой позиции). Значение –30 в **Low Twist Limit->Limit** и 60 в **High Twist Limit->Limit** ограничивают вращение вокруг оси закручивания (*оранжевый гизмо*) в пределах между –30 градусов и 60 градусов.

Группа параметров **Swing 1 Limit** ограничивает вращение вокруг оси качения (*зеленая ось*). Угол лимита симметричен. Таким образом, значение равное, например, 30 ограничит вращение от –30 до 30.

Ось **Swing 2 Limit** не имеет гизмо, но эта ось ортогональна по отношению к двум другим осям. Как и у предыдущей оси, лимит является симметричным, таким образом, значение равное, например 40 будет ограничивать вращение вокруг оси от –40 и до 40 градусов.

Вы можете использовать свойства **Break Force** и **Break Torque**, чтобы установить лимиты силы по отношению к соединению. Если они меньше бесконечности, то **Fixed Joint** будет разрушено, и соединение больше не будет поддаваться ограничениям.

6.8.2. Неподвижное сочленение. Компонент Fixed Joint

Компонент **Fixed Joint** (неподвижное соединение) ограничивает движение определенного объекта, связывая его с другим объектом. Этот процесс похож на Parenting (определение текущего объекта как «дочерний» по отношению к другому, «родительскому», объекту), но реализован с помощью физики, а не иерархии компонентов **Transform**. Чаще всего этот компонент используется в случае, если в определенный момент времени может потребоваться разъединить два объекта, или наоборот, соединить два объекта без необходимости изменения иерархии.

Таблица 6.9. Свойства компонента Fixed Joint

Свойство	Функция
Connected Body	Необязательная ссылка на другой объект с Rigidbody, к которому присоединяется текущий объект. Если поле оставить пустым, объект присоединяется к заданной точке в пространстве

Break Force	Сила, которую требуется приложить к объекту, чтобы разорвать соединение
Break Torque	Крутящий момент, который необходимо приложить к объекту, чтобы разорвать соединение
Enable Collision	Если включено, два соединенных тела будут сталкиваться друг с другом
Enable Preprocessing	Отключение предварительной обработки помогает стабилизировать невозможные для выполнения конфигурации

При создании игр иногда возникают случаи, когда требуется, чтобы объекты двигались вместе (временно или постоянно). Компоненты Fixed Joint позволяют упростить реализацию подобных ситуаций, поскольку вам не приходится менять положение объекта в иерархии с помощью скриптов. Недостаток подобного решения в том, что вам придется добавлять компоненты Rigidbody на объекты, которые требуется соединить с помощью Fixed Joint.

> **Примечание.** Для корректной работы компонентов Fixed Joint требуется компонент Rigidbody

Например, с помощью этого компонента можно реализовать «липкую гранату». Для этого нужно написать скрипт, позволяющий определить столкновение с другим объектом, имеющим компонент Rigidbody (например, врагом), после чего создать Fixed Joint, который присоединит гранату к этому Rigidbody, заставляя ее оставаться «прилипшей», даже если враг будет двигаться.

Для определения пределов прочности соединений можно использовать свойства **Break Force** и **Break Torque**. Если их значение меньше чем Infinity (бесконечность) и приложенная к соединению сила оказывается больше этих значений, соединение Fixed Joint разрывается и перестает удерживать два объекта вместе.

6.8.3. Configurable Joint

Настраиваемые сочленения (*configurable joint*) чрезвычайно настраиваемы, поскольку они включают в себя все функциональные возможности других

типов соединений. Вы можете использовать их для создания чего угодно, от адаптированных версий существующих соединений до узкоспециализированных соединений вашего собственного дизайна.

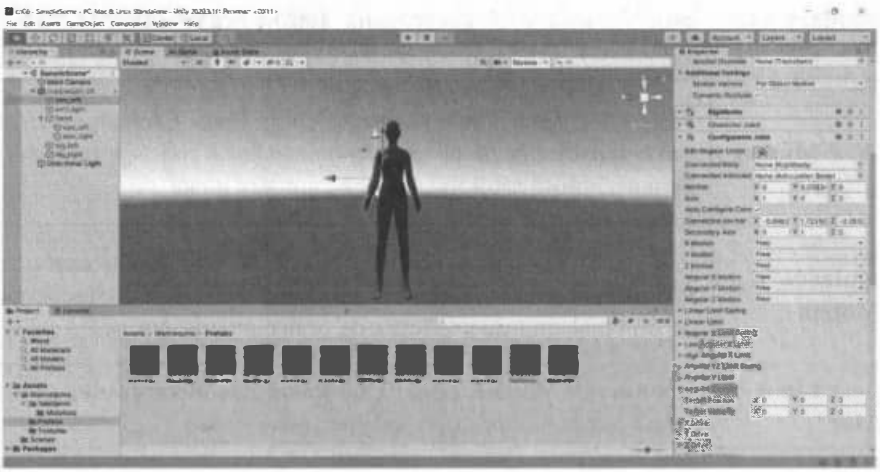


Рис. 6.8. Свойства настраиваемого сочленения

Таблица 6.10. Настраиваемые сочленения

Свойство	Функция
Connected Body	Другой объект Rigidbody, с которым соединен сустав. Вы можете установить это в *None*, чтобы указать, что соединение прикреплено к фиксированной позиции в пространстве, а не к другому Rigidbody
Anchor	Точка, где расположен центр сочленения. Все симуляции, основанные на физике, будут использовать эту точку как центр, при расчетах
Axis	Ось в локальном пространстве, задающая естественное вращение объекта, основанное на симуляции физики
Auto Configure Connected Anchor	Если включено, то позиция Connected Anchor будет автоматически рассчитана для соответствия глобальной позиции свойства привязки. Это поведение по умолчанию. Если свойство отключено, вы можете настроить положение подключенного якоря вручную

Connected Anchor	Ручная конфигурация позиции Connected Anchor
Secondary Axis	Вместе свойства Axis и Secondary Axis определяют локальную систему координат сочленения. Третья ось будет ортогональной этим двум
X, Y, Z Motion	Разрешает движение по осям X, Y или Z как *Свободное*, *Полностью заблокированное* или *Ограниченное* в соответствии с ограничивающими свойствами, описанными ниже
Angular X, Y, Z Motion	Разрешает вращение вокруг осей X, Y или Z как свободное, полностью заблокированное или ограниченное в соответствии с ограничивающими свойствами, описанными ниже
Linear Limit Spring	Пружинное усилие, прикладываемое для вытягивания объекта назад, когда он проходит крайнее положение
Spring	Сила пружины. Если это значение установлено в ноль, то предел будет непроходимым; значение, отличное от нуля, сделает предел эластичным
Damper	Уменьшение силы пружины пропорционально скорости движения сустава. Установка значения выше нуля позволяет суставу «гасить» колебания, которые в противном случае продолжались бы бесконечно
Linear Limit	Ограничивает линейное движение сустава (то есть перемещение на расстояние, а не вращение), определяемое как расстояние от начала соединения
Limit	Расстояние в мировых единицах от начала координат до предела
Bounciness	Сила отскока, прикладываемая к объекту, чтобы оттолкнуть его назад, когда он достигает предельного расстояния
Contact Distance	Контактная дистанция. Минимальное расстояние (между позицией соединения и лимитом), при котором будет применяться ограничение. Высокое значение снижает вероятность нарушения предела при быстром движении объекта, но это приводит к некоторому снижению производительности, поскольку предел будет пересчитываться физическим моделированием чаще

Angular X Limit Spring	Пружинный момент, применяется для вращения объекта назад, когда он проходит за предельный угол соединения
Spring	Крутящий момент пружины. Если это значение установлено в ноль, то предел будет непроходимым; значение, отличное от нуля, сделает предел эластичным
Damper	Уменьшение крутящего момента пружины пропорционально скорости вращения соединения. Установка значения выше нуля позволяет суставу «гасить» колебания, которые в противном случае продолжались бы бесконечно
Low Angular X Limit	Нижний предел вращения сустава вокруг оси X, указанный как угол от исходного вращения сустава
Limit	Ограничивает угол
Bounciness	Крутящий момент, приложенный к объекту, когда его вращение достигает предельного угла
Contact Distance	Контактная дистанция – между углом соединения и лимитом, на которой лимит будет применяться. Высокое значение снижает вероятность нарушения предела при быстром движении объекта, но это приводит к некоторому снижению производительности, поскольку предел будет пересчитываться физическим моделированием чаще
High Angular XLimit	Похоже на свойство Low Angular X Limit, описанное выше, но определяет верхний угловой предел вращения соединения, а не нижний предел
Angular YZ Limit Spring	Подобно свойству Angular X Limit Spring , но применяется к ротации вокруг осей Y и Z
Angular Y Limit	Аналогично свойствам Angular X Limit, но применяется к оси Y и рассматривает верхний и нижний пределы углов как одинаковые
Angular Z Limit	Аналогично свойства Angular X Limit, но применяется к оси Z и рассматривает верхний и нижний пределы углов как одинаковые
Target Position	Целевая позиция, в которую должна двигаться движущая сила сустава

Target Velocity	Желаемая скорость, с которой сустав (соединение) должен двигаться в целевое положение под действием движущей силы
XDrive	Движущая сила, которая перемещает соединение линейно вдоль его локальной оси X
Mode	Режим определяет, должен ли сустав перемещаться для достижения указанной позиции (Position), заданной скорости (Velocity) или того и другого
Position Spring	Усилие пружины, которое перемещает сустав к его целевому положению. Используется только в том случае, если режим движения установлен на **Velocity** или **Position and Velocity**
Position Damper	Уменьшение силы пружины пропорционально скорости движения сустава. Установка значения выше нуля позволяет суставу «гасить» колебания, которые в противном случае продолжались бы бесконечно. Используется только в том случае, если режим движения установлен на **Velocity** или **Position and Velocity**
Maximum Force	Сила, используемая для ускорения соединения к его целевой скорости. Используется только в том случае, если режим движения установлен на **Velocity** или **Position and Velocity**
YDrive	Аналогично свойству X Drive, описанному выше, но применяется к оси Y соединения
ZDrive	Аналогично свойству X Drive, описанному выше, но применяется к оси Z соединения
Target Rotation	Целевое вращение, к которому должен стремиться привод соединения (сустава). Указывается в виде кватерниона[1]
Target Angular Velocity	Угловая скорость, к которой должно стремиться вращательное соединение. Задается в виде вектора, длина которого определяет скорость вращения, а направление – ось вращения
Rotation Drive Mode	Способ, которым движущая сила будет применена к объекту, чтобы повернуть его к целевой ориентации. Если режим установлен в **X and YZ**, крутящий момент будет применяться к этим осям, как определено в свойствах **Angular X/YZ**, описанных ниже. Если используется режим **Slerp**, то свойства **Slerp Drive** будут определять крутящий момент привода

Angular X Drive	Определяет, как соединение будет вращаться под действием крутящего момента вокруг его локальной оси X. Используется только в том случае, если для свойства **Rotation Drive Mode**, описанного выше, установлено значение X & YZ
Mode	Режим определяет, должно ли соединение двигаться, чтобы достичь заданного углового положения (Position), заданной угловой скорости (Velocity) или обоих
Position Spring	Крутящий момент пружины, которая вращает соединение в направлении его целевого положения. Используется только в том случае, если режим движения установлен на **Velocity** или **Position and Velocity**
Position Damper	Уменьшение крутящего момента пружины пропорционально скорости движения сустава. Установка значения выше нуля позволяет суставу «гасить» колебания, которые в противном случае продолжались бы бесконечно. Используется только в том случае, если режим движения установлен на **Velocity** или **Position and Velocity**
Maximum Force	Крутящий момент, используемый для ускорения сустава к его целевой скорости. Используется только в том случае, если режим движения установлен на **Velocity** или **Position and Velocity**
Angular YZDrive	Работает аналогично свойству **Angular X Drive**, описанному выше, но применяется к осям Y и Z
Slerp Drive	Определяет, как шарнир будет вращаться под действием крутящего момента вокруг всех локальных осей. Используется только в том случае, если для свойства **Rotation Drive Mode**, описанного выше, установлено значение **Slerp**

Mode	Режим определяет, должно ли сочленение двигаться, чтобы достичь заданного углового положения (**Position**), заданной угловой скорости (**Velocity**) или обоих
Position Spring	Крутящий момент пружины, который вращает шарнир в направлении его целевого положения. Используется только в том случае, если режим движения установлен на **Velocity** или **Position and Velocity**
Position Damper	Уменьшение крутящего момента пружины пропорционально скорости движения сустава. Установка значения выше нуля позволяет суставу «гасить» колебания, которые в противном случае продолжались бы бесконечно. Используется только в том случае, если режим движения установлен на **Velocity** или **Position and Velocity**
Maximum Force	Крутящий момент, используемый для ускорения сустава к его целевой скорости. Используется только в том случае, если режим движения установлен на **Velocity** или **Position and Velocity**
Projection Mode	Определяет, как соединение будет возвращено к своим ограничениям, когда оно неожиданно выйдет за их пределы (например, из-за того, что физический движок не может согласовать текущую комбинацию сил в симуляторе). Возможные значения – **None** (никак) и **Position and Rotation** (позиция и вращение)
Projection Distance	Расстояние, на которое сустав должен выйти за пределы своих ограничений, прежде чем физический движок попытается вернуть его в приемлемое положение
Projection Angle	Угол сочленения должен повернуться за пределы установленного ограничения, прежде чем физический движок попытается вернуть его в приемлемое положение
Configured in World Space	Должны ли значения, установленные различными свойствами цели и диска, рассчитываться в мировом пространстве, а не в локальном пространстве объекта?
Swap Bodies	Если этот параметр включен, соединение будет вести себя так, как если бы компонент был присоединен к подключенному жесткому телу (т.е. к другому концу соединения)
Break Force	Если соединение выталкивается за пределы своих ограничений силой, превышающей это значение, то соединение будет постоянно «сломано» и удалено

Break Torque	Если шарнир поворачивается за пределы своих ограничений крутящим моментом, превышающим это значение, то шарнир будет постоянно «сломан» и удален
Enable Collision	Должен ли объект с сочленением иметь возможность сталкиваться с подключенным объектом (в отличие от простого прохождения друг через друга)?
Enable Preprocessing	Включает препроцессинг. Если предварительная обработка отключена, то определенные «невозможные» конфигурации соединения будут поддерживаться более стабильными, а не выходить из-под контроля

Как и другие типы сочленений, Configurable Joins позволяет ограничивать движение объекта, а также приводить его к целевой скорости или положению, используя силы. Данный тип соединения – самый гибкий и позволяет реализовать все возможные варианты.

Вы можете ограничить поступательное движение и вращение на каждой оси независимо, используя свойства **X, Y, Z Motion** и **X, Y, Z Rotation**. Если включен параметр **Configured In World Space**, то движения будут ограничиваться осями мира, а не локальными осями объекта. Каждое из этих свойств может быть установлено в **Locked**, **Limited** или **Free**:

- **Locked** – закрытая (*locked*) ось не разрешает, какого либо движения вообще. Например, если вы заблокировали ось Y, объект не сможет двигаться вверх или вниз
- **Limited** – ограниченная (*limited*) ось позволяет свободное перемещение между предварительно определенными пределами. Например, можно ограничить движение танковой башни, ограничив ее Y-вращение определенным диапазоном углов
- **Free** – разрешает любое значение

Вы можете ограничить поступательное перемещение, используя свойство **Linear Limit**, определяющее максимальное расстояние, на которое сустав (соединение) может перемещаться от своей исходной точки (измеряется по каждой оси отдельно). Например, вы можете ограничить шайбу для воздушного хоккея, заблокировав соединение по оси **Y** (в мировом пространстве),

оставив ее свободным по оси **Z** и установив предел для оси **X**, чтобы он соответствовал ширине стола; шайба будет вынуждена оставаться в игровой зоне. Чтобы стало понятнее все сказанное только что, посмотрите на рис. 6.9.

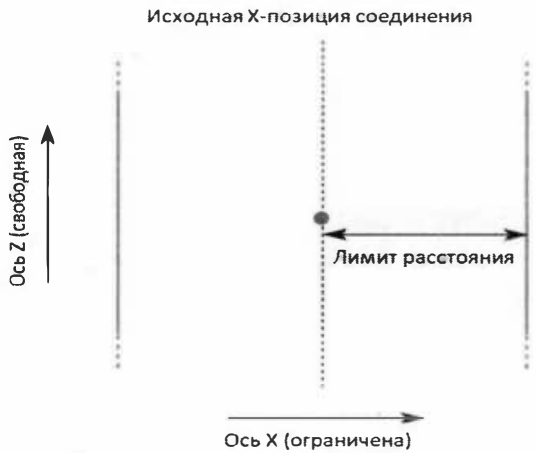


Рис. 6.9. Ограничение движение по осям

Вы также можете ограничить вращение, используя свойства *Angular Limit*. В отличие от линейного предела, они позволяют указывать разные предельные значения для каждой оси. Кроме того, вы также можете определить отдельные верхний и нижний пределы угла поворота для оси **X** (две другие оси используют один и тот же угол по обе стороны от исходного вращения). Например, вы можете построить «качающийся стол», используя плоскую плоскость с ограниченным соединением, чтобы позволить небольшой наклон в направлениях **X** и **Z**, оставляя Y-вращение заблокированным.

6.8.4. Мастер тряпичной куклы

Если у вас есть персонаж, то вы можете настроить его движения вручную, а можете использовать мастер тряпичной куклы, встроенный в Unity. Данный мастер позволяет быстро создать свой Ragdoll (объект в виде тряпичной куклы, которая подчиняется законам физики).

Все, что вам просто нужно перетащить части тела на соответствующие поля свойств в мастере. Затем нужно выбрать **Create** и Unity автоматически сгенерирует все коллайдеры (*Collider*), твердые тела (*Rigidbody*) и сочленения (*Joint*), которые создадут для вас Ragdoll.

«Тряпичные куклы» используют меши с привязкой к костям (*skinned meshes*), то есть меш персонажа, оснащенный костями в пакете 3D моделирования. Поэтому, вы должны построить ragdoll персонажей в пакетах 3D моделирования, таких как Maya или Cinema4D.

После того как вы создали и оснастили вашего персонажа, сохраните ассет, как обычно в вашей папке проекта. Когда вы переключитесь в Unity, вы увидите файл ассета персонажа. Выберите этот файл, в инспекторе появится диалог **Import Settings**. Убедитесь, что **Mesh Colliders** не включено.

Создайте экземпляр персонажа перетаскиванием его из окна **Project** в окно **Hierarchy**. Разверните его иерархию трансформации (Transform Hierarchy) нажатием на маленькую стрелочку слева от названия экземпляра в иерархии. Теперь вы готовы начать присваивать ваши части ragdoll'a.

Откройте Ragdoll Wizard выбрав **GameObject, 3D Object, Ragdoll** из строки меню. Теперь вы увидите сам мастер (рис. 6.10). Заполните свойства мастера и нажмите кнопку **Create**. Поля имеют значение (если вы не дружите с английским):

- Pelvis – таз;
- Left Hips – левое бедро;
- Left Knee – левое колено;
- Left Foot – левая нога;
- Right Hips – правое бедро;
- Right Knee – правое колено;
- Right Foot – правая нога;
- Left Arm – левая рука;
- Left Elbow – левый локоть;
- Right Arm – правая рука;
- Right Elbow – правый локоть;
- Middle Spine – средний позвоночник;
- Head – голова;
- Total Mass – общая масса;
- Strength – сила.

Рис. 6.10. Окно Ragdoll Wizard

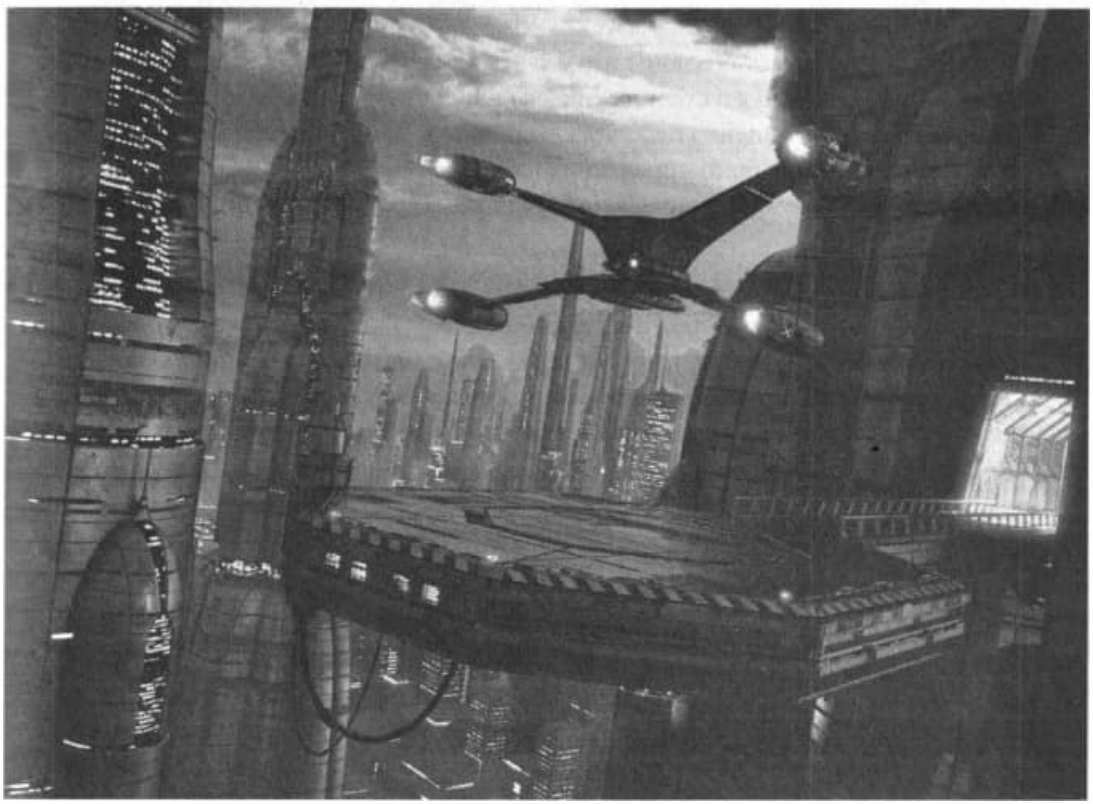
Финальный шаг – сохраните настроенный ragdoll в качестве префаба. Выберите **Assets, Create, Prefab Variant** из строки меню. Вы увидите файл с названием New Prefab, появившийся в окне **Project**. Переименуйте его в **Ragdoll Prefab**. Перетащите экземпляр ragdoll персонажа из окна **Hierarchy** на файл Ragdoll Prefab. Теперь у вас есть полностью настроенный, пригодный к повторному использованию персонаж для использования столько раз в вашей игре, сколько вам угодно.

Использовать мастер просто, но только при условии, что у вас есть модель, пригодная для использования в виде Ragdoll. В противном случае придется настраивать соединения с помощью компонентов Joint.

Далее будут рассмотрены основы скриптинга, что позволит вам писать собственные скрипты для Unity.

Глава 7.

Основы скриптинга



Unity хоть и упрощает создание игр, но все же без написания скриптов никуда не денешься. Мы уже сталкивались с написанием сценариев – когда делали сценарий перемещения персонажа, сценарий слежения за камерой, далее будут показаны сценарии управления главным меню и экраном настроек. В следующих главах будет продемонстрирована серия скриптов управления персонажем.

Скриптинг – необходимая составляющая всех игр. Даже самые простые игры нуждаются в скриптах для реакции на действия игрока и организации событий геймплея. Кроме того, скрипты могут быть использованы для создания графических эффектов, управления физическим поведением объектов или реализации пользовательской ИИ системы для персонажей игры.

Цель данной главы – не научить вас писать код с нуля, а объяснить основные понятия, которые вам нужно знать при программировании в Unity.

7.1. Создание скриптов

Поведение игровых объектов определяется прикрепленными к ним компонентами. Компоненты очень гибкие и с их помощью можно реализовать самые разные возможности. Но рано или поздно вы заметите, что возможностей компонентов вам уже недостаточно. И тут на помощь приходит

скриптинг – процесс написания сценариев на языке C#. Ранее в Unity можно было использовать собственный диалект – UnityScript – он был похож на JavaScript, но разработчики самой Unity отказались от него в пользу C#.

В этой главе мы не будем рассматривать синтаксис C# – вы должны знать хотя бы основы программирования на этом языке. Если вы круглый новичок, тогда поищите другую книгу, посвященную C# – таких книг гораздо больше, чем книг по Unity.

Создать скрипт, как вы уже знаете, можно с помощью команды меню **Assets, Create, C# Script**. Новый сценарий появится в окне обозревателя, и вы сможете переименовать его по своему усмотрению.

После того, как сценарий создан, дважды щелкните по нему, чтобы запустить редактор кода. Если у вас установлен только Visual Studio, то будет открыта эта IDE для редактирования кода сценария. Но я рекомендую установить редактор Visual Studio Code (рис. 7.1) – тогда не нужно будет запускать целую IDE для редактирования простенького сценария.

Рис. 7.1. Редактор Visual Studio Code

7.2. Структура файла скрипта. Присоединение скрипта к объекту

По умолчанию содержимое скрипта будет примерно таким, как показано в лист. 7.1. Почем примерно таким – потому, что название класса будет зависеть от имени сценария, которое вы введете при его создании.

Листинг 7.1. Пример сценария Unity

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Используется для инициализации
    void Start () {

    }

    // Вызывается 1 раз за кадр
    void Update () {

    }
}
```

Скрипт взаимодействует с внутренними механизмами Unity за счет создания класса, наследованного от встроенного класса, называемого MonoBehaviour. Вы можете думать о классе как о своего рода плане для создания нового типа компонента, который может быть прикреплен к игровому объекту. Каждый раз, когда вы присоединяете скриптовый компонент к игровому объекту, создается новый экземпляр объекта, определенный планом. Имя класса берется из имени, которое вы указали при создании файла. Имя класса и имя файла должны быть одинаковыми, для того, чтобы скриптовый компонент мог быть присоединен к игровому объекту.

Метод Update() – это место для размещения кода, который будет обрабатывать обновление кадра для игрового объекта. Это может быть движение, срабатывание действий и ответная реакция на ввод пользователя, в основном все, что должно быть обработано с течением времени в игровом процессе. Чтобы позволить методу Update() выполнять свою работу, часто бывает полезно инициализировать переменные, считать свойства и осуществить связь с другими игровыми объектами до того, как будут совершены какие-либо действия. Метод Start() будет вызван Unity до начала игрового процесса (т.е.

до первого вызова функции Update()), и это идеальное место для выполнения инициализации.

> **Внимание!** Вы можете быть удивлены, что инициализация объекта выполняется не в функции-конструкторе. Это потому, что создание объектов обрабатывается редактором и происходит не в начале игрового процесса, как вы могли бы ожидать. Если вы попытаетесь определить конструктор для скриптового компонента, он будет мешать нормальной работе Unity и может вызвать серьезные проблемы с проектом.

Скрипт определяет только план компонента и, таким образом, никакой его код не будет активирован до тех пор, пока экземпляр скрипта не будет присоединен к игровому объекту. Вы можете прикрепить скрипт перетаскиванием ассета скрипта на игровой объект в панели **Hierarchy** или через окно **Inspector** выбранного игрового объекта. Имеется также подменю **Scripts** в меню **Component**, которое содержит все скрипты, доступные в проекте, включая те, которые вы создали сами.

Сейчас мы еще раз рассмотрим этот процесс. Вы только что создали скрипт по умолчанию. Отредактируйте его метод Start() так:

```
void Start()
{
    Debug.Log("Всем привет!");
}
```

Создайте пустой игровой объект с помощью команды меню **Gameobject, Create Empty**. Перетащите на него созданный вами скрипт (рис. 7.2).

Рис. 7.2. Сценарий MainPlayer.cs назначен ну стому игровому объекту GameObject

Запустите игру, нажав кнопку **Play**. Переключитесь на вкладку **Console** – вы увидите сообщение, сгенерированное с помощью метода Debug.Log(), см рис. 7.3.

Рис. 7.3. Всем привет!

7.3. Переменные в сценарии

При создании сценария вы создаете свой собственный новый тип компонента, который можно прикрепить к игровым объектам, как и любой другой компонент.

Подобно тому, как другие компоненты часто имеют свойства, которые можно редактировать в инспекторе, вы также можете разрешить редактирование значений в вашем скрипте из инспектора. Рассмотрим небольшой пример и добавим в наш сценарий переменную:

```
public string userName;
// Вызывается перед обновлением кадра
void Start()
{
     Debug.Log("Всем привет! Меня зовут " + userName);
}
```

После того, как вы сохраните этот сценарий, в инспекторе появится новое свойство, позволяющее установить значение этой переменной (рис. 7.4).

Рис. 7.4. Новое поле в инспекторе объектов

Внимание! Чтобы переменная появилась в инспекторе объектов, вы обязательно должны объявить ее как public.

Unity позволяет изменять значение переменных скрипта в запущенной игре. Это очень полезно чтобы увидеть эффекты от изменений сразу же, без остановки и перезапуска. Когда проигрывание оканчивается, значения

переменных сбрасываются в то состояние, которое они имели до нажатия кнопки **Play**. Это гарантирует, что вы свободно можете играть с настройками объектов, не боясь что-то испортить.

7.4. Программное управление игровыми объектами

7.4.1. Обращение к компонентам

Довольно часто скрипту нужно обратиться к компоненту. Чаще всего нужен доступ к компонентам, которые присоединены к тому же игровому объекту. Компонент на самом деле является экземпляром класса, поэтому первым шагом будет получение ссылки на экземпляр компонента, с которым вы хотите работать. Сделать это можно с помощью метода GetComponent(). Пример кода:

```
void Start () {
    Rigidbody rb = GetComponent<Rigidbody>();
}
```

Как только у вас есть ссылка на экземпляр компонента, вы можете устанавливать значения его свойств, тех же, которые вы можете изменить в окне **Inspector**:

```
void Start () {
    Rigidbody rb = GetComponent<Rigidbody>();

    // Изменяем массу твердого тела
    rb.mass = 50;
}
```

Программно разработчик может не только устанавливать свойства компонента, но и вызывать его методы – такая возможность недоступна в инспекторе:

```
void Start () {
    Rigidbody rb = GetComponent<Rigidbody>();
```

```
    // Применяем силу к Rigidbody.
    rb.AddForce(Vector3.up * 50f);
}
```

Помните, что один и тот же скрипт вы можете присоединять к разным объектам.

7.4.2. Обращение к другим объектам

Вторая ситуация – обращение из скрипта к другим объектам. В следующих главах будет показано, как присоединить к скрипту AudioMixer и управлять его параметрами. В этой главе мы покажем теоретическую сторону такого действия. Создайте в классе переменную:

```
public GameObject enemy;
```

Как только вы сохраните сценарий, в инспекторе появится свойство *Enemy*. Создайте пустой GameObject, переименуйте его в Enemy1 (или в любое другое имя) и перетащите на свойство *Enemy* – так вы назначите объект скрипту и скрипт сможет получить доступ к его свойствам и методам (рис. 7.5).

Рис. 7.5. Свойству назначен объект

Соединение объектов через переменные наиболее полезно, когда вы имеете дело с отдельными объектами, имеющими постоянную связь. Вы можете использовать массив для хранения связи с несколькими объектами одного типа, но связи все равно должны быть заданы в редакторе Unity, а не во время выполнения. Часто удобно находить объекты во время выполнения, и Unity предоставляет два основных способа сделать это, описанных ниже.

7.4.3. Поиск дочерних объектов

Иногда игровая сцена может использовать несколько объектов одного типа, таких как враги, путевые точки и препятствия. Может возникнуть необхо-

димость отслеживания их в определенном скрипте, который управляет или реагирует на них (например, все путевые точки могут потребоваться для скрипта поиска пути). Можно использовать переменные для связывания этих объектов, но это сделает процесс проектирования утомительным, если каждую новую путевую точку нужно будет перетащить в переменную в скрипте. Аналогично, при удалении путевой точки придется удалять ссылку на отсутствующий объект. В случаях, наподобие этого, чаще всего удобно управлять набором объектов, сделав их дочерними одного родительского объекта. Дочерние объекты могут быть получены, используя компонент **Transform** родителя (так как все игровые объекты неявно содержат Transform):

```
using UnityEngine;

public class WaypointManager : MonoBehaviour {
    public Transform[] waypoints;

    void Start() {
        waypoints = new Transform[transform.childCount];
        int i = 0;

        foreach (Transform t in transform) {
            waypoints[i++] = t;
        }
    }
}
```

Вы можете также найти заданный дочерний объект по имени, используя метод Transform.Find():

```
transform.Find("Enemy");
```

Это может быть полезно, когда объект содержит дочерний элемент, который может быть добавлен или удален в игровом процессе. Хороший пример – оружие, которое может быть подобрано и выброшено.

Объект или коллекция объектов могут быть также найдены по их тегу, используя методы GameObject.FindWithTag() и GameObject.FindGameObjectsWithTag():

```
GameObject player;
GameObject[] enemies;
```

```
void Start() {
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}
```

7.4.4. Создание и уничтожение игровых объектов

Вместо того чтобы создавать врагов игрока вручную, вы можете поручить это сценарию. Для создания игрового объекта используется метод Instantiate():

```
public GameObject enemy;

void Start() {
    for (int i = 0; i < 5; i++) {
        Instantiate(enemy);
    }
}
```

Заметьте, что дублирующийся объект не обязан присутствовать на сцене. Гораздо чаще используется префаб, который был перетащен на открытую переменную (public variable) из файлов проекта в панели **Project**. Также, копируя игровой объект (GameObject), вы копируете все компоненты оригинального объекта.

Также есть функция **Destroy**, которая уничтожит объект после того, как загрузка кадра будет завершена или опционально после короткой паузы (пауза задается вторым параметром):

```
void OnCollisionEnter(Collision otherObj) {
    if (otherObj.gameObject.tag == "Zombie") {
        Destroy(gameObject,.5f);
    }
}
```

Обратите внимание, что функция **Destroy** может уничтожать отдельные компоненты без влияния на сам объект. Частая ошибка – писать что-то вроде этого:

```
Destroy(this);
```

На самом деле такой вызов уничтожит только вызывающий скриптовый компонент, вместо того, чтобы уничтожить игровой объект, к которому присоединен этот скрипт.

7.5. Методы событий

7.5.1. Основные методы

Unity периодически передает управление скрипту при вызове определенных объявленных в нем функций. Как только функция завершает исполнение, управление возвращается обратно к Unity. Эти функции известны как методы событий, т.к. их активирует Unity в ответ на события, которые происходят в процессе игры. Вы уже знакомы как минимум с двумя функциями событий – Start() и Update(). Но это далеко не единственные методы, которые вы можете использовать. Основными методами являются следующие:

- FixedUpdate()
- LateUpdate()
- OnGUI()
- OnDisable()
- OnEnable()

Игра – это что-то вроде анимации, в которой кадры генерируются на ходу. Ключевым моментом в программировании игр является изменение позиции, состояния и поведения объектов в игре прямо перед прорисовкой кадра. Такой код в Unity обычно размещают в функции Update(). Метод Update() вызывается перед прорисовкой кадра и перед расчетом анимаций. Пример:

```
void Update() {
    float distance = speed * Time.deltaTime * Input.
GetAxis("Horizontal");
    transform.Translate(Vector3.right * distance);
}
```

Физический движок Unity также обновляется фиксированными по времени шагами, аналогично рендерингу кадра. Метод FixedUpdate() вызывается прямо перед каждым обновлением физических данных. Т.к. обновление физики и кадра происходит не с одинаковой частотой, то вы получите более точные результаты от кода физики, если поместите его в функцию FixedUpdate, а не в *Update*.

```
void FixedUpdate() {
```

```
    Vector3 force = transform.forward * dForce * Input.
GetAxis("Vertical");
    rigidbody.AddForce(force);
}
```

Также иногда полезно иметь возможность внести дополнительные изменения в момент, когда у всех объектов в сцене отработали функции Update() и FixedUpdate() и рассчитались все анимации. В качестве примера, камера должна оставаться привязанной к целевому объекту; подстройка ориентации камеры должна производиться после того, как целевой объект сместился. Другим примером является ситуация, когда код скрипта должен переопределить эффект анимации (допустим, заставить голову персонажа повернуться к целевому объекту в сцене). В ситуациях такого рода можно использовать функцию LateUpdate.

```
void LateUpdate() {
    Camera.main.transform.LookAt(target.transform);
}
```

В Unity есть система для прорисовки элементов управления GUI поверх всего происходящего в сцене и реагирования на клики по этим элементам. Этот код обрабатывается несколько иначе, нежели обычное обновление кадра, так что он должен быть помещен в функцию OnGUI, которая будет периодически вызываться.

```
void OnGUI() {
    GUI.Label(labelRect, "Game Over");
}
```

Функция OnDisable() вызывается, когда объект отключается, а OnEnable() – когда включается. В дальнейших главах мы построим обработчики кнопок Settings (вызывает экран с настройками) и Back (вызывает главное меню) посредством установки обработчиков прямо в инспекторе. Если бы мы писали код, то в функции OnEnable() для объекта Settings нужно было отключить объект MainMenu, чтобы он не был виден на экране, а в функции OnEnable() для MainMenu – отключить объект Settings.

7.5.2. Порядок выполнения функций событий

В Unity функции событий вызываются не просто так, а в определенном порядке:

1. **Reset()** – вызывается для инициализации свойств скрипта, когда он только присоединяется к объекту. Используется редко.
2. **Первая загрузка сцены**. Следующие функции вызываются при запуске сцены (один раз для каждого объекта на сцене):
   - Awake() – всегда вызывается до любых функций Start() и после того, как префаб был вызван в сцену (если GameObject неактивен на момент старта, Awake не будет вызван, пока GameObject не будет активирован, или функция в каком-нибудь прикрепленном скрипте не вызовет Awake)
   - OnEnable() – вызывается, если объект активен – сразу после включения объекта
   - OnLevelWasLoaded() – вызывается, чтобы проинформировать, что был загружен новый уровень игры
3. **Первое обновление кадра**:
   - Start() – вызывается до обновления первого кадра, если скрипт включен
4. **Между кадрами**:
   - OnApplicationPause() – эта функция вызывается в конце кадра, во время которого вызывается пауза, что эффективно между обычными обновлениями кадров. Один дополнительный кадр будет выдан после вызова OnApplicationPause, чтобы позволить игре отобразить графику, которая указывает на состояние паузы
5. **В процессе игры** – вызываются методы Update(), FixedUpdate(), LateUpda(), см. далее.
6. **В процессе рендеринга**:
   - OnPreCull() – вызывается до того, как камера отсечет сцену. Отсечение определяет, какие объекты будут видны в камере. OnPreCull() вызывается прямо перед тем, как начинается отсечение
   - OnBecameVisible/OnBecameInvisible() – вызывается тогда, когда объект становится видимым/невидимым любой камере
   - OnWillRenderObject() – вызывается один раз для каждой камеры, если объект в поле зрения
   - OnPreRender() – вызывается перед тем, как камера начнет рендерить сцену

- OnRenderObject() – вызывается, после того, как все обычные рендеры сцены завершатся
- OnPostRender() – вызывается после того, как камера завершит рендер сцены
- OnRenderImage (доступен только в Pro-версии) – вызывается после завершения рендера сцены, для возможности пост-обработки изображения экрана
- OnGUI() – вызывается несколько раз за кадр и отвечает за элементы интерфейса (GUI). Сначала обрабатываются события макета и раскраски, после чего идут события клавиатуры/мышки для каждого события
- OnDrawGizmos() – используется для прорисовки *гизмо* в окне Scene View в целях визуализации. На практике используется очень редко

7. **При разрушении объекта**:
   - OnDestroy() – вызывается после всех обновлений кадра, в последнем кадре объекта, пока он еще существует
8. **При выходе**:
   - OnApplicationQuit() – вызывается для всех игровых объектов перед тем, как приложение закрывается. В редакторе вызывается тогда, когда игрок останавливает игровой режим
   - OnDisable() – вызывается, когда объект отключается или становится неактивным

> **Примечание**. Для объектов, добавленных в сцену сразу, функции Awake() и OnEnable() для всех скриптов будут вызваны до вызова Start(), Update() и т.д. Естественно, для объектов вызванных во время игрового процесса такого не будет.

В процессе игры основная функция, которую вы будете использовать – это функция Update(), описанная ранее – большая часть задач выполняется именно в этой функции. Что касается FixedUpdate(), часто случается, что этот метод вызывается чаще, чем Update(). Метод FixedUpdate() может быть вызван несколько раз за кадр, если FPS низок и функция может быть и вовсе не вызвана между кадрами, если FPS высок. Все физические вычисления и обновления происходят сразу после FixedUpdate(). При применении расчетов передвижения внутри FixedUpdate(), вам не нужно умножать ваши

значения на Time.deltaTime. Потому что FixedUpdate вызывается в соответствии с надежным таймером, независящим от частоты кадров.

Метод LateUpdate() вызывается раз в кадр, после завершения *Update*. Любые вычисления, произведенные в Update(), будут уже выполнены на момент начала LateUpdate(). Часто LateUpdate() используют для преследующей камеры от третьего лица. Если вы перемещаете и поворачиваете персонажа в Update(), вы можете выполнить все вычисления перемещения и вращения камеры в LateUpdate(). Это обеспечит то, что персонаж будет двигаться до того, как камера отследит его позицию.

7.6. Сопрограммы

Функции не всегда позволяют реализовать желаемое. Дело в том, что если вы вызвали функцию, то она должна выполниться до конца, прежде чем она вернет какое-либо значение. Фактически, это означает, что любые действия, происходящие в функции, должны выполниться в течение одного кадра; вызовы функций не пригодны для, скажем, процедурной анимации или любой другой временной последовательности. В качестве примера мы рассмотрим задачу по уменьшению прозрачности объекта до его полного исчезновения. Вот пример неправильного кода:

```
void ObjFade() {
    for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
    }
}
```

Если вы попробуете эту функцию на практике, то она не даст вам желаемого эффекта. Для скрытия объекта нам нужно было постепенно уменьшить прозрачность, а значит иметь некоторую задержку для того, чтобы были отображены промежуточные значения параметра. Однако функция выполняется в полном объеме, за одно обновление кадра. Промежуточные значения, увы, не будут видны, и объект исчезнет мгновенно.

С подобными ситуациями можно справиться, добавив в функцию Update() код, изменяющий прозрачность кадр за кадром. Однако для подобных задач зачастую удобнее использовать корутины или сопрограммы.

Сопрограмма похожа на функцию, которая может приостановить выполнение и вернуть управление Unity, но затем продолжить с того места, где остановилась в следующем кадре. В C # сопрограмма объявляется так:

```
IEnumerator Fade() {
    for (float f = 1f; f >= 0; f -= 0.1f) {
        Color c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield return null;
    }
}
```

По сути, сопрограмма – это функция, объявленная с типом IEnumerator и оператором **yield return**, включенным где-то в теле. Оператор **yield** – это точка, в которой выполнение будет приостановлено и возобновлено в следующем кадре. Чтобы запустить сопрограмму, вам нужно использовать функцию StartCoroutine():

```
void Update() {
    if (Input.GetKeyDown("f")) {
        StartCoroutine("Fade");
    }
}
```

Можно заметить, что счетчик цикла в функции Fade сохраняет правильное значение во время работы корутины. Фактически, любая переменная или параметр будут корректно сохранены между вызовами оператора **yield**.

На практике использование сопрограмм будет продемонстрировано в последней части этой книги, когда мы будем создавать реальную игру.

7.7. Атрибуты

Атрибуты (Attributes) – это специальные маркеры, которые могут быть помещены перед классом, свойствами или функциями в скрипте, чтобы указать особое поведение. Например, атрибут HideInInspector может быть добавлен перед объявлением свойства для предотвращения отображения этого свойства в инспекторе, даже если оно публичное. Пример:

```
[HideInInspector]
public float strength;
```

Атрибут **Range** позволяет указать диапазон значений, который может принимать переменная:

```
[Range(float,float)]
float someFloat;
```

Атрибут **Tooltip** позволяет установить подсказку для переменной. Подсказка будет отображена в инспекторе:

```
[Tooltip(string)]
public string anyVariable;
```

Дополнительную информацию об атрибутах можно получить в документации по C#:

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/

7.8. Важные классы

При написании скриптов вам необходимо ознакомиться с тремя базовыми классами в Unity:

- **MonoBehaviour** – Базовый класс для всех новых сценариев Unity, предоставляет вам список всех функций и событий, доступных для стандартных сценариев, прикрепленных к игровым объектам. Если вы ищете какое-либо взаимодействие или контроль над отдельными объектами в вашей игре, начните с этого класса. Подробная справка доступна по адресу https://docs.unity3d.com/ru/current/ScriptReference/MonoBehaviour.html
- **Transform** – у каждого игрового объекта есть три важных характеристики – позиция, ротация и масштаб. Все эти характеристики определяются компонентом Transform. Соответственно, если вам нужно перемещать, вращать или изменять масштаб объекта в игре, вам нужно изучить этот класс. См. https://docs.unity3d.com/ru/current/ScriptReference/Transform.html
- **Rigidbody** – для большинства элементов игрового процесса физический движок предоставляет самый простой набор инструментов для перемещения объектов, обнаружения триггеров и столкновений. Данный класс предоставляет все свойства и функции, необходимые для работы с фи-

зикой в Unity. Описание данного класса доступно по адресу *https://docs.unity3d.com/ru/current/ScriptReference/Rigidbody.html*

7.9. Обработка исключений

Обработка исключений – стандартная практика при программировании на C# и если вы знаете этот язык программирования, вы должны знать, как работать с исключениями. В контексте Unity вам нужно обрабатывать исключение NullReferenceExpection, которое происходит, когда вы пытаетесь получить доступ к ссылочной переменной, которая не ссылается ни на один объект. Если ссылочная переменная не ссылается на объект, она будет рассматриваться как **null**. Среда выполнения сообщит вам, что вы пытаетесь получить доступ к несуществующему объекту именно с помощью NullReferenceException.

Рассмотрим следующий пример кода:

```
// Используйте для инициализации
    void Start () {
        GameObject go = GameObject.Find("player");
        Debug.Log(go.name);
    }
```

Но что делать, если у нас нет объекта с именем *player*? Вы получите исключение, и игра прекратит свою работу с ошибкой. Следующий код более правильный:

```
        GameObject go = GameObject.Find("player ");
        if (go) {
            Debug.Log(go.name);
        } else {
            Debug.Log("Нет объекта с именем player");
        }
```

Самый правильный вариант – использовать блоки **try catch**:

```
void Start () {
        try {
            GameObject go = GameObject.Find("player ");
     Debug.Log(go.name);
        }
```

```
        catch (NullReferenceException ex) {
            Debug.Log("Нет объекта с именем player ");
        }
    }
```

7.10. Сохранение и загрузка игрового процесса

Во многих играх вы наверняка видели кнопки **Save** и **Load**, а в сленге геймеров даже есть такое выражение «засейвиться». Сохранения представляют собой файлы, в которых хранится информация о текущем состоянии игровых объектов:

- позиции игрока;
- уровне;
- мане;
- здоровье;
- опыте и так далее.

Чтобы можно было все это удобно преобразовать в файл, используется сериализация — специальный инструмент, позволяющий сохранить объект в формате JSON или XML. Лучше сохранять все в виде бинарных файлов, потому что так игроки не смогут изменить характеристики своего персонажа в обычном текстовом редакторе.

Чтобы сохранить данные (координаты, наличие предметов в инвентаре, здоровье), создается класс SaveData (лист. 7.2).

Листинг 7.2. Класс SaveData

```
//Обязательно нужно указать, что класс должен сериализоваться
[System.Serializable]
public class SaveData
{
   // Поля с параметрами игрока
   public float currHP;
   public float HP;

   public float currMP;
   public float MP;

   public float currXP;
   public float XP;
```

```
   public int level;

//В Unity позиция игрока записана с помощью класса Vector3, но
его //нельзя сериализовать. Чтобы обойти эту проблему, данные
о позиции //будут помещены в массив типа float.
   public float[] position;

   public SaveData(Character character) //Конструктор класса
   {
      //Получение данных, которые нужно сохранить
      HP = character.HP;
      currHP = character.currHP;

      level = character.level;

      position = new float[3] //Получение позиции
      {
         character.transform.position.x,
         character.transform.position.y,
         character.transform.position.z
      };
   }

}
```

Теперь напишем отдельный класс SaveLoad, который будет сохранять характеристики игрока (метод Save ()) и восстанавливать их (метод Load()).

Листинг 7.3. Класс SaveLoad

```
using UnityEngine;
//Библиотека для работы с файлами
using System.IO;
//Библиотека для работы бинарной сериализацией
using System.Runtime.Serialization.Formatters.Binary;

// Создание статичного класса позволит использовать
// методы без объявления его экземпляров
public static class SaveLoad {

// Путь к сохранению. Вы можете использовать любое расширение
   private static string path = Application.persistentDataPath
+ "/r.dat";
//Создание сериализатора
```

```
private static BinaryFormatter formatter = new
BinaryFormatter();

// Метод сохранения
   public static void Save(Character character)
   {

// Создание файлового потока
      FileStream fs = new FileStream (path, FileMode.Create);
// Получение данных
      SaveData data = new SaveData(character);
// Сериализация данных
      formatter.Serialize(fs, data);

      fs.Close(); //Закрытие потока

   }

//Метод загрузки
   public static SaveData Load ()
   {
      // Проверка существования файла сохранения
      if(File.Exists(path)) {
// Открытие файлового потока
         FileStream fs = new FileStream(path, FileMode.Open);
// Получение данных
         SaveData data = formatter.Deserialize(fs) as SaveData;

         fs.Close(); //Закрытие потока
         return data; //Возвращение данных
      }
      else
      {
         //Если файл не существует, будет возвращено null
         return null;
      }

   }
}
```

Осталось только в главном меню создать кнопки **Load** и **Save** и в обработчиках нажатия кнопок вызывать соответствующие методы, например:

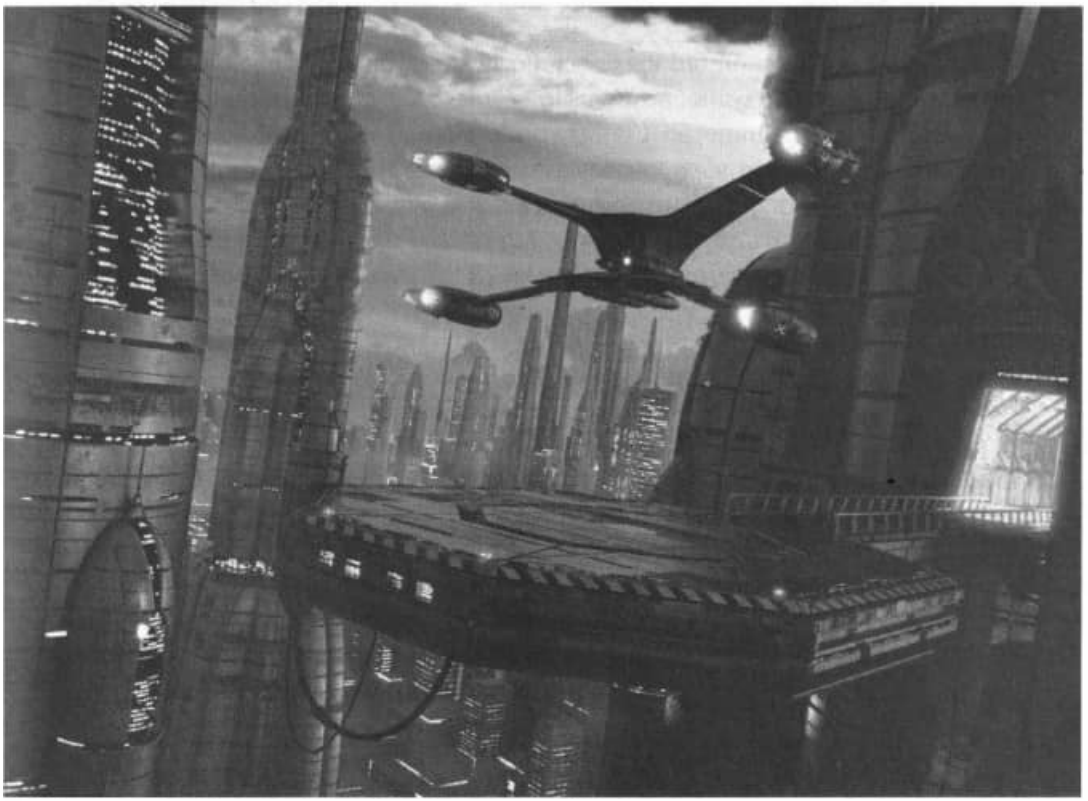
```
// Обработчик Load
// Читаем данные из файла
```

```
SaveData data = SaveLoad.Load();
if(!data.Equals(null)) //Если данные есть
{
   HP = data.HP;
   currHP = data.currHP;

   level = data.level;

   transform.position = new Vector3(data.position[0], data.
position[1], data.position[2]);
}
// Обработчик Save:
SaveLoad.Save();
```

На этом все. Далее мы поговорим о работе со звуком в Unity.

Глава 8.

Звук в Unity



Игра была бы неполной без какого-либо звука, будь то фоновая музыка или звуковые эффекты. Аудиосистема Unity является гибкой и мощной. Unity поддерживает большинство стандартных форматов аудиофайлов и имеет сложные функции для воспроизведения звуков в трехмерном пространстве, при необходимости с такими эффектами, как эхо и фильтрация. Unity также может записывать звук с любого доступного микрофона на компьютере пользователя для использования во время. В этой главе вы узнаете, как работать со звуком в Unity.

8.1. Что нужно знать о звуке в Unity

В реальной жизни звуки испускаются источниками звука и слышатся слушателями. Способ восприятия звука зависит от ряда факторов. Слушатель может сказать, в каком направлении идет звук, и также может судить о его отдаленности – по громкости и качеству. Быстро движущийся источник звука (например, падающая бомба или проезжающая полицейская машина) будет менять высоту звука при движении в результате эффекта Доплера. Кроме того, окружение будет влиять на способ отражения звука, поэтому голос внутри пещеры будет иметь эхо, а тот же голос на открытом воздухе – нет.

Для имитация эффекта положения, Unity требует, чтобы звуки исходили из аудиоисточников, прикрепленных к объектам. Издающиеся звуки затем

улавливаются аудиослушателем, подключенным к другому объекту, чаще всего к основной камере. Затем Unity может симулировать эффекты расстояния и положения источника от объекта слушателя и соответствующим образом воспроизвести их для пользователя. Относительная скорость объектов источника и слушателя также может использоваться для имитации эффекта Доплера для дополнительной реалистичности.

Однако Unity не может рассчитывать эхо-сигналы исключительно из геометрии сцены, но вы можете имитировать их, добавляя аудиофильтры к объектам. Например, вы можете применить фильтр **Echo** к звуку, который должен исходить из пещеры. В ситуациях, когда объекты могут перемещаться и иметь более сильное эхо, вы можете добавить зону реверберации к сцене. Типичный пример – включать автомобили, проезжающие через туннель. Если вы разместите зону реверберации внутри туннеля, то звуки двигателя машины будут слышны, когда они входят, и эхо будет затухать по мере выхода машин из туннеля.

Audio Mixer позволяет смешивать различные аудиоисточники, применять к ним эффекты и выполнять мастеринг. Кроме аудиоисточников есть еще и слушатели, которые выполняют роль микрофонов. Их можно использовать для записи звука с микрофона для последующего воспроизведения.

8.2. Поддерживаемые звуковые файлы

Unity может импортировать аудиофайлы в форматах AIFF, WAV, MP3 и Ogg так же, как и другие ресурсы – просто перетащите звуковой файл в область обозревателя проекта. При импорте аудиофайла создается аудиоклип, который затем можно перетащить на источник звука или использовать из сценария. Таблица 8.1 содержит список поддерживаемых форматов.

Таблица 8.1. Поддерживаемые звуковые форматы

Формат	**Расширения**
MPEG layer 3	.mp3
Ogg Vorbis	.ogg
Microsoft Wave	.wav
Audio Interchange File Format	.aiff / .aif
Ultimate Soundtracker module	.mod

Impulse Tracker module	.it
Scream Tracker module	.s3m
FastTracker 2 module	.xm

8.3. Обзор AudioMixer

AudioMixer – это компонент Unity, на который могут ссылаться источники звука (AudioSources), чтобы обеспечить более сложную маршрутизацию и микширование аудиосигнала, генерируемого из AudioSources.

Посредством AudioMixer смешивать различные источники звука, применять к ним эффекты, когда звук направляется из AudioSource в AudioListener.

Для отображения окна Audio Mixer выберите соответствующую команду из меню Window. В окне Audio Mixer (см. рис. 8.1) отображаются микшеры, группы и снапшоты. Группа Audio Mixer, по сути, является смесью аудио-сигналов. С помощью микшера вы можете регулировать громкость каждого типа аудио-сигнала, например, можно сделать тише фоновую музыку, но громче аудио-эффекты (или наоборот). Также можно применить определенные звуковые эффекты к каждой составляющей группы.

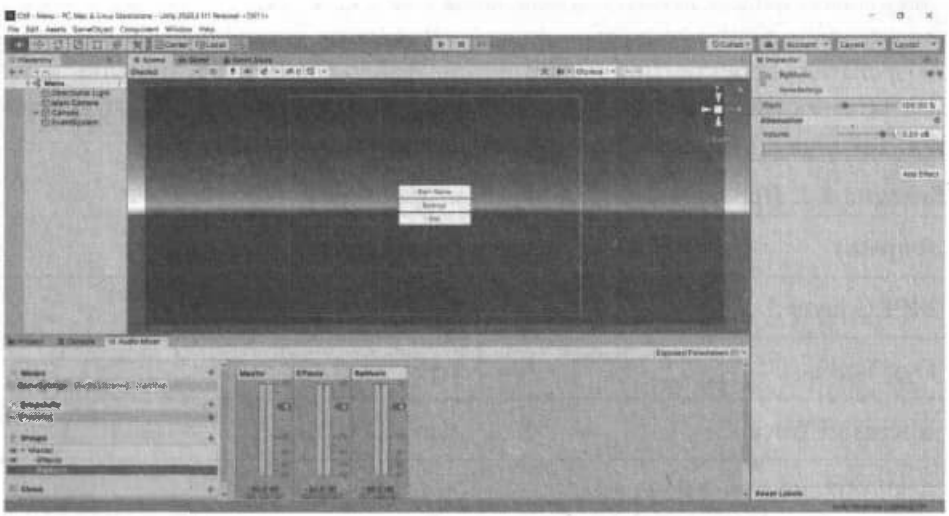


Рис. 8.1. Окно "Audio Mixer"

Вы можете создать один или несколько аудио-микшеров и иметь более одного активного одновременно. Аудио-микшер всегда содержит мастер-группу (**Master**). Затем можно добавить другие группы, чтобы определить структуру микшера.

Снапшоты (**Snapshots**) позволяют захватить настройки всех параметров в группе как снимок. Если вы создаете список снимков, вы можете переходить между ними в игровом процессе, чтобы создавать различные настроения или темы.

Также вы можете настроить различные представления (**Views**). Вы можете отключить видимость определенных групп в микшере и установить это как представление. Затем вы можете переходить между представлениями по мере необходимости.

8.4. Свойства AudioClip

Поговорим об импорте звуковых ассетов – ведь перед тем, как звуковой файл воспроизвести, его нужно импортировать в проект Unity.

Unity поддерживает широкий спектр форматов исходных файлов. Каждый раз, когда файл импортируется, он перекодируется в формат, подходящий для цели сборки и типа звука. Это можно выбрать с помощью параметра **Compression Format** в инспекторе (рис. 8.2).

Рис 8.2. Свойства аудио-клипа

Чтобы звук был ближе к оригиналу, предпочтительно использовать форматы PCM/Vorbis/MP3. PCM очень облегчает требования к процессору, поскольку звук несжатый и может быть просто считан из памяти. Форматы Vorbis/MP3 адаптивно отбрасывают менее слышимую информацию с помощью ползунка качества (**Quality**).

ADPCM – это компромисс между использованием памяти и процессора в том смысле, что он использует лишь немного больше процессорного времени, чем несжатый PCM, но дает постоянный коэффициент сжатия 3.5, что в целом примерно в 3 раза хуже, чем сжатие, которое может быть достигнуто с помощью форматов Vorbis или MP3. Кроме того, ADPCM позволяет автоматически оптимизировать или вручную устанавливать частоты дискретизации, которые – в зависимости от частотного содержания звука и приемлемой потери качества – могут дополнительно уменьшить размер звуковых файлов.

Как правило, сжатое аудио лучше всего использовать для длинных файлов, например, для фоновой музыки или диалога, в то время как Native формат лучше подходит для коротких звуковых эффектов вроде выстрела. Следует подбирать степень сжатия с помощью слайдера **Quality**. Начните с высокого качества и постепенно уменьшайте его до уровня, когда потеря качества уже заметна. После этого начните понемногу увеличивать обратно качество до тех пор, пока заметная на слух потеря качества не пропадет.

Остальные свойства звукового файла приведены в табл. 8.1. Чтобы увидеть эти свойства, просто щелкните на звуковом файле в области Project.

Таблица 8.1. Свойства звукового файла

Свойство	Описание
Load Type	Метод загрузки аудио ассетов во время работы приложения, используемый Unity
Decompress On Load	Распаковывать аудио файлы сразу же после загрузки. Используйте эту опцию для маленьких сжатых звуков, чтобы избежать чрезмерного потребления ресурсов из-за распаковки "на лету". Учтите, что распаковка файлов после загрузки занимает примерно в десять раз больше памяти по сравнению с их размером в сжатом виде, поэтому не используйте эту опцию для больших файлов

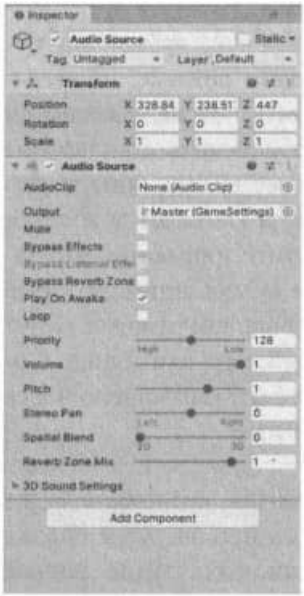
Compressed In Memory	Сохранять звуки в памяти в сжатом виде и распаковывать при проигрывании. Эта опция не сильно сказывается на производительности (особенно для сжатых файлов Ogg/Vorbis), поэтому используйте ее только для больших файлов, на которых распаковка при загрузке потребляла бы слишком много памяти. Заметьте, что из-за технических ограничений, эта опция автоматически переключается на Stream From Disc (см. описание ниже) для Ogg Vorbis ассетов на платформах, которые используют FMOD аудио
Streaming	Декодировать звуки на лету. Этот метод использует минимальный объем памяти для буферизации сжатых данных, которые постепенно считываются с диска и декодируются на лету. Обратите внимание, что декомпрессия происходит в отдельном потоковом потоке
Compression Format	Формат звука, который будет использован при проигрывании во время работы приложения
PCM	Это нативный формат. Он является наиболее качественным, но и производит файлы большего размера, поэтому лучше всего использовать его для очень коротких аудио эффектов
ADPCM	Этот формат полезен для звуков, которые содержат много шума и должны воспроизводиться в больших количествах, таких как шаги, удары, оружие. Коэффициент сжатия в 3.5 раза меньше, чем у PCM, но загрузка процессора намного ниже, чем у форматов MP3/Vorbis, что делает его предпочтительным выбором для вышеупомянутых категорий звуков

Vorbis/MP3	Это сжатый формат. Сжатие приводит к меньшему размеру файлов, но с некоторыми потерями в качестве звука по сравнению с нативным форматом. Данный формат лучше всего использовать для звуков средней длины и для музыки
HEVAG	Это родной формат, используемый на PS Vita. Спецификации этого очень похожи на спецификации формата ADPCM
Sample Rate Setting	Форматы сжатия PCM и ADPCM позволяют автоматически оптимизировать или вручную снижать частоту дискретизации
Preserve Sample Rate	Сохраняет частоту дискретизации без изменений (по умолчанию)
Optimize Sample Rate	Автоматически оптимизирует частоту дискретизации в соответствии с анализируемым самым высоким частотным содержанием
Override Sample Rate	Позволяет вручную изменять частоту дискретизации, поэтому эффективно его можно использовать для отбрасывания частотного содержимого
Force To Mono	Если этот параметр включен, аудиоклип будет сведен к одноканальному звуку. После понижающего микширования сигнал нормализуется по пику, поскольку процесс понижающего микширования обычно приводит к сигналам, которые являются более тихими, чем исходные, следовательно, сигнал с нормализацией по пику дает больший запас для последующих регулировок через свойство громкости AudioSource

Load In Background	Если этот параметр включен, аудиоклип будет загружаться в фоновом режиме, не вызывая остановок основного потока. Отключен по умолчанию, чтобы обеспечить стандартное поведение Unity, когда все аудиоклипы заканчивают загружаться, когда начинается воспроизведение сцены. Обратите внимание, что запросы воспроизведения аудиоклипов, которые все еще загружаются в фоновом режиме, будут отложены до завершения загрузки клипа. Состояние загрузки может быть запрошено через свойство AudioClip.loadState
Preload Audio Data	Если этот параметр включен, аудиоклип будет предварительно загружен при загрузке сцены. Если этот флаг не установлен, аудио-данные будут загружены или в первый AudioSource.Play()/AudioSource.PlayOneShot() или загружены через AudioSource.LoadAudioData() и снова выгружены через AudioSource.UnloadAudioData()
Quality	Уровень сжатия, применяемый к сжатому клипу. Расчетный размер файла можно посмотреть под слайдером. Лучше всего настроить это значение (с помощью ползунка слайдера) так, чтобы качество звучания оставалось на приемлемом уровне, но размер файла при этом соответствовал вашим требованиям

8.5. Источники звука

Когда вы импортировали клипы в проект, нужно позаботиться об объектах Audio Source – они будут управлять воспроизведением звуковых файлов (AudioClip).



Если AudioClip является 3D клипом, источник проигрывается в заданном положении в пространстве и будет приглушаться в зависимости от расстояния. Аудио может быть распределено по колонкам (например, из стерео в 7.1) с помощью свойства **Spread** и трансформироваться между 3D и 2D с помощью свойства **PanLevel**. Можно контролировать зависимость этих эффектов от расстояния с помощью кривых затухания. Также если слушатель находится в одной или нескольких зонах реверберации, то к источнику применяются реверберации (только для Unity Pro). Для обогащения аудио-ряда, к источнику можно применять отдельные аудио фильтры.

Рис. 8.3. Свойства объекта "AudioSource"

Свойство	Описание
Audio Clip	Ссылка на аудио клип для проигрывания этим источником
Output	Звук может выводиться через аудиослушатель (AudioListener) или микшер (AudioMixer)
Mute	Если включено, звук все еще будет проигрываться, но не будет слышен
Bypass Effects	Используется для быстрого пропуска всех эффектов, примененных к источнику звука. Простой путь включения/отключения всех эффектов
Bypass Listener Effects	Используется для быстрого включения/отключения всех эффектов, примененных к слушателю
Bypass Reverb Zones	Служит для быстрого включения/отключения всех зон реверберации
Play On Awake	Если включено, звук начнет воспроизводиться сразу после загрузки сцены. Если отключено, вам потребуется запустить звук программно, с помощью метода Play()

Loop	Включите для бесконечного повтора звукового клипа
Priority	Определяет приоритет данного источника звука среди всех остальных источников в сцене (0 = наиболее важный, 256 = наименее важный, 128 по умолчанию). Используйте 0 для музыки, чтобы избежать ее случайного переключения
Volume	Насколько громок звук на расстоянии одной мировой единицы измерения (одного метра) от слушателя (AudioListener)
Pitch	Степень изменения высоты тона при замедлении/ускорении аудио-клипа. Величина 1 означает нормальную скорость воспроизведения
Stereo Pan	Устанавливает позицию в стере-поле 2D-звука
Spatial Blend	Устанавливает степень влияния 3D движка на источник звука
Reverb Zone Mix	Устанавливает количество выходного сигнала, который направляется в зоны реверберации. Величина является линейной в диапазоне (0 - 1), но допускает усиление в 10 дБ в диапазоне (1 - 1,1), которое может быть полезным для достижения эффекта ближнего и отдаленного звука
3D Sound Settings	Настройки, которые применяются пропорционально параметру Spatial Blend
Doppler Level	Определяет количество эффекта доплера, применяемого к данному источнику (при значении 0 эффект применяться не будет)
Spread	Устанавливает угол распространения для 3D-стерео или мультиканального звука в пространстве динамиков

Min Distance	Громкость звука будет максимальной, насколько это возможно, на протяжении MinDistance. Вне MinDistance она будет постепенно снижаться. Увеличьте MinDistance звука, чтобы сделать его «громче» в трёхмерном мире; снизьте, чтобы сделать звук «тише» в трехмерном мире
Max Distance	Расстояние, на котором звук перестаёт затухать. За пределами этой точки его громкость останется на уровне, на котором она была бы на расстоянии MaxDistance единиц от слушателя и больше не будет затухать
Rolloff Mode	Как быстро звук угасает. Чем выше значение, там ближе должен быть слушатель к звуку прежде, чем его можно будет услышать (определяется по графику)
Logarithmic Rolloff	Громкость звука высока, когда вы близко к источнику, но при удалении от объекта она довольно быстро падает
Linear Rolloff	Чем вы дальше от источника звука, тем хуже вы его слышите
Custom Rolloff	Звук источника аудио ведет себя соответственно графику затуханий

В большинстве случаев работа с AudioSource осуществляется так: устанавливаются все свойства, а затем в сценарии в нужный момент запускается воспроизведение звука методом Play(). Рассмотрим сценарий, управляющий воспроизведением звука. Сценарий запускает воспроизведение звука в методе Start(), затем при нажатии на кнопки UI он делает паузу и продолжает воспроизведение. Само аудио хранится в источнике звука audioData.

Листинг 8.1. Управление воспроизведением

```
using UnityEngine;

[RequireComponent(typeof(AudioSource))]
public class ExampleScript : MonoBehaviour
{
    public AudioSource audioData;
```

```
    void Start()
    {
        audioData = GetComponent<AudioSource>();
        // Запускаем воспроизведение
        audioData.Play(0);
        Debug.Log("started");
    }

    void OnGUI()
    {
        if (GUI.Button(new Rect(10, 70, 150, 30), "Pause"))
        {
  // Пауза
            audioData.Pause();
            Debug.Log("Pause: " + audioData.time);
        }

        if (GUI.Button(new Rect(10, 170, 150, 30),
"Continue"))
        {
  // Продолжаем воспроизведение
            audioData.UnPause();
        }
    }
}
```

В следующей главе мы покажем, как можно регулировать громкость аудио с помощью слайдера. Мы создадим сцену установки параметров игры и на практическом примере покажем, как устанавливать уровень громкости (рис. 8.4).

Рис. 8.4. Слайдер Volume, который будет разработан нами в следующих главах

8.6. Аудио-слушатели

Слушатель аудио (AudioListener) ведет себя как микрофон. Он получает входящие данные с любого источника звука (Audio Source) в сцене и проигрывает звуки через динамики. Для большинства приложений имеет смысл добавить слушатель к главной камере –объекту **Main Camera**. Если слушатель аудио находится в рамках зоны реверберации (Reverb Zone), то реверберация применяется ко всем слышимым звукам в сцене (только для Unity Pro). Более того, к слушателю можно добавлять аудио эффекты, чтобы применить их ко всем слышимым в сцене звукам.

Компонент AudioListener, наверное, является самым простым компонентом. У него нет свойств. Его достаточно просто добавить, чтобы он заработал. По умолчанию, он всегда добавлен на объект **Main Camera** (рис. 8.5).

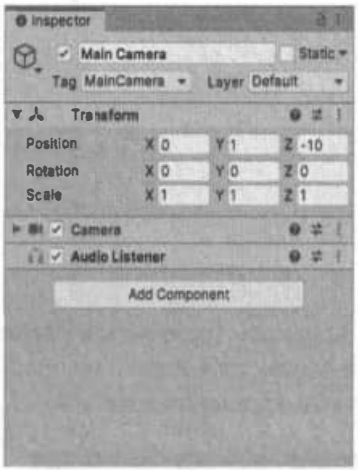


Рис. 8.5. Свойства Main Camera: добавлен компонент "Audio Listener"

Компонент Audio Listener работает в связке с источниками звука (компонент Audio Source), позволяя вам создавать акустическое окружение в играх. Если Audio Listener добавлен к игровому объекту в сцене, то любые достаточно близкие к слушателю источники будут слышны в динамиках. В каждой сцене может быть только один Audio Listener для корректной работы системы.

Если источники – 3D, тогда слушатель будет имитировать положение, скорость и ориентацию звука в 3D пространстве (очень точно настроить приглушение и поведение в 3D/2D вы сможете у компонента Audio Source). В 2D режиме будет игнорироваться любая 3D обработка. Например, если ваш персонаж уходит с улицы в ночной клуб, музыка в ночном клубе вероятно должна быть 2D, в то время как голоса отдельных персонажей, находящихся в клубе, должны быть моно, с реалистичным расположением в сцене, которое обеспечивает Unity.

Вам следует добавлять компонент Audio Listener или к объекту **Main Camera** или к игровому объекту, представляющего собой игрока. Попробуйте оба варианта, чтобы понять, какой из них больше подходит для вашей игры.

8.7. Запись звука. Класс Microphone

Класс Microphone нужен для записи звука с таких физических носителей как микрофон вашего компьютера или любого доступного мобильного устройства.

С помощью этого класса можно запустить и остановить запись с встроенного микрофона, получить список доступных устройств для записи (микрофонов), а также узнать о их текущем состоянии.

Для класса Microphone не существует своего компонента, но получить доступ к этому классу можно и через скрипт.

Свойство *devices* позволяет получить список доступных микрофонов:

Листинг 8.2. Список доступных микрофонов

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour {
    void Start() {
        foreach (string device in Microphone.devices) {
            Debug.Log("Name: " + device);
        }
    }
}
```

Данный код получает список микрофонов и выводит их на консоль.

Определившись с микрофоном, можно использовать следующие методы для осуществления записи и для управления нею:

- Start – начинает запись
- End – заканчивает запись
- GetDeviceCaps – получает частотные возможности устройства
- GetPosition – получает позицию в сэмплах записи
- isRecoding – возвращает *true*, если ведется запись с микрофона, переданного в качестве параметра

Методу Start() нужно передать следующие параметры:

- deviceName – имя устройства

- loop – указывает, нужно ли запись продолжать запись при достижении lengthSec
- lengthSec – длина (в секундах) записи
- frequency – частота дискретизации аудиоклипа, полученного при записи

Следующий код получает аудио-запись со встроенного микрофона длиной 10 секунд, частота дискретизации 44100. После того, как аудио-клип получен, будет запущено его воспроизведение.

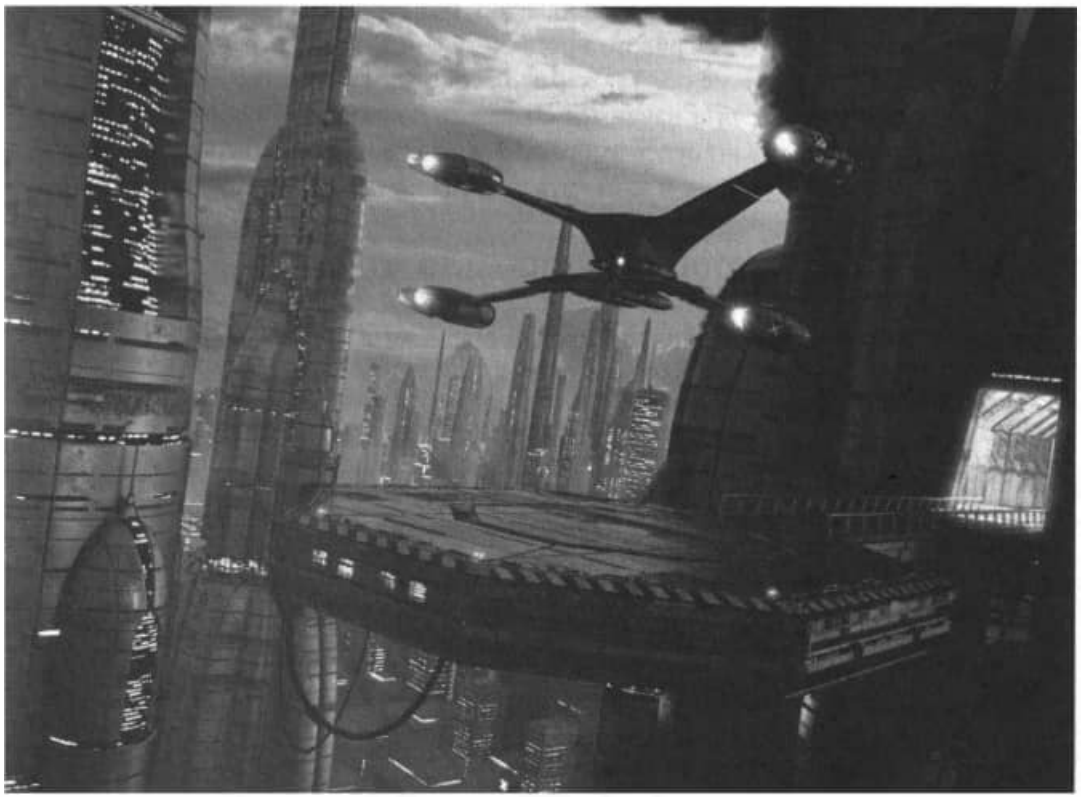
Листинг 8.3. Захват аудио

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour {
    void Start() {
        AudioSource aud = GetComponent<AudioSource>();
        aud.clip = Microphone.Start("Built-in Microphone",
false, 10, 44100);
        aud.Play();
    }
}
```

На этом все, а в следующих главах мы поговорим о разработке главного меню и экрана настроек для нашей игры. Экран настроек будет предполагать помимо всего прочего еще и регулировку уровня громкости.

Глава 9.

Разработка интерфейса пользователя



Ни одна уважающая себя игра не обходится без внутриигрового меню. Как правило, в таком меню есть команды запуска игры, создания новой игры, сохранения и загрузки игрового процесса и т.д. В этой главе будет показано, как создать такое меню. Также мы рассмотрим основные элементы интерфейса пользователя и научимся создавать полноценный экран настроек игры, в котором можно будет выбрать режим отображения игры, отрегулировать уровень громкости и изменить разрешение экрана.

9.1. Создаем простое меню для игры

9.1.1. Подготовка к созданию меню

Создайте две сцены:

- Game;
- Menu.

Понятно, что первая сцена – это будет основная сцена игры. Можете переименовать уже имеющуюся сцену, которая создается по умолчанию. Вторая сцена – это сцена меню. Как будет сейчас выглядеть сцена **Game** – совершенно неважно. Поместите любые объекты на сцену, лишь бы на ней что-то было (рис. 9.1).

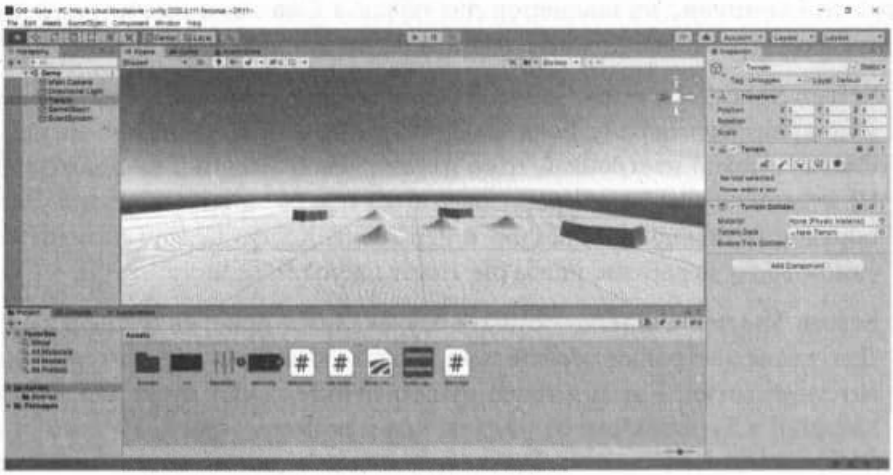


Рис. 9.1. Примерная сцена игры

Рис. 9.2. Созданы две сцены

А вот сценой **Menu** мы сейчас займемся. Откройте сцену **Menu** и добавьте объект **Panel** (команда меню **Gameobject, UI, Panel**). Сразу добавляется **Canvas** (холст) и дочерним объектом к нему добавляется **Panel** (панель), см. рис. 9.3.

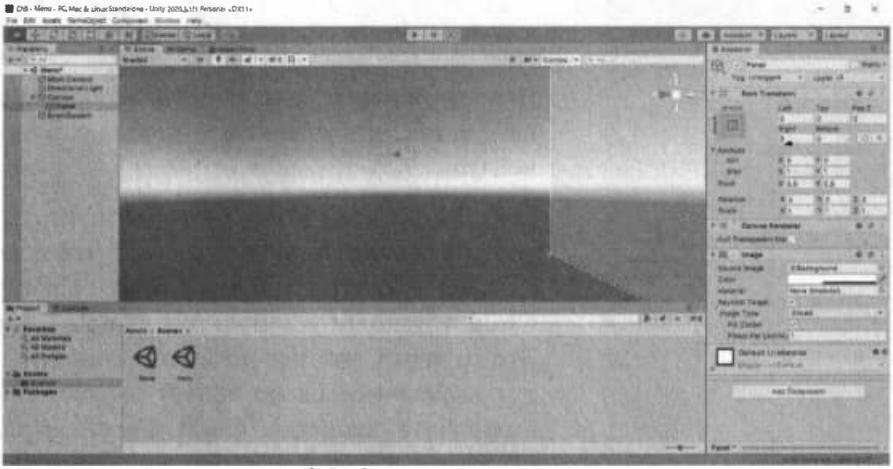


Рис. 9.3. Созданы холст и панель

Обратите внимание на инспектор для объекта **Canvas**, а именно на компонент Canvas (рис. 9.4). Для него возможны следующие значения:

- **Screen Space – Overlay** – способ рендеринга, при котором Canvas напрямую привязан к экрану. Если изменить разрешение экрана или размер, то Canvas с такой настройкой тоже изменит свой масштаб на подходящий. UI в таком случае будет прорисовываться поверх других объектов. Важен порядок размещения объектов в иерархии. Холст должен находиться в самом верху иерархии, иначе он может пропасть из виду
- **Screen Space – Camera** – в таком случае, холст прикрепляется в камере. Для такой настройки обязательно нужно указать камеру, которой соответствует холст. Как и в предыдущем пункте, холст будет менять свой масштаб в зависимости от разрешения и размера экрана, а также от области видимости камеры. Так же для холста с такими настройками важно размещение относительно других объектов. На переднем плане будут объекты, которые находятся ближе к камере, не зависимо от того, это UI или другие GameObjects
- **World Space** – холст размещается, как любой другой объект без привязки к камере или экрану, он может быть ориентирован как вам угодно, размер холста задается с помощью RectTransform, но то, как его будет видно во время игры, будет зависеть от положения камеры

В нашем случае подойдут первые два варианта – выбирайте или **Screen Space – Overlay** или **Screen Space – Camera**. Если выбрали второй вариант, не забудьте выбрать камеру в свойствах Canvas.

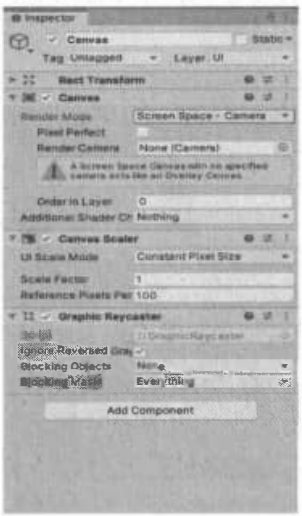


Рис. 9.4. Свойства объекта "Canvas"

Используя инструмент **Hand Tool**, разверните панель так, чтобы вам было удобно работать с ней (рис. 9.5).

Когда панель выделена, щелкните по кнопке **Add Component** и добавьте компонент **Layout, Vertial Layout Group**. Свойство **Child Alignment** установите в **Middle Center**. Это означает, что все дочерние элементы будут выравниваться по центру экрана. Также выключите свойство **Child Force Expand**,

чтобы дочерние элементы выстраивались строго друг под другом.

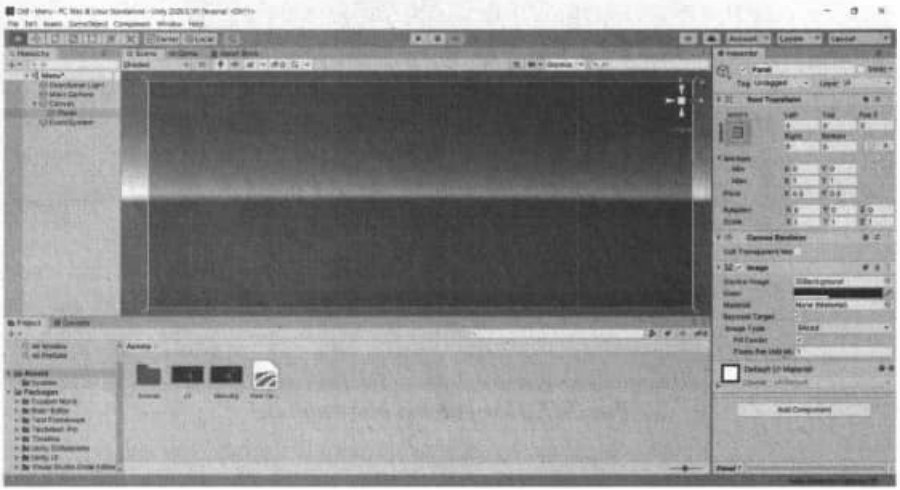


Рис. 9.5. Панель развернута «лицом» к разработчику

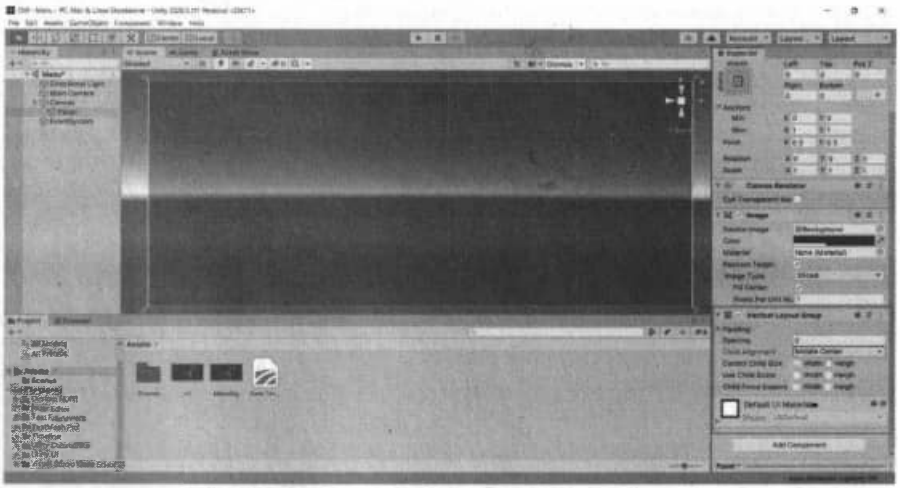


Рис. 9.6. Свойства компонента "Vertial Layout Group"

Добавьте две кнопки (команда **Gameobject, UI, Button**). Обе кнопки в окне иерархии нужно переместить на панель, чтобы они стали дочерними компонентами для нашей панели. Вы увидите, как кнопки выстроятся одна под другой (рис. 9.7).

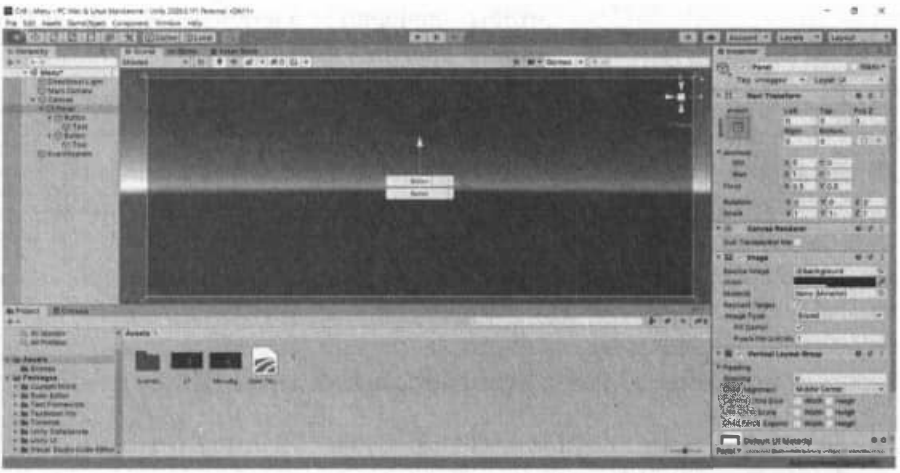


Рис. 9.7. Помещены обе кнопки

Запустите игру, чтобы убедиться, что наше меню выглядит так, как вам нужно. У вас должно получиться примерно то же, что показано на рис. 9.8. На надписи пока не обращайте внимания, далее мы займемся визуальным оформлением меню.

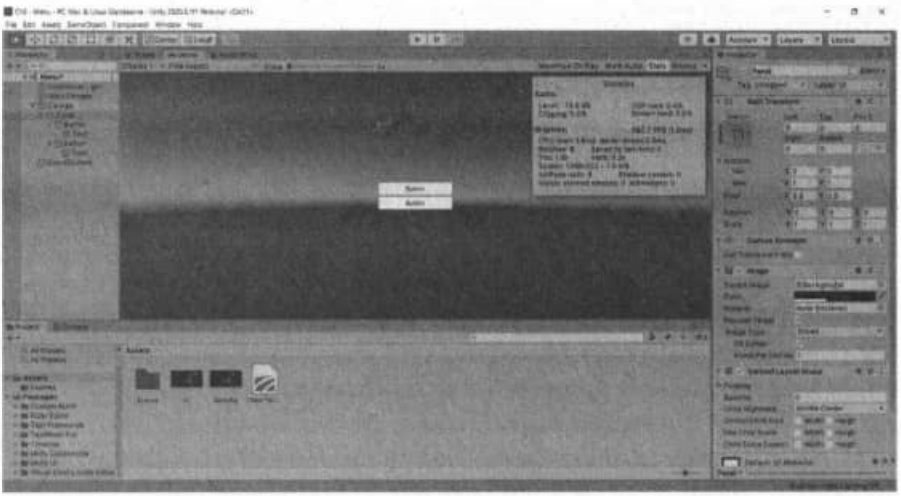


Рис. 9.8. Промежуточный результат

9.1.2. Фоновое изображение для меню

Сейчас наше меню выглядит слишком просто. Добавим фоновое изображение. Сделать это в Unity не просто, поскольку прежде, чем использовать лю-

бое фоновое изображение, нужно превратить его в спрайт. Последовательность действий будет следующей:

1. Импортируйте в проект любое изображение (PNG/JPG), которое вы хотите использовать в качестве фонового изображения меню. Просто перетащите графический файл в область обозревателя проекта.
2. Выделите графический файл в обозревателе проекта. В инспекторе будут отображены его свойства. Измените свойство **Texture Type** на **Sprite (2D and UI)**, рис. 9.9.
3. Щелкните на элементе **Panel** в окне иерархии. При переходе к панели Unity спросит вас, хотите ли вы сохранить изменения в изображении. Соглашайтесь.
4. Перетащите фоновый спрайт на свойство **Source Image** панели (рис. 9.10).
5. Переключитесь на **Main Camera**. Свойство **Clear Flags** установите в **Depth Only**, иначе камера будет отображать небосвод, а нам этого не нужно.
6. Запустите игру. Вот теперь наше меню выглядит значительно привлекательнее (рис. 9.11).

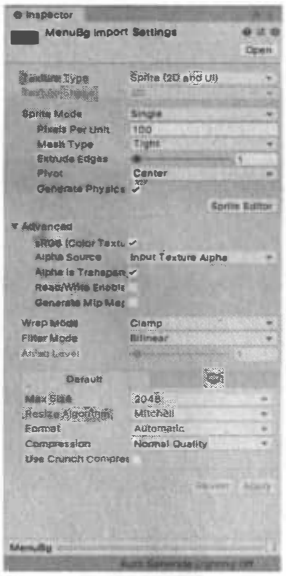


Рис 9.9. Конвертируем JPEG изображение в спрайт

Рис. 9.10. Свойства панели

Рис. 9.11. Меню с фоновым изображением

9.1.3. Программирование кнопок

Вернемся к нашей панели. Выберите верхнюю кнопку. Переименуйте ее в StartButton, а свойство **Text** ее элемента **Text** установите в Start Game, как показано на рис. 9.12.

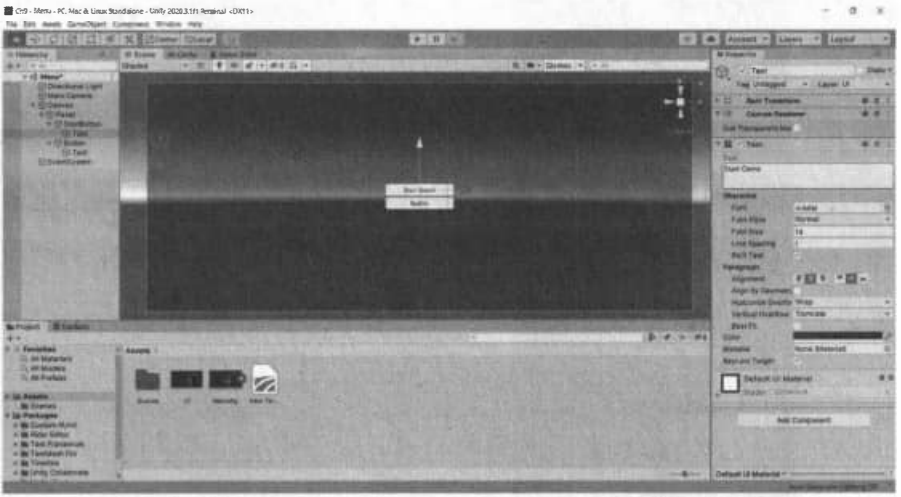


Рис. 9.12. Переименование кнопки

Вторую кнопку назовите **Exit**, а в качестве значения дочернего свойства **Text** установите тот же текст – **Exit**. У вас должно получиться две кнопки – **Start Game** и **Exit** (рис. 9.13).

Рис. 9.13. Обе кнопки созданы

Далее начнется немного сложный с первого взгляда момент, поэтому будьте внимательны. Создайте пустой объект (**Gameobject, Create Empty**). Переименуйте этот объект в **MainMenu**, сделайте его дочерним для **Canvas**, а **Panel** сделайте дочерним элементом для **MainMenu**. Получиться должно, как на рис. 9.14.

Рис. 9.14. Создан пустой объект "MainMenu"

Далее мы создадим сценарий MainMenu.cs и привяжем его к объекту MainMenu. Щелкните правой кнопкой мыши на элементе **Assets**, выберите команду **Create, C# Script**. Установите имя скрипта MainMenu.cs и дважды щелкните по нему, чтобы открыть редактор кода (рис. 9.15).

Рис. 9.15. Редактор кода

Код скрипта приведен в лист. 9.1.

Листинг 9.1. Код MainMenu.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class MenuControls : MonoBehaviour
{
  public void PlayPressed()
    {
        SceneManager.LoadScene("Game");
    }

   public void ExitPressed()
    {
        Application.Quit();
    }
}
```

Как вы уже догадались, этот скрипт задает действия для кнопок StartGame и Exit. Однако обратите внимание, что наши методы называются совершенно произвольно и их имена никак не сочетаются с названием кнопок. Так и есть – вы можете использовать любые имена, но методы обязательно должны быть объявлены как public. Также вы должны импортировать UnityEngine. SceneManagement, чтобы работал метод SceneManager.LoadScene().

Сохраните сценарий и перетащите его на компонент MainMenu. Должно получиться так, как показано на рис. 9.16.

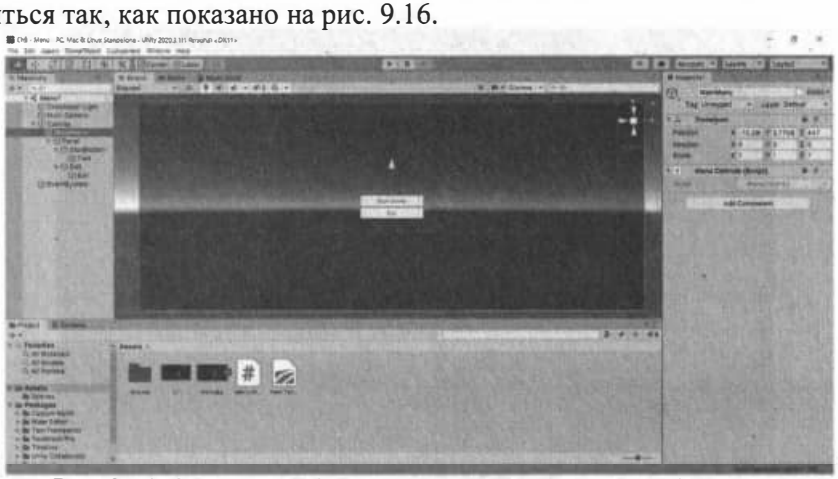


Рис. 9.16. Объекту MainMenu назначен сценарий MainMenu.cs

Осталось назначить обработчики для кнопок. Щелкните на кнопке Start Game. Выполните следующие действия (рис. 9.17):

- По умолчанию обработчик OnClick() для кнопки не назначен (9.17а). Нажмите кнопку +
- На поле выбора объекта (по умолчанию будет значение None) из иерархии объектов перетащите наш объект MainMenu (рис. 9.17б)
- Из списка **Function** выберите метод PlayPressed(), рис. 9.17в

Рис. 9.17. Процесс назначения обработчика для кнопки

Аналогично, назначьте обработчик для кнопки **Exit**, но выберите метод ExitPressed().

9.1.4. Изменение параметров сборки

Процесс сборки игры будет подробно описан в дальнейших главах этой книги. А пока выберите команду меню **File, Build Settings**. В область **Scenes In Build** перетащите обе созданные нами сцены. Сцены можно перетаскивать из обозревателя проекта. Если этого не сделать, при попытке нажать кнопку Start Game вы получите сообщение об ошибке, изображенное на рис. 9.19.

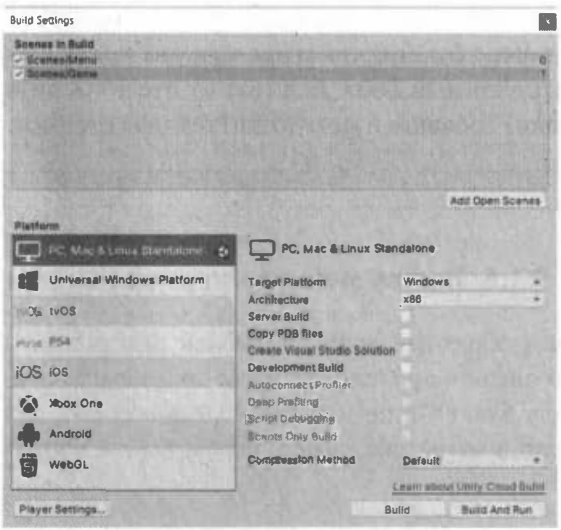


Рис. 9.18. Параметры сборки

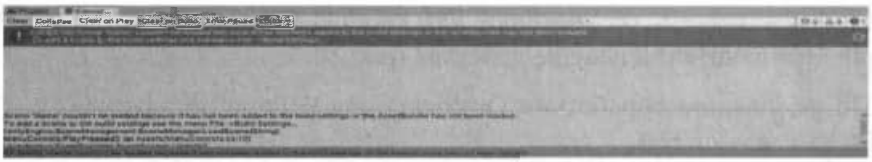


Рис. 9.19. Ошибка сборки

Закройте окно **Build Settings** и запустите игру. Нажмите кнопку Start Game, вы увидите, как будет загружена сцена **Game** (рис. 9.20). Наша кнопка работает!

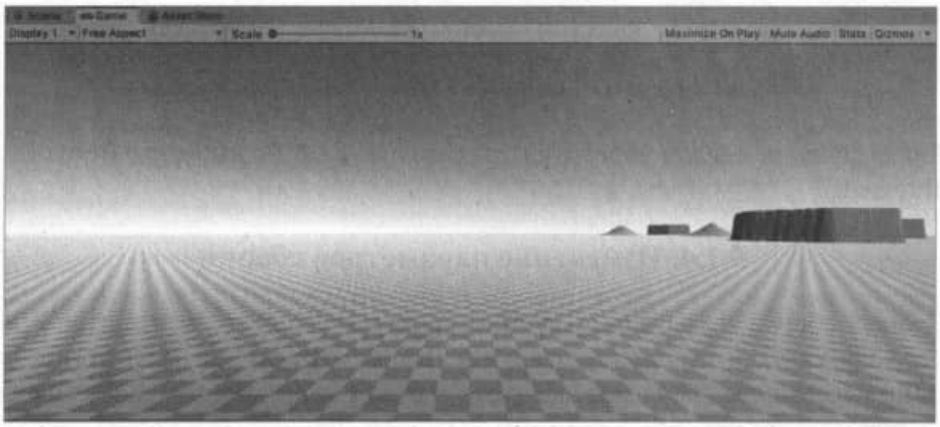


Рис. 9.20. Загружена сцена "Game"

Вторую кнопку можно будет протестировать уже после сборки игры – когда будет запущен исполнимый файл. Если вы хотите проверить, срабатывает ли обработчик кнопки, добавьте в метод ExitPressed() строчку:

```
Debug.Log("Exit pressed!");
```

9.1.5. Вызов меню из игровой сцены

Теперь проделаем обратный трюк, а именно вызов сцены меню из игры. Сейчас ситуация такая – при запуске игры пользователь видит сцену меню, выбирает команду **Start Game** и видит игровую сцену. Но если он захочет вернуться обратно в меню, например, чтобы выйти из игры, он не сможет этого сделать.

Сейчас мы реализуем вызов сцены меню по нажатию клавиши Esc:

1. Загрузите сцену Game и создайте пустой объект (команда меню **Gameobject, Create Empty**).
2. Щелкните правой кнопкой мыши на элементе **Assets** и создайте скрипт (**Create, C# Script**) с названием MenuController.cs.
3. Отредактируйте код этого скрипта так, как показано в лист. 9.2.
4. Перетащите созданный сценарий на пустой игровой объект.

Листинг 9.2. Код сценария MenuController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class MenuController : MonoBehaviour
{
    // Перед первым обновлением кадра
    void Start()
    {

    }

    // Вызывается раз за кадр
    void Update()
    {
        if(Input.GetKey(KeyCode.Escape))
        {
     SceneManager.LoadScene("Menu");
        }
    }
}
```

Сценарий просто как 2+2, здесь мы проверяем, не нажата ли клавиша Esc. Если нажата, вызываем сцену **Menu**. Запустите игру и убедитесь, что нажатие Esc приводит к отображению сцены меню.

Теперь наше меню полностью создано и мы можем приступить к изучению других элементов графического интерфейса.

9.2. Элементы графического интерфейса

На данный момент мы познакомились с тремя элементами UI (User Interface) – холстом, панелью и кнопкой. Однако в меню Gameobject, UI есть еще много элементов:

- **Text** – текстовая надпись
- **Image** – изображение
- **Raw Image** – также используется для отображения картинки, но в отличие от Image может выводить любую текстуру, а не только спрайт
- **Toggle** – переключатель
- **Slider** – слайдер (ползунок)
- **Scrollbar** – полоса прокрутки
- **Dropdown** – выпадающий список
- **Input Field** – поле ввода

Далее мы рассмотрим эти элементы.

9.2.1. Текстовая надпись

Элемент управления **Text** отображает неинтерактивный фрагмент текста для пользователя. Он может использоваться для предоставления подписей или меток для других элементов UI или для отображения инструкций или другого текста. По сути, с элементом **Text** мы уже знакомы – он добавляется в качестве дочернего при создании кнопки (элемент Button).

Свойства текстовой надписи приведены в табл. 9.1.

Таблица 9.1. Свойства элемента управления Text

Свойство	Описание
Text	Отображаемый текст
Character	Параметры начертания символов
Font	Гарнитура шрифта, используемого для вывода текста
Font Style	Стиль начертания текста. Может быть Normal (обычный), Bold (жирный), Italic (курсив) и Bold And Italic (жирный курсив)
Font Size	Размер шрифта

Line Spacing	Междустрочный интервал
Rich Text	Должно ли содержимое Text интерпретироваться как текст в формате RTF
Paragraph	Параметры абзаца
Alignment	Горизонтальное и вертикальное выравнивание текста
Align by Geometry	Используйте экстенты геометрии глифа для выполнения горизонтального выравнивания, а не метрики глифа
Horizontal Overflow	Метод, используемый для обработки ситуации, когда текст слишком широкий, чтобы поместиться в прямоугольнике. Доступные опции – Wrap и Overflow
Vertical Overflow	Метод, используемый для обработки ситуации, когда обернутый текст слишком высок, чтобы поместиться в прямоугольнике. Доступные опции – Truncate и Overflow
Best Fit	Должен ли Unity игнорировать свойства размера и просто пытаться разместить текст в прямоугольнике элемента управления?
Color	Цвет, используемый для отображения текста
Material	Материал, используемый для рендеринга текста

9.2.2. Элементы для отображения изображений. Загрузка изображений из Интернета

Объекты **Image** и **RawImage** используются для отображения изображений. Первый объект может отображать только текстуры, которые являются спрайтами. Как преобразовать текстуру в спрайт, было показано в разделе 9.1, когда мы устанавливали фоновое изображение для меню.

Второй объект в качестве изображения может использовать любую текстуру без предварительного преобразования ее в спрайт.

Рассмотрим свойства объекта **Image** (табл. 9.2).

Таблица 9.2. Свойства объекта Image

Свойство	Описание
Source Image	Исходное изображение. Текстура должна быть в формате спрайта
Color	Цвет, применяемый к изображению
Material	Материал, используемый для рендеринга изображения
Raycast Target	Должно ли изображение быть целью для рейкастинга
Preserve Aspect	Позволяет предупредить нарушение пропорций изображения
Set Native Size	Кнопка, устанавливающая размеры объекта так, чтобы они соответствовали размеру текстуры

Свойств у Raw Image всего четыре – **Texture, Color** и **Material** – они такие же, как у объекта Image. Свойство **UV Rectangle** задает смещение и размер изображения в UV-прямоугольнике, координаты являются нормализированными (указываются в диапазоне от 0,0 до 1,0). Края изображения растягиваются так, чтобы заполнить пространство вокруг UV-прямоугольника.

Исходное изображение не обязательно должно является спрайтом, вы можете использовать этот объект для отображения любой текстуры, которая поддерживается в Unity. Например, вы можете показать изображение, загруженное с Сети посредством класса WWW. Если вы заинтересовались, то сделать это можно так, как показано в лист. 9.3.

Листинг 9.3. Загрузка изображения с Сети и его отображение в Raw Image

```
using UnityEngine;
using System.Collections;

public class ExampleClass : MonoBehaviour {
    public string url = "http://example.com/image.jpg";
    IEnumerator Start() {
        WWW www = new WWW(url);
        yield return www;
        Renderer renderer = GetComponent<Renderer>();
```

```
        renderer.material.mainTexture = www.texture;
    }
}
```

9.2.3. Переключатель

С помощью переключателя (Toggle) можно включать/выключать какую-либо опцию. Другое название переключателя – чекбокс. Как выглядит чекбокс, показано на рис. 9.21.

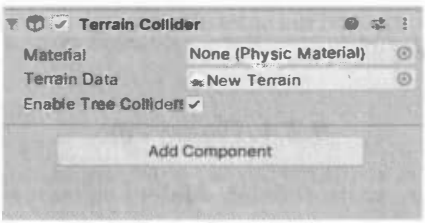


Рис. 9.21. Примеры чекбоксов

Свойства данного объекта приведены в табл. 9.3.

Таблица 9.3. Свойства переключателя

Свойство	Описание
Interactable	Если включено, переключатель может принимать ввод. Если выключено, переключатель может только информировать пользователя о том, что какая-то опция включена/выключена, а управлением переключателем осуществляется через скрипт
Transition	Определяет способ, как элемент управления визуально реагирует на действия пользователя
Navigation	Определяет последовательность элементов управления
Is On	Будет ли переключатель включен по умолчанию
Toggle Transition	Способ, которым переключатель реагирует графически, когда его значение изменяется
Graphic	Изображение, используемое для чекбокса
Group	Задает объект Toggle Group – группу переключателей, к которой относится данный чекбокс

Примечание. Объект Toggle Group – это невизуальный объект, используемый для создания группы переключателей. Внутри группы активным может быть только один из переключателей. Группы переключения полезны везде, где пользователь должен сделать выбор из взаимоисключающего набора параметров. Типичные примеры включают выбор типов персонажей игрока, настройки скорости (медленная, средняя, быстрая и т.д.). Вы можете иметь более одного объекта Toggle Group в сцене одновременно, так что вы можете создать несколько отдельных групп при необходимости.

9.2.4. Ползунок

Главным свойством ползунка (объект **Slider**) является десятичное значение Value, которое пользователь может изменять в пределах между минимальным и максимальным значением путем перетаскивания графического ползунка. Ползунок может быть горизонтальным или вертикальным. Он также генерирует событие OnValueChanged, возникающее при изменении значения Value. Свойства объекта приведены в табл. 9.4.

Таблица 9.4. Свойства объекта Slider

Свойство	Описание
Interactable	Будет ли этот компонент принимать входные данные от пользователя или контролироваться только через скрипт?
Transition	Определяет, как элемент управления визуально реагирует на действия пользователя
Navigation	Определяет последовательность элементов управления
Fill Rect	Графика, используемая для области заполнения элемента управления
Hand Rect	Графика, используемая для «ползунка» элемента управления
Direction	Направление, в котором значение ползунка будет увеличиваться при перетаскивании ручки. Возможные варианты: Left To Right (слева направо), Right To Left (справа налево), Bottom To Top (снизу вверх) и Top To Bottom (сверху вниз)

Min Value	Минимальное значение ползунка
Max Value	Максимальное значение ползунка
Whole Numbers	Должен ли ползунок быть ограничен только целыми числами
Value	Текущее числовое значение слайдера. Если значение установлено в инспекторе, оно будет использоваться в качестве начального значения, но оно изменится во время выполнения при изменении значения

9.2.5. Полоска прокрутки

Полоска прокрутки (Scroll Rect) может использоваться, когда контент, занимающий много места, должен отображаться в небольшой области. Scroll Rect предоставляет функциональные возможности для прокрутки этого содержимого.

Обычно **Scroll Rect** комбинируется с **Mask** (невидимый элемент UI, используемый для изменения поведения дочерних элементов) для создания представления прокрутки, где виден только прокручиваемый контент. Его также можно комбинировать с одной или двумя полосами прокрутки, которые можно перетаскивать для прокрутки по горизонтали или вертикали.

Свойства объекта Scroll Rect приведены в табл. 9.5.

Таблица 9.5. Свойства объекта Scroll Rect

Свойство	Описание
Content	Ссылка на элемент UI, который нужно прокрутить, например, большое изображение
Horizontal	Включает горизонтальную прокрутку
Vertical	Включает вертикальную прокрутку
Movement Type	Возможны значения Unrestricted, Elastic или Clamped. Используйте Elastic или Clamped, чтобы содержимое оставалось в границах Scroll Rect. Режим Elastic возвращает контент, когда он достигает края прокрутки

Elasticity	Количество отказов, используемых в режиме эластичности
Inertia	Когда установлена инерция, содержимое будет продолжать двигаться, когда указатель отпущен после перетаскивания. Если инерция не установлена, содержимое будет перемещаться только при перетаскивании
Deceleration Rate	Когда инерция установлена, скорость замедления определяет, как быстро содержимое перестает двигаться. Уровень 0 немедленно остановит движение. Значение 1 означает, что движение никогда не замедлится
Scroll Sensitivity	Чувствительность к колесу прокрутки
Viewport	Ссылка на область просмотра Rect Transform, которая является родителем содержимого Rect Transform
Horizontal Scrollbar	Необязательная ссылка на элемент горизонтальной полосы прокрутки (Scrollbar)
Visibility	Должна ли полоса прокрутки автоматически скрываться, когда она не нужна
Spacing	Пространство между полосой прокрутки и окном просмотра
Vertical Scrollbar	Необязательная ссылка на элемент вертикальной полосы прокрутки
Visibility	Должна ли полоса прокрутки автоматически скрываться, когда она не нужна
Spacing	Пространство между полосой прокрутки и окном просмотра

9.2.6. Выпадающий список

Выпадающий список (объект **Dropdown**) используется для предоставления пользователю возможности выбора одного из элементов.

Элемент управления показывает текущий выбранный вариант. После щелчка на списке открывается список параметров, и пользователь может выбрать

другое значение из списка. При выборе нового значения список снова закрывается, а в элементе управления отображается новая выбранная опция. Список также закрывается, если пользователь нажимает на сам элемент управления или где-либо еще внутри холста (Canvas).

Список генерирует событие On Value Changed, которое генерируется при изменении значения списка.

Свойства объекта Dropdown приведены в табл. 9.6.

Таблица 9.6. Свойства объекта Dropdown

Свойство	Описание
Interactable	Будет ли этот компонент принимать входные данные от пользователя или контролироваться только через скрипт?
Transition	Определяет, как элемент управления визуально реагирует на действия пользователя
Navigation	Определяет последовательность элементов управления.
Template	Позволяет выбрать шаблон выпадающего списка
Caption Text	Компонент Text для хранения текста выбранного в данный момент параметра (необязательный)
Caption Image	Компонент Image для хранения изображения выбранного в данный момент параметра (необязательный)
Item Text	Компонент Text для хранения текста элемента (необязательный)
Item Image	Компонент Image для хранения изображения элемента (необязательный)
Value	Индекс текущей выбранной опции. 0 – первый вариант, 1 – второй и т.д.
Options	Список возможных вариантов. Для каждого варианта можно указать текст и картинку

9.2.7. Поле ввода

Поле ввода (Input Field) позволяет организовать текстовое поле ввода, позволяющее получить какую-то информацию от игрока, например, имя игрока. Свойства данного объекта описаны в табл. 9.7.

Таблица 9.7. Свойства поля ввода

Свойство	Описание
Interactable	Логическое значение, определяющее, можно ли взаимодействовать с полем ввода или нет
Transition	Переходы используются для установки того, как изменяется поле ввода при изменении его состояния (возможны состояния Normal, Highlighted, Pressed, Disabled
Navigation	Определяет последовательность элементов управления
TextComponent	Ссылка на элемент Text, используемый в качестве содержимого поля ввода
Text	Начальное значение. Значение, которое отображается по умолчанию до редактирования пользователем
Character Limit	Значение максимального количества символов, которое можно ввести в поле ввода
Content Type	Тип контента
Standard	Можно ввести любой символ
Autocorrected	Автокоррекция отслеживает ввод пользователя и предлагает ввести ему наиболее подходящий вариант замены, заменяя набранный пользователем текст автоматически, если пользователь явно не отменяет действие
Integer Number	Только целые числа
Decimal Number	Только цифры
Alphanumeric	Можно вводить, как буквы, так и цифры
Name	Автоматически делает заглавными буквы первую букву каждого слова

Email Address	Позволяет вводить буквенно-цифровую строку, похожую на e-mail.
Password*	Используется для ввода пароля, закрывает введенные символы звездочкой
Pin	Скрывает символы со звездочкой. Позволяет вводить только целые числа
Custom	Пользовательский формат, задаваемый параметрами Line Type, Input Type, Keyboard Type, Character Validation
Line Type	Определяет, как тест отформатирован внутри текстового поля
Single Line	Разрешает ввести только одну строку
Multi Line Submit	Позволяет ввести несколько строк. Новая строка используется только при необходимости
Multi Line Newline	Позволяет ввести несколько строк. Пользователь может использовать новую строку, нажав клавишу Enter
Placeholder	Ссылка на элемент Text, который будет использован в качестве заполнителя, например, «Введите имя…»
Caret Blink Rate	Определяет частоту мерцания для курсора
Selection Color	Цвет фона выделенной части текста
Hide Mobile Input (только для iOS)	Скрывает собственное поле ввода, прикрепленное к экранной клавиатуре на мобильных устройствах. Обратите внимание, что это работает только на устройствах с iOS

Внимание! Чтобы получить текст, введенный пользователем в поле ввода, используйте свойство Text объекта InputType, а не свойство Text компонента, отображающего текст. Например, при вводе пароля свойство компонента Text может содержать звездочки, в то время как свойство Text поля ввода – реальный пароль.

9.3. Создание сцены с параметрами игры

9.3.1. Подготовка элементов управления

Познакомившись с элементами управления, попытаемся усовершенствовать наш проект ch9, добавив в него еще одну сцену – сцену установки опций игры.

Первым делом нужно изменить нашу сцену **Menu**, добавив в нее еще одну кнопку – кнопку с надписью **Settings**. Новую кнопку назовите SettingsButton и разместите ее так, чтобы она находилась между уже созданными двумя кнопками (рис. 9.22).

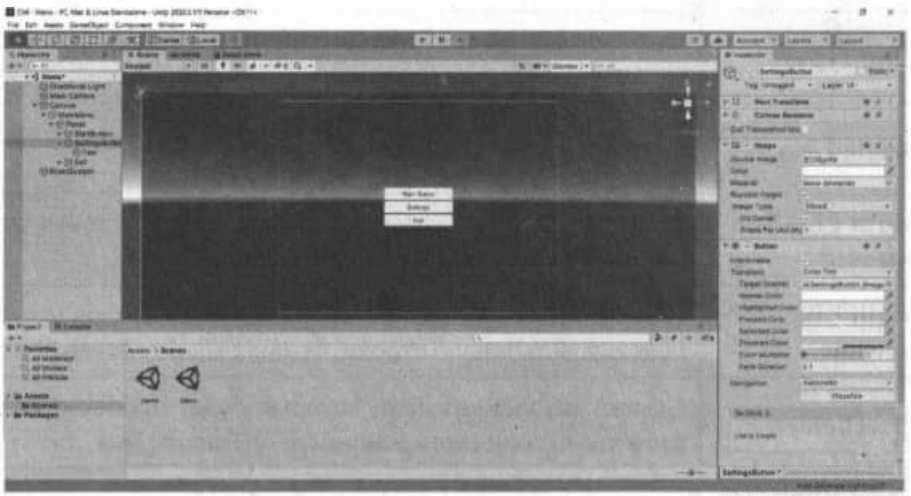
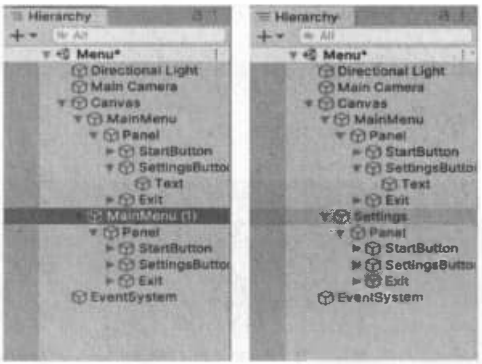


Рис. 9.22. Добавлена еще одна кнопка

Затем по аналогии можно создать еще одну сцену и назвать ее **Settings**, кнопка SettingsButton будет вызывать эту сцену. Так вы могли бы сделать и сами, а я покажу вам один трюк, позволяющий на одной сцене содержать, как главное меню, так и меню установки параметров игры. Во-первых, вы узнаете, что так можно, во-вторых, не будете плодить сцены. В-третьих, вам не придется заново настраивать холст и панель для новой сцены. В общем, одни преимущества, а недостатков нет.

Итак, у нас есть объект MainMenu, который мы создали раньше. Выберите его и нажмите Ctrl + D, чтобы сделать дубликат. У вас появится новый объект, который является дочерним для **Canvas**. Должно получиться, как показано на рис. 9.23.



а. б.

Рис. 9.23. Два объекта MainMenu

Новый объект MainMenu(1) переименуйте в Settings. Сразу после этого отключите MainMenu – так будет удобнее работать с меню **Settings**. Удалите созданные кнопки с панели **Settings.** Теперь нам нужно добавить:

- Переключатель Fullscreen, он будет переключать игру между полноэкранным и оконным режимом (рис. 9.24)

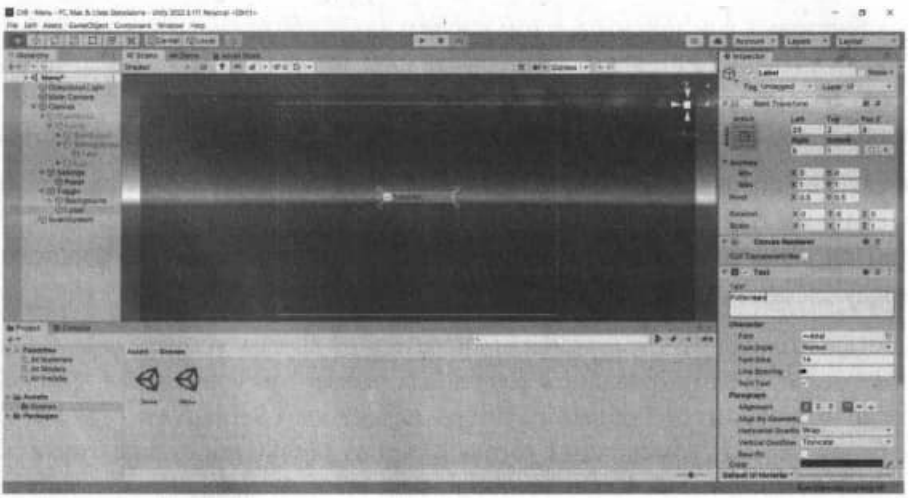


Рис. 9.24. Добавлен переключатель. Обратите внимание, что для изменения надписи мы редактируем свойство "Text" дочернего элемента "Label"

- Элемент **Text** с текстом Volume
- Ползунок **Volume**, который будет регулировать громкость

- Элемент **Text** с текстом Resolution
- Выпадающий список (Dropdown) для выбора разрешений экрана
- Кнопку с надписью **Back**

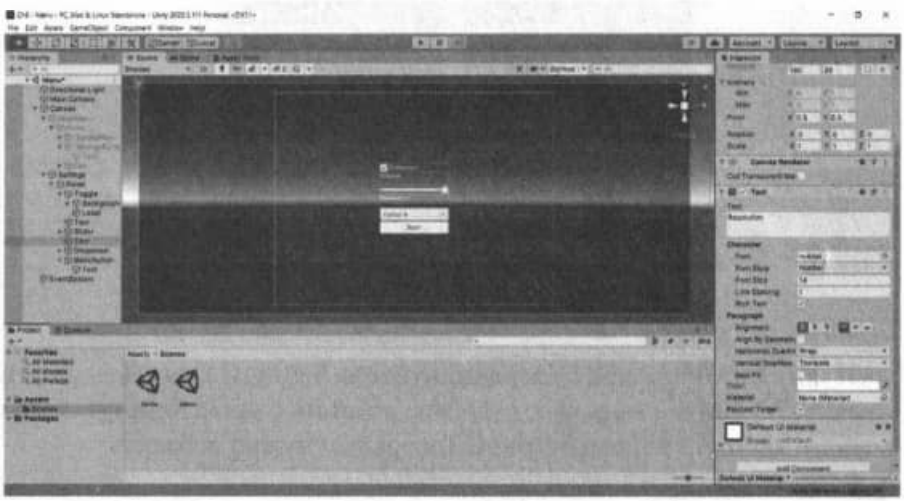
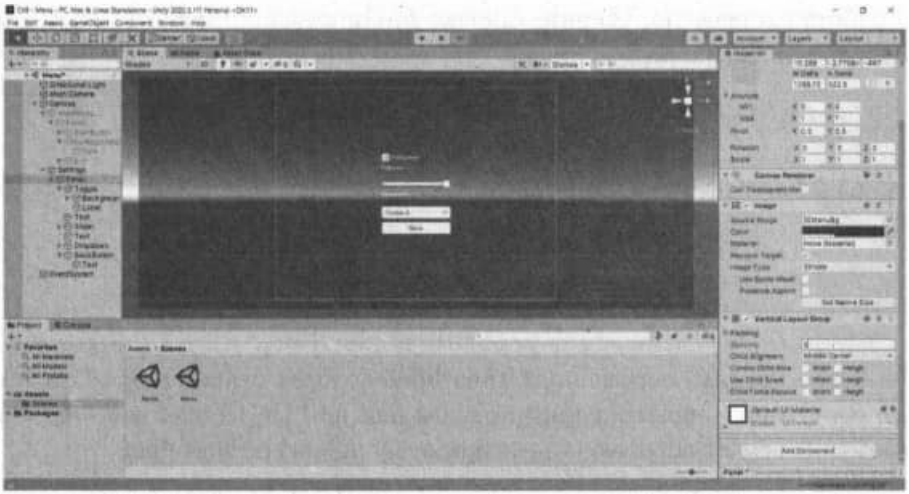


Рис. 9.25. Добавили необходимые элементы

Для текстовых компонентов, в том числе и для чекбокса установите свойство **Color** в значение FFF – так вы сделаете цвет шрифта белым. У нас темный фон, поэтому черный цвет по умолчанию плохо смотрится – его почти не видно. У вас должно получиться что-то вроде того, что показано на рис. 9.25.

Обратите внимание, что смотрятся наши элементы так себе – они находятся слишком близко друг к другу. Выделите Panel и установите ее свойство **Spacing** в значение, большее 1, например, 5. Так вы увеличите расстояние между элементами (рис. 9.26).

С остальными параметрами можете играться по своему усмотрению. Сейчас займемся программированием созданных элементов управления. Создайте отдельный скрипт для объекта Settings, назовем его Settings.cs – так мы уже не будем вносить изменений в готовый скрипт MainMenu.cs и не будем смешивать виски с колой. Как обычно, щелчок правой кнопкой мыши на папке **Assets**, команда **Create, C# Script**. По умолчанию среда сгенерирует скрипт, изображенный на рис. 9.27.



Внимание! Поскольку мы создавали дубликат объекта MainMenu, ему уже назначен сценарий MainMenu.cs. Не забудьте удалить для объекта Settings компонент MainMenu.cs и назначить (путем перетаскивания) другой сценарий – Settings.cs.

Рис. 9.27. Скрипт Settings.cs по умолчанию

9.3.2. Переключатель полноэкранного режима

Начнем с нашего переключателя. Он занимается переключением между полноэкранным режимом – когда включен, и оконным – когда выключен. Чтобы

он работал корректно, уберите галочку **Is On** с нашего переключателя, а в Settings.cs добавьте код:

```
public bool isFullScreen;

public void FullScreenToggle()
  {
      isFullScreen = !isFullScreen;
      Screen.fullScreen = isFullScreen;
  }
```

Мы будем хранить переменную типа **bool**, которая будет отображать текущее состояние – полноэкранный режим или нет. По изменению **toggle** эта переменная будет переключаться на противоположное значение.

У экрана есть свойство Screen.fullScreen типа **bool**. Можно просто присваивать значение нашей переменной isFullScreen этому свойству.

Как видите, все просто. Осталось только в область **On Value Changed()** нашего переключателя перетащить объект **Settings** и выбрать функцию FullScreenToggle(), как мы уже делали ранее. Нужные параметры приведены на рис. 9.28.

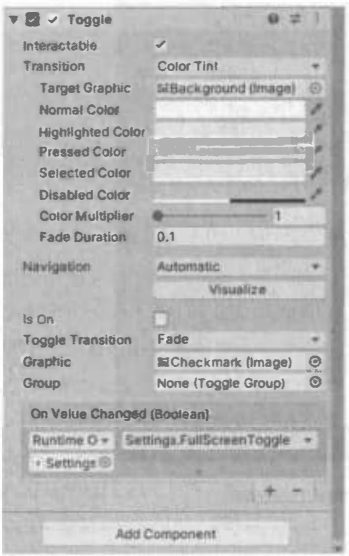


Рис. 9.28. Параметры переключателя

Запустите игру командой **File, Build and Run**. Убедитесь, что переключатель работает корректно. Просто запустить сцену не выйдет. Нужно откомпилировать игру и запустить ее – только, когда игра будет работать как самостоятельное приложение, наш переключатель начнет работать.

9.3.3. Регулировка уровня громкости

Для работы с настройками звука нам для начала понадобится AudioMixer, а также какой-нибудь трек, на котором мы будем проверять работу наших настроек.

Добавим эти два элемента. Сначала добавляем AudioMixer. Правой кнопкой мыши в окне Project: **Create, AudioMixer**. Назовите его **GameSettings** (рис. 9.29).

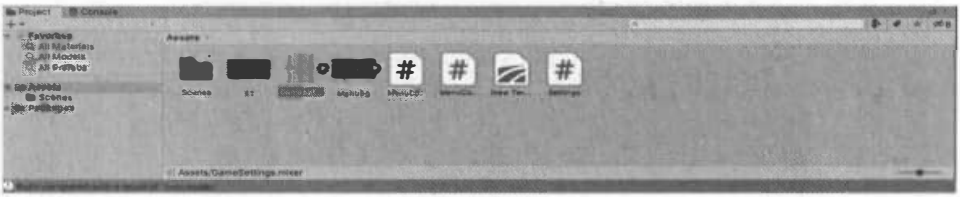


Рис. 9.29. AudioMixer создан

После этого откройте окно AudioMixer командой **Window, Audio, Audio Mixer** (рис. 9.30). Что бы контролировать параметры миксера через скрипт, их нужно сделать видимыми для этого скрипта. Эта процедура называется **Expose Parameters**. Для этого щелкаем на Mixer и в инспекторе находим **Volume** и щелкаем правой кнопкой мыши. Выбираем команду **Expose 'Volume (of Master)'** to script, см. рис. 9.31.

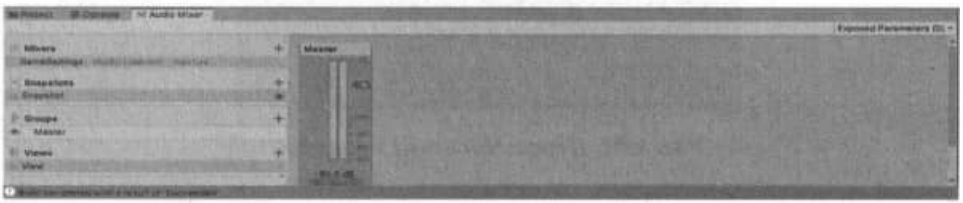


Рис. 9.30 Окно Audio Mixer

Обратите внимание на параметр Exposed Parameters в верхней левой части окна **Audio Mixer**. Было **Exposed Parameters (0)**, стало **Exposed Parameters (1)**. Теперь там есть параметр. Щелкаем на него и называем наш параметр masterVolume (это важно[1]). Следует запомнить имя, которое ему присваиваем – его нужно будет указать в коде (рис. 9.32).

1 Если вы укажете другое имя параметра или не укажете его вообще, то при выполнении сценария получите сообщение Exposed name does not exist и не сможете регулировать уровень громкости

Рис. 9.31. Делаем регулировку громкости доступной в скрипте

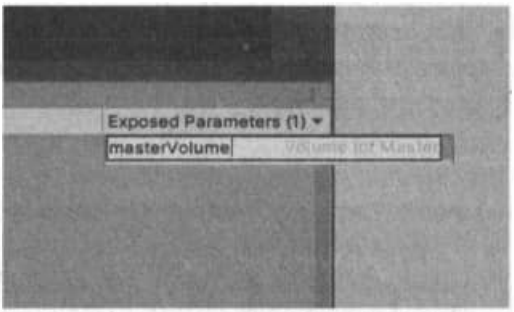


Рис. 9.32. Переименование параметра

Переходим в Settings.cs и создаем поле AudioMixer, чтобы получить ссылку на него в коде:

```
using UnityEngine.Audio;
...
  public AudioMixer am;
```

После этого добавляем метод обработки громкости:

```
  public void AudioVolume(float sliderValue)
    {
        am.SetFloat("masterVolume", sliderValue);
    }
```

Метод SetFloat будет принимать значения нашего слайдера и присваивать это значение параметру "masterVolume". Первым делом перейдите к объекту **Settings** и присвойте наш микшер **GameSettings** свойству **AudioMixer** (рис. 9.33).

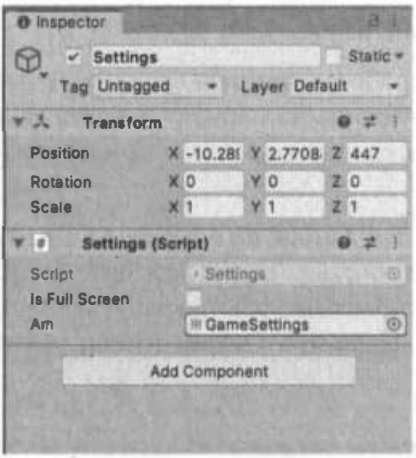


Рис. 9.33. Назначаем микшер

Осталось прикрепить наш метод к событиям слайдера. Находим в инспекторе слайдера поле **On Value Changed** и точно так же прикрепляем объект. Вот только теперь нам надо не просто выбирать метод из списка, а использовать поле **Dynamic float**. Как видите, там уже есть наш метод, и он будет получать переменную от самого слайдера.

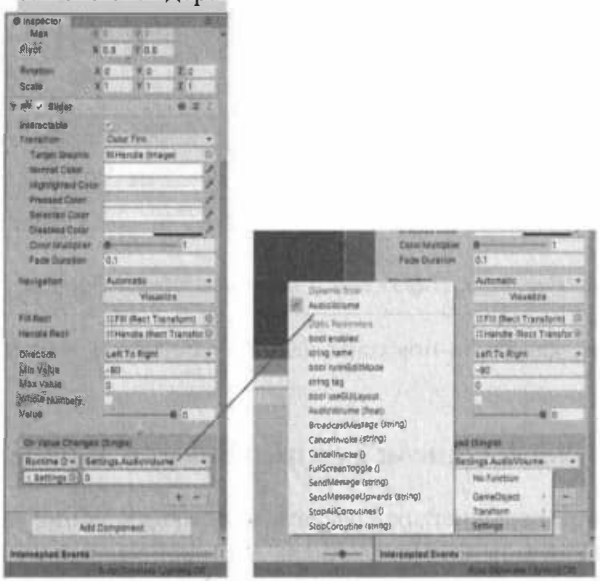


Рис. 9.34. Свойства слайдера

Обратите внимание, что мы напрямую привязываем значение слайдера к значениям аудио-миксера. В аудио миксере громкость изменяется от -80 до 20. Нам же достаточно менять от -80 (нет звука) до 0 (нормальный звук). В настройках слайдера минимальное значение выставляем на -80, максимальное на 0, что и показано на рис. 9.34.

Осталось проверить, работает ли наш слайдер. Добавим компонент **Audio Source.** Его нужно добавлять на Canvas. Скачайте какой-то MP3-файл и импортируйте его в проект путем простого перетаскивания в область обозревателя проекта.

Далее измените свойство **AudioClip** объекта **Audio Source** – выберите ранее импортированный аудио-файл. Также включите свойства **Play On Awake** и **Loop**, чтобы звук воспроизводился при запуске приложения и длился бесконечно. В качестве параметра **Output** нужно указать наш объект **Game Settings**.

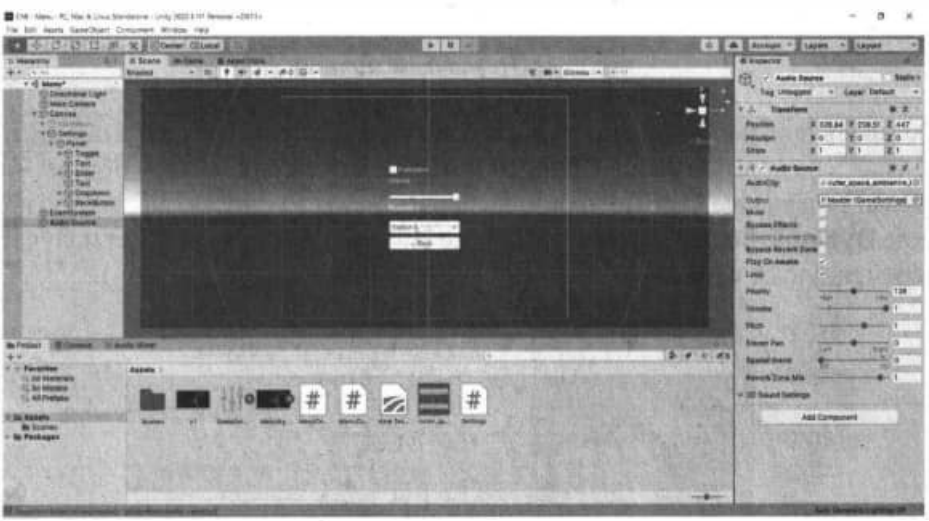


Рис. 9.35. Параметры объекта "Audio Source"

Запустите игру, как было показано выше и убедитесь, что наш слайдер работает.

9.3.4. Список с разрешениями экрана

Экраны у всех разные и наперед угадать, какие разрешения на них будут поддерживаться невозможно. Поэтому для настроек разрешения экрана нужно

сначала получить все возможные разрешения, а потом заполнить список разрешений этими значениями.

Первое что нам понадобится – массив типа Resolution[] где мы будем хранить значения разрешений экрана.

Однако у пунктов выпадающего списка тип – string. Поэтому создаем список List<>, в который мы будем сохранять значения возможных разрешений. Для работы со списками необходимо подключить:

```
using System.Collections.Generic;
```

Также нам понадобится ссылка на соответствующий Dropdown. Для работы с UI элементами следует также прописать:

```
using UnityEngine.UI;
```

В скрипте добавляем следующие поля:

```
Resolution[] rsl;
List<string> resolutions;
public Dropdown dropdown;
```

Инициализацию и заполнение проводим в методе Awake. Этот метод вызывается при запуске объекта, соответственно выполняется раньше, чем все остальные методы.

Получаем значения и каждое из них добавляем в List в формате ширина*высота. После этого очищаем список Dropdown и заполняем его новыми опциями.

```
public void Awake()
    {
        resolutions = new List<string>();
        rsl = Screen.resolutions;
        foreach (var i in rsl)
        {
            resolutions.Add(i.width +"x" + i.height);
        }
        dropdown.ClearOptions();
        dropdown.AddOptions(resolutions);
    }
```

Теперь нужно создать метод, который будет менять разрешение экрана. Как и в предыдущих пунктах – принимать значение будем от UI элемента. Создаем функцию, которая принимает **int**

```
public void Resolution(int r)
    {
        Screen.SetResolution(rsl[r].width, rsl[r].height,
isFullScreen);
    }
```

В метод SetResolution() необходимо передать параметры – ширина, высота и булевскую переменную, отвечающую за полный экран. У нас такая уже есть – это isFullScreen. Передаем ее в функцию.

Дальше не забываем подключить к соответствующему событию наш метод Resolution из группы Dynamic Int, а так же добавить ссылку на нужный Dropdown (рис. 9.36).

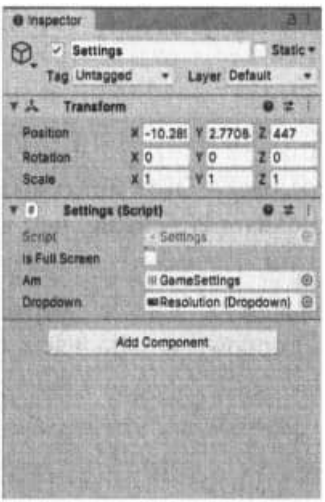


Рис. 9.36. Параметры скрипта. Ссылка на выпадающий список. Если он у вас называется иначе, значение параметра "Dropdown" у вас будет отличаться.

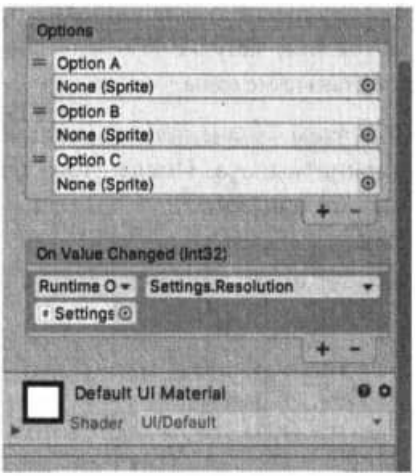


Рис. 9.37. Параметры выпадающего списка. Реакция на событие On Value Changed

9.4. Реакция на кнопки Settings и Back

На данный момент наша игра запускается сразу экрана с настройками, а на главную сцену, как и в основное меню, перейти невозможно. Исправим это.

Для кнопки **Settings** нам не нужно писать никакой код, так как некоторый функционал уже встроен.

Суть в том, что мы будем активировать GameObject. Кнопка **Settings** должна деактивировать объект **MainMenu** и активировать объект Settings. Кнопка **Back**, наоборот, должна *деактивировать объект Settings* и *активировать объект MainMenu*. Выбираем кнопку **Settings** и в OnClick() перетаскиваем наш объект Settings. В функциях выбираем **GameObject.SetActive()** и ставим галочку. Аналогично для объекта MainMenu, но галочку уже не ставим (рис. 9.38).

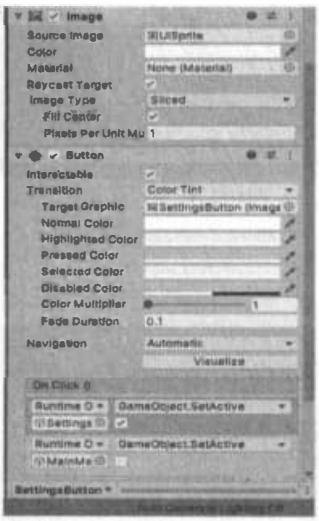


Рис. 9.38. Обработчик нажатия кнопки "Settings". Скрываем MainMenu, отображаем "Settings"

Ну а для кнопки **Back**, которая находится в меню опций, можно таким же образом вызвать метод SetActive() для объекта **Settings**, но на этот раз нам нужно инактивировать наш объект, поэтому мы просто не ставим галочку. Зато ставим галочку для объекта MainMenu.

Рис. 9.39. Обработчик нажатия для кнопки "Back". Деактивируем "Settings", активируем MainMenu

Теперь деактивируйте объект **Settings** и активируйте объект MainMenu. Запустите игру и посмотрим, как она выглядит. На рис. 9.40 показано главное меню, на рис. 9.41 – панель с настройками. Убедитесь, что кнопки **Settings** и **Back** работает, как следует. Если они не работают, значит, вы что-то сделали не так.

Рис. 9.40. Главное меню

Рис. 9.41. Настройки игры

Полный код скрипта Settings.cs приведен в лист. 9.4.

Листинг 9.4. Сценарий Settings.cs

```
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;
using UnityEngine.Audio;
using System.Collections.Generic;
using UnityEngine.UI;

public class Settings : MonoBehaviour
{

  public bool isFullScreen;          // признак полноэкранного
режима
  public AudioMixer am;              // аудио-микшер

  Resolution[] rsl;
  List<string> resolutions;          // массив разрешений
  public Dropdown dropdown;          // ссылка на список с
разрешениями

  public void FullScreenToggle()
    {
        isFullScreen = !isFullScreen;
        Screen.fullScreen = isFullScreen;
    }

  public void AudioVolume(float sliderValue)
    {
        am.SetFloat("masterVolume", sliderValue);
    }


  public void Awake()
    {
        resolutions = new List<string>();
        rsl = Screen.resolutions;    // получаем разрешения
экрана
        // Для каждого разрешения заполняем список
        foreach (var i in rsl)
        {
            resolutions.Add(i.width +"x" + i.height);
        }
        // Очищаем любые элементы
        dropdown.ClearOptions();
        // Добавляем новые элементы
        dropdown.AddOptions(resolutions);
    }

    public void Resolution(int r)
```

```
    {
        // Устанавливаем разрешение
        Screen.SetResolution(rsl[r].width, rsl[r].height,
isFullScreen);
    }
}
```

На этом все. Мы не только разобрались с элементами UI, но и построили экран настроек и главное меню для нашей игры, то есть разобрались, как использовать элементы UI. В следующих главах мы поговорим об анимации: как заставить наш персонаж двигаться.

Глава 10.

Анимация

В компьютерной игре без анимации никак. Хотите, чтобы движение персонажа были реалистичными – нужно использовать анимацию, иначе его перемещение по сцене будет напоминать ситуацию, когда ребенок переставляет игрушку, у которой не двигаются конечности, по поверхности. Вроде бы и перемещается, но руки и ноги не двигаются. Также анимация нужна для создания всякого рода визуальных эффектов, например, вращения ручки при открытии сейфа, вращении колес автомобиля и т.д.

10.1. Обзор системы анимации

Unity обладает сложной системой анимации, которая обеспечивает:

1. Простой рабочий процесс и настройку анимации для всех элементов Unity, включая объекты, персонажей и свойства.
2. Поддержка импортированных анимационных клипов и анимации, созданной в Unity
3. Гуманоидная анимация ретаргетинга – возможность применять анимацию от одной модели персонажа к другой.
4. Упрощенный рабочий процесс для выравнивания анимационных клипов.
5. Удобный предварительный просмотр анимационных клипов, переходов и взаимодействий между ними. Это позволяет аниматорам работать более независимо от программистов, создавать прототипы и просматривать свои анимации до того, как будет написан реальный код.

6. Управление сложными взаимодействиями между анимациями с помощью инструмента визуального программирования.
7. Анимация различных частей тела с различной логикой.
8. Особенности наслоения и маскировки

Система анимации Unity основана на концепции анимационных роликов, которые содержат информацию о том, как определенные объекты должны со временем изменять свое положение, вращение или другие свойства. Представим, что есть модель человека, которая может стоять (*idle*), идти (*walk*), бежать (*run*) и прыгать (*jump*). Так вот для каждого из этих состояний должен быть свой анимационный ролик.

Каждый клип можно рассматривать как одну линейную запись. Анимационные клипы из внешних источников создаются художниками или аниматорами с помощью сторонних инструментов, таких как 3D Max или Maya.

Анимационные клипы затем организуются в структурированную систему, похожую на блок-схему, называемую **Animator Controller**. Анимационный контроллер действует как «конечный автомат», отслеживающий, какой клип должен воспроизводиться в данный момент, и когда анимации должны меняться или смешиваться. Также он содержит информацию о переходах из одного состояния в другое. Например, человек, который бежит, не может остановиться мгновенно, поэтому он сначала должен перейти в состояние *walk*, а затем – в *idle*.

Очень простой анимационный контроллер может содержать только один или два клипа, например, для состояний простоя и ходьбы. Более продвинутый анимационный контроллер может содержать десятки гуманоидных анимаций для всех действий главного героя и может смешиваться между несколькими клипами одновременно, чтобы обеспечить плавное движение при перемещении игрока по сцене.

Система анимации Unity также имеет множество специальных функций для обработки гуманоидных персонажей, которые дают вам возможность перенастроить гуманоидную анимацию из любого источника в вашу собственную модель персонажа. Для этого в Unity используется система Avatar, позволяющая человекоподобным персонажам отображаться в общий внутренний формат. Для получения более подробной информации обратитесь к документации[1].

[1] См. https://docs.unity3d.com/ru/current/Manual/class-Avatar.html

Все эти части – анимационные клипы, анимационный контроллер и аватар – собраны в игровом объекте посредством компонента Animator. У данного компонента есть ссылка на анимационный контроллер и, если нужно, на аватар для этой модели. Анимационный контроллер, в свою очередь, содержит ссылки на используемые анимационные клипы.

Рассмотрим диаграмму, изображенную на рис. 10.1:

1. Анимационные клипы импортируются из внешнего источника или создаются в Unity. Чтобы потренироваться, вы можете загрузить бесплатные модели из магазина ассетов. Вот, например, неплохой пакет *https://assetstore.unity.com/packages/3d/animations/basic-motions-free-pack-154271*.
2. Анимационные клипы размещаются и располагаются в анимационном контроллере. Область 2 содержит вид анимационного контроллера в окне Animator. Состояния модели (которые могут представлять анимацию или вложенные подсостояния) отображаются как узлы, соединенные линиями. Анимационный контроллер существует как ассет в окне обозревателя проекта.

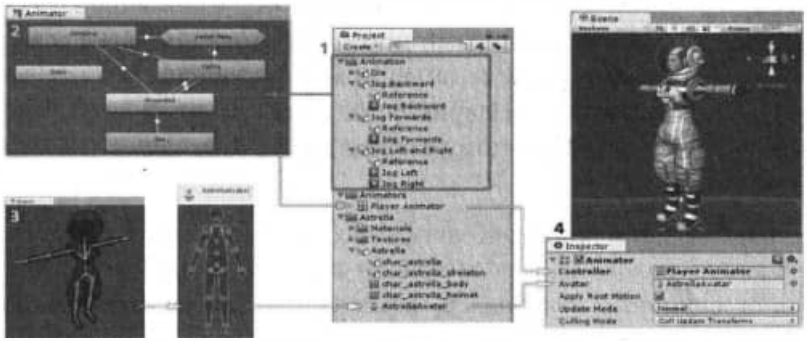


Рис. 10.1. Система анимации

3. Модель персонажа имеет особую конфигурацию костей, которые были преобразованы в общий формат аватара Unity. Виден скелет, а также основные точки, в которых может происходить движение костей.
4. При анимации модели персонажа к ней добавляется компонент Animator. В окне **Inspector** виден компонент **Animator**, которому назначены анимационный контроллер и аватар. Аниматор использует их вместе, чтобы оживить модель. Ссылка на аватар необходима только при анимации человекоподобного персонажа. Для других типов анимации требуется только анимационный контроллер.

10.2. Вращающийся куб

Разумеется, анимировать можно не только человекоподобные модели. При желании вы можете анимировать все, что угодно в Unity. Рассмотрим простой пример анимации модели куба. Изложенные здесь принципы вы сможете использовать на практике при анимации более сложных моделей.

Создайте новый проект ch10 и добавьте на сцену модель куба (меню **Gameobject, 3D Object, Cube**), рис. 10.2.

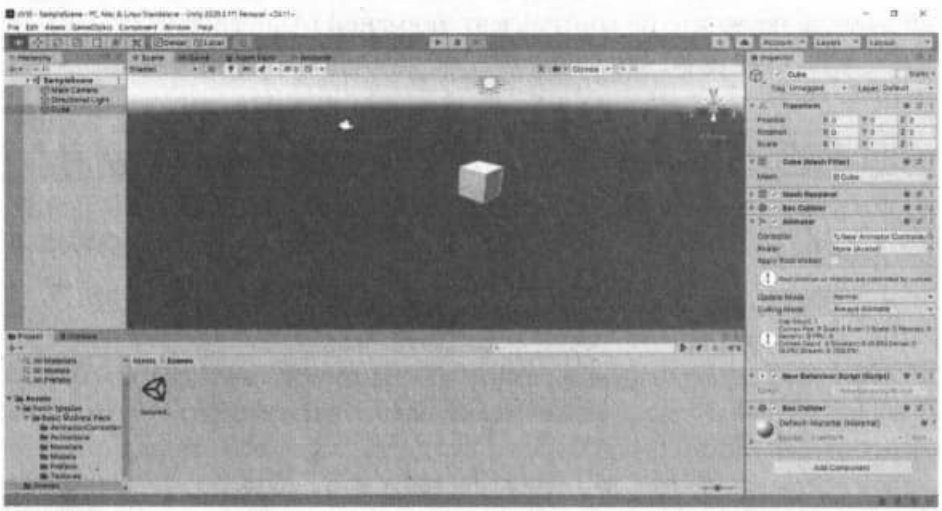


Рис. 10.2. Модель куба добавлена на сцену

В окне обозревателя проектов создайте все, что вам понадобится при работе с этой сценой. Щелкните правой кнопкой мыши на **Assets** и затем выберите поочередно следующие команды меню:

- **Create, Animation controller** – создает анимационный контроллер
- **Create, Animation** – создает анимацию
- **Create, C# Script** – создает скрипт

Созданные ассеты показаны на рис. 10.3. Имена можете оставить по умолчанию, а можете переименовать так, как вам больше нравится. Далее я буду работать с именами по умолчанию.

Далее будет показано, как создать анимацию куба, которая будет активироваться по нажатию левой кнопки мыши.

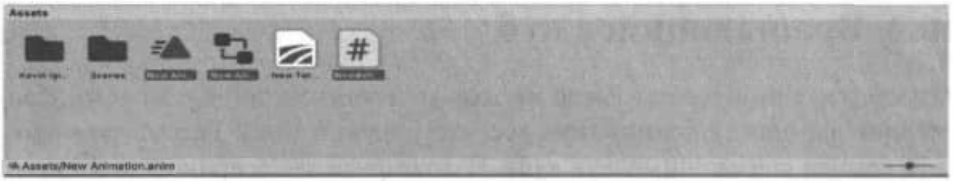


Рис. 10.3. Созданные ассеты

Первым делом нужно создать состояние, которое заставит куб вращаться. Это состояние мы будем вызывать из нашего скрипта. Для создания состояний дважды щелкните по контроллеру анимации (Animation controller). По умолчанию, у вас будет три состояния, которые никак не связаны друг с другом (рис. 10.4).

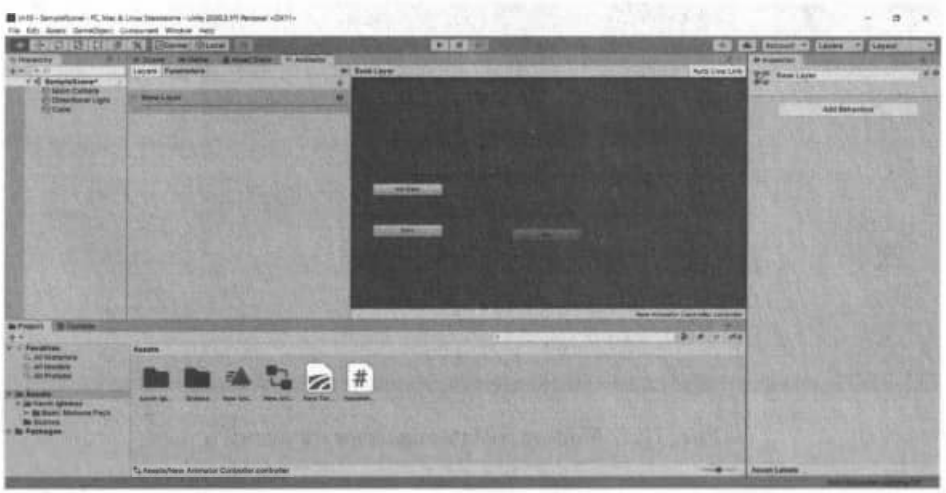


Рис. 10.4. Состояния анимационного контроллера по умолчанию

Чтобы создать состояние, щелкните на пустой области правой кнопкой мыши и выберите команду **Create state, Empty**. Создайте состояние – оно будет отображено в виде оранжевого прямоугольника. Переименуйте его в Normal. Связь между Entry и Normal будет создана автоматически. Это означает, что по умолчанию объект переходит в состояние Normal.

Создайте второе состояние и назовите его MouseDownState. Должно получиться, как показано на рис. 10.5.

Теперь нужно создать переходы из состояния Normal в MouseDownState и обратно. Линии со стрелками называются переходами. Для создания перехода щелкните правой кнопкой мыши на состоянии Normal и выберите команду **Make Transition**. Далее щелкните на состоянии MouseDownState, чтобы

создать переход из Normal в MouseDownState. Снимите флажок **Has Exit Time**. Должно получиться, как показано на рис. 10.6.

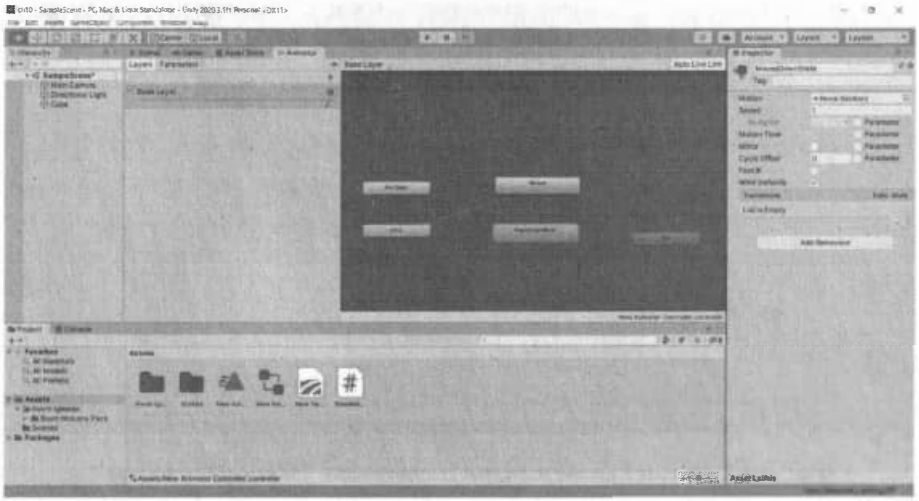


Рис. 10.5. Процесс настройки анимационного контроллера

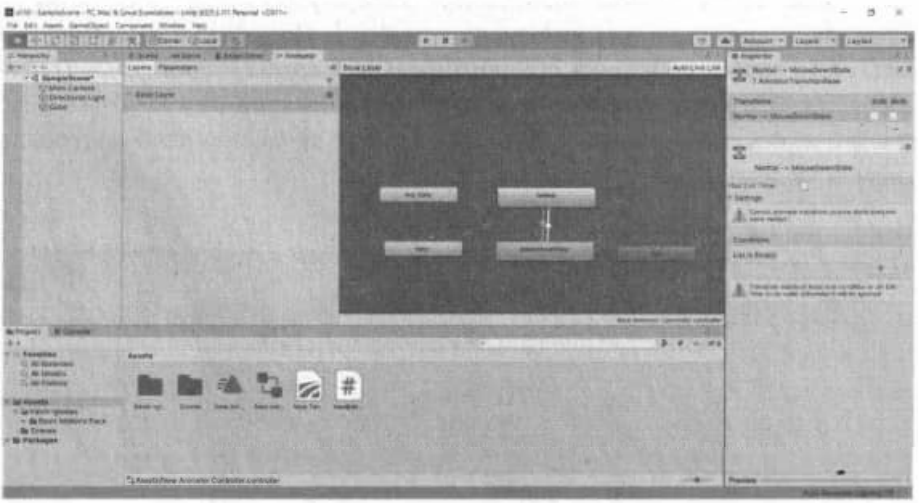


Рис. 10.6. Состояния настроены

Состояния настроены, однако сей факт не заставит наш куб вращаться. Нам нужно добавить анимацию в состояние MouseDownState. Выберите это состояние и перетащите анимацию, которую вы недавно создали, в поле **Motion**, как показано на рис. 10.7. Этим вы назначили анимацию данному состоянию, то есть когда наш объект будет входить в данное состояние, будет воспроизводиться назначенная вами анимация.

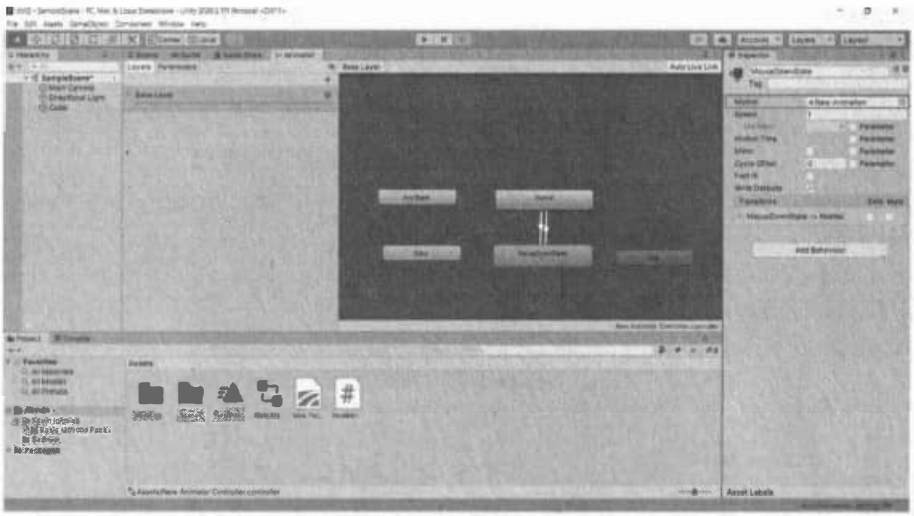


Рис. 10.7. Анимация назначена состоянию "MouseDownState"

Теперь у нас есть анимационный контроллер, внутри которого созданы два состояния и одному из них назначена анимация. Но сам куб еще ничего не знает о том, что мы собрались его анимировать. Поэтому вернитесь на сцену, выберите куб, а затем перетащите контроллер анимации на куб. То же самое сделайте и для сценария (рис. 10.8). Созданный анимационный контроллер назначен модели куба.

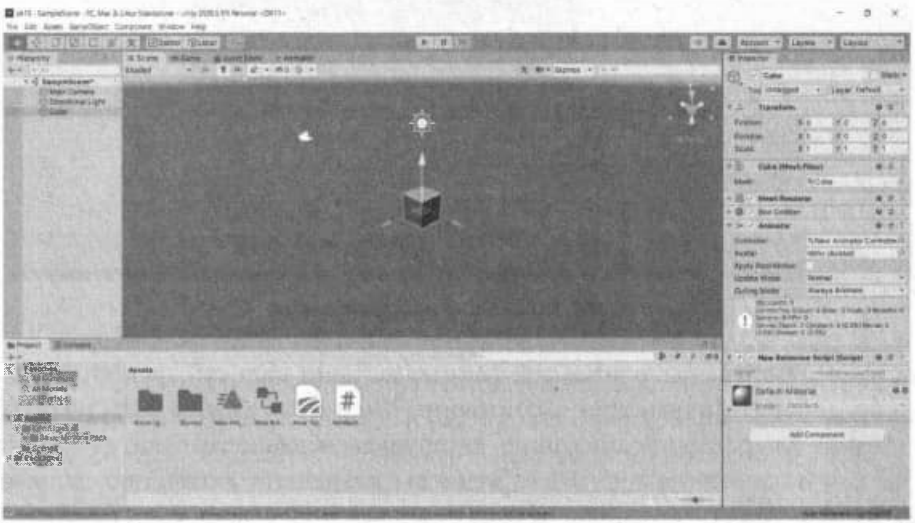


Рис. 10.8. Кубу назначены анимационный контроллер и сценарий

Убедись, что куб все еще выбран и нажмите кнопку **Add Component** внизу инспектора свойств. Добавьте компонент Physics, Box Collider. Он нужен, чтобы определить, был ли произведен щелчок мышью на игровом объекте.

Вот теперь мы можем приступить, собственно, к созданию самой анимации. Выберите куб в иерархии и нажмите Ctrl + 6. Это откроет окно **Animation**. Добавьте анимацию вращения (Rotation). Для выбора типа анимации используйте кнопку **Add Property**, затем выберите тип анимации – **Transform**. В списке на рис. 10.9 нет свойства **Rotation**, поскольку оно уже было выбрано. По окончанию всех действий у вас должна быть строчка **Cube:Rotation** в окне анимации.

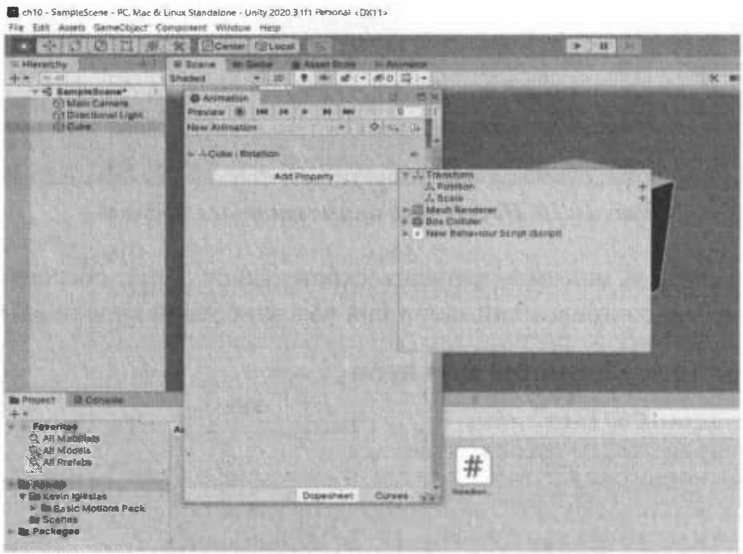


Рис. 10.9. Настройка анимации для куба

Теперь самое интересное. Разверните окно **Animation** (рис. 10.10). Нужно заставить куб вращаться. Идеально, если у вас два монитора – тогда окно анимации можно переместить на второй монитор, чтобы вы видели, как вращается модель. Если у вас один монитор, то окно анимации можно поместить под сцену, чтобы вам был виден куб.

Итак, для начальной позиции установите свойство **Rotation.z** равным 0. Затем перетащите маркер времени вправо, например, на одну минуту вправо, и для конечной позиции установите значение 90 градусов, как показано на рис. 10.10. Можно указать промежуточные значения, если вы хотите, чтобы куб вращался неравномерно, например, для времени 0:45 установить зна-

чение 45 градусов. Тогда куб будет немного дергаться в процессе поворота. Нажмите кнопку **Play**, чтобы посмотреть что получилось.

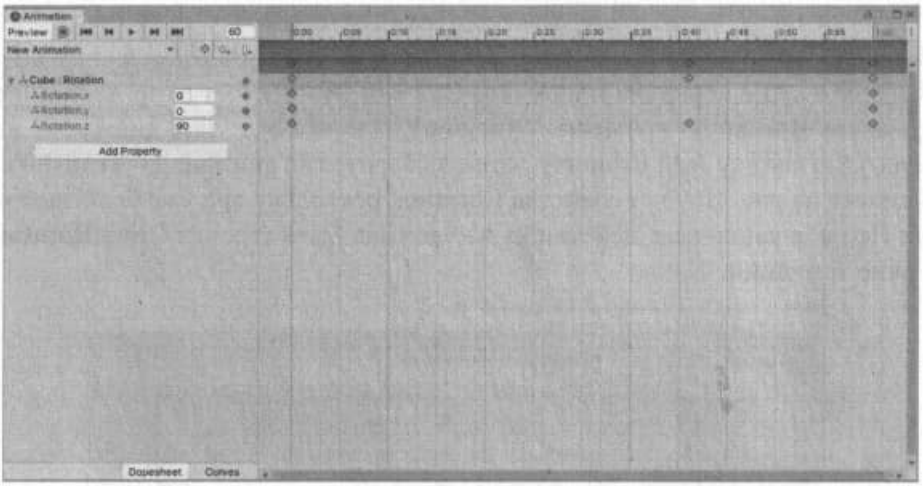


Рис. 10.10. Настройка анимационного эффекта

Осталось дело за малым – написать скрипт (лист. 10.1), обеспечивающий переход куба в состояние анимации при нажатии левой кнопки мыши.

Листинг 10.1. Сценарий для куба

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class NewBehaviourScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        // Определить, нажата ли левая кнопка мыши
   if (Input.GetMouseButtonDown (0))
    {
     var ray = Camera.main.ScreenPointToRay(Input.
mousePosition);
```

```
    RaycastHit raycastHit;

     if (Physics.Raycast (ray, out raycastHit, 100))
     {
        // получаем коллайдер, по которому щелкнули
        var colliderHit = raycastHit.collider;
        // получить объект игры, к которому привязан
        //  коллайдер
        var gameObjectHit = colliderHit.gameObject;

        // получить аниматор gameObjects
       var animator = gameObjectHit.GetComponent <Animator>
();

        // воспроизведение анимации
        animator.Play ("MouseDownState");
     }
  }
   }
}
```

Запустите игру, если вы все сделали правильно, наш куб будет вращаться при щелчке мыши на нем (рис. 10.11).

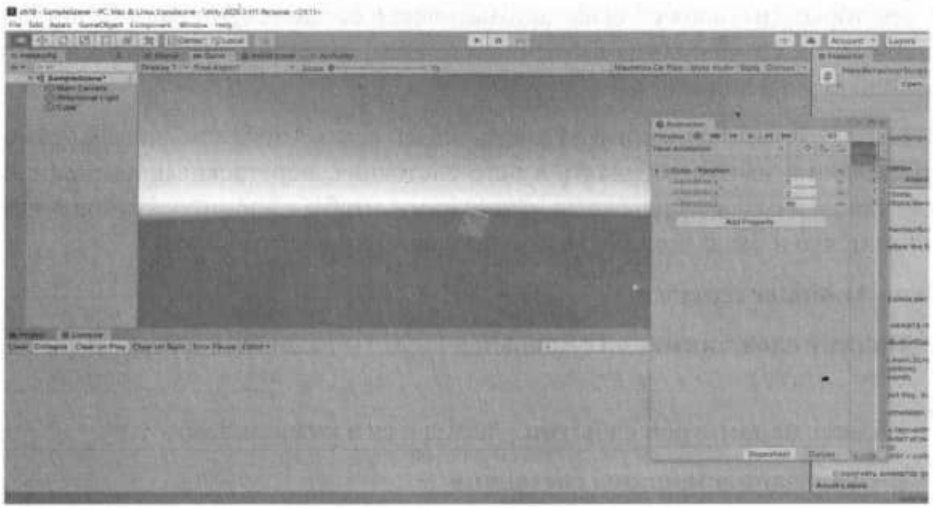


Рис. 10.11. Куб вращается

Если подытожить, то вы только что узнали:

- Как настроить состояния персонажа и переходы между ними с помощью анимационного контроллера

- Как создать анимацию для игрового объекта
- Как перевести игровой объект в нужное состояние анимации

Пример хоть и прост, но приведенные в нем принципы могут использоваться для работы с более сложными моделями.

10.3. Анимационный контроллер

Данный раздел полностью посвящен анимационному контроллеру. В прошлом разделе вы уже попробовали поработать с ним на практике и представляете, зачем он нужен.

Анимационный контролер позволяет настраивать и управлять набором анимаций персонажа или иного анимированного **Game Object**.

Контроллер имеет ссылки на используемые им анимационные клипы, и управляет различными анимационными состояниями и переходами между ними, используя, так называемый **State Machine** (конечный автомат), который может быть представлен в виде блок-схемы, или простой программы, написанной на языке визуального программирования в Unity.

В некоторых ситуациях для вас автоматически создается Animator Controller, например, в случае, когда вы начинаете анимировать новый GameObject с помощью окна анимации.

В других случаях вы захотите создать новый ассет Animator Controller самостоятельно и начать добавлять в него состояния, перетаскивая анимационные клипы и создавая переходы между ними, чтобы сформировать конечный автомат, что и было показано в предыдущем разделе.

Окно **Animator** содержит:

- **Виджет слоя анимаций** (Animation Layer) – находится в верхнем левом углу
- **Виджет параметров события** – находится в нижнем левом углу
- **Визуализацию машины состояний**

> **Примечание.** Окно Animator будет всегда отображать машину состояний из последнего выбранного ассета .controller, независимо от того какая сцена загружена.

Состояния анимации (*animation states*) – основные элементы машины состояний анимации (Animation State Machine). Каждое состояние содержит индивидуальную анимационную последовательность (или *blend tree*), которая будет проигрываться, когда персонаж находится в этом состоянии. В том случае, когда игра вызовет переход состояния, персонаж сменит состояние на другое, чья последовательность анимации будет использована.

Когда вы выбираете состояние из инспектора Animator Controller, вы увидите свойства для этого состояния (рис. 10.12, табл. 10.1).

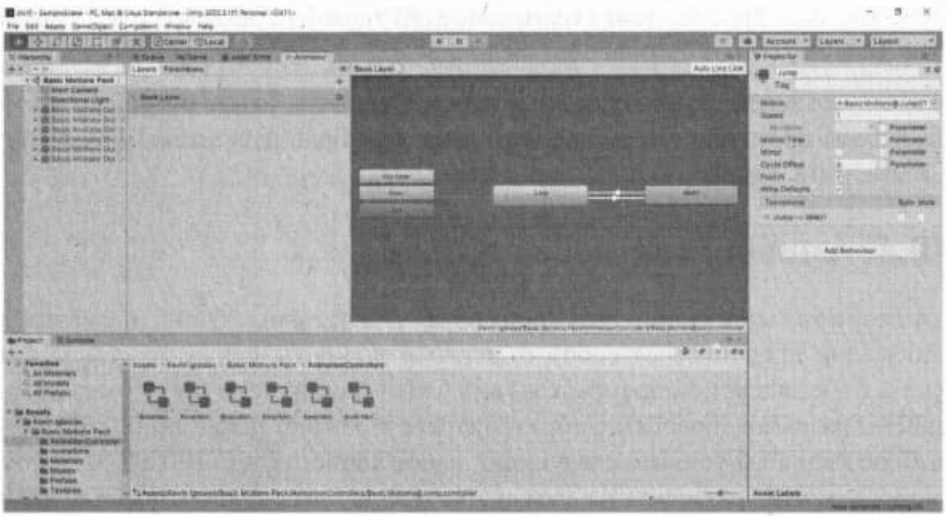


Рис. 10.12. Анимационные состояния: свойства

Таблица 10.1. Свойства анимационного состояния

Свойство	Описание
Speed	Стандартная скорость анимации
Motion	Клип анимации, назначенный на это состояние
Foot IK	Нужно ли учитывать IK ног для этого состояния
Write Defaults	Записывает ли AnimatorStates значения по умолчанию для свойств, которые не анимированы его движением
Mirror	Должно ли состояние зеркалироваться. Применимо только к человекоподобной анимации
Transitions	Список переходов из этого состояния

Состояние по умолчанию отображается оранжевым цветом. Это состояние, в котором машина будет при активации. Вы можете изменить то, какое состояние будет стандартным, нажав правой кнопкой на нужном состоянии и выбрав из контекстного меню **Set As Layer Default State**.

Any State – особое состояние, которое всегда существует. Оно подходит для ситуаций, когда вы хотите переключиться на определенное состояние, независимо от текущего состояния. Это более быстрый способ, нежели добавление исходящего перехода каждому состоянию. Учтите, что в виду своей функции, Any State не может быть конечной точкой перехода (т.е. переход к "any state" не может быть использован для выбора случайного состояния).

Как вы уже знаете, новое состояние можно добавить, нажав правой кнопкой на пустом пространстве в окне **Animator** и выбрав из контекстного меню **Create State, Empty**.

10.4. Анимационные клипы

Анимационные клипы – это крошечные строительные блоки анимации в Unity. Они представляют собой отдельные элементы движения, такие как RunLeft (бег влево), Jump (прыжок) или Crawl (ползание) и могут быть объединены разными способами, образуя живые и реалистичные анимации. При выборе клипа вы увидите следующий набор свойств (рис. 10.13). При этом свойства можно разделить на две группы – те, которые применяются ко всем клипам набора (представлены в табл. 10.2а) и те, которые устанавливаются для каждого клипа отдельно (табл. 10.2б).

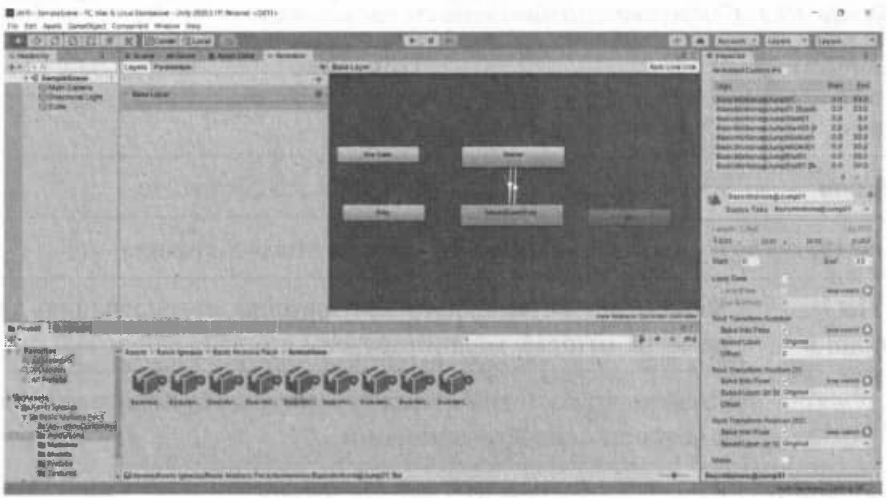


Рис. 10.13. Свойства анимационного клипа

Таблица 10.2а. Свойства анимационного клипа (применяются ко всем клипам в наборе)

Свойство	Описание
Import Animation	Следует ли импортировать анимацию из этого ресурса? Если этот флажок не установлен, все остальные параметры на этой странице скрыты, и анимация не импортируется
Bake Animations	Анимации, созданные с использованием IK или Simulation, будут запекаться для пересылки кинематических ключевых кадров. Эта опция доступна только для файлов Maya, 3dsMax и Cinema4D
Resample Curves	Если включено, кривые анимации будут пересэмплироваться в каждом кадре. Вы должны включить этот параметр, если у вас есть проблемы с интерполяцией между ключами в вашей оригинальной анимации. Отключите его, чтобы сохранить кривые анимации, как они были изначально созданы. Этот параметр доступен только для общих анимаций, но не для анимации человекоподобных моделей
Import Animation	Следует ли импортировать анимацию из этого ресурса? Если этот флажок не установлен, все остальные параметры на этой странице скрыты, и анимация не импортируется
Anim. Compression	Тип сжатия, используемый при импорте анимации
- Off	Без сжатия
- Keyframe Reduction	Удаляет лишние ключевые кадры
- Optimal	Сжатие на усмотрение Unity

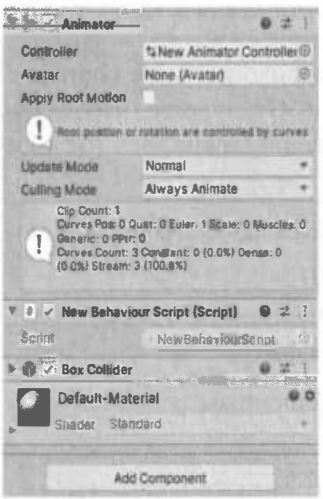
Таблица 10.2б. Свойства анимационного клипа (устанавливаются отдельно для каждого клипа)

Свойство	Описание
Name	*Имя клипа*
Source Take	Метка в исходном файле анимации, которую следует использовать в качестве данного клипа (эта опция отобразится, если в анимационном файле более одной метки). Это то, что разделяет анимации между собой в Motionbuilder, Maya и других пакетах для 3D моделирования. Unity может импортировать такие метки как отдельные клипы, или вы можете создать клип из всего файла либо из отдельной метки
Start	Начальный кадр клипа
End	Конечный кадр клипа
Loop Time	Включите эту опцию, чтобы анимационный клип проигрывался и начинался заново при достижении окончания
Loop Pose	Включите для плавного зацикливания движения
Cycle Offset	Смещение для цикла зацикленной анимации, если нам требуется начать его с другого времени
Root Transform Rotation	Корневая трансформация вращения
Bake into Pose	Включите, чтобы «запечь» корневое вращение в движение костей. Отключите, чтобы корневое вращение хранилось как root motion
Based Upon	На чем основывается корневое вращение
- Original	Сохраняет вращение таким, каким оно было создано в исходном файле
- Body Orientation	Сохраняет прямое направление верхней части тела
Offset	Смещение корневого вращения (в градусах)

Root Transform Position (Y)	Позиция корневой трансформации (оси XZ)
Bake into Pose	Включите опцию, чтобы «запечь» вертикальное корневое движение в движение костей. Отключите, чтобы вертикальное корневое движение хранилось как root motion
Based Upon	На чем основывается вертикальное корневое положение
- Original	Сохраняет вертикальное положение таким, каким оно было создано в исходном файле
- Center of Mass	Сохраняет центр масс совмещенным с положением корневой трансформации
- Feet	Сохраняет ступню совмещенной с положением корневой трансформации
Offset	Смещение вертикального корневого положения
Root Transform Position (XZ)	Позиция корневой трансформации (оси XZ)
Bake into Pose	Включите, чтобы «запечь» горизонтальное корневое движение в движение костей. Отключите, чтобы горизонтальное корневое движение хранилось как root motion
Based Upon	На чем основывается горизонтальное корневое положение
- Original	Сохраняет горизонтальное положение таким, каким оно было создано в исходном файле
- Center of Mass	Сохраняет центр масс совмещенным с положением корневой трансформации
Offset	Смещение горизонтального корневого положения
Mirror	Зеркально отразить этот клип (поменять местами левую и правую части)

Additive Reference Pose	Когда этот параметр включен, позволяет определить опорную позу, используемую в качестве основы для аддитивной анимации
Mask	Маска для тела (Body mask) и маска трансформаций (Transform mask), примененные к данному анимационному клипу
Curves	Кривые, связанные с параметрами
Events	Используется для создания нового события на клипе
Motion	Позволяет определить пользовательский корневой узел движения
Import Messages	Предоставляет вам информацию о том, как была импортирована ваша анимация, включая дополнительный отчет 'Retargeting Quality Report'

10.5. Компонент "Animator"



Компонент **Animator** используется для назначения анимации GameObject в вашей сцене. Компонент **Animator** требует ссылки на анимационный контроллер (Animator Controller), определяющий, какие анимационные клипы использовать, и контролирует, когда и как смешивать и переходить между ними.

Если ваш объект является человекоподобным персонажем, обязательно назначение свойства Avatar (рис. 10.14).

Рис. 10.14. Свойства компонента "Animator"

Таблица 10.3. Свойства компонента Animator

Свойство	Функция
Controller	Анимационный контроллера данного персонажа
Avatar	Аватар персонажа (только для человекоподобных персонажей)
Apply Root Motion	Что должно контролировать положение персонажа – сама анимация или код?
Update Mode	Позволяет выбрать, когда Animator обновляется и какую временную шкалу он должен использовать
Normal	Аниматор обновляется синхронно с вызовом Update(), а скорость аниматора соответствует текущему графику. Если шкала времени замедлена, анимации также будет замедлена
Animate Physics	Аниматор будет синхронизироваться с вызовом FixedUpdate(), то есть в связке с физической системы. Этот режим используется, если вы анимируете движение объектов с физическими взаимодействиями, такими как персонажи, которые могут толкать объекты твердого тела вокруг
Unscaled Time	Аниматор обновляется синхронно с вызовом Update, но скорость аниматора игнорирует текущую шкалу времени и анимируется со скоростью 100% независимо. Это полезно для анимации системы графического интерфейса с нормальной скоростью при использовании измененных временных шкал для специальных эффектов или для приостановки игрового процесса
Culling Mode	Режим куллинга для анимаций (в каких случаях анимация не должна проигрываться)
Always Animate	Всегда анимировать, не производить куллинг

Cull Update Transforms	Retarget, IK и запись трансформаций отключены, когда средства визуализации не видны
Cull Completely	Анимация полностью выключена, если не видны рендеры

Информационное поле в нижней части компонента **Animator** предоставляет статистику по всем клипам, используемых контроллером **Animator**.

Анимационный клип содержит данные в форме «кривых», которые задают, как значение изменяется со временем. Эти кривые могут описывать положение или вращение объекта, изгиб мышцы в человекоподобной системе анимации или другие анимированные значения в клипе, такие как изменение цвета материала.

Таблица 10.4 объясняет, что представляет каждый элемент в этих данных.

Таблица 10.4. Информация о кривых

Метка	Описание
Clip Count	Общее число анимационных клипов, используемых анимационным контроллером, который назначен Animator
Curves (Pos, Rot & Scale)	Общее количество кривых, используемых для анимации положения, поворота или масштаба объектов. Они предназначены для анимированных объектов, которые не являются частью стандартной человекоподобной установки. При анимации человекоподобного аватара эти кривые показывают количество лишних немышечных костей
Muscles	Количество кривых мышечной анимации, используемых аниматором для анимации гуманоидов. Эти кривые используются для анимации стандартных гуманоидных мышц аватара. Наряду со стандартными движениями мышц для всех гуманоидных костей в стандартном аватаре Unity, он также включает две «мышечные кривые», которые хранят положение движения корня и анимацию вращения
Generic	Количество числовых (float) кривых, используемых аниматором для анимации других свойств, таких как цвет материала

PPtr	Общее количество кривых спрайтовой анимации (только для 2D)
Curves Count	Общее объединенное количество кривых анимации
Constant	Количество анимационных кривых, которые оптимизируются как постоянные (неизменные) значения. Unity выбирает это автоматически, если ваши файлы анимации содержат кривые с неизменными значениями
Dense	Количество анимационных кривых, оптимизированных с использованием «плотного» метода хранения данных (дискретные значения, которые интерполируются между собой линейно). Этот метод использует значительно меньше памяти, чем «потоковый» метод
Stream	Количество анимационных кривых с использованием «потокового» метода хранения данных (значения со временем и данные касательной для изогнутой интерполяции). Эти данные занимают значительно больше памяти, чем при использовании Dense-метода

10.6. Пакет "Basic Motions"

Хорошие человекоподобные модели стоят достаточно дорого. Не думаю, что у начинающего читателя есть желание покупать подобные пакеты пусть даже за 100$. При создании серьезной игры подобные вложения – капля в море на фоне стоимости разработки моделей персонажей с нуля. Но на данном этапе, когда нужно освоить Unity, хочется поработать с человекоподобными моделями без каких-либо движений.

В магазине ассетов Unity есть бесплатные пакеты, используя которые вы можете изучить базовые движения человекоподобной модели. Один из таких пакетов – Basic Motions Free:

https://assetstore.unity.com/packages/3d/animations/basic-motions-free-pack-154271

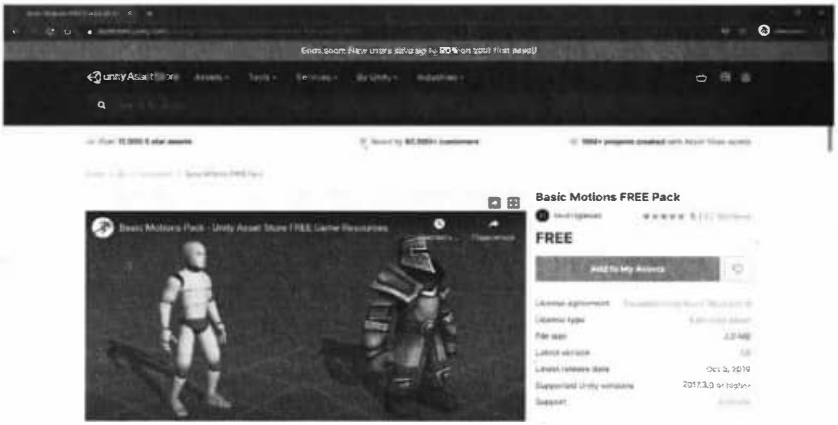


Рис. 10.15. Пакет "Basic Motions Free" в магазине ассетов Unity

Это не единственный бесплатный пакет. Вы также можете обратить свое внимание на следующие пакеты:

- Space Robot Kyle (*https://assetstore.unity.com/packages/3d/characters/robots/space-robot-kyle-4696*) – модель космического робота
- 3D Game Kit (*https://assetstore.unity.com/packages/templates/tutorials/3d-game-kit-115747*) – готовый пакет трехмерной игры, в котором есть все, в том числе и модели персонажей
- Zombie (*https://assetstore.unity.com/packages/3d/characters/humanoids/zombie-30232*) – анимированный набор зомби, поддерживающий различные анимационные состояния

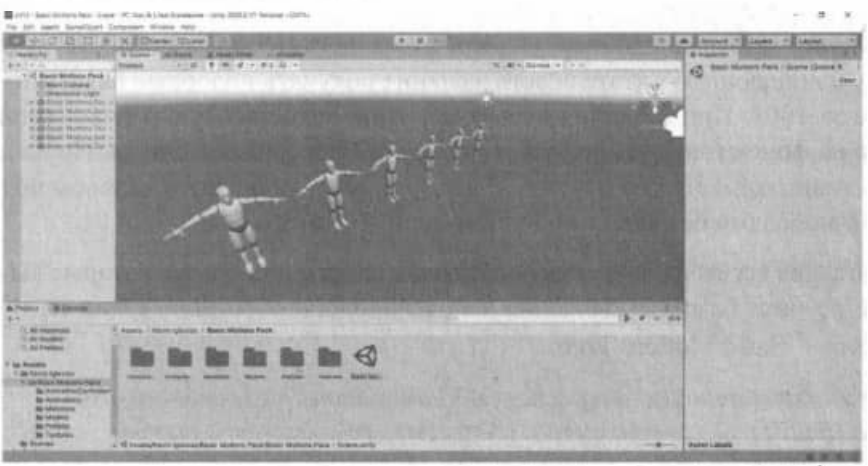


Рис. 10.16. Сцена "Basic Motions Free"

Все эти пакеты вы сможете освоить самостоятельно, а мы пока сконцентрируемся на пакете **Basic Motions Free**. Загрузите и импортируйте его. Откройте сцену Basic Motion Pack. Вы увидите пять персонажей (рис. 10.16). Каждый из них будет демонстрировать одно состояние анимации. Запустите игру, чтобы увидеть, что произойдет (рис. 10.17).

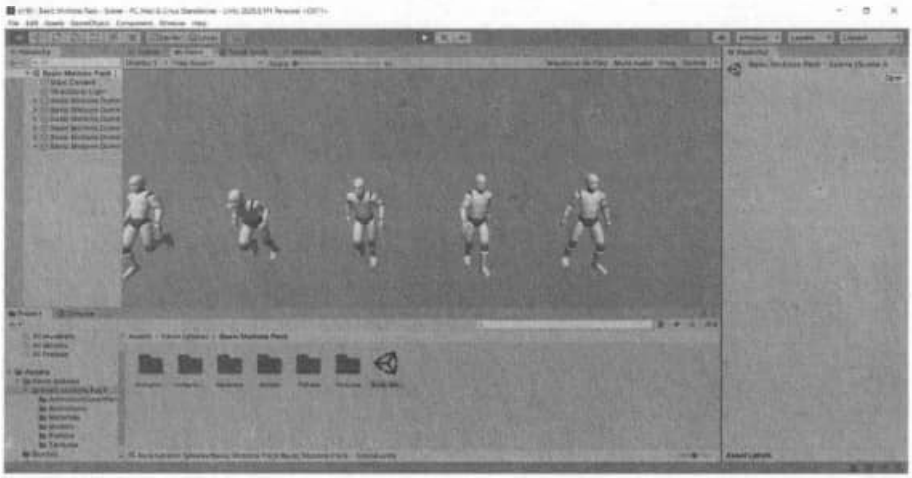


Рис. 10.17. Игра запущена: каждый персонаж демонстрирует определенный тип движения

Для каждого типа движения создан собственный анимационный контроллер. Перейдите в папку AnimationController и откройте контроллер BasicMotions@Jump.controller (рис. 10.18). Состояние *Jump* – это состояние прыжка. Из этого состояния, как видно на рис. 10.18, можно перейти в состояние покоя и обратно.

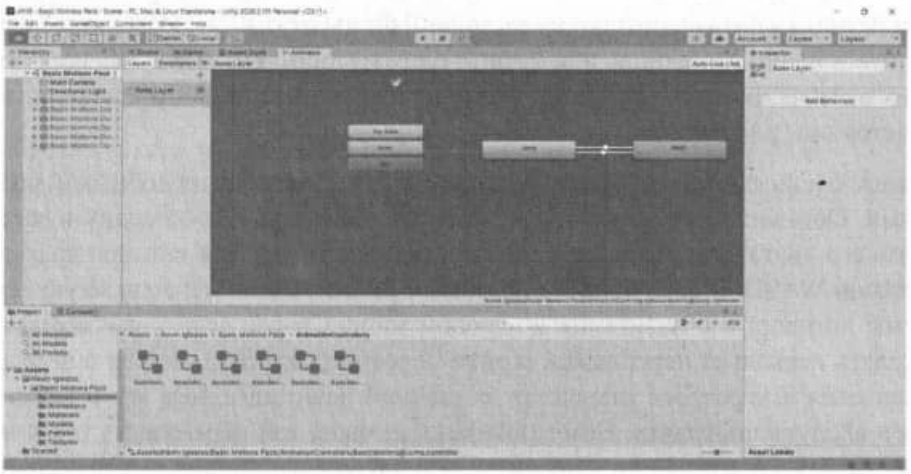


Рис. 10.18. Просмотр состояний анимационного контроллера

«Пройдитесь» по созданным персонажам в окне иерархии. При щелчке на каждом персонаже в окне **Animator** будет отображаться связанным с ним анимационный контроллер и сможете понять, что делает тот или иной персонаж (рис. 10.19).

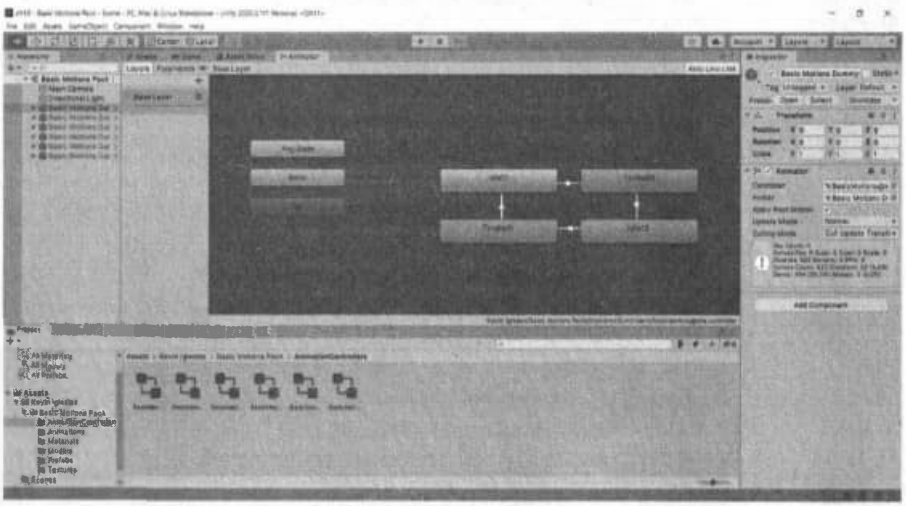


Рис. 10.19. Анимационный контроллер для первого персонажа. Переход из одного состояния простоя в другое

Также обратите внимание на компонент **Animator** для каждого персонажа – задается анимационный контроллер и аватар (свойство **Avatar**).

Наверное, при запуске сцены вы обратили внимание, что бегущие персонажи (которым назначено одно из состояний бега) бегут/идут постоянно. Перейдите в папку Animations и выберите одну из анимаций бега. Вы заметите, что включено свойство **Loop Time**, чтобы анимация воспроизводилась постоянно (рис. 10.20).

Пакет очень достойный для обучения работе с человекоподобными моделями. Осталось дело за малым – добавить объект на новую сцену и заставить его двигаться, например, при щелчке на нем мышкой или при нажатии клавиш WASD. В следующих главах мы рассмотрим не менее важную тему – мы поговорим о навигации в игровом мире. Скорее всего, вам захочется создать несколько персонажей (кроме персонажа игрока), но как они будут двигаться в игре? Все зависит от созданной навигации, чем мы и займемся в следующих главах. Будет показано сначала, как перемещать обычный цилиндр, но осмысленно – по карте. Мы научим наш цилиндр перемещать-

ся по лабиринту, искать оптимальный путь, научим обходить движущиеся препятствия. А затем превратим цилиндр в модель человека и научим его двигаться.

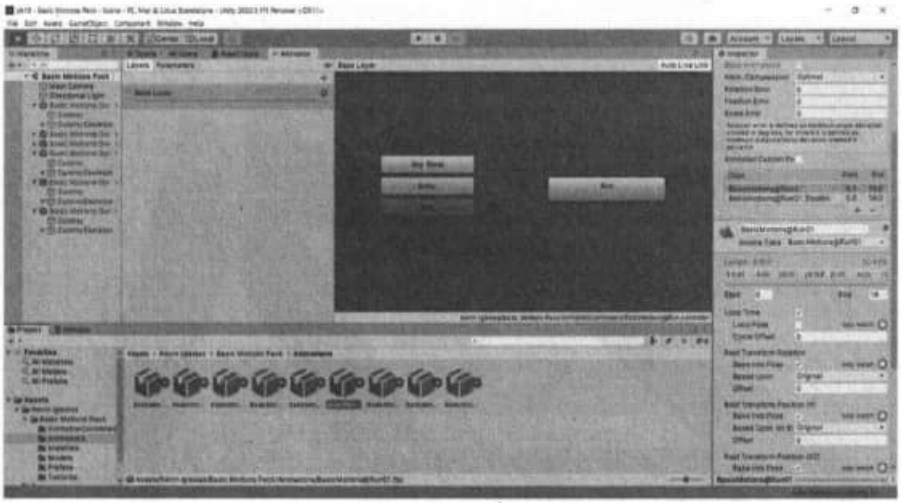
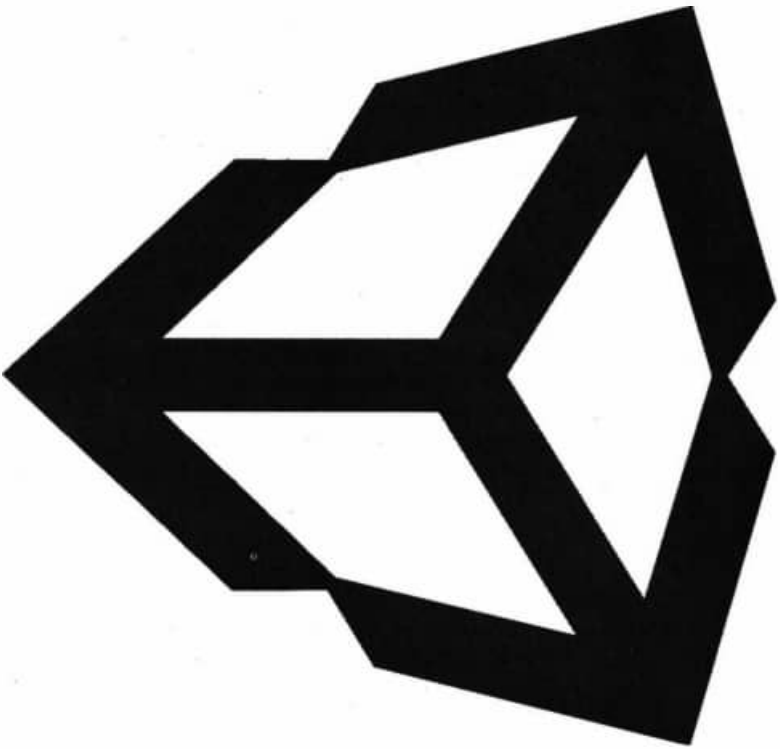


Рис. 10.20. Настройка анимационного клипа



Глава 11.

Навигация в игре

Навигационная система позволяет вам создавать персонажей, разумно перемещающихся перемещаться по игровому миру, используя навигационные сетки, которые создаются автоматически из вашей геометрии сцены. Динамические препятствия позволяют вам изменять навигацию персонажей во время выполнения. В этой главе описываются системы навигации и поиска путей Unity.

11.1. Основные компоненты навигационной системы

Система навигации Unity позволяет создавать персонажей, которые могут перемещаться по игровому миру. Основными компонентами системы являются:

- **Структура NavMesh** (сокращение от Navigation Mesh) – это структура данных, описывающая поверхности, по которым можно пройти в игровом мире и позволяющая найти путь прохождения от одного места к другому в вашем игровом мире. Структура данных создается автоматически из геометрии вашего уровня.

- **Компонент NavMesh Agent** поможет вам создавать персонажей, которые избегают друг друга при движении к своей цели. Для этого агенты используют NavMash и стараются не встречаться друг с другом.
- **Компонент Off-Mesh Link** позволяет включать навигационные ярлыки, которые не могут быть представлены с помощью пешеходной поверхности. Например, перепрыгивание через канаву или забор – все это можно описать с помощью этого компонента.
- **Компонент NavMesh Obstacle** позволяет описывать движущиеся препятствия, которых должны избегать агенты во время навигации по миру. Примером такого препятствия может послужить катящаяся бочка. Агенты будут «видеть» такие препятствия и будут стараться их обойти.

11.2. Как работает навигационная система

Когда вам нужно разумно перемещать персонажей игры, вам нужно решить две проблемы: выяснить куда идти (то есть найти место назначения на сцене) и выяснить, как именно туда пройти. Эти две задачи тесно связаны, но сами по себе – разные. Выяснить куда идти – это более глобальная задача, выяснить как именно – более локальная, поскольку нужно учитывать только направление движения и способы обхода препятствий.

Навигационная система нуждается в собственных данных, чтобы представить зоны прохождения в игровой сцене. Обходные зоны определяют места на сцене, где агент может стоять и двигаться. В Unity агенты описаны как цилиндры. Проходная зона автоматически строится из геометрии сцены путем тестирования мест, где может стоять агент. Затем локации соединяются с поверхностью, лежащей поверх геометрии сцены. Эта поверхность называется сеткой навигации (для краткости здесь и далее NavMesh).

NavMesh хранит эту поверхность в виде выпуклых многоугольников, что является очень удобным представлением, поскольку мы знаем, что между любыми двумя точками внутри многоугольника нет препятствий. Помимо границ полигонов, мы храним информацию о том, какие полигоны являются соседями друг к другу. Это позволяет нам рассуждать о всей области, по которой мы можем перемещаться.

Чтобы найти путь между двумя местоположениями в сцене, нам сначала нужно сопоставить начальные и конечные местоположения с их ближайшими полигонами. Затем мы начинаем поиск с начального местоположения, посещая всех соседей, пока не достигнем конечного полигона. Отслежива-

ние посещенных полигонов позволяет нам найти последовательность полигонов, которая приведет от начала к месту назначения. Распространенным алгоритмом поиска пути является A*, который использует Unity.

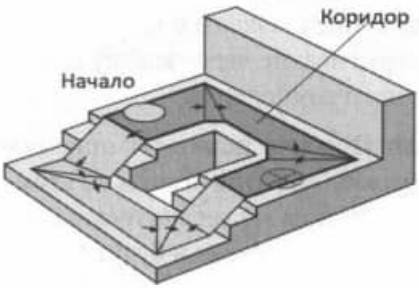


Рис. 11.1. Поиск пути

Последовательность полигонов, которые описывают путь от начала до конечного полигона, называется коридором. Агент достигнет пункта назначения, всегда поворачивая к следующему видимому углу коридора. Если у вас есть простая игра, в которой на сцене движется только один агент, то можно найти все углы коридора одним махом и оживить персонажа, чтобы он двигался вдоль отрезков, соединяющих углы.

При работе с несколькими агентами, перемещающимися одновременно, они должны отклоняться от первоначального пути, избегая друг друга. Попытка исправить такие отклонения, используя путь, состоящий из отрезков, вскоре становится очень трудной и подверженной ошибкам.

11.3. Обход препятствий

По пути следования к месту назначения вычисляется желаемая скорость, необходимая для достижения пункта назначения. Использование желаемой скорости перемещения агента может привести к столкновению с другими агентами.

Предотвращение препятствий выбирает новую скорость, которая балансирует между движением в желаемом направлении и предотвращением будущих столкновений с другими агентами и краями навигационной сетки. Unity использует взаимные скоростные препятствия (RVO) для прогнозирования и предотвращения столкновений.

Однако на практике препятствия – это не только другие агенты. Это могут быть стены, ящики, бочки, транспортные средства. С препятствиями можно справиться с помощью локального обхода препятствий или глобального поиска пути.

Когда препятствие движется, лучше всего справляться с использованием местных препятствий. Таким образом, агент может прогнозируемо избежать препятствия. Когда препятствие становится стационарным и может рассматриваться как *блокирующее путь всех агентов*, препятствие можно внести в глобальную навигацию, то есть на сетку навигации.

В разделе 11.1 мы упоминали компонент Off-Mesh Link. Связи между полигонами NavMesh описываются с помощью ссылок внутри системы поиска пути. Иногда необходимо разрешить агенту перемещаться по местам, которые нельзя пройти, например, перепрыгивая через забор или проходя через закрытую дверь. В этих случаях нужно знать такие места (расположение двери, место, где можно перепрыгнуть забор т.д.).

Эти действия могут быть аннотированы с помощью ссылок вне сетки (Off-Mesh Link), которые сообщают указателю пути, что маршрут существует через указанную ссылку. Чтобы было понятнее, посмотрите на рис. 11.2, где визуально продемонстрировано, что представляет собой Off-Mesh Link.

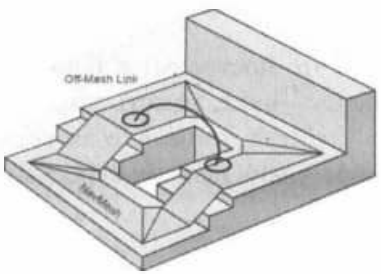


Рис. 11.2. Ссылка вне сетки

11.4. Создание NavMesh

Продемонстрируем создание NavMesh на практике. Создайте следующую сцену: пол и несколько стен по сцене. Поместите на сцену цилиндр – он будет вместо персонажа, перемещаемого по сцене. Можете создать подобную сцену самостоятельно, а можете использовать сцену из проекта ch11. В нем

будет несколько сцен, демонстрирующих работу навигационной системы. На рис. 11.3 показана наша сцена.

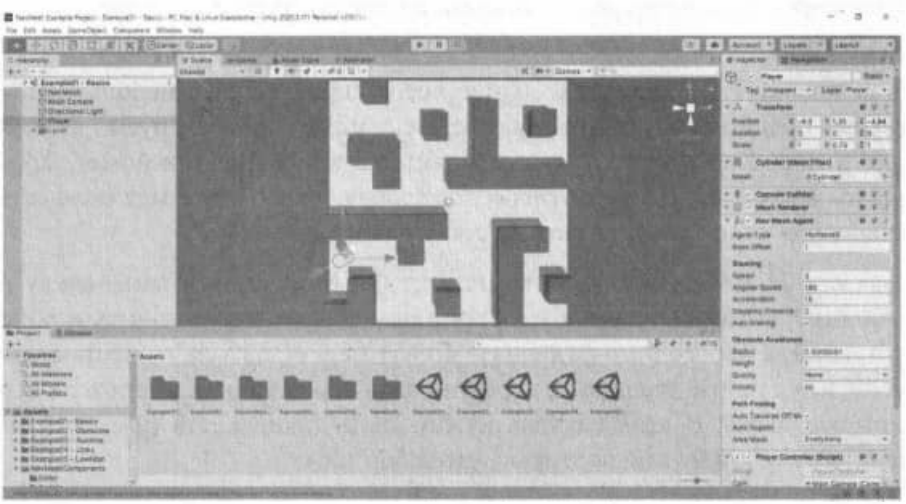


Рис. 11.3. Созданная сцена

На сцене есть несколько объектов – камера, направленный свет, Player (это будет наш персонаж), а также объект Level, состоящий из стен (Walls) и пола (Ground).

Первое, что мы сделаем – это построим (в терминологии Unity этот процесс называется «выпечкой» – **bake**, то есть выражение «запечь» NavMash не должно вас смущать) NavMesh, которая будет содержать области нашего уровня, доступные для «прогулки» нашему персонажу.

Выполните следующие действия:

1. Создайте пустой объект (**GameObject, Create Empty**);
2. Переименуйте его в NavMesh;
3. Перетащите его в самый верх иерархии;
4. Выберите объект NavMesh в иерархии. В окне инспектора нажмите кнопку **Add Component** и добавьте компонента **Nav Mesh Surface**;
5. У вас появится область **Nav Mesh Surface (script)** с некоторыми параметрами. Пока не обращайте на них внимания. Просто нажмите кнопку Bake для «запекания» навигационной карты;

6. Убедитесь, что включен флажок **Show NavMesh** в области **Navmesh Display**.

Посмотрим, что у нас получилось (рис. 11.4). Синяя область – это и есть наш NavMesh. По этой области может передвигаться наш персонаж. Синяя область – доступная для перемещения, белая область и сами объекты – недоступны. Вокруг стационарных объектов, таких как стены, создается белая область, чтобы персонаж не мог «войти» в стену.

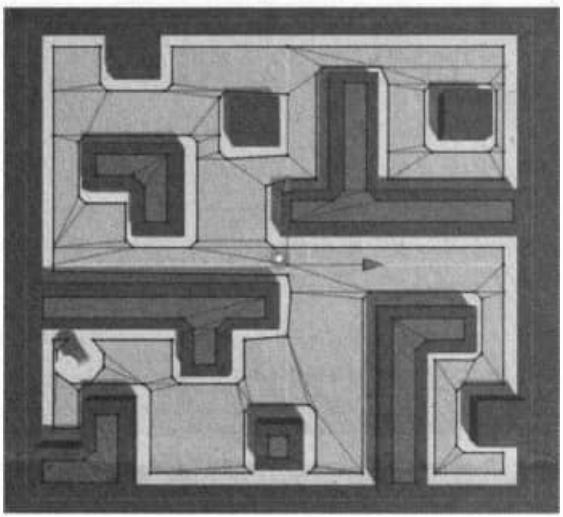


Рис. 11.4. Созданный NavMesh

Но поздравлять вас пока рано. Обратите внимание (на рис. 11.4 специально так отрегулирован вид, чтобы сей момент был заметен), что область вокруг нашего цилиндра помечена белым, а не синим, следовательно, по ней он не сможет прогуливаться. Почему так? А потому, что наш NavMesh ничего не знает о нашем игроке. Ведь он не знает, что цилиндр, помещенный на сцену, будет нашим персонажем.

Решить данную проблему можно с помощью слоев. Посмотрите на параметр **Include Layers**. Он содержит слои, которые будут использоваться при построении карты. Принцип следующий: создать слой для нашего игрока и исключить его из NavMesh.

Теперь приступим к реализации:

1. Откройте окно **Tag & Layers** (**Edit, Project Settings, Tag & Layers**) и создайте слой Player (рис. 11.5);
2. Выберите в иерархии наш объект **Player** и для него установите слой Player – измените параметр **Layer** (рис. 11.6);
3. Перейдите к объекту NavMesh. Разверните список **Include Layers** и снимите флажок со слоя Player (рис. 11.7);
4. Нажмите кнопку **Bake**.

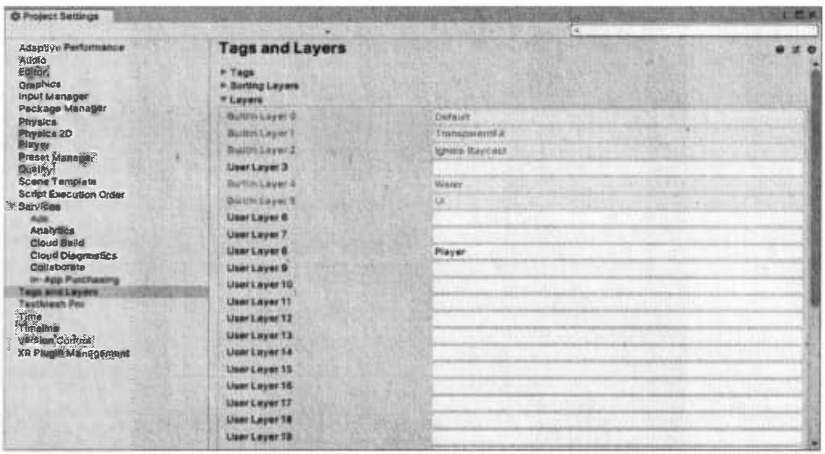


Рис. 11.5. Создан слой "Player"

Рис. 11.6. Установка слоя для игрока

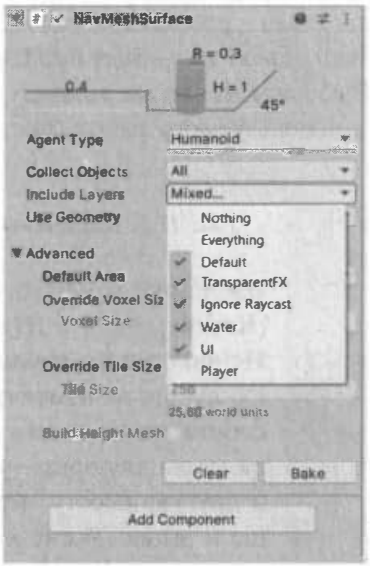


Рис. 11.7. Слой "Player" не будет использован при построении NavMesh

Теперь мы получили совсем другую картину (рис. 11.8). Наш цилиндр стоит на синей области, а это означает, что она доступна для «прогулки» по ней. Вот теперь наша навигационная сетка создана.

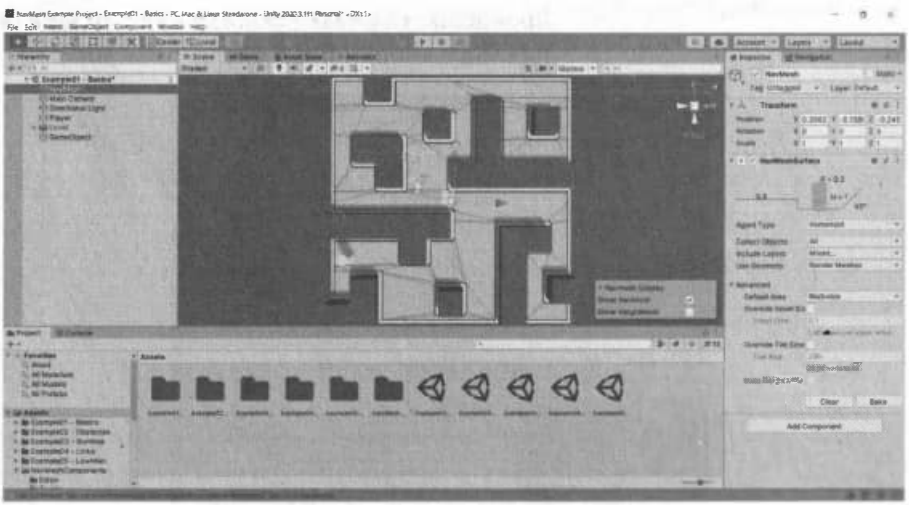


Рис. 11.8. Навигационная сетка создана

Обратите внимание, что наша карта (NavMesh) создана с учетом особенностей нашего игрока, точнее его размеров. Параметр **Agent Type** задает тип

агента. Именно с его помощью и происходит расчет недоступных для прохода областей – там, где наш объект не сможет пройти. По умолчанию используется тип *Humanoid*. Выберите из списка команду **Open Agent Settings**. Вы увидите параметры для используемого агента (рис. 11.9).

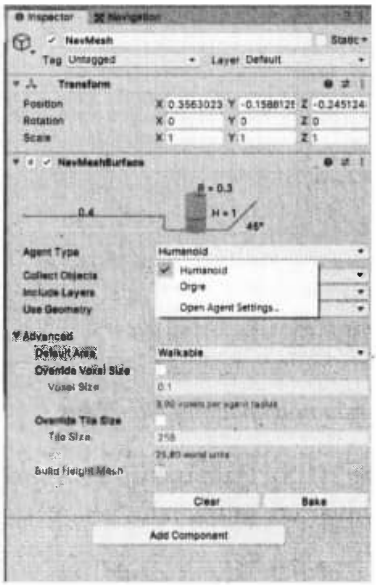


Рис. 11.9. Параметры используемого агента

Вы можете изменить название (**Name**), радиус (**Radius**), высоту (**Height**), высоту шага (**Step Height**), максимальный подъем (**Max Slope**). Но лучше не изменять данный тип, а создать собственный, нажав кнопку +. Создадим параметры для орка – установим другой радиус и другую высоту, что сделает наш агент толще и выше. Также можно изменить и другие параметры. Вернитесь в **Инспектор** для нашего объекта NavMesh, выберите другой тип объекта (который вы только что создали) и нажмите кнопку **Bake** (рис. 11.11). Карта будет перестроена с учетом введенных параметров агента. Да, с такими размерами далеко не пройдешь. Верните предыдущий тип агента и нажмите кнопку **Bake** снова.

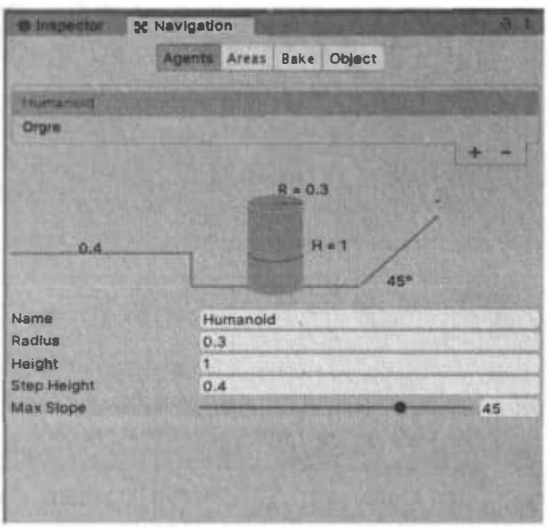


Рис. 11.10. Создание другого типа агента

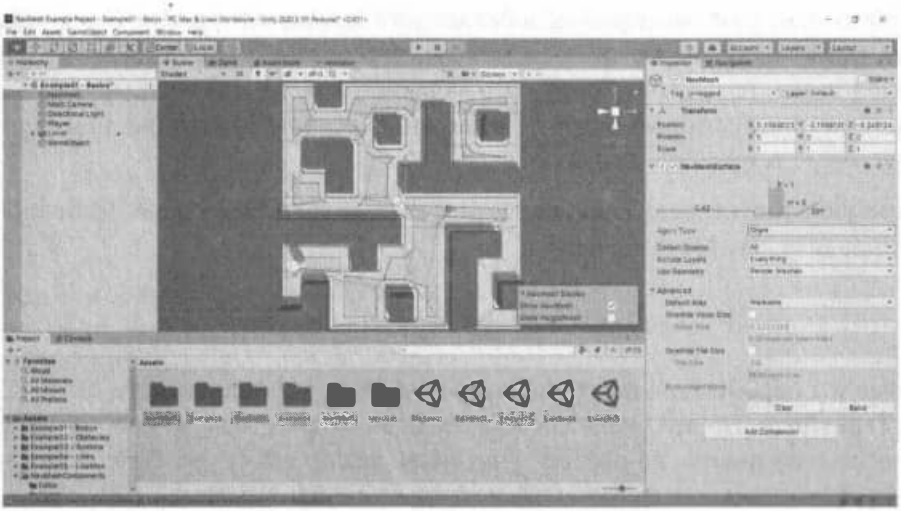


Рис. 11.11. Карта построена с учетом другого типа агента

11.5. Перемещение персонажа по сцене

Ранее мы создали два типа агента – для человека и орка. Наш персонаж пока еще не знает, кто он. Чтобы он узнал, кто он, выделите объект Player в иерархии и с помощью кнопки **Add Component** добавьте новый компонент – **Nav Mesh Agent**. Свойство **Agent Type** как раз и позволяет задать тип персонажа (рис. 11.12).

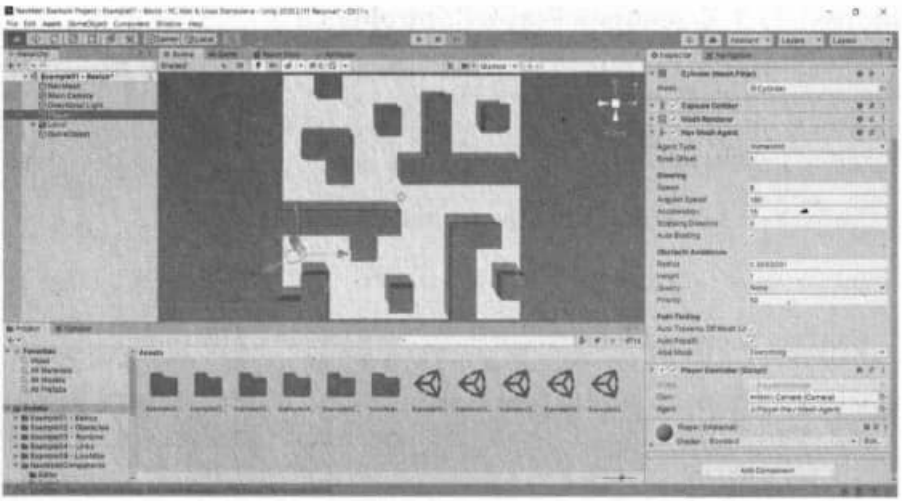


Рис. 11.12. Добавлен компонент "Nav Mesh Agent"

Однако основное назначение добавленного компонента иное. С помощью Nav Mesh Agent мы можем перемещать персонаж по созданной навигационной сетке, находить маршруты в заданное положение и т.д. Давайте напишем скрипт, который будет перемещать персонаж в позицию, по которой пользователь щелкнул мышью.

Перемещение игрока осуществляется с помощью метода setDestination() компонента Nav Mesh Agent:

```
agent.SetDestination(hit.point);
```

Ранее мы перемещали персонаж посредством установки свойств компонента **Transform**. Но это очень сложно. Ведь во-первых, нам нужно вычислять точные координаты, во-вторых, учитывать все препятствия. Теперь нам это не нужно делать, поскольку у нас есть компоненты NavMesh.

Координаты мыши мы получаем так:

```
  if (Input.GetMouseButtonDown(0))
        {
            Ray ray = cam.ScreenPointToRay(Input.
mousePosition);
            RaycastHit hit;
if (Physics.Raycast(ray, out hit))
                {
```

Полный код сценария PlayerController.cs приведен в лист. 11.1.

Листинг 11.1. Сценарий PlayerController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class PlayerController : MonoBehaviour {

  public Camera cam;

    public NavMeshAgent agent;

  // Use this for initialization
  void Start () {

  }
```

```
    // Update is called once per frame
    void Update () {
        if (Input.GetMouseButtonDown(0))
            {
                Ray ray = cam.ScreenPointToRay(Input.
mousePosition);
                RaycastHit hit;

                if (Physics.Raycast(ray, out hit))
                {
                    agent.SetDestination(hit.point);
                }
            }
    }
}
```

Наш сценарий принимает два параметра – cam и agent. Первый – это наша камера. Именно от нее мы будем получать необходимые координаты. Второй – это наш агент, который будет перемещать персонаж.

Перетащите созданный скрипт на объект Player. В качестве параметра **Cam** выберите Main Camera, в качестве параметра **Agent** – Player Nav Mesh Agent, то есть Nav Mesh Agent персонажа – просто перетащите этот компонент на поле Agent.

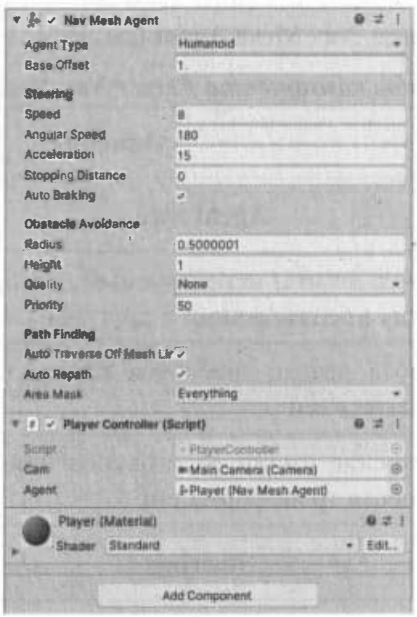


Рис. 11.13. Установка свойств персонажа

Запустите игру. Посмотрим, что получилось. Вы увидите, как объект переместился в указанное положение.

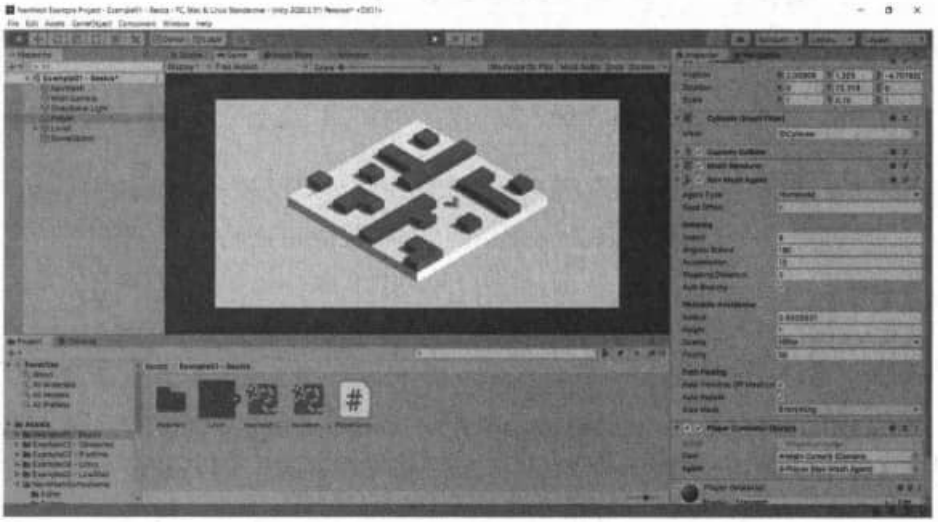


Рис. 11.14. Персонаж переместился в другую точку

На данный момент персонаж перемещается очень медленно. Чтобы сделать его быстрее, вы можете изменить параметры **Speed**, **Angular Speed** и **Acceleration** компонента **Nav Mesh Agent** (см. табл. 11.1)

Таблица 11.1. Свойства компонента Player Nav Mesh Agent

Свойство	Описание
Agent Size	
Radius	Радиус агента, используемый для расчета столкновений между препятствиями и другими агентами
Height	Высота зазора, требуемая агенту для прохождения под препятствием
Base offset	Смещение цилиндра столкновения относительно точки поворота трансформации
Steering	

Speed	Максимальная скорость движения (в мировых единицах в секунду)
Angular Speed	Максимальная скорость вращения (градусов в секунду)
Acceleration	Максимальное ускорение (в мировых единицах в секунду, в квадрате)
Stopping distance	Дистанция остановки
Auto Braking	Когда включено, агент будет замедляться при достижении пункта назначения. Вы должны отключить это для поведения такого патрулирования, когда агент должен плавно перемещаться между несколькими точками
Obstacle Avoidance	
Quality	Качество предотвращения препятствий. Если у вас большое количество агентов, вы можете сэкономить процессорное время за счет снижения качества обхода препятствий
Priority	Агенты с более низким приоритетом будут игнорироваться этим агентом при выполнении избегания. Значение должно быть в диапазоне 0–99, где более низкие числа указывают на более высокий приоритет
Path Finding	
Auto Traverse OffMesh Link	Установите в значение *true*, чтобы автоматически перемещаться по ссылкам вне сетки. Вы должны отключить это, если хотите использовать анимацию или какой-то особый способ обхода ссылок вне сетки
Auto Repath	Когда этот параметр включен, агент будет пытаться снова найти путь, когда он достигнет конца частичного пути. Когда нет пути к месту назначения, генерируется частичный путь к ближайшему достижимому месту к месту назначения

Area Mask	Маска области описывает, какие типы областей агент будет учитывать при поиске пути. Когда вы готовите сетки для выпекания NavMesh, вы можете установить тип каждой области сетки. Например, вы можете пометить лестницы специальным типом области и запретить пользоваться лестницами определенным типам персонажей, например, оркам

11.6. Компонент Nav Mesh Modifier

Компонент Nav Mesh Modifier позволяет модифицировать навигационную карту. Выделите наши стены, добавьте компонент **Nav Mesh Modifier** к ним. После этого установите свойство **Ignore from Build** и перегенерируйте карту. Вы увидите, что стены исключены из карты (рис. 11.15).

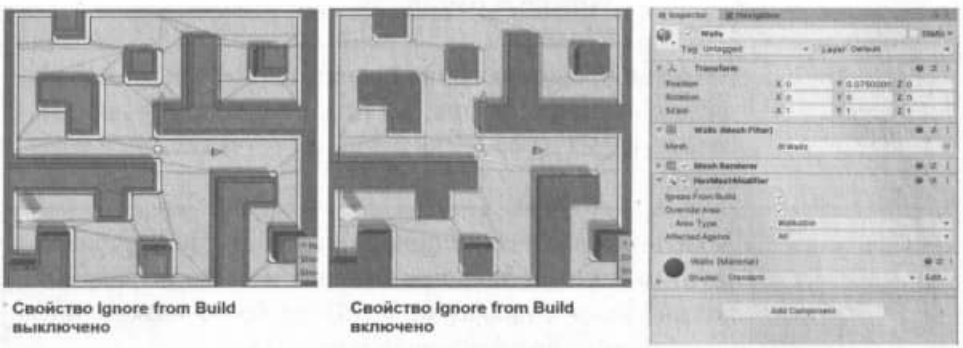


Рис. 11.15. Стены исключены из карты

Аналогичным образом можно исключать из карты другие объекты. Параметр **Override Area**, если включен позволяет перезаписывать тип области. Включите его и выберите тип области (Area Type) – **Not Walkable**. Перегенерируйте карту, посмотрим, что получилось (рис. 11.16).

Сравните с нашими более ранними NavMesh: внутри стен больше нет доступного для прогулки пространства. Параметр **Affected Agents** позволяет выбрать агентов, которых касается модификация NavMesh. По умолчанию используется значение **All**, но вы можете выбрать конкретный тип агента.

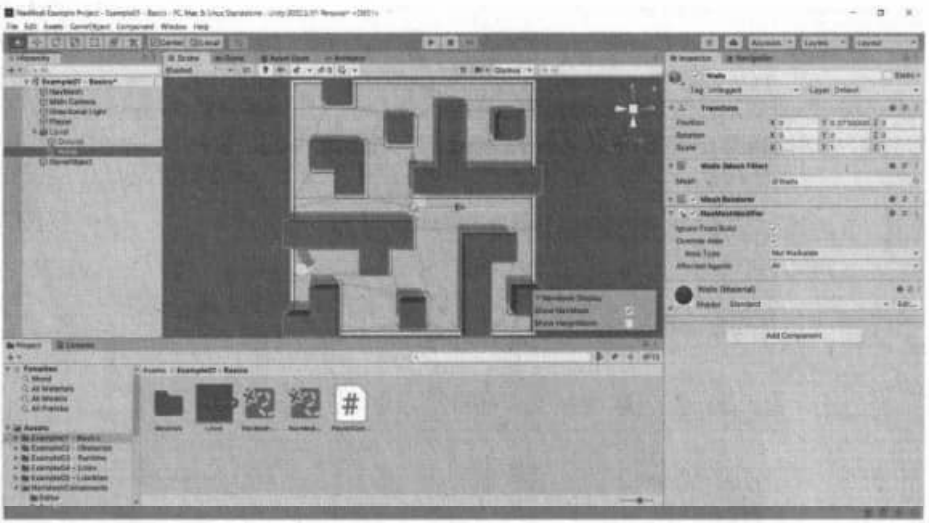


Рис. 11.16. Стены помечены как недоступные для прогулки

11.7. Обход движущихся препятствий

Ранее мы создали навигационную сетку, содержащую стены. Наш персонаж мог перемещаться по уровню, но не мог проходить сквозь стены. Со стенами было относительно просто – они не двигались. В этом разделе мы усложним задачу и попробуем научить персонаж избегать движущихся препятствий.

Компонент **NavMesh Obstacle** позволяет вам описывать движущиеся препятствия, которых агенты NavMesh должны избегать при навигации по уровню. Хорошим препятствием подобного рода являются бочка или ящик. Пока препятствие движется, агенты делают все возможное, чтобы избежать его, но как только препятствие становится неподвижным, оно будет вырезать дыру в NavMesh, чтобы агенты могли изменить свой путь, чтобы обойти его. Если стационарное препятствие заблокировало путь, агенты смогут найти другой маршрут. Именно работа этого компонента далее и будет продемонстрирована.

В проекте ch11 откройте сцену Example2, которая содержит только один зеленый цилиндр (рис. 11.17).

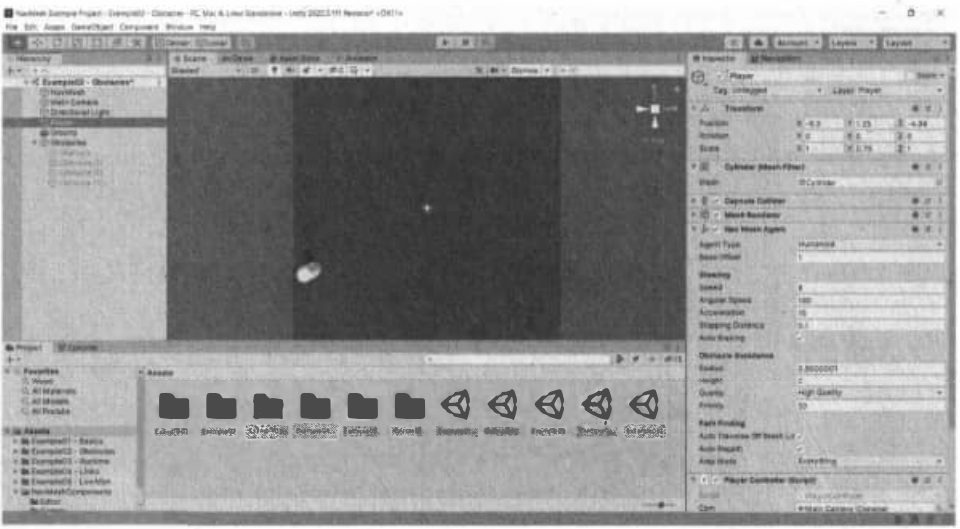


Рис. 11.17. Подготовка сцены

Добавим на сцену препятствия – ящики. Для удобства препятствия уже созданы и добавлен скрипт их анимации. Вам нужно только активировать объект Obstacles (рис. 11.18).

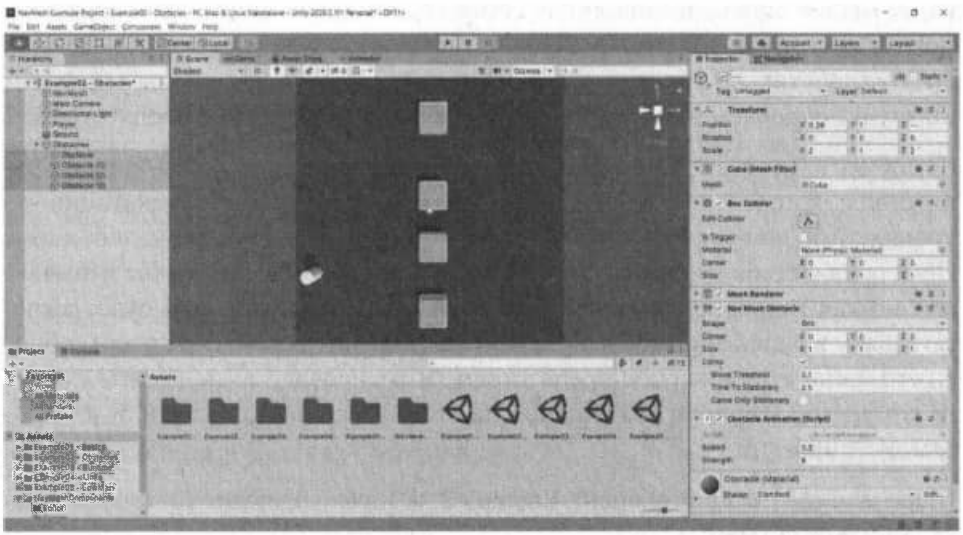


Рис. 11.18. Добавлены препятствия

Скрипт для анимации представлен в лист. 11.2.

Листинг 11.2. Сценарий перемещения препятствий

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ObstacleAnimation : MonoBehaviour {

   public float speed = .2f;
   public float strength = 9f;

   private float randomOffset;

   // Use this for initialization
   void Start () {
      randomOffset = Random.Range(0f, 2f);
   }

   // Update is called once per frame
   void Update () {
      Vector3 pos = transform.position;
      pos.x = Mathf.Sin(Time.time * speed + randomOffset) *
strength;
      transform.position = pos;
   }
}
```

Сценарий очень прост и суть его заключается в установке свойства *position* для компонента **transform**. Позиция вычисляется на основании скорости и силы. Для удобства эти два параметра мы сделали *public*, чтобы их можно было редактировать с помощью инспектора свойств.

Запустите игру и убедитесь, что сценарий работает – препятствия двигаются (рис. 11.19). Скорость движения отдельного ящика можно регулировать с помощью упомянутых ранее параметров.

В качестве препятствия мы используем объект Куб и наши объекты уже полностью готовы быть препятствиями. Если же вы хотите произвольный объект превратить в препятствие, вам нужно добавить компонент **Nav Mesh Obstacle** (рис. 11.20).

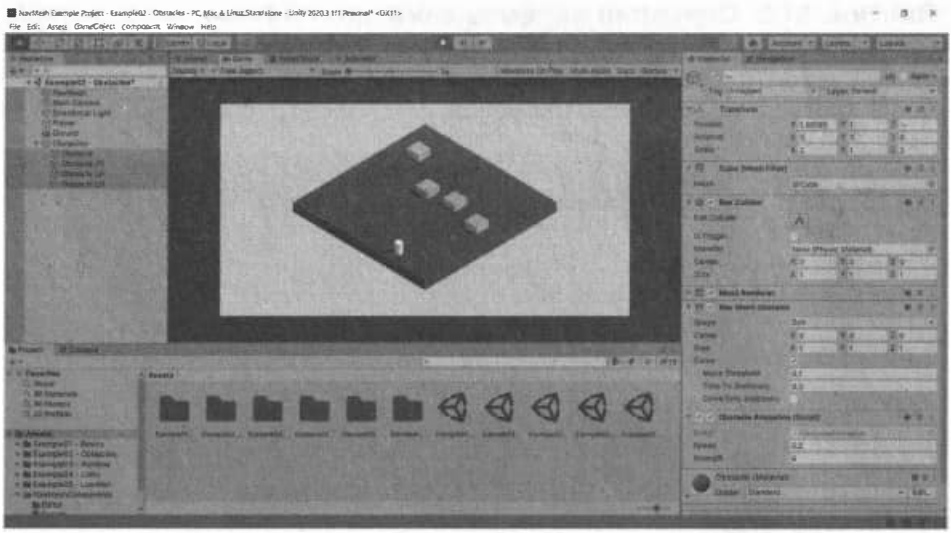
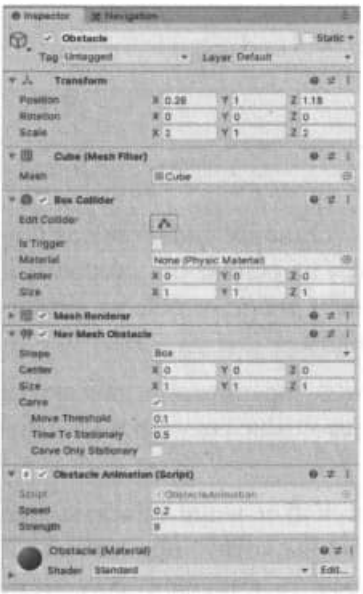


Рис. 11.19. Препятствия двигаются



Свойство **Shape** задает форму препятствия (мы используем форму **Box** – ящик). Параметр **Size** задает размер препятствия. Очень важное свойство **Carve** – нужно его включить, иначе наша NavMesh не будет реагировать на препятствие. При включенном Carve в навигационной строке будет вырезана «дыра» – подобный эффект мы наблюдали со стенами. Описание всех свойств компонента Nav Mesh Obstacle приведено в таблице 11.2.

Рис. 11.20. Параметры компонента Nav Mesh Obstacle

Таблица 11.2. Свойства компонента Nav Mesh Obstacle

Свойство	Описание
Shape	Форма геометрии препятствия: капсула (capsule) или коробка (box)

Для коробки:	
Center	Центр относительно позиции компонента transform
Size	Размер коробки
Для капсуля:	
Center	Центр относительно позиции компонента transform
Radius	Радиус цилиндрического препятствия
Height	Высота цилиндрического препятствия
Carve	Когда включено, в NavMesh создается «дыра»
Когда Carve включено:	
Move Threshold	Пороговое расстояние для обновления движущегося отверстия в NavMesh
Time To Stationary	Время ожидания, пока препятствие не будет считаться стационарным
Carve Only Stationary	Если включено, препятствие будет вырезано только тогда, когда оно неподвижно

Теперь попробуем «запечь» карту. Как обычно, выберите NavMesh и нажмите кнопку Bake. На рис. 11.21 показана созданная навигационная карта. Обратите внимание на «дыры» вокруг наших препятствий.

Для анимации персонажа будем использовать сценарий PlayerController.cs, который мы разработали ранее. Просто перетащите его на объект **Player** и запустите игру. Чтобы вы ни делали, как бы вы ни старались, вы не сможете поместить объект персонажа (цилиндр) на движущееся препятствие.

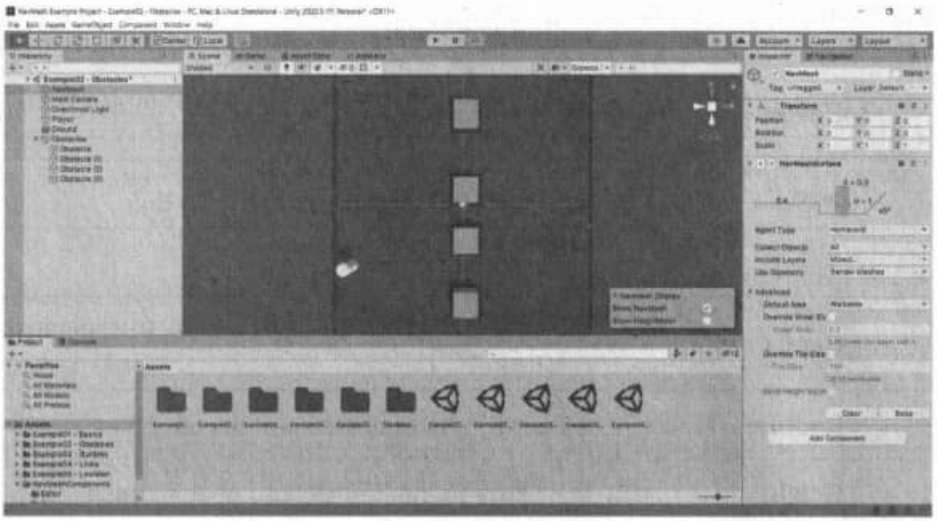


Рис. 11.21. Созданная навигационная карта

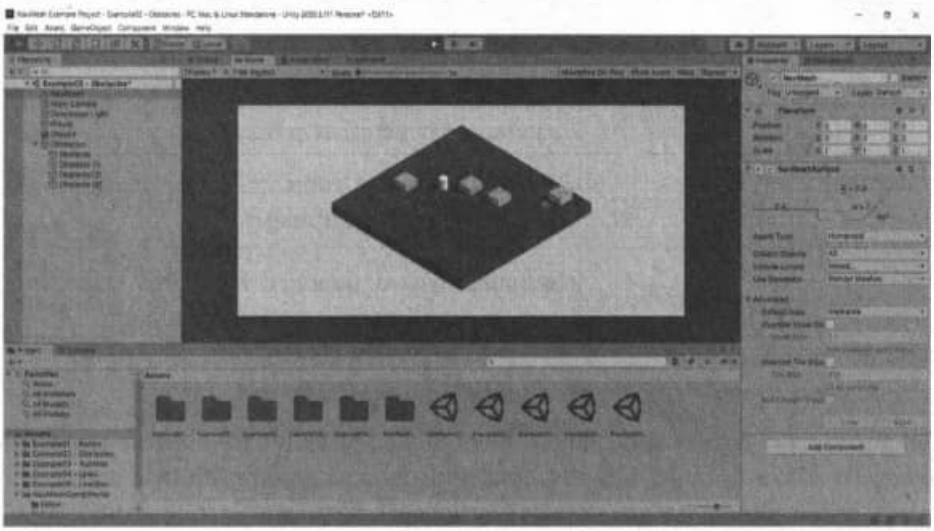


Рис. 11.22. Игра запущена

11.8. Автоматическое генерирование карты и уровня

Уровень, то есть имеющиеся на нем препятствия, можно генерировать случайным образом. Понятное дело, что тогда и навигационную сетку за-

пекать придется автоматически. Сделать это можно с помощью метода BuildNavMesh(). Но сначала мы попробуем работать без этого метода.

Откройте сцену example3 в проекте ch11. По умолчанию на ней ничего нет. Но в ассетах виден префаб для **Player** (рис. 11.23). Этому префабу присвоен сценарий PlayerController. В этот сценарий мы внесли изменение. В методе **Start** мы находим главную камеру и присваиваем свойству **cam** – на всякий случай, если вдруг мы забудем установить это свойство. Новый вариант сценария PlayerController.cs приведен в лист. 11.3.

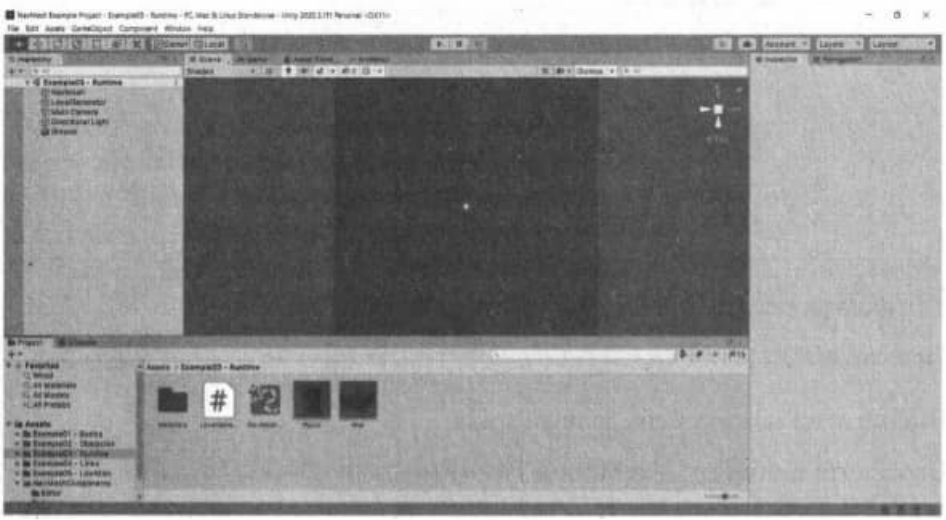


Рис. 11.23. Сцена для примера 3. Префаб Player в ассетах для примера 3

Листинг 11.3. Новый вариант PlayerController.cs

```
using UnityEngine;
using UnityEngine.AI;
using System.Collections;
using System.Collections.Generic;


public class PlayerController : MonoBehaviour {

   public Camera cam;

     public NavMeshAgent agent;

   void Start() {
      // Присваиваем главную камеру свойству Cam
```

```
        cam = Camera.main;
    }

    // Update is called once per frame
    void Update () {
        if (Input.GetMouseButtonDown(0))
            {
                Ray ray = cam.ScreenPointToRay(Input.
mousePosition);
                RaycastHit hit;
            Debug.Log("Hit");

                if (Physics.Raycast(ray, out hit))
                {
                    agent.SetDestination(hit.point);
                }
            }
    }
}
```

Как видите, обратиться к главной камере можно так:

```
Camera.main;
```

Возьмите на заметку себе данный трюк.

Перейдите к объекту NavMesh и нажмите кнопку **Bake**. Нам нужно запечь карту по умолчанию, иначе Unity будет «ругаться», что карта не найдена. Как видите, вся область является доступной для прогулок нашего персонажа (рис. 11.24).

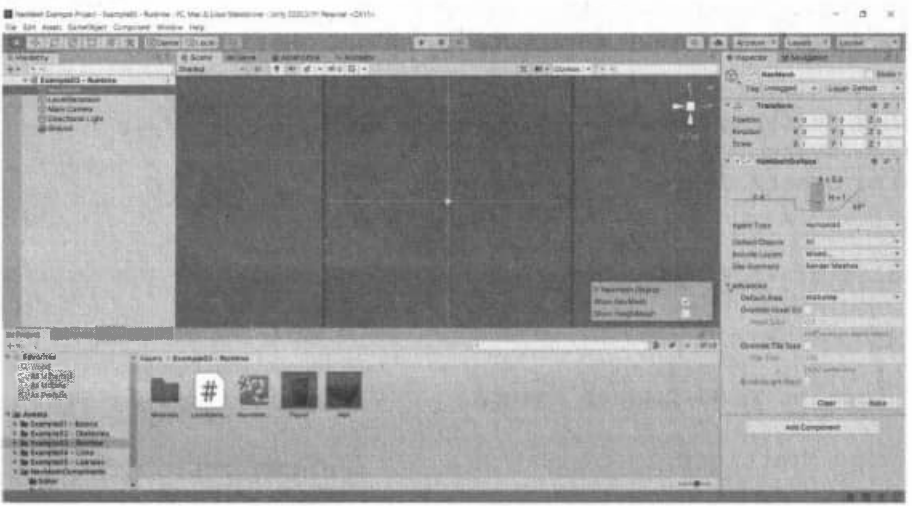


Рис. 11.24. Запеченная навигационная карта

Пустому объекту LevelGenerator присвоен скрипт LevelGenerator. Промежуточная его версия приведена в лист. 11.4.

Листинг 11.4. Генерирование уровня без запекания карты

```
using UnityEngine;
using UnityEngine.AI;

public class LevelGenerator : MonoBehaviour {

  public int width = 10;
  public int height = 10;

  public GameObject wall;
  public GameObject player;

  private bool playerSpawned = false;

  // Инициализация. Вызываем GenerateLevel()
  void Start () {
    GenerateLevel();
  }

  // Создаем уровень
  void GenerateLevel()
  {
    // Цикл по сетке
    for (int x = 0; x <= width; x+=2)
    {
       for (int y = 0; y <= height; y+=2)
       {
     // Поместить стену здесь?
     if (Random.value > .7f)
     {
       // Создаем стену
       Vector3 pos = new Vector3(x - width / 2f, 1f, y -
height / 2f);
       Instantiate(wall, pos, Quaternion.identity, transform);
     } else if (!playerSpawned) // Поместить игрока?
     {
       // Помещаем персонаж
       Vector3 pos = new Vector3(x - width / 2f, 1.25f, y -
height / 2f);
       Instantiate(player, pos, Quaternion.identity);

       playerSpawned = true;
```

```
        }
      }
    }
  }

}
```

Итак, у нас есть сценарий, генерирующий уровень. Есть префаб персонажа, к которому присоединен сценарий PlayerController.cs. Запустите игру и попробуйте перемещаться по уровню. Как показано на рис. 11.25, персонаж вместо того, чтобы найти маршрут от исходного положения к конечному, просто пошел по пути наименьшего сопротивления – сквозь стены. Если мы создаем игру «Каспер и Ко», такое поведение будет вполне подходящим, но в большинстве случаев ни люди, ни орки, ни зомби не могу проходить сквозь стены.

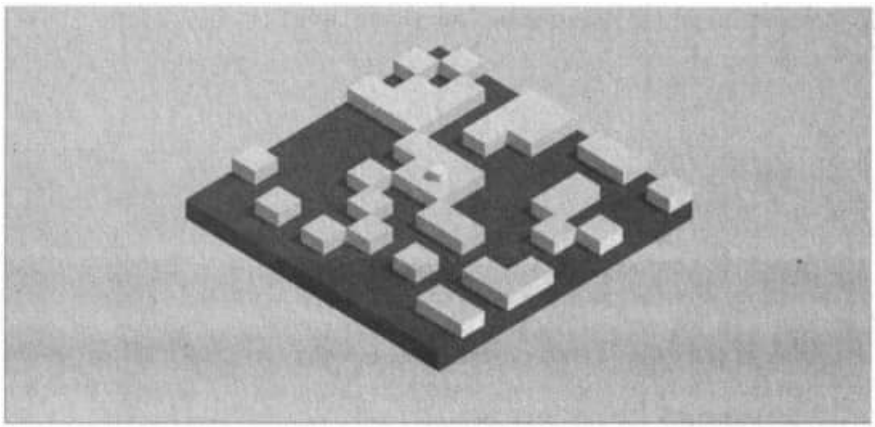


Рис. 11.25. Персонаж прошел сквозь стены

Такое получилось, поскольку карта NavMesh у нас заполнила всю поверхность. То есть для Unity персонаж может перемещаться где угодно. Нам нужно модифицировать сценарий следующим образом:

- Добавить свойство NavMeshSurface:

  ```
  public NavMeshSurface surface;
  ```

- В метод Start() после создания уровня вызвать метод запекания карты:

  ```
  surface.BuildNavMesh();
  ```

Важно добавить метод запекания карты после создания уровня. Посмотрим, что получилось. А получилось вот что – игрок больше уже не может проходить сквозь стены (рис. 11.26).

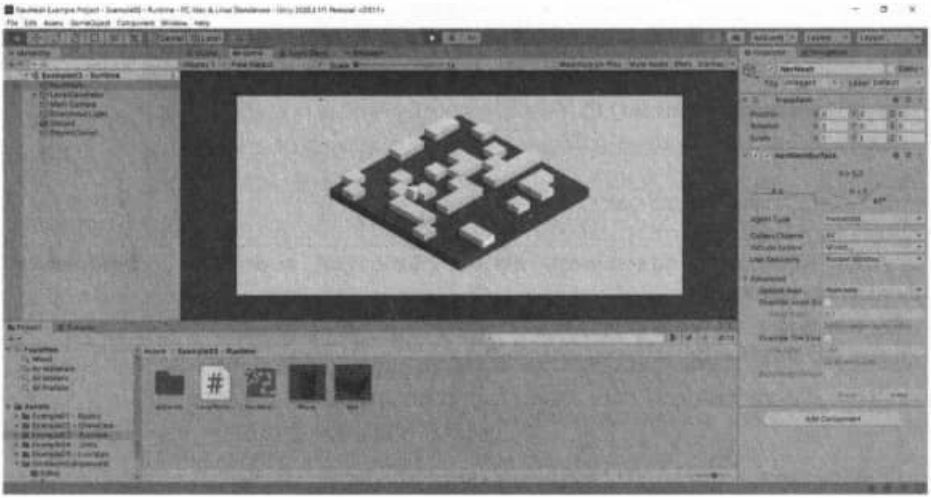


Рис. 11.26. Нормальная работа игры

Если переключиться в режим **Scene** при работающей игре, мы сможем просмотреть сгенерированную карту (рис. 11.27). Как видите, теперь наш NavMesh содержит все препятствия.

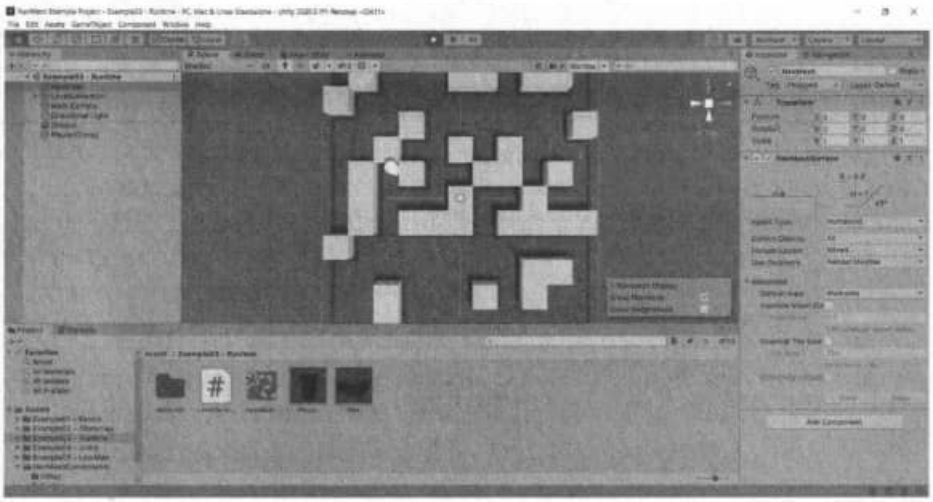


Рис. 11.27. Созданный автоматическим образом NavMesh

Полная версия сценария генерирования уровня приведена в лист. 11.5.

Листинг 11.5. Сценарий создания уровня

```
using UnityEngine;
using UnityEngine.AI;

public class LevelGenerator : MonoBehaviour {

  public int width = 10;
  public int height = 10;

  public GameObject wall;
  public GameObject player;
  // Не забудьте назначить этому свойству значение, перетащив
объект NavMesh
  // на него в инспекторе
  public NavMeshSurface surface;

  private bool playerSpawned = false;

  // Инициализация. Вызываем GenerateLevel()
  void Start () {
    // Генерируем уровень
    GenerateLevel();
    // Строим карту
    surface.BuildNavMesh();
  }

  // Создаем уровень
  void GenerateLevel()
  {
    // Цикл по сетке
    for (int x = 0; x <= width; x+=2)
    {
       for (int y = 0; y <= height; y+=2)
       {
     // Поместить стену здесь?
     if (Random.value > .7f)
     {
       // Создаем стену
       Vector3 pos = new Vector3(x - width / 2f, 1f, y -
height / 2f);
       Instantiate(wall, pos, Quaternion.identity, transform);
     } else if (!playerSpawned) // Поместить игрока?
     {
       // Помещаем персонаж
```

```
        Vector3 pos = new Vector3(x - width / 2f, 1.25f, y -
height / 2f);
        Instantiate(player, pos, Quaternion.identity);

        playerSpawned = true;
      }
        }
    }
  }

}
```

11.9. Анимированный персонаж

Перемещающийся цилиндр – это, безусловно, хорошо. Наш цилиндр вполне себе красивый – зеленого цвета. Он правильно перемещается по карте, не проходит сквозь стены. Но это перемещающийся по сцене цилиндр – это явно не то, что вы хотите от Unity в нынешнем году. Реальная игра будет содержать анимированный человекоподобный персонаж, а не цилиндр.

Откройте сцену example5. Вы увидите все тот же зеленый цилиндр, стоящий в лабиринте, созданном из красных стен (рис. 11.28).

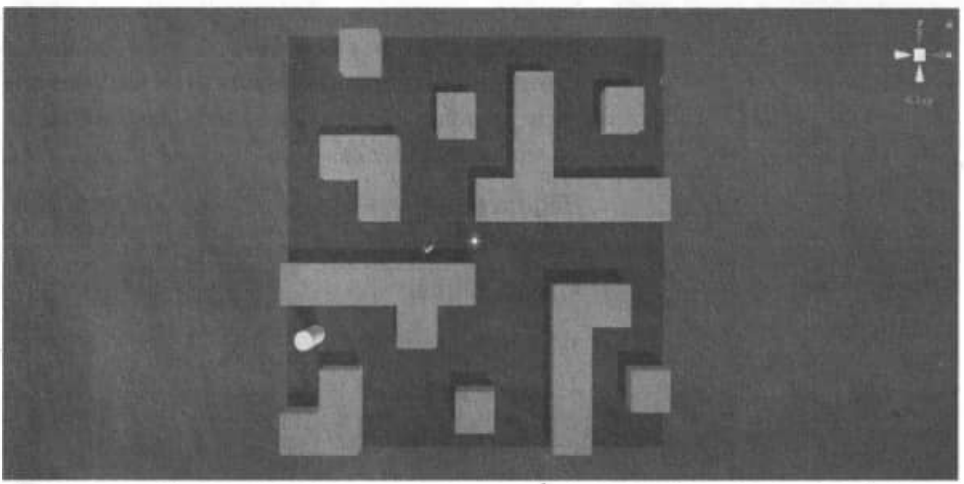


Рис. 11.28. Цилиндр в лабиринте

Попробуем заменить этот цилиндр на модель человечка. Первым делом сгенерируем нашу карту – выберите NavMesh и нажмите кнопку **Bake**. После чего вы увидите такую картину (рис. 11.29).

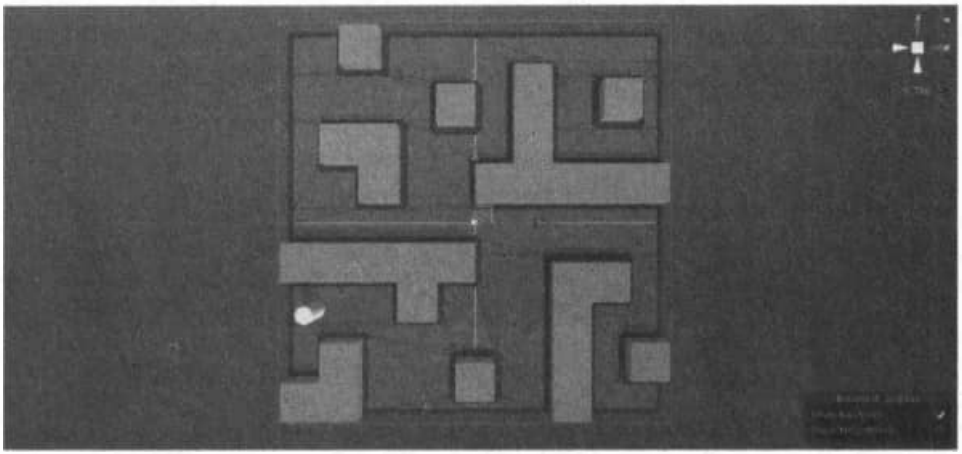


Рис. 11.29. Навигационная карта уровня

Выделите цилиндр – это наш персонаж. Присоедините к нему скрипт PlayerController.cs, назначьте поля Cam и Agent, как показано на рис. 11.30.

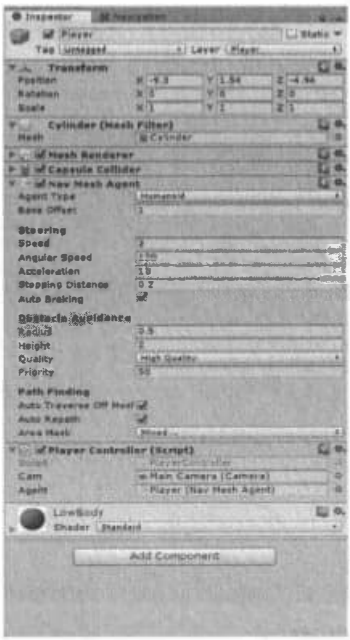


Рис. 11.30. Параметры персонажа

Запустите игру и убедитесь, что персонаж перемещается по сцене (рис. 11.31). Значит, с картой и основными объектами у нас все хорошо и можно приступить к замене цилиндра на модель человека.

Вернитесь к свойствам игрока. Удалите следующие компоненты:

- Mesh Filter
- Capsule Collider
- Mesh Renderer

Примечание. Для удаления компонента щелкните правой кнопкой мыши по названию и выберите команду **Remove Component**.

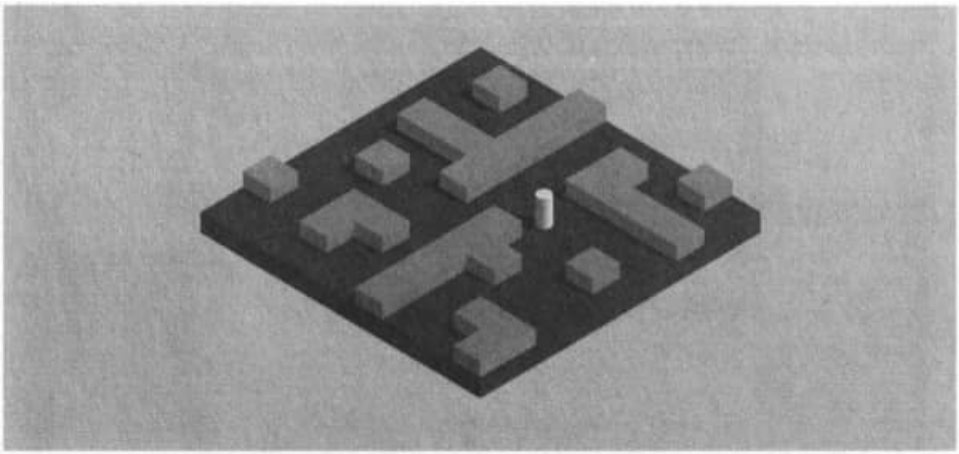


Рис. 11.31. Персонаж перемещается по сцене

У вас должны остаться только компоненты Transform, NavMeshAgent и PlayerController (скрипт), как показано на рис. 11.32. Теперь наш персонаж – невидимый цилиндр.

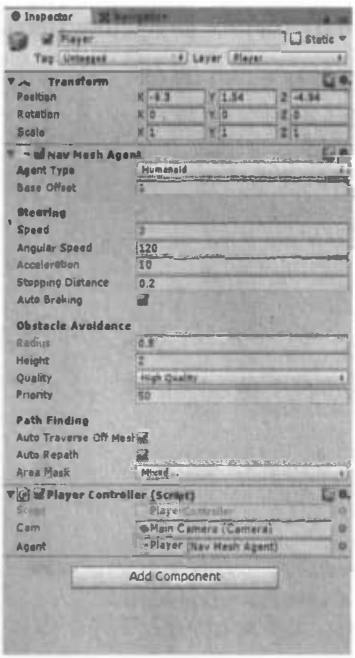


Рис. 11.32. Удалили все лишнее

Найдите модель человечка в ассетах. На рис. 11.33 изображено окно **Project**, чтобы вам было понятнее, где искать, а также сама модель, которую мы будем использовать.

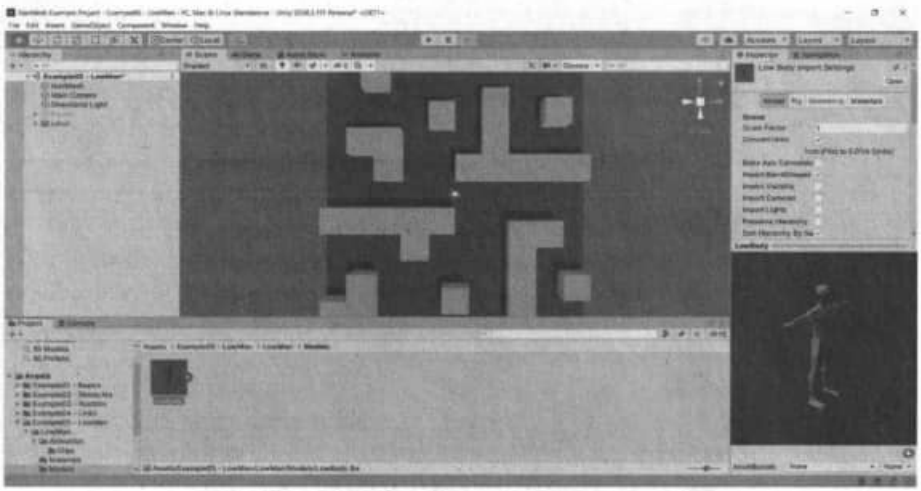


Рис. 11.33. Модель человечка

Перетащите модель человека на объект **Player** в окно иерархии. Так чтобы модель LowBody стала дочерним объектом для Player. Теперь вместо цилиндра у нас появился человек, висящий в воздухе. Если запустить игру прямо сейчас, то вы увидите довольно занимательный вид парящего над сценой человека, причем модель человека разворачивается в сторону движения (рис. 11.35).

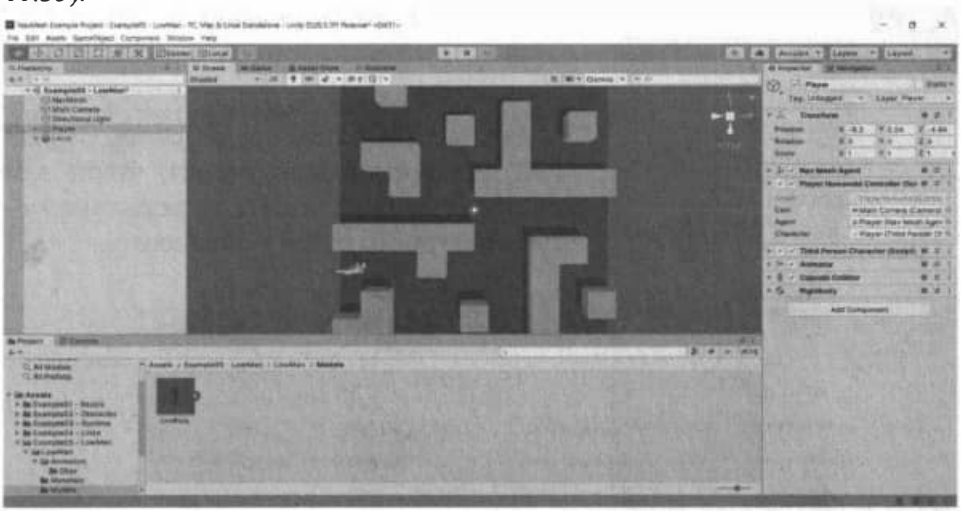


Рис. 11.34. Модель цилиндра заменена на модель человека

Явно не то, что нам нужно. Удалите компонент **Animator** из объекта Low-Body (выделите объект LowBody в окне иерархии и удалите компонент

подобно тому, как вы это уже делали). Должен остаться только компонент **Transform**.

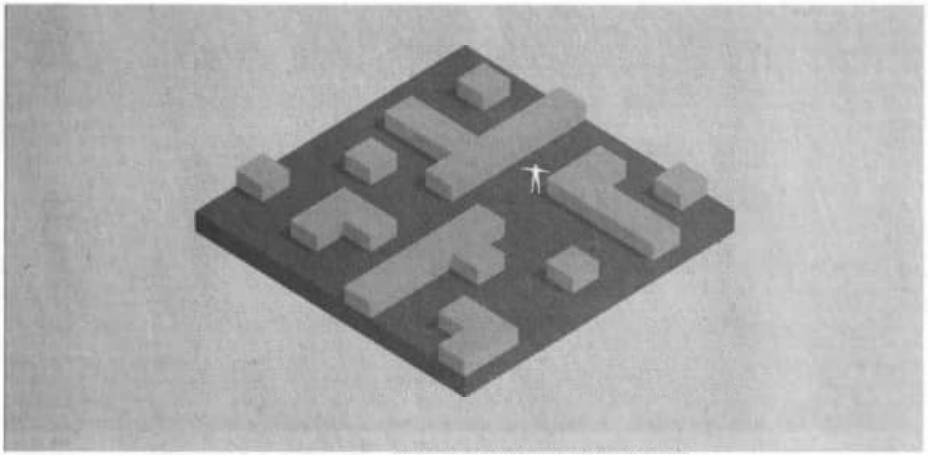


Рис. 11.35. Явно не то, чего мы хотели

Следующая наша задача – разместить человека на сцене:

1. Щелкните правой кнопкой мыши на компоненте **Transform** объекта LowBody и выполните команду **Reset**.
2. Выделите объект Player, перейдите к группе параметров **Nav Mesh Agent**, установите свойство **Base Offset** равным 0. На рис. 11.36 слева **Base Offset** = 1, справа **Base Offset** = 0. Теперь коллайдер NavMesh-агента соответствует нашей модели.
3. Используя инструмент **MoveTool**, переместите объект **Player** на сцену. Сверьте параметры компонента **Transform** (рис. 11.37)
4. Выберите объект LowBody и для свойства Scale установите значения 1.2 по всем осям. Так вы сделаете модель чуть больше.

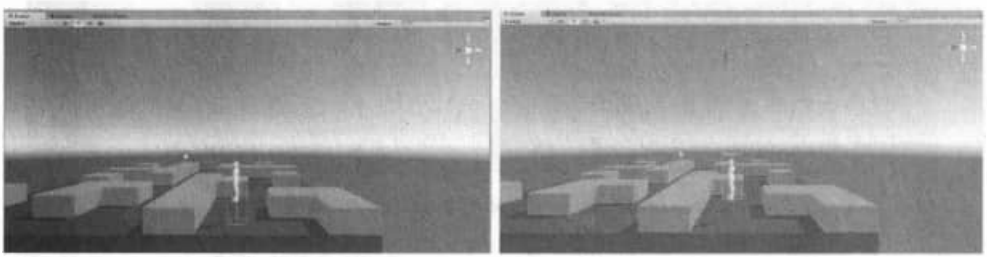


Рис. 11.36. Установка свойства "Base Offset"

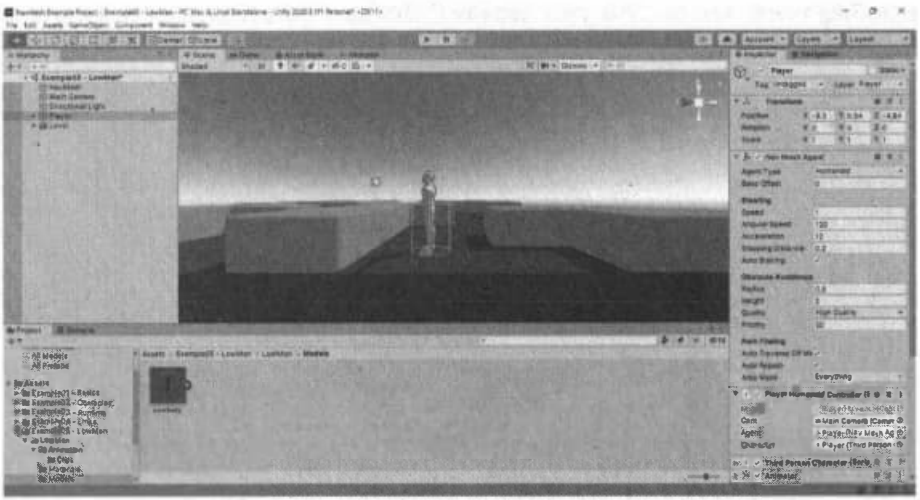


Рис. 11.37. Модель человека помещена на сцену и теперь не парит в воздухе

Перейдите в раздел **Materials**, выберите материал LowBodyMaterial и просто перетащите его на наш объект LowBody. В итоге у вас должен получиться не просто человечек, а зеленый человечек (рис. 11.38).

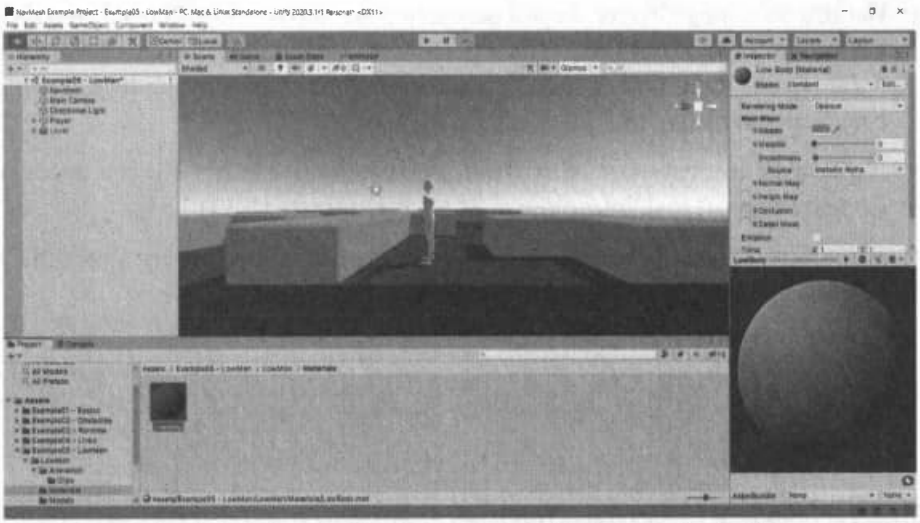


Рис. 11.38. К модели применен материал

Запустите игру еще раз, чтобы убедиться, что точно наша модель не парит над сценой. Если вы следовали рекомендациям из этой книги, все у вас будет хорошо (рис. 11.39).

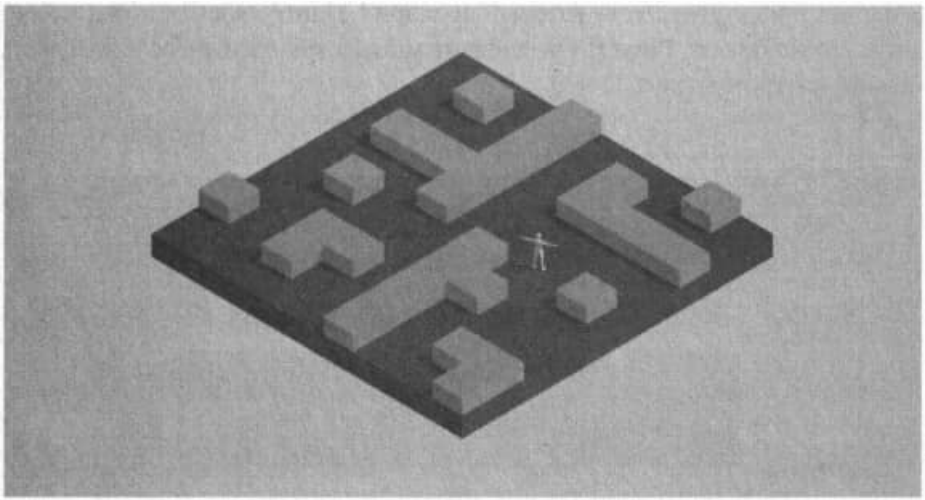


Рис. 11.39. Модель человечка не парит над сценой

Модель не парит, но и не ходит. Нам же хотелось, чтобы он ходил или бегал по сцене. Вы должны понимать, что анимация модели человечка немного сложнее анимации цилиндра. У цилиндра простая форма и его достаточно просто «анимировать». По сути его нужно только перемещать по сцене. С моделью человека все сложнее – его конечности должны двигаться, учитывая направление движения, чтобы все выглядело реалистично (рис. 11.40).

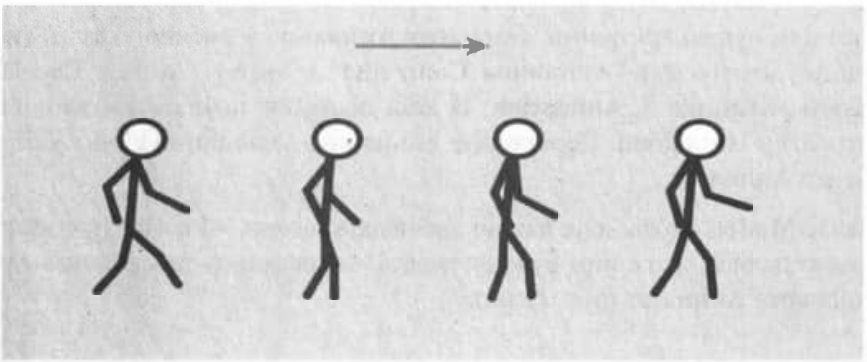


Рис. 11.40. Анимация человечка

Во время движения модели нам, кроме направления движения нужно учитывать еще и скорость. С какой скоростью движения персонаж? Будет ли он идти или бежать? От этого зависит, какая анимация должна воспроизводиться – анимация бега или прогулки.

В нашем проекте уже есть готовый сценарий ThirdPersonController. Перетащите его на объект **Player**. Он добавит множество параметров, которые вы можете настроить (рис. 11.41).

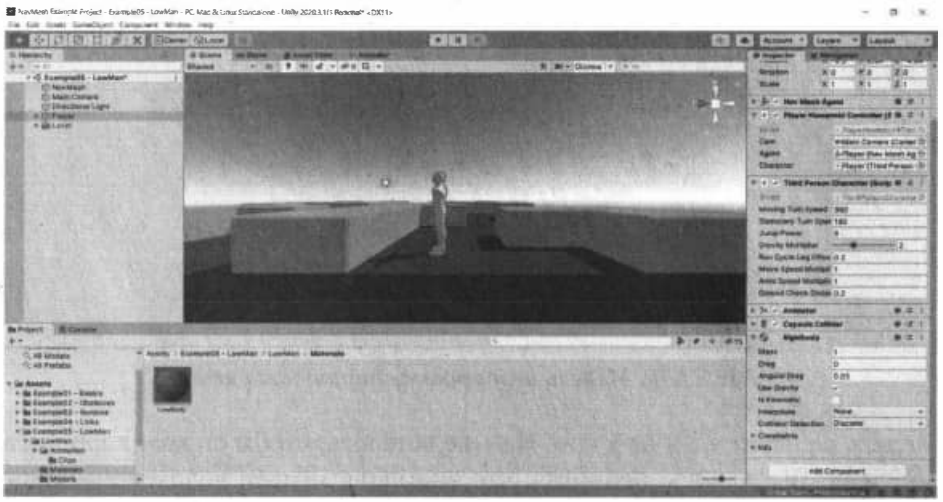


Рис. 11.41. Новые параметры персонажа

Сейчас нам нужно изменить параметры компонента **Capsule Collider**. Установите Center.Y = 1 и Height = 2, чтобы коллайдер соответствовал нашей модели – см. рис. 11.41. В компоненте **NavMesh Agent** установите **Speed** = 1.

Далее нам нужно настроить компонент **Animator**, а именно указать анимационный контроллер (**Animation Controller**) и аватар (Avatar). Перейдите в **Assets, Example 5, Animation**. В этой папке вы найдете анимационный контроллер **Humanoid**. Перетащите его на поле **Animation Controller** компонента **Animator**.

В папке **Models** вы найдете аватар для нашей модели – **LowBodyAvatar**. Вы уже догадались, что с ним нужно сделать – перетащить на свойство **Avatar** компонента **Animator** (рис. 11.42).

> **Примечание.** Если вы сейчас запустите игру, то наш персонаж хоть и не будет реалистично двигаться, но руки будут опущены вниз, а не разведены в стороны.

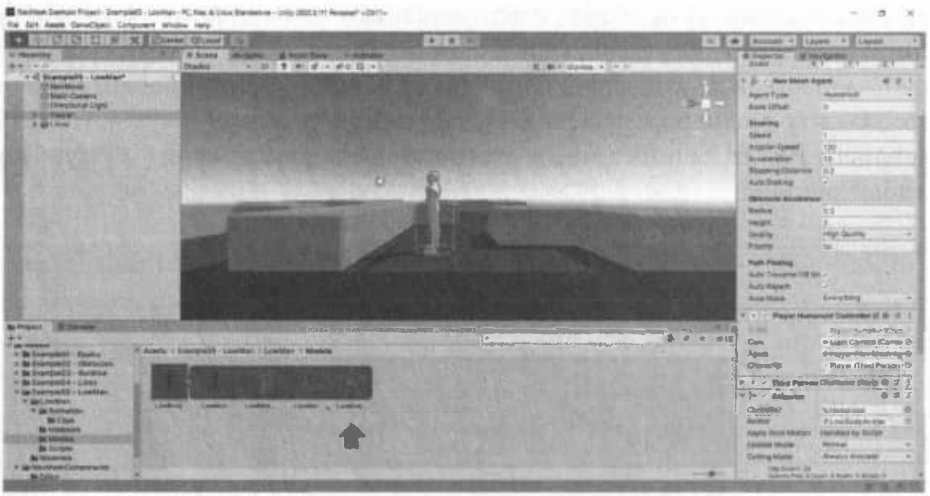


Рис. 11.42. Настройка компонента "Animator"

Откройте окно **Animator** (команда **Window, Panels, Animator**). В нем вы найдете состояния анимационного контроллера нашего персонажа. В группе **Locomotions** собраны различные состояния движения (рис. 11.44)

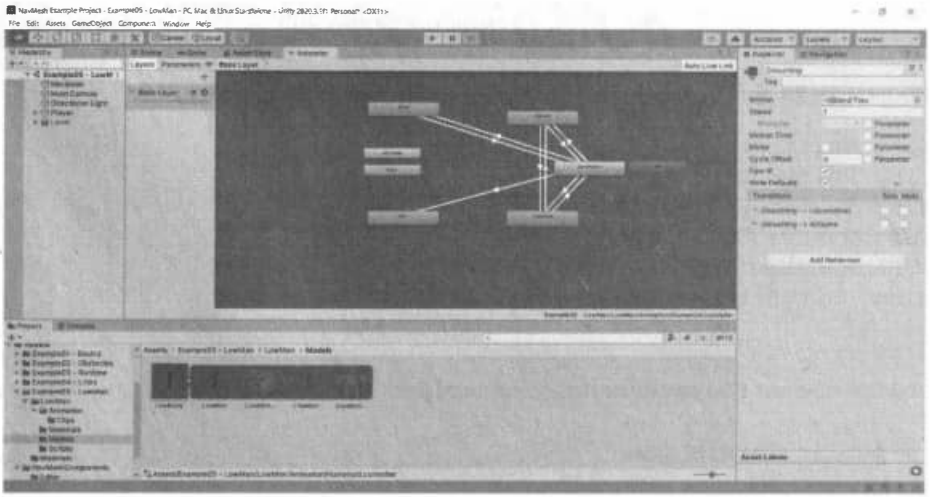


Рис. 11.43. Состояния персонажа

Задача нашего скрипта ThirdPersonCharacter.cs – выбрать одно из этих состояний в зависимости от ситуации. Нам нужно внести изменения в PlayerController.cs. Однако поскольку после внесенных изменений его уже нельзя

будет использовать в примерах с перемещением цилиндра, то мы создадим другой сценарий – PlayerHumanoidController (лист. 11.6). Данный сценарий будет задействовать сценарий ThirdPersonCharacter и анимировать нашего персонажа во время перемещения. Нужно выделить объект Player и удалить компонент PlayerController, после этого добавить новый скрипт – PlayerHumanoidController.

Рис. 11.44. Состояния движения

Листинг 11.6. Сценарий PlayerHumanoidController.cs

```
using UnityEngine;
using UnityEngine.AI;
using System.Collections;
using System.Collections.Generic;
// Подключаем ThirdPerson
using UnityStandardAssets.Characters.ThirdPerson;


public class PlayerHumanoidController : MonoBehaviour {

    public Camera cam;

    public NavMeshAgent agent;
    // Новое свойство
    public ThirdPersonCharacter character;

    void Start()
    {
```

```
        // Запрещаем navMesh-агенту обновлять вращение
        // этим будет заниматься ThirdPersonController
        agent.updateRotation = false;
    }

    // Update is called once per frame
    void Update ()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Ray ray = cam.ScreenPointToRay(Input.
mousePosition);
            RaycastHit hit;

            if (Physics.Raycast(ray, out hit))
            {
                agent.SetDestination(hit.point);
            }
        }

       // вызываем метод Move() из ThirdCharacterController
        // Первый параметр - куда мы хотим пойти (вектор)
    // Второй - должен ли персонаж ползти
        // Третий - должен ли персонаж прыгать
        if (agent.remainingDistance > agent.stoppingDistance)
        {
     // Перемещаем персонаж
            character.Move(agent.desiredVelocity, false,
false);
        }
        else
        {
     // Останавливаем персонаж
            character.Move(Vector3.zero, false, false);
        }

    }
}
```

Заполните поля нового компонента – перетащите камеру, NavMesh-агент и ThirdPersonController – на поле **Character**. Должно получиться, как показано на рис. 11.45.

Запустите игру. Наш персонаж двигается и корректно останавливается! Мы это сделали.

Рис. 11.45. Свойства персонажа

На всякий случай в книге приводится код сценария ThirdPersonController.cs. Моменты, на которые стоит обратить внимание, выделены жирным.

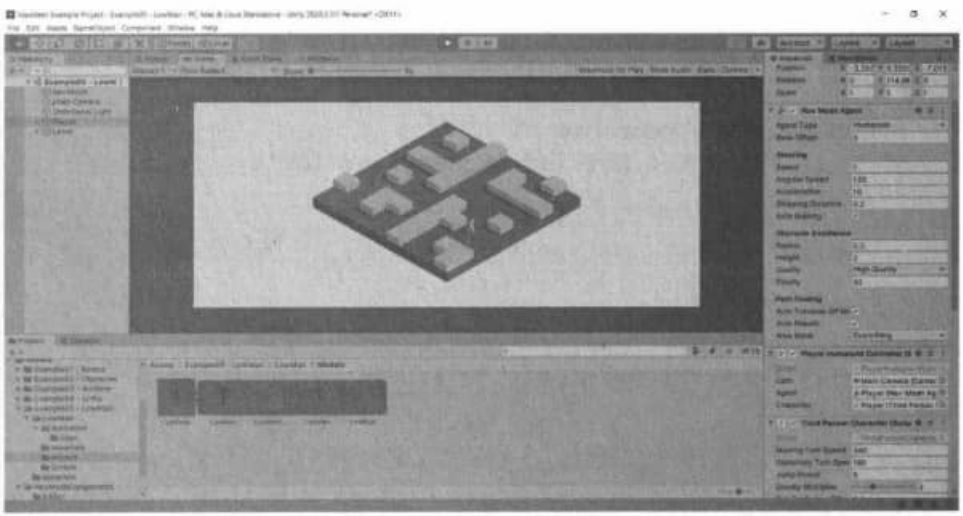


Рис. 11.46. Персонаж двигается

Листинг 11.7. Сценарий ThirdPersonController.cs

```
using UnityEngine;

namespace UnityStandardAssets.Characters.ThirdPerson
```

```
{
   [RequireComponent(typeof(Rigidbody))]
   [RequireComponent(typeof(CapsuleCollider))]
   [RequireComponent(typeof(Animator))]
   public class ThirdPersonCharacter : MonoBehaviour
   {
      [SerializeField] float m_MovingTurnSpeed = 360;
      [SerializeField] float m_StationaryTurnSpeed = 180;
      [SerializeField] float m_JumpPower = 6f;
      [Range(1f, 4f)][SerializeField] float m_GravityMultiplier
= 2f;
      // Значение зависит от выбранной модели, нужно подбирать под
      // конкретную модель
      [SerializeField] float m_RunCycleLegOffset = 0.2f;
      [SerializeField] float m_MoveSpeedMultiplier = 1f;
      [SerializeField] float m_AnimSpeedMultiplier = 1f;
      [SerializeField] float m_GroundCheckDistance = 0.2f;

      Rigidbody m_Rigidbody;
      Animator m_Animator;
      bool m_IsGrounded;
      float m_OrigGroundCheckDistance;
      const float k_Half = 0.5f;
      float m_TurnAmount;
      float m_ForwardAmount;
      Vector3 m_GroundNormal;
      float m_CapsuleHeight;
      Vector3 m_CapsuleCenter;
      CapsuleCollider m_Capsule;
      bool m_Crouching;


      void Start()
      {
         // Получаем необходимые объекты
         m_Animator = GetComponent<Animator>();
         m_Rigidbody = GetComponent<Rigidbody>();
         m_Capsule = GetComponent<CapsuleCollider>();
         m_CapsuleHeight = m_Capsule.height;
         m_CapsuleCenter = m_Capsule.center;

         m_Rigidbody.constraints = RigidbodyConstraints.
FreezeRotationX | RigidbodyConstraints.FreezeRotationY |
RigidbodyConstraints.FreezeRotationZ;
         m_OrigGroundCheckDistance = m_GroundCheckDistance;
      }
```

```
public void Move(Vector3 move, bool crouch, bool jump)
{
// move - куда нужно пойти
// crounch - нужно ли ползти
// jump - нужно ли прыгать

   // конвертируем мировой вектор moveInput в локальную
   // версию
   if (move.magnitude > 1f) move.Normalize();
   move = transform.InverseTransformDirection(move);
   CheckGroundStatus();
   move = Vector3.ProjectOnPlane(move, m_GroundNormal);
   m_TurnAmount = Mathf.Atan2(move.x, move.z);
   m_ForwardAmount = move.z;

   ApplyExtraTurnRotation();

// управление скоростью отличается, если мы на земле или
// в воздухе (при прыжке)
   if (m_IsGrounded)
   {
      HandleGroundedMovement(crouch, jump);
   }
   else
   {
      HandleAirborneMovement();
   }

   ScaleCapsuleForCrouching(crouch);
   PreventStandingInLowHeadroom();

   // отправляем параметры в аниматор
   UpdateAnimator(move);
}


void ScaleCapsuleForCrouching(bool crouch)
{
   if (m_IsGrounded && crouch)
   {
      if (m_Crouching) return;
      m_Capsule.height = m_Capsule.height / 2f;
      m_Capsule.center = m_Capsule.center / 2f;
      m_Crouching = true;
```

```
        }
        else
        {
            Ray crouchRay = new Ray(m_Rigidbody.position +
Vector3.up * m_Capsule.radius * k_Half, Vector3.up);
            float crouchRayLength = m_CapsuleHeight - m_Capsule.
radius * k_Half;
            if (Physics.SphereCast(crouchRay, m_Capsule.
radius * k_Half, crouchRayLength, Physics.AllLayers,
QueryTriggerInteraction.Ignore))
            {
                m_Crouching = true;
                return;
            }
            m_Capsule.height = m_CapsuleHeight;
            m_Capsule.center = m_CapsuleCenter;
            m_Crouching = false;
        }
    }

    void PreventStandingInLowHeadroom()
    {
        // не вставать в зоны, где можно ползти
        if (!m_Crouching)
        {
            Ray crouchRay = new Ray(m_Rigidbody.position +
Vector3.up * m_Capsule.radius * k_Half, Vector3.up);
            float crouchRayLength = m_CapsuleHeight - m_Capsule.
radius * k_Half;
            if (Physics.SphereCast(crouchRay, m_Capsule.
radius * k_Half, crouchRayLength, Physics.AllLayers,
QueryTriggerInteraction.Ignore))
            {
                m_Crouching = true;
            }
        }
    }


    void UpdateAnimator(Vector3 move)
    {
        // update the animator parameters
        m_Animator.SetFloat("Forward", m_ForwardAmount, 0.1f,
Time.deltaTime);
        m_Animator.SetFloat("Turn", m_TurnAmount, 0.1f, Time.
deltaTime);
```

```
        m_Animator.SetBool("Crouch", m_Crouching);
        m_Animator.SetBool("OnGround", m_IsGrounded);
        if (!m_IsGrounded)
        {
            m_Animator.SetFloat("Jump", m_Rigidbody.velocity.y);
        }

        // рассчитать, какая нога позади, чтобы оставить
// эту ногу в задней части анимации прыжка
     clip times of 0.0 and 0.5)
        float runCycle =
            Mathf.Repeat(
                m_Animator.GetCurrentAnimatorStateInfo(0).
normalizedTime + m_RunCycleLegOffset, 1);
        float jumpLeg = (runCycle < k_Half ? 1 : -1) * m_
ForwardAmount;
        if (m_IsGrounded)
        {
            m_Animator.SetFloat("JumpLeg", jumpLeg);
        }

// Множитель скорости анимации позволяет настраивать общую
// скорость ходьбы / бега в инспекторе, что влияет
// на скорость движения из-за корневого движения

        if (m_IsGrounded && move.magnitude > 0)
        {
            m_Animator.speed = m_AnimSpeedMultiplier;
        }
        else
        {
            // не используем это в воздухе
            m_Animator.speed = 1;
        }
    }


    void HandleAirborneMovement()
    {
        // применяем гравитацию
        Vector3 extraGravityForce = (Physics.gravity * m_
GravityMultiplier) - Physics.gravity;
        m_Rigidbody.AddForce(extraGravityForce);

        m_GroundCheckDistance = m_Rigidbody.velocity.y < 0 ?
m_OrigGroundCheckDistance : 0.01f;
```

```
    }


    void HandleGroundedMovement(bool crouch, bool jump)
    {
        // проверяем условия, можем ли мы прыгнуть
        if (jump && !crouch && m_Animator.
GetCurrentAnimatorStateInfo(0).IsName("Grounded"))
        {
            // jump!
            m_Rigidbody.velocity = new Vector3(m_Rigidbody.
velocity.x, m_JumpPower, m_Rigidbody.velocity.z);
            m_IsGrounded = false;
            m_Animator.applyRootMotion = false;
            m_GroundCheckDistance = 0.1f;
        }
    }

    void ApplyExtraTurnRotation()
    {
        // помогаем персонажу быстрее поворачиваться
        float turnSpeed = Mathf.Lerp(m_StationaryTurnSpeed, m_
MovingTurnSpeed, m_ForwardAmount);
        transform.Rotate(0, m_TurnAmount * turnSpeed * Time.
deltaTime, 0);
    }


    public void OnAnimatorMove()
    {
        // эта функция перезаписывает корневое движение
        // это позволяет модифицировать позиционную
// скорость до ее применения
        if (m_IsGrounded && Time.deltaTime > 0)
        {
            Vector3 v = (m_Animator.deltaPosition * m_
MoveSpeedMultiplier) / Time.deltaTime;

            // мы сохраняем существующую часть текущей скорости.
        v.y = m_Rigidbody.velocity.y;
            m_Rigidbody.velocity = v;
        }
    }


    void CheckGroundStatus()
```

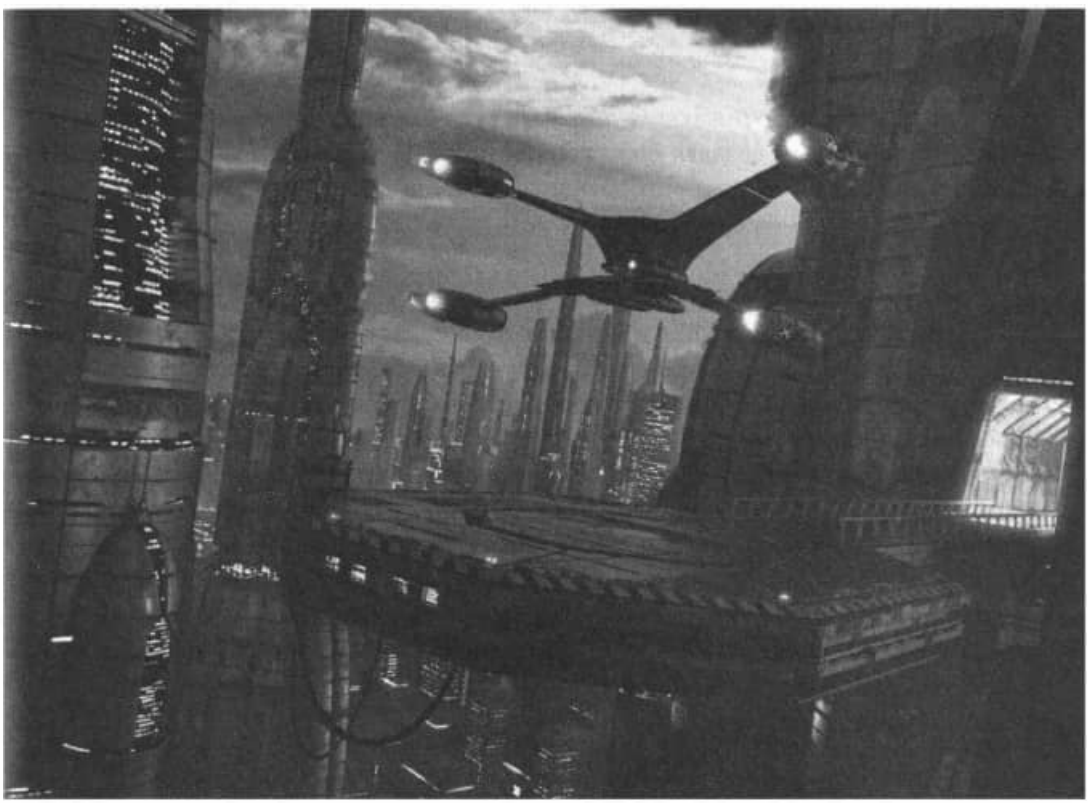
```
    {
        RaycastHit hitInfo;
#if UNITY_EDITOR
                Debug.DrawLine(transform.position + (Vector3.up
* 0.1f), transform.position + (Vector3.up * 0.1f) + (Vector3.
down * m_GroundCheckDistance));
#endif
        // 0.1f небольшое смещение для запуска луча
// изнутри персонажа
// позиция преобразования в типовых активах
// лежит в основе персонажа.
        if (Physics.Raycast(transform.position + (Vector3.up *
0.1f), Vector3.down, out hitInfo, m_GroundCheckDistance))
        {
            m_GroundNormal = hitInfo.normal;
            m_IsGrounded = true;
            m_Animator.applyRootMotion = true;
        }
        else
        {
            m_IsGrounded = false;
            m_GroundNormal = Vector3.up;
            m_Animator.applyRootMotion = false;
        }
    }
  }
}
```

Теперь можно переходить к чтению следующей части книги, в которой мы разработаем игру – нашу интерпретацию **Бомбермена**. Процесс будет рассмотрен от начала и до конца – вплоть до компиляции игры. Не переключайтесь, будет интересно!

ЧАСТЬ III. РАЗРАБОТКА ИГРЫ

Глава 12.

Подготовительные мероприятия



Наконец-то настало время аккумулировать все полученные знания и использовать их во благо – создать собственную игру. В этой части книги будет рассмотрено создание игры – от начали и до конца.

12.1. Поиск идеи

Во время работы над книгой я постоянно думал, какую игру создать, чтобы продемонстрировать возможности Unity на практике. Пересмотрев несколько зарубежных книг по Unity, пришел к выводу, что всем им присущ один недостаток: они начинают разработку какой-то сложной игры, например, шутера или гоночной игры, но не доводят процесс ее разработки до конца. Например, создают одну-две сцены, а дальше – проявите фантазию. С одной стороны, можно понять авторов таких книг: разработка сложной игры вряд ли впишется в объем одной книги, а писать трехтомник вроде «Искусства программирования» Д. Кнута сейчас нерентабельно с никто его печатать не будет, а тем более покупать. С другой стороны, кто будет понимать читателей, которых бросили на полпути? Получается, что такие книги создают больше вопросов, чем ответов.

Поэтому создание сложной игры отбросил сразу – даже если начать объяснение основ в самом начале книги, то вряд ли бы к концу книги она была бы закончена. А по сему, нужно было придумать какую-то простенькую игру, на примере которой продемонстрировать все этапы ее создания. Хотя в Unity можно создавать и 2D-игры, такую возможность также не рассматривал – не для этого вы прочитали большую часть этой книги, разбирались с трехмерными моделями, чтобы «на выходе» получить двухмерную змейку. Это не интересно, да и никто сейчас в такие игры не играет.

Поэтому однозначно будем создавать 3D-игру, но не слишком сложную, чтобы можно было создать ее полностью.

Первое, с чего нужно начать создание игры – поиск идеи и вдохновения. Возможно, у вас есть какой-то сюжет, который вы бы хотели обыграть в игре. А возможно, вас вдохновила другая игра, которую вы бы хотели сделать лучше. Возможно, это будет игра детства, в которую вы играли еще на ПК вроде ZX-80. Для меня такой игрой стал Bomberman. Именно такую игру (естественно трехмерную), мы реализуем с помощью Unity. Игра будет на два игрока – так что вы будете играть не с компьютером, а со своим другом.

Смысл игры заключается в следующем. У нас есть поле, на котором находится небольшой лабиринт. Препятствия на сцене в виде стен лабиринта позволяют игрокам спрятаться от взрыва бомб. Игрок управляет персонажем, находящимся в лабиринте, состоящем из неразрушаемых стен. Он может оставлять бомбу, взрывающуюся через некоторое фиксированное время. Взрыв бомбы распространяется по незаполненным квадратам сцены, не по всем, а только по тем, которые находятся в радиусе взрыва. Если в этот радиус попадает один из игроков, он умирает.

В нашей игре мы сделаем два игрока. Одним можно будет управлять клавишами WASD, вторым – стрелками. Первый игрок выбрасывает бомбу с помощью клавиши **Пробел**, второй – **Enter**. Игрок может установить несколько бомб – по одной в каждом пустом квадрате.

С момента установки до взрыва бомбы должно пройти некоторое время. Взрывы могут вызывать цепные реакции. Например, время до взрыва бомбы составляет пусть 3 секунды. Игрок ставит бомбу 1, затем через 2 секунды ставит бомбу 2, находящуюся в радиусе поражения бомбы 1. Когда бомба 1 взорвется, это породит взрыв бомбы 2 – она не будет дожидаться срабатывания таймера и взорвется раньше.

Оригинальная игра Bomberman вышла в начале 80-ых и впервые мне посчастливилось играть в нее еще на ПК, к которому в качестве носителя

данных подключался кассетный магнитофон. Bomberman можно воссоздавать снова и снова – это вечная игра, которая всегда будет интересна и актуальна, независимо, какой год на дворе.

Оригинальная игра была двухмерной, но с помощью Unity мы создадим ее трехмерный вариант – так игра не будет казаться динозавром на фоне современных игр.

12.2. Что будет в игре

Создавая эту игру, вы научитесь базовым принципам разработки игр:

- Сбрасывать бомбы и привязывать их игровой поверхности. Аналогичным образом можно устанавливать мины в каком-то шутере от первого лица
- Обрабатывать взрывы с двумя типами объектов – с игроками и другими бомбами
- Собственно, порождать взрывы
- Обрабатывать смерть игрока для вычисления победителя. В нашей игре может победить игрок 1, игрок 2 или же результатом может быть ничья, если погибли оба игрока

12.3. Подготовка ассетов

Прежде, чем приступить к программированию игры, нужно позаботиться об ассетах для игры. Во-первых, вам понадобятся модели игроков и других объектов, которые вы собираетесь использовать в игре (например, деревья, кусты, если есть желание создать стрелялку в лесу). Во-вторых, материалы. В-третьих, набор звуковых эффектов, например, звук взрывающейся бомбы, звук выстрела и т.д. Надеюсь, после прочтения первых двух частей этой книги, вы это понимаете.

Чтобы вы сконцентрировались именно на разработке самой игры, все ассеты будут уже подготовлены мною для вас. Откройте проект с игрой (он находится в папке виртуального диска для этой книги, размещенного на сайте издательства), в нем вы найдете следующие типы ассетов:

- **Animation Controllers** – анимационные контроллеры. Содержит контроллер анимации игроков, включая логику для анимации конечностей игроков, когда они ходят

- **Materials** – материалы, необходимые для построения игры
- **Models** – модели персонажей, бомбы, а также материалы, необходимые для игрока и для бомбы
- **Music** – содержит саундтрек
- **Physics Materials** – физические материалы игроков, специальный вид материалов, которые добавляют физические свойства поверхностям. Для упрощения нашей игры оба игрока перемещаются по уровню без трения
- **Prefabs** – префабы, содержат бомбы и префабы взрывов
- **Scenes** – содержит одну сцену, сцену игры
- **Scripts** – содержит все необходимые скрипты
- **Sound Effects** – звуковые эффекты – для бомбы и для взрыва
- **Textures** – здесь находятся текстуры игроков

Благодаря тому, что все ассеты уже разработаны, вы можете сконцентрироваться на самом процессе разработке игры, а не думать о том, где достать ассеты.

Далее вы можете действовать одним из двух способов:

1. **Создать новый проект и создать своего Bomberman с нуля, используя готовые ассеты.**
2. **Открыть финальный проект и смотреть, как все устроено.**

Оба варианта правильны: выбирайте тот, который ближе к вам. Если вы уверены в себе и чувствуете, что сами сможете разработать игру, тогда выбирайте первый вариант. Если что, всегда можно подсмотреть то, что не получается в готовом проекте. Совсем уж новички, как в программировании, так и в разработке игр, могут выбрать второй вариант. Я же буду показывать все уже на готовом проекте, поэтому считаю, что вы выбрали вариант 2.

12.4. Разработка сцены

Откройте проект и сцену **Game**. Сцена состоит из блоков стен (Blocks) и блоков пола (Floor), см. рис. 12.1.

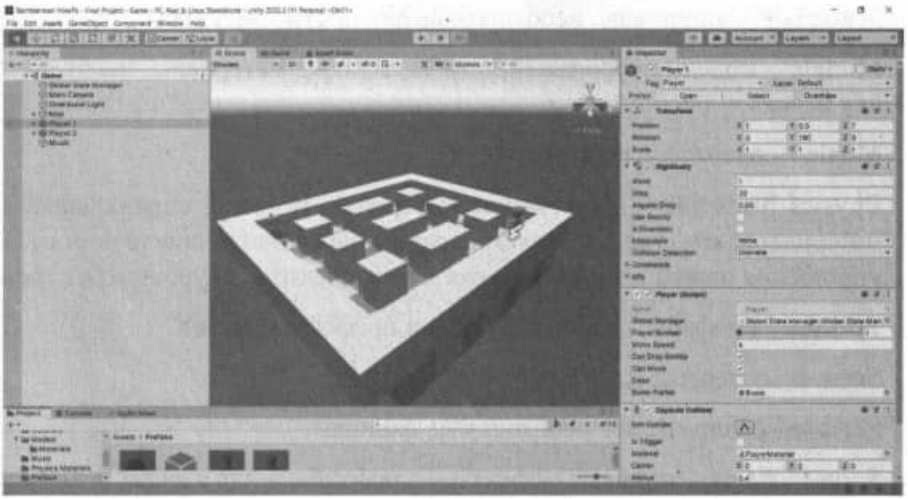


Рис. 12.1. Сцена игры

Сначала разберемся с блоками стен. Они выстроены вдоль сцены, чтобы игрок не смог выйти за ее пределы, также некоторые блоки расставлены на самой сцене, чтобы игрок мог спрятаться за ними и укрыться от взрыва (рис. 12.2).

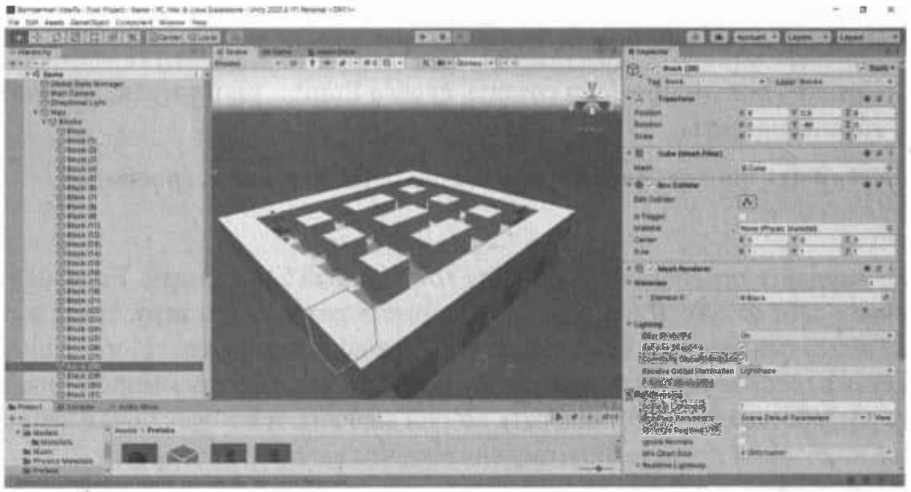


Рис. 12.2. Как устроены блоки

К каждому блоку присоединен **Box Collider** – он идеально подходит для нашего блока. Также блоку назначен материал **Block** – специально разработанный для блока (рис. 12.3).

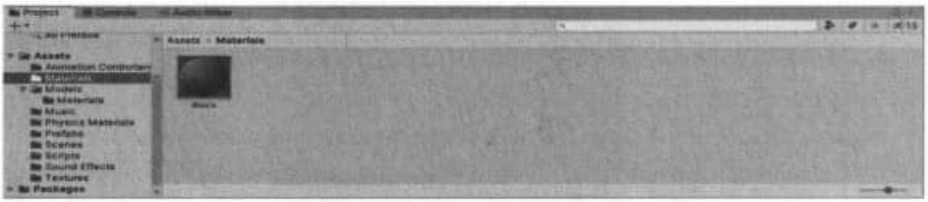


Рис. 12.3. Материал для блока

Пол также создан с помощью блоков, но разного цвета. Цвет задается с помощью компонента **Mesh Renderer** – выбирается материал нужного цвета. Как и для стен, для элементов пола добавлены **Box Collider**. Позиция блоков пола ниже позиции блоков стен, что вполне разумно.

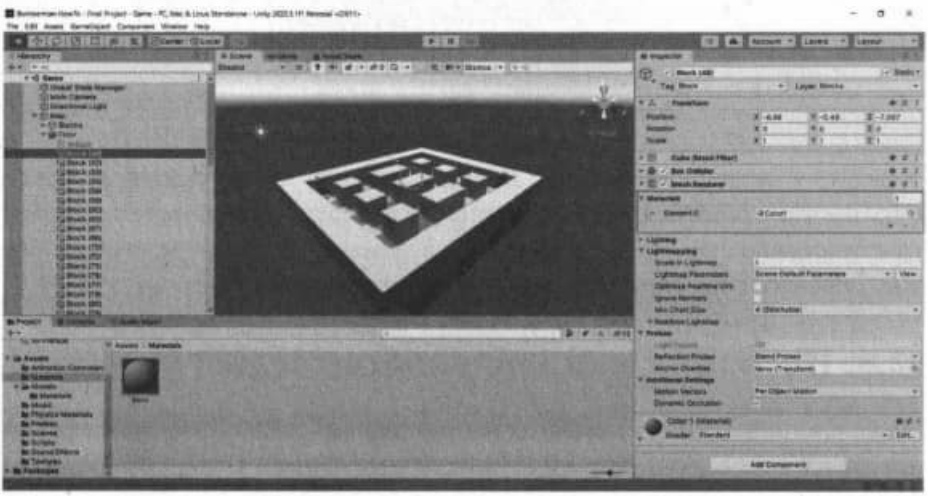


Рис. 12.4. Настройка "Mesh Renderer" для блоков пола

12.5. Настройка главной камеры

Камера должна смотреть на сцену сверху вниз, чтобы обоим игрокам было удобно играть. Мы не можем привязать камеру к игроку, поскольку игроков два. Поэтому нужно выбрать такое положение камеры, чтобы удобно было обоим игрокам. Настройки **Main Camera** показаны на рис. 12.5.

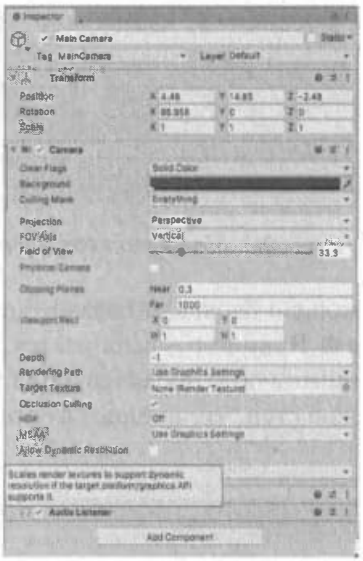


Рис. 12.5. Настройки "Main Camera"

Обратите внимание на свойство **Background**: оно задает фон экрана за сценой. Все что в режиме проектирования (на вкладке **Scene**) отмечено серым, в игре будет заполнено цветом фона. Наша камера работает в режиме перспективы, так что наша игра полностью трехмерная, а не 2.5D.

12.6. Источник направленного света

Также нам понадобится источник направленного света. Его параметры приведены на рис. 12.6. В принципе, ничего необычного в них нет. Мы равномерно освещаем всю сцену.

12.7. Отслеживание состояния

Для отслеживания состояния игры мы создадим пустой объект с именем **Global State Manager**. Ему мы назначим скрипт, который будет отслеживать состояние игроков – GlobalStateManager.cs. Скрипт будет принимать три параметра **Size, Element 0, Element 1**. *Первый параметр* – это количество игроков, *второй параметр* – это первый игрок (в качестве его значения мы указываем объект Player 1), соответственно, *третий параметр* – это второй игрок. Если пожелаете добавить еще игрока, не забудьте создать свойство Element 2 в этом сценарии и увеличить параметр Size.

Рис. 12.6. Параметры направленного света

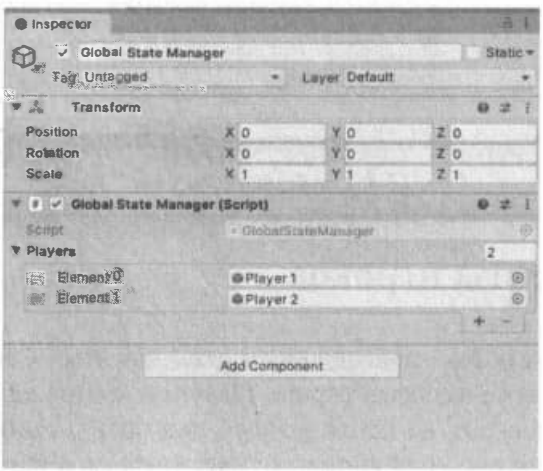


Рис. 12.7. Свойства объекта "Global State Manager"

Примечание. Не беспокойтесь, все исходные коды всех сценариев будут показаны. Пока мы просто разбираемся, что и как устроено!

12.8. Музыкальное сопровождение

Для воспроизведения фоновой музыки используется объект **Music**. При желании, изменив свойство AudioClip, вы можете задать собственный аудиоклип. Свойство **Play On Awake** обеспечивает автоматическое воспроизведение звукового файла (рис. 12.8).

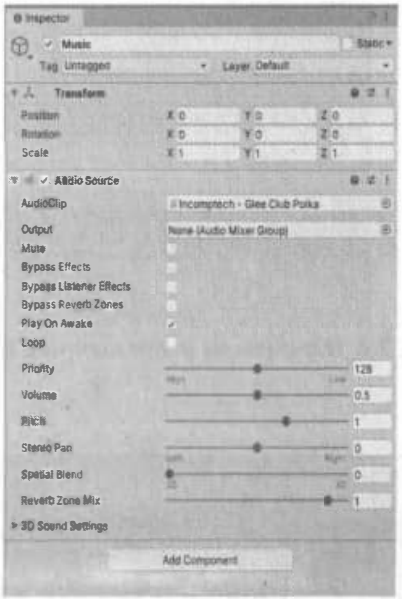


Рис. 12.8. Свойства объекта "Music"

12.9. Объекты игроков

К объектам игроков присоединены компоненты **Capsule Collider** и **Rigidbody**, а также назначен скрипт **Player.cs**, контролирующий поведение игрока. Чтобы игрок не проваливался под сцену, свойство **Use Gravity** отключено – нам оно не обязательно, масса игрока установлена на уровне 1 килограмма.

При желании вы можете изменить уже готовую модель игрока (рис. 12.10) в любом 3D-редакторе, если вас не устраивает стандартная.

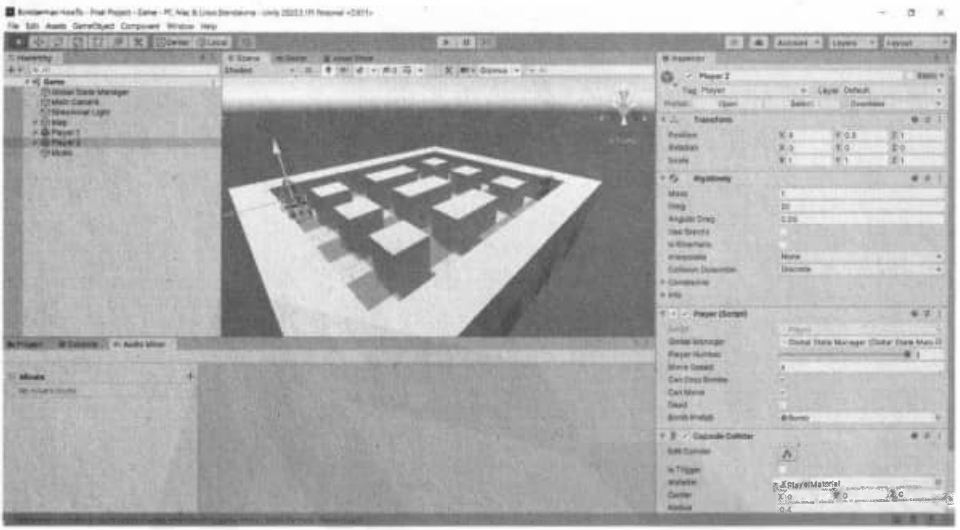


Рис. 12.9. Свойства игрока

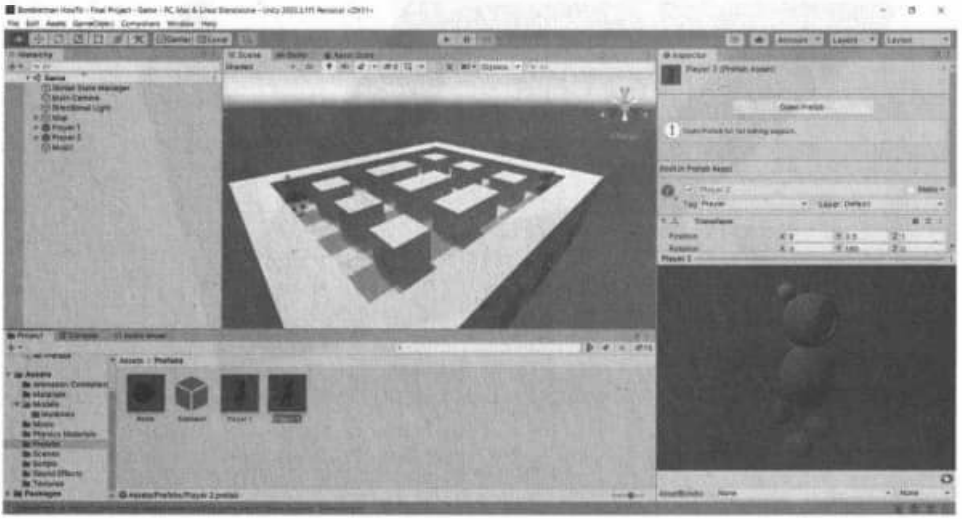
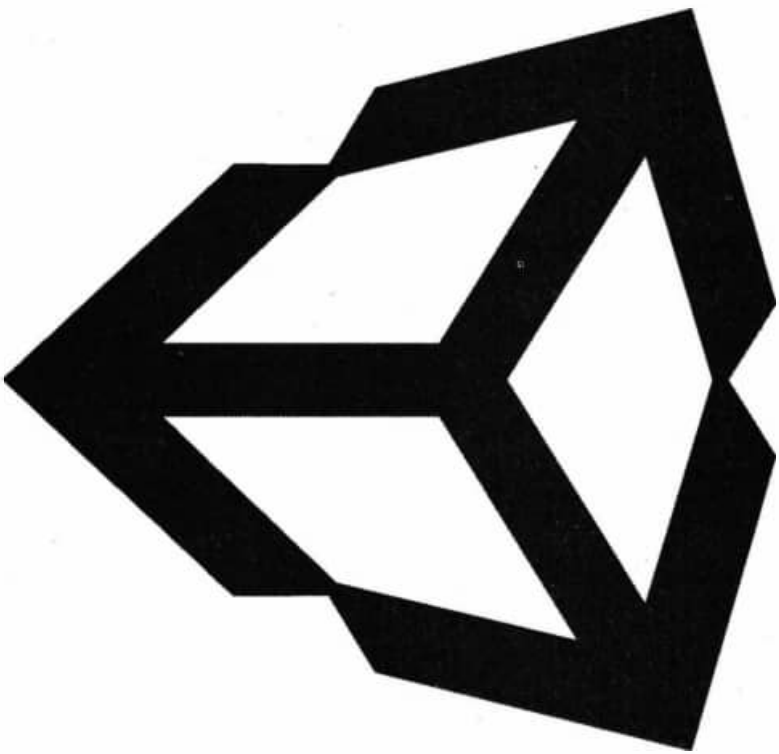


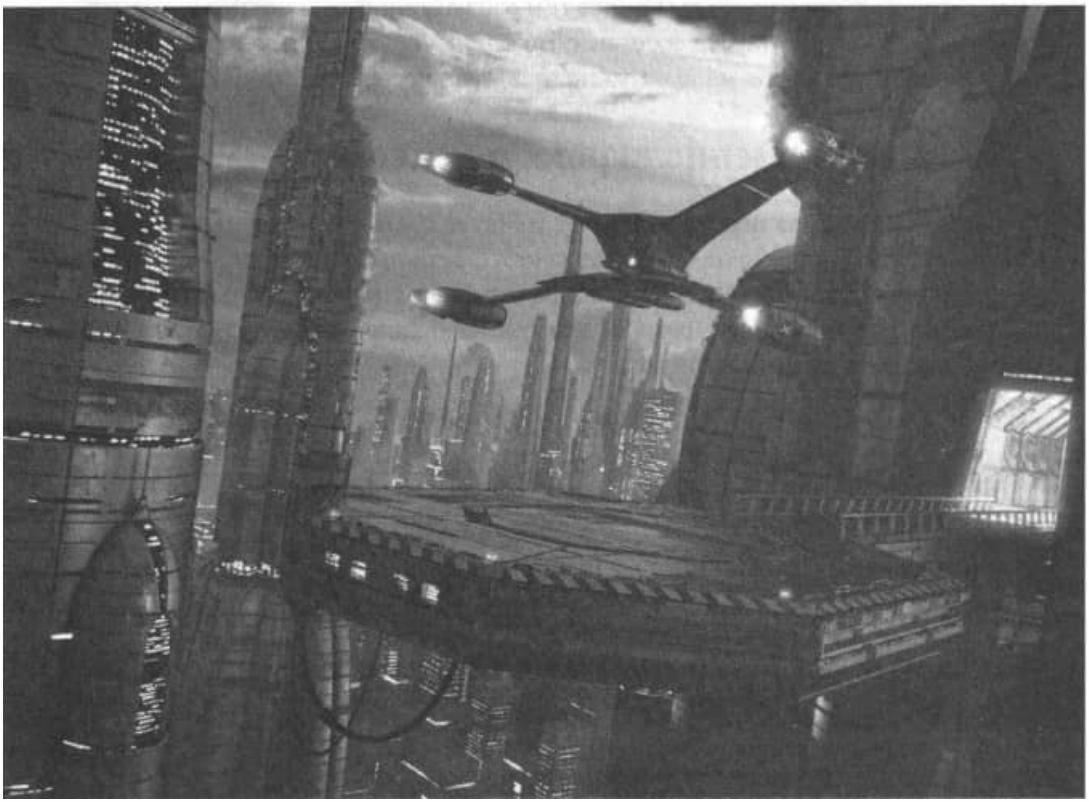
Рис. 12.10. Модель игрока

Теперь, когда мы знаем, как устроена сцена и объекты на ней, можем приступить к реализации геймплея.



Глава 13.

Геймплей в Unity



Ранее вы открыли уже готовый проект и ознакомились с основными элементами сцены, с доступными ассетами. Но одних только ассетов и расставленных на сцене объектов мало – нужно заставить все это двигаться.

13.1. Перемещение игрока по сцене

Итак, у нас есть два игрока – Player 1 и Player 2 – каждому из них назначен сценарий Player.cs. Мы не будем назначать отдельные сценарии для каждого игрока – так будет проще. Даже в более сложных играх отдельные сценарии назначаются кардинально разным персонажам, например, персонажам врагов в шутере и персонажу игрока.

Взгляните на свойства, определяемые скриптом:

- **Global State Manager** – ссылка на менеджер состояния
- **Player Number** – задает номер игрока, значения от 1 до 2
- **Move Speed** – скорость перемещения игрока
- **Can Drop Bombs** – игрок (если включено) может сбрасывать бомбы
- **Can Move** – игрок (если включено) может перемещаться
- **Dead** – признак смерти игрока (если включено, игрок умер)
- **Bomb Prefab** – префаб бомбы

В коде скрипта эти свойства определяются так:

```
    // Менеджер состояния
    public GlobalStateManager globalManager;
    // Параметры игрока
    [Range (1, 2)] //задает номер игрока, доступные значения 1
и 2
    public int playerNumber = 1;
    //Скорость перемещения игрока
    public float moveSpeed = 5f;
    // Может ли игрок размещать бомбы
    public bool canDropBombs = true;
    // Может ли игрок перемещаться
    public bool canMove = true;
    // Игрок мертв?
    public bool dead = false;
```

Метод Update() вызывается каждый кадр. В нем мы вызываем метод Update-Movement, который обновляет перемещения игроков:

```
void Update ()
    {
        UpdateMovement ();       // обновить перемещение
    }

    private void UpdateMovement ()
    {
     // Сбрасываем анимацию прогулки в состояние простоя
        animator.SetBool ("Walking", false);
        if (!canMove)
        {   // Возвращаемся, если игрок не может двигаться
            return;
        }

        // В зависимости от номера игрока,
  // используем другой способ перемещения
        if (playerNumber == 1)
        {
            UpdatePlayer1Movement();
        } else
        {
            UpdatePlayer2Movement();
        }
    }
```

Далее, мы останавливаем анимацию и проверяем, может ли игрок двигаться. Если не может, тогда выходим из метода. Если может, мы проверяем номер игрока и в зависимости от его номера, мы вызываем соответствующий метод перемещения – UpdatePlayer1Movement() или UpdatePlayer2Movement().

Эти методы практически равносильны, но используют разные методы ввода для перемещения того или иного игрока. Если нажаты клавиши WASD, то выполняется перемещение первого игрока, если клавиши-стрелки – второго. При нажатии клавиши **W** или стрелки вверх, игрок идет вверх, **S** (или стрелки вниз) – вниз, **A** (или стрелки влево) – влево, **D** (или стрелки вправо) – вправо.

```
// Перемещение игрока 1
private void UpdatePlayer1Movement ()
    {
        if (Input.GetKey (KeyCode.W))
        { // Идем вверх
            rigidBody.velocity = new Vector3 (rigidBody.
velocity.x, rigidBody.velocity.y, moveSpeed);
            myTransform.rotation = Quaternion.Euler (0, 0, 0);
            animator.SetBool ("Walking", true);
        }

        if (Input.GetKey (KeyCode.A))
        { // Идем влево
            rigidBody.velocity = new Vector3 (-moveSpeed,
rigidBody.velocity.y, rigidBody.velocity.z);
            myTransform.rotation = Quaternion.Euler (0, 270,
0);
            animator.SetBool ("Walking", true);
        }

        if (Input.GetKey (KeyCode.S))
        { // Идем вниз
            rigidBody.velocity = new Vector3 (rigidBody.
velocity.x, rigidBody.velocity.y, -moveSpeed);
            myTransform.rotation = Quaternion.Euler (0, 180,
0);
            animator.SetBool ("Walking", true);
        }

        if (Input.GetKey (KeyCode.D))
        { // Идем вправо
            rigidBody.velocity = new Vector3 (moveSpeed,
rigidBody.velocity.y, rigidBody.velocity.z);
```

```
            myTransform.rotation = Quaternion.Euler (0, 90,
0);
            animator.SetBool ("Walking", true);
        }

        if (canDropBombs && Input.GetKeyDown (KeyCode.Space))
        { // Бросаем бомбу
            DropBomb ();
        }
    }
```

Методика перемещения объекта игрока следующая:

- Мы устанавливаем вектор скорости для **Rigidbody**, при этом мы устанавливаем только скорость перемещения по оси X (она задается свойством moveSpeed), а для осей Y и Z не трогаем
- Мы устанавливаем свойство *rotation* посредством метода **Quanterion**. Euler()[1], который обеспечивает поворот модели игрока на определенный угол по определенной оси – ось и угол определяются в зависимости от нажатой клавиши перемещения
- Для аниматора устанавливаем состояние Walking в *true*

Метод UpdatePlayer2Movement() аналогичен методу перемещения первого игрока, но отличается только другим способом ввода. Также в этих методах есть код сброса бомбы, но об этом мы поговорим в следующем разделе.

13.2. Сброс бомбы

Для сброса бомбы Игрок 1 должен нажать **Пробел**, а Игрок 2 – **Enter** (причем все равно какой – или основной, или на цифровой клавиатуре):

```
  if (canDropBombs && Input.GetKeyDown (KeyCode.Space))
  { // Бросаем бомбу
       DropBomb();
   }
```

```
 if (canDropBombs && (Input.GetKeyDown (KeyCode.KeypadEnter)
 || Input.GetKeyDown (KeyCode.Return)))
 {   // Сброс бомбы. Мы реагируем, как на основной Enter,
// так и на Enter на цифровой клавиатуре.
```

1 См. https://docs.unity3d.com/ScriptReference/Quaternion.Euler.html

```
        DropBomb();
    }
```

Метод DropBomb() обеспечивает сброс бомбы. Работает он так: сначала проверяет, назначен ли игроку префаб бомбы. Мы не используем один и тот же префаб, чтобы была возможность при желании назначить разные префабы разным игрокам. Для установки бомбы используется метод Instantiate(), которому нужно передать префаб (bombPrefab), а также позицию префаба на сцене и информацию о его вращении:

```
    private void DropBomb ()
    {
     // Сначала нужно проверить, назначен ли префаб бомбы
        if (bombPrefab)
        {   // Создаем бомбу и помещаем ее на плитку
            Instantiate (bombPrefab,
                new Vector3 (Mathf.RoundToInt (myTransform.
position.x),
     bombPrefab.transform.position.y,
Mathf.RoundToInt (myTransform.position.z)),
                 bombPrefab.transform.rotation);
        }
    }
```

Бомба помещается в том же месте, что и игрок, поэтому игроку еще нужно успеть отбежать от нее.

На данный момент пока идет все хорошо, но есть один момент, на который нужно обратить внимание. Если не использовать округление, то игрок может поставить бомбу в любом месте карты, даже между плитками:

```
Instantiate(bombPrefab, myTransform.position, bombPrefab.
transform.rotation);
```

Для решения этой проблемы, мы округляем позицию игрока, дабы бомба оказалась именно в плитке, а не посредине между плитками. Для этого используется метод Mathf.RoundToInt(). Чтобы вы поняли, зачем мы используем округление, посмотрите на рис. 13.1.

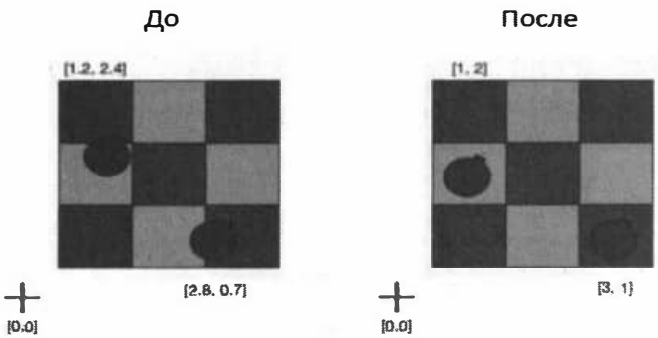


Рис. 13.1. Необходимость использования округления

13.3. Делаем взрыв

Настало время создать взрыв. Начнем с создания сценария взрыва – создайте сценарий Bomb.cs. Затем этот сценарий нужно перетащить на префаб бомбы или же открыть префаб бомбы, нажать кнопку **Add Component** в инспекторе объектов и выбрать скрипт Bomb.cs. В готовом проекте всего этого делать не нужно, но если вы создаете все с нуля, то эти действия необходимо выполнить.

Параметры, которые будут передаваться в этот скрипт, выглядят так:

```
public AudioClip explosionSound;
public GameObject explosionPrefab;
public LayerMask levelMask;
```

Первый параметр – это ассет звука, который будет воспроизведен во время взрыва бомбы. **Второй** – это префаб объекта взрыва. **Третий** – маска слоя, гарантирующая, что лучи, проверяющие наличие свободных мест, попадут только в блоки на уровне.

Скрипт принимает ассет звукового файла и префаб взрыва. На эти поля нужно переместить соответствующие объекты – звуковой файл и префаб взрыва – если создаете игру с нуля.

Чтобы вызвать взрыв, внутри метода Start() мы вызываем метод Invoke(), принимающий два параметра:

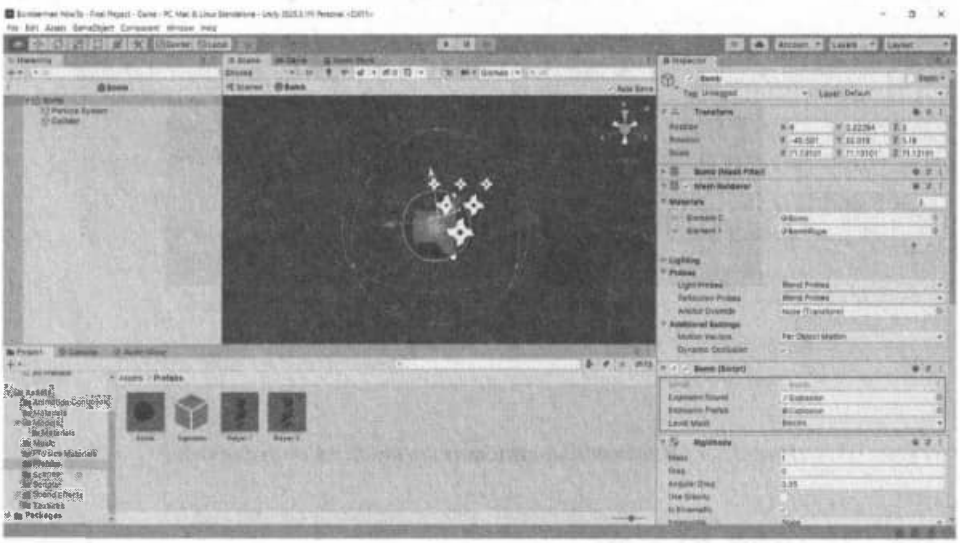


Рис. 13.2. Параметры скрипта Bomb.cs. Скрипт присвоен префабу "Bomb"

```
Invoke("Explode", 3f);
```

Первый параметр задает имя вызываемого метода, а второй – задержку в секундах перед вызовом этого метода. Другими словами, наша бомба должна взорваться спустя 3 секунды после того, как она будет размещена игроком.

Далее нужно разработать сам метод Explode(). В самом простом случае его код будет таким:

```
    void Explode ()
    {
        // Воспроизводим звук взрыва
        AudioSource.PlayClipAtPoint (explosionSound,
transform.position);

        // Создаем первый взрыв в позиции бомбы
        Instantiate (explosionPrefab,
transform.position, Quaternion.identity);

        GetComponent<MeshRenderer>().enabled = false;
        transform.Find("Collider").gameObject.
SetActive(false);
  Destroy (gameObject, .3f); // Уничтожаем бомбу
    }
```

Данный метод взорвет бомбу только в месте ее установки. Сначала мы воспроизводим звук, затем отображаем префаб взрыва и скрываем бомбу спустя 0.3 секунды после взрыва. Два вызова между Instantiate() и Destroy() наводят порядок с мешем и коллайдером бомбы. Первый вызов скрывает меш, второй уничтожает коллайдер, чтобы игроки могли перемещаться сквозь область взрыва.

13.4. Маска слоя

Давайте определимся, что происходит во время взрыва. Взрыв влияет на игроков и другие бомбы. Он не должен влиять на стены, которые должны защищать игрока от взрыва. Нам нужно знать, является ли объект стеной или нет. Лучший способ сделать это – использовать маску слоя (LayerMask).

LayerMask выборочно отфильтровывает определенные слои и обычно используется с raycasts. В этом случае вам нужно отфильтровать только блоки, чтобы луч не попал ни в что другое.

В верхнем правом углу раскройте меню **Layers** и выберите команду **Edit Layers**. Щелкните по текстовому полю рядом с **User Layer 8** и введите название нового уровня – **Blocks**. Этот уровень мы будем использовать для наших блоков.

В иерархическом представлении выберите **Blocks** (для выбора всех блоков на сцене) внутри объекта контейнера **Map**, затем из списка **Layer** выберите **Blocks**, как показано на рис. 13.3.

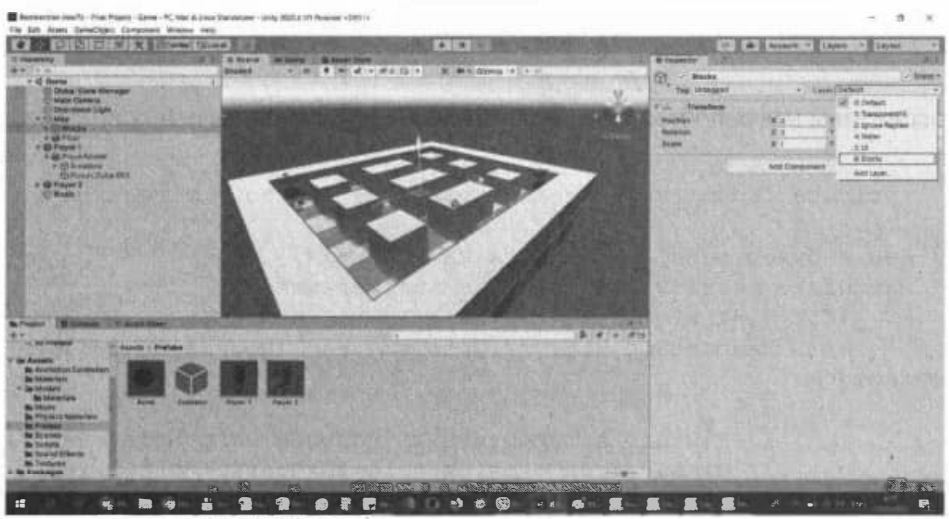


Рис. 13.3. Изменение уровня для всех объектов

Затем появится окно **Change Layer**. Нажмите кнопку **Yes, change children**, чтобы применить изменение слоя для всех блокам, которые есть на карте.

У нашего скрипта Bomb.cs есть параметр **levelMask**. В качестве его значения выберите созданный нами уровень – **Blocks** (рис. 13.4).

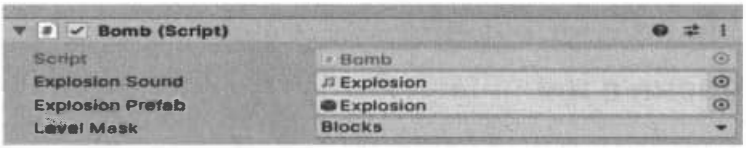


Рис. 13.4. Изменение маски уровня

13.5. Делаем больше взрывов

На данный момент одна бомба порождает один только взрыв. Чтобы игра была интереснее, нужно взрывать еще и соседние плитки. Для этого мы будем использовать так называемые сопрограммы.

> **Примечание.** Сопрограмма – это, по сути, функция, которая позволяет приостановить свое выполнение и вернуть управление Unity. Позже выполнение этой функции возобновится с того места, где оно было остановлено в последний раз. Люди часто путают сопрограммы с многопоточностью. Это не одно и то же: сопрограммы работают в одном потоке и возобновляются в промежуточные моменты времени.

Рассмотрим код сопрограммы, порождающей взрывы на соседних плитках:

```
    private IEnumerator CreateExplosions (Vector3 direction)
    {
// Взрыв будет распространяться на 2 плитки,
// начиная с места взрыва во всех направлениях
        for (int i = 1; i < 3; i++)
        {   RaycastHit hit; // Содержит всю информацию о
рейкасте

            // Рейкаст в определенной позиции на расстоянии i
            // Из-за маски слоя он поражает только блоки, но
не игроков
// или бомбы
```

```
            Physics.Raycast (transform.position + new Vector3
(0, .5f, 0),
direction, out hit, i, levelMask);

            if (!hit.collider)
            { // Свободное пространство, делаем новый взрыв
                Instantiate (explosionPrefab, transform.
position + (i * direction),
explosionPrefab.transform.rotation);
            } else
            { // Останавливаем порождения взрыва в этом
направлении
                break;
            }
    // Ждем 50 мс перед проверкой следующей локации
            yield return new WaitForSeconds (.05f);
    }
}
```

На первый взгляд код кажется очень сложным. Но на самом деле это не так:

1. Первым делом в цикле **for** мы определяем количество плиток (метров), на которые будет распространятся взрыв. Наш взрыв будет распространятся на 2 метра (от 1 до 3 – 2 метра, 2 соседние плитки, 1 плитка = 1 метр).
2. Объект **Raycast Hit** содержит всю информацию о распространении взрыва.
3. Следующая важная строка кода обеспечивает распространение взрывов из центра бомбы в направлении, которое вы передадите при вызове сопрограммы StartCoroutine. Параметр **i** указывает расстояние, которое должны пройти лучи. Наконец, мы используем маску слоя с именем levelMask, чтобы гарантировать, что лучи заденут только блоки, но не игроков и бомбы.
4. Если **Raycast** ничего не задел, то это пустая плитка.
5. В пустой плитке нужно создать новый взрыв.
6. **Raycast** попадает в блок.
7. Как только первый луч попадет в блок, мы прерываем цикл. Это гарантирует, что взрыв не пройдет сквозь стены.
8. Ожидание 0.05 секунды перед следующей итерацией **for** делает взрыв более убедительным.

Теперь вы понимаете причину, по которой мы использовали округление. Посмотрите на рис. 13.5. Белым на схеме изображены блоки. Если бомба выровнена по центру плитки, лучи распространяются правильно (слева). Если же бомба не выровнена по центру, лучи, исходящие из центра бомбы, могут упереться в блоки, которые они не должны поражать.

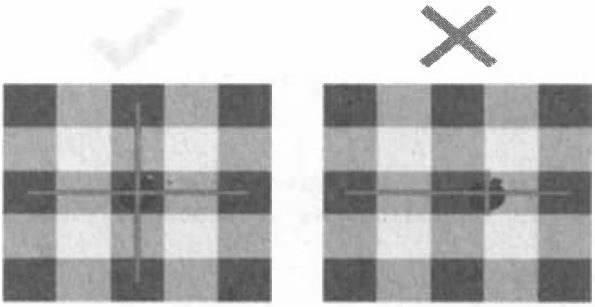


Рис. 13.5. Распространение взрывов

Осталось в методе Explode() добавить вызов нужных нам сопрограмм:

```
// Для каждого направления начинаем цепочку взрывов
StartCoroutine (CreateExplosions (Vector3.forward));
StartCoroutine (CreateExplosions (Vector3.right));
StartCoroutine (CreateExplosions (Vector3.back));
StartCoroutine (CreateExplosions (Vector3.left));
```

13.6. Цепные реакции

Когда взрыв одной бомбы касается другой, следующая бомба должна взорваться – это сделает нашу игру более реалистичной и интересной. На практике это довольно легко организовать с помощью триггеров. Добавьте в сценарий Bomb.cs новый метод OnTriggerEnter():

```
public void OnTriggerEnter (Collider other)
{
    if (!exploded && other.CompareTag ("Explosion"))
    {
        CancelInvoke ("Explode"); // Отменяем вызов Explode
```

```
            Explode (); // Взрыв!
        }
    }
```

Метод OnTriggerEnter() является предопределенным в MonoBehaviour. Он вызывается при столкновении триггерного коллайдера и твердого тела. Коллайдер other является коллайдером игрового объекта, который вызвал срабатывание триггера. В этом случае вам необходимо проверить сталкивающийся объект и заставить следующую бомбу взорваться.

Во-первых, вам нужно знать, взорвалась ли бомба. Для этого мы будем использовать переменную:

```
private bool exploded = false;
```

Наш код делает четыре вещи:

1. Проверяет, не взорвалась ли бомба
2. Проверяет, есть ли у триггер-коллайдера **other** тег Explosion
3. Отменяет уже вызванный вызов Explode, чтобы бомба не взорвалась дважды
4. Взрывает бомбу

Но где устанавливать переменную *exploded*? Вполне логичным местом для этого – метод Explode(), сразу после отключения меша:

```
GetComponent<MeshRenderer> ().enabled = false; // Отключаем меш
exploded = true;
```

Запустите игру и поставьте несколько бомб рядом друг с другом. Теперь у вас есть серьезная разрушительная огневая мощь. Один маленький взрыв может поджечь всю нашу сцену – вот вам пример цепных реакций (рис. 13.6).

Полный код сценария Bomb.cs приведен в лист. 13.1. Данный код является уже законченным и больше не будет модифицироваться.

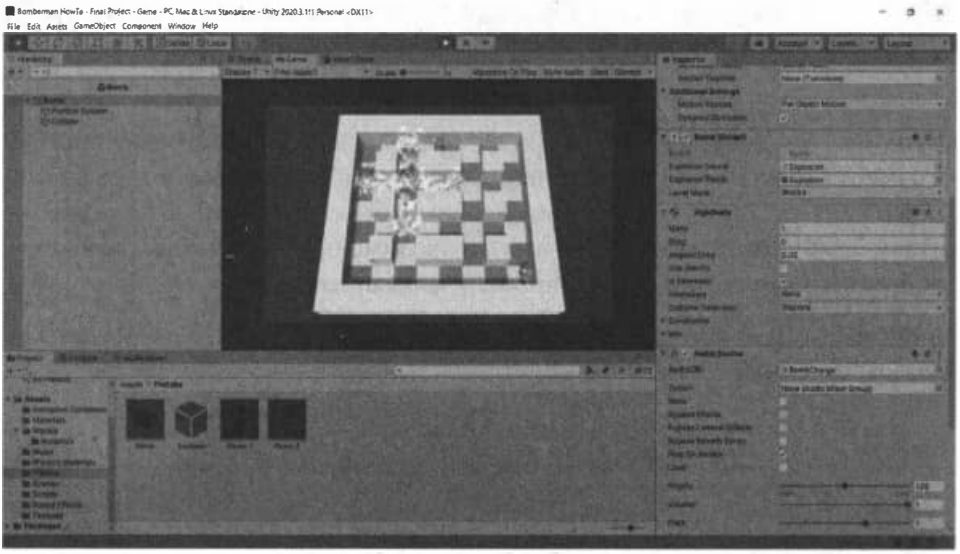


Рис. 13.6. Пример цепной реакции

Листинг 13.1. Полный код Bomb.cs

```
using UnityEngine;
using System.Collections;
using System.Runtime.CompilerServices;

public class Bomb : MonoBehaviour
{
    public AudioClip explosionSound;
    public GameObject explosionPrefab;
    public LayerMask levelMask;

    private bool exploded = false;

    // Инициализация
    void Start ()
    {
       Invoke ("Explode", 3f); // Вызываем взрыв через 3 секунды
    }

    void Explode ()
    {
        // Воспроизводим звук взрыва
        AudioSource.PlayClipAtPoint (explosionSound,
transform.position);
```

```
        // Создаем первый взрыв в позиции бомбы
        Instantiate (explosionPrefab,
transform.position, Quaternion.identity);

        // Для каждого направления начинаем цепочку взрывов
        StartCoroutine (CreateExplosions (Vector3.forward));
        StartCoroutine (CreateExplosions (Vector3.right));
        StartCoroutine (CreateExplosions (Vector3.back));
        StartCoroutine (CreateExplosions (Vector3.left));
       // Отключаем меш
        GetComponent<MeshRenderer> ().enabled = false;

        exploded = true;
  // Отключаем коллайдер
  transform.Find ("Collider").gameObject.SetActive (false);
  // Уничтожаем бомбу в 0.3с после того, как все
  // сопрограммы закончат работу
  Destroy (gameObject, .3f);
    }

    public void OnTriggerEnter (Collider other)
    {
        if (!exploded && other.CompareTag ("Explosion"))
        {
// Отменяем вызов Explode, иначе бомба взорвется дважды
            CancelInvoke ("Explode");
            Explode (); // Взрыв!
        }
    }

    private IEnumerator CreateExplosions (Vector3 direction)
    {
// Взрыв будет распространяться на 2 плитки,
// начиная с места взрыва во всех направлениях
        for (int i = 1; i < 3; i++)
        {
            RaycastHit hit; // Содержит всю информацию о рейкасте

            // Рейкаст в определенной позиции на расстоянии i
            // Из-за маски слоя он поражает только блоки,
// но не игроков или бомбы
            Physics.Raycast (transform.position + new Vector3
(0, .5f, 0),
direction, out hit, i, levelMask);
```

```
            if (!hit.collider)
            { // Свободное пространство, делаем новый взрыв
                Instantiate (explosionPrefab,
transform.position + (i * direction),
explosionPrefab.transform.rotation);
            } else
            { // Останавливаем порождения взрыва в этом направлении
                break;
            }
// Ждем 50 мс перед проверкой следующей локации
            yield return new WaitForSeconds (.05f);
        }
    }
}
```

Наши бомбы взрываются, но игроки по-прежнему живы. Значит, нам осталось обработать событие смерти игрока, иначе в игре не будет смысла.

13.7. Обработка смерти игрока

Вернемся к нашему сценарию Player.cs. Переменная **dead** является признаком того, жив игрок или нет:

```
public bool dead = false;
```

Также нам нужен менеджер состояния – GlobalStateManager. Ранее мы о нем упоминали и создали, что сценарий Player.cs принимает ссылку на него. Менеджер состояний реализуется посредством сценария GlobalStateManager.cs и уведомляет обо всех смертях игроков, он же определяет, какой из игроков выиграл.

Внутри OnTriggerEnter() у нас уже есть проверка, позволяющая узнать, не пострадал ли игрок от взрыва, но все, что этот метод сейчас делает – просто регистрирует данный факт на консоли:

```
    public void OnTriggerEnter (Collider other)
    {
        if (!dead && other.CompareTag ("Explosion"))
        { // Игрок поражен взрывом
            Debug.Log ("P" + playerNumber + " hit by
explosion!");
```

Нам этого мало. После Debug.Log нужно добавить следующие строки:

```
dead = true;
globalManager.PlayerDied (playerNumber);
Destroy (gameObject);
```

Вот, что мы делаем:

- Устанавливаем флаг **dead** в *true*
- Уведомляем менеджер состояний о том, что игрок мертв
- Уничтожаем игровой объект

Если вы создаете игру с нуля, перейдите в окно иерархии, выделите оба объекта игрока и перетащите объект GlobalStateManager на свойство Global Manager обоих игроков. Теперь каждый игрок, который встанет на месте взрыва, погибнет.

Тем временем мы полностью дописали наш сценарий Player.cs и можем рассмотреть весь его код (лист. 13.2).

Листинг 13.2. Сценарий Player.cs

```
using UnityEngine;
using System.Collections;
using System;

public class Player : MonoBehaviour
{
    // Менеджер состояния
    public GlobalStateManager globalManager;

    // Параметры игрока
    [Range (1, 2)] //задает номер игрока, доступные значения 1
и 2
    public int playerNumber = 1;
    //Скорость перемещения игрока
    public float moveSpeed = 5f;
    // Может ли игрок размещать бомбы
    public bool canDropBombs = true;
    // Может ли игрок перемещаться
    public bool canMove = true;
    // Игрок мертв?
    public bool dead = false;
```

```
    // При желании можно ограничить количество бомб, которые
    // есть у игрока,
    // но это требует
    // дополнительного кодирования, мы не ограничивали к-во
    // бомб по умолчанию
    //private int bombs = 2;

    // Префабы
    public GameObject bombPrefab;

    // Доступ к компонентам
    private Rigidbody rigidBody;
    private Transform myTransform;
    private Animator animator;

    // Инициализация
    void Start ()
    {
        // Кэшируем компоненты для лучшей производительности и
   // чтобы дальнейший код был проще
        rigidBody = GetComponent<Rigidbody> ();
        myTransform = transform;
        animator = myTransform.Find ("PlayerModel").
GetComponent<Animator> ();
    }

    // Update вызывается каждый кадр
    void Update ()
    {
        UpdateMovement ();      // обновить перемещение
    }

    private void UpdateMovement ()
    {
   // Сбрасываем анимацию прогулки в состояние простоя
        animator.SetBool ("Walking", false);
        if (!canMove)
        {   // Возвращаемся, если игрок не может двигаться
            return;
        }

        // В зависимости от номера игрока,
  // используем другой способ перемещения
        if (playerNumber == 1)
        {
```

```
            UpdatePlayer1Movement ();
        } else
        {
            UpdatePlayer2Movement ();
        }
    }

    // Обновляем движение игрока 1, его вращение при
    // перемещении
    // Для перемещения будем использовать WASD, для выброса
    // бомбы - пробел
    private void UpdatePlayer1Movement ()
    {
        if (Input.GetKey (KeyCode.W))
        { // Идем вверх
            rigidBody.velocity = new Vector3 (rigidBody.
velocity.x, rigidBody.velocity.y, moveSpeed);
            myTransform.rotation = Quaternion.Euler (0, 0, 0);
            animator.SetBool ("Walking", true);
        }

        if (Input.GetKey (KeyCode.A))
        { // Идем влево
            rigidBody.velocity = new Vector3 (-moveSpeed,
rigidBody.velocity.y, rigidBody.velocity.z);
            myTransform.rotation = Quaternion.Euler (0, 270,
0);
            animator.SetBool ("Walking", true);
        }

        if (Input.GetKey (KeyCode.S))
        { // Идем вниз
            rigidBody.velocity = new Vector3 (rigidBody.
velocity.x, rigidBody.velocity.y, -moveSpeed);
            myTransform.rotation = Quaternion.Euler (0, 180,
0);
            animator.SetBool ("Walking", true);
        }

        if (Input.GetKey (KeyCode.D))
        { // Идем вправо
            rigidBody.velocity = new Vector3 (moveSpeed,
rigidBody.velocity.y, rigidBody.velocity.z);
            myTransform.rotation = Quaternion.Euler (0, 90,
0);
            animator.SetBool ("Walking", true);
```

```
        }

        if (canDropBombs && Input.GetKeyDown (KeyCode.Space))
        { // Бросаем бомбу
            DropBomb ();
        }
    }

    // Обновляем движение игрока 1, его вращение при
    // перемещении
    // Для перемещения будем использовать стрелки, для
    //выброса бомбы - Enter
    private void UpdatePlayer2Movement ()
    {
        if (Input.GetKey (KeyCode.UpArrow))
        { // Вверх
            rigidBody.velocity = new Vector3 (rigidBody.
velocity.x, rigidBody.velocity.y, moveSpeed);
            myTransform.rotation = Quaternion.Euler (0, 0, 0);
            animator.SetBool ("Walking", true);
        }

        if (Input.GetKey (KeyCode.LeftArrow))
        { // Влево
            rigidBody.velocity = new Vector3 (-moveSpeed,
rigidBody.velocity.y, rigidBody.velocity.z);
            myTransform.rotation = Quaternion.Euler (0, 270,
0);
            animator.SetBool ("Walking", true);
        }

        if (Input.GetKey (KeyCode.DownArrow))
        { // Вниз
            rigidBody.velocity = new Vector3 (rigidBody.
velocity.x, rigidBody.velocity.y, -moveSpeed);
            myTransform.rotation = Quaternion.Euler (0, 180,
0);
            animator.SetBool ("Walking", true);
        }

        if (Input.GetKey (KeyCode.RightArrow))
        { // Вправо
            rigidBody.velocity = new Vector3 (moveSpeed,
rigidBody.velocity.y, rigidBody.velocity.z);
            myTransform.rotation = Quaternion.Euler (0, 90,
0);
```

```
            animator.SetBool ("Walking", true);
        }

        if (canDropBombs && (Input.GetKeyDown (KeyCode.
KeypadEnter) || Input.GetKeyDown (KeyCode.Return)))
        { // Сброс бомбы. Мы реагируем, как на основной Enter,
    // так и на Enter на цифровой клавиатуре.
            DropBomb ();
        }
    }

    /// Бросаем бомбу под игроком
    private void DropBomb ()
    {
        if (bombPrefab)
        { // Сначала нужно проверить, назначен ли префаб бомбы
            // Создаем бомбу и помещаем ее на плитку
            Instantiate (bombPrefab,
                new Vector3 (Mathf.RoundToInt (myTransform.
position.x), bombPrefab.transform.position.y, Mathf.RoundToInt
(myTransform.position.z)),
                bombPrefab.transform.rotation);
        }
    }

    public void OnTriggerEnter (Collider other)
    {
        if (!dead && other.CompareTag ("Explosion"))
        { // Игрок поражен взрывом
            Debug.Log ("P" + playerNumber + " hit by
explosion!");

            dead = true;
// Уведомляем менеджер состояний о том, что игрок мертв
            globalManager.PlayerDied (playerNumber);
            Destroy (gameObject);
        }
    }
}
```

13.8. Объявляем победителя

Пришло время написать сценарий GlobalStateManager.cs. Откройте этот файл. В нем есть две переменные:

```
private int deadPlayers = 0;
private int deadPlayerNumber = -1;
```

Первая переменная – это счетчик погибших игроков. *Вторая* – номер погибшего игрока. Метод PlayerDier() вызывается для погибшего игрока. Ему передают номер игрока, который погиб. Код увеличивает счетчик погибших игроков. Также с задержкой 0.3с вызывается метод CheckPlayersDeath()

```
public void PlayerDied (int playerNumber)
{
    deadPlayers++;

    if (deadPlayers == 1)
    {
        deadPlayerNumber = playerNumber;
        Invoke ("CheckPlayersDeath", .3f);
    }
}
```

Задача этого метода определить, какой из игроков умер, и выполнить какие-либо действия. Мы будем только выводить сообщения на консоль, подтверждающие смерть того или иного игрока:

```
void CheckPlayersDeath ()
{
    if (deadPlayers == 1)
    { //Один игрок умер, второй - победитель

        if (deadPlayerNumber == 1)
        { //P1 умер, P2 - победитель
            Debug.Log ("Player 2 is the winner!");

        } else
        { //P2 умер, P1 - победитель
            Debug.Log ("Player 1 is the winner!");

        }
    } else
```

```
        {  //Оба умерли
            Debug.Log ("The game ended in a draw!");

        }
    }
```

Также нам бы хотелось вывести сообщение игрокам и сообщить, кто победитель. Для этого мы будем использовать метод GUI.Label(), но проблема в том, что этот метод можно вызывать только внутри метода OnGUI(), поэтому мы не можем вызвать его после Debug.Log() в методе CheckPlayersDeath(). Получились, по сути, два одинаковых метода – один мы будем использовать для каких-либо действий после смерти игрока, например, для логгирования счетчика побед/поражений игрока, а второй (OnGUI) – для информирования игрока о выигрыше.

```
   void OnGUI() {
     if (deadPlayers == 1)
        { //Один игрок умер, второй - победитель

            if (deadPlayerNumber == 1)
            { //P1 умер, P2 - победитель
                GUI.Label(new Rect(10, 10, 200, 20), "Player 2
is the winner!");
            } else
            { //P2 умер, P1 - победитель
                GUI.Label(new Rect(10, 10, 200, 20), "Player 1
is the winner!");
            }
        } else if  (deadPlayers == 2) {
            GUI.Label(new Rect(10, 10, 200, 20), "Game
over!");
        }

    }
```

Полный код GlobalStateManager.cs приведен в лист. 13.3.

Листинг 13.3. Полный код сценария GlobalStateManager.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class GlobalStateManager : MonoBehaviour
```

```
{
    public List<GameObject> Players = new List<GameObject> ();

    private int deadPlayers = 0;
    private int deadPlayerNumber = -1;

    public void PlayerDied (int playerNumber)
    {
        deadPlayers++;

        if (deadPlayers == 1)
        {
            deadPlayerNumber = playerNumber;
            Invoke ("CheckPlayersDeath", .3f);
        }
    }

   // отображаем различные сообщения во время игры
   void OnGUI() {
     if (deadPlayers == 1)
        { //Один игрок умер, второй - победитель

            if (deadPlayerNumber == 1)
            { //P1 умер, P2 - победитель
                GUI.Label(new Rect(10, 10, 200, 20), "Player 2
is the winner!");
            } else
            { //P2 умер, P1 - победитель
                GUI.Label(new Rect(10, 10, 200, 20), "Player 1
is the winner!");
            }
        } else if  (deadPlayers == 2) {
            GUI.Label(new Rect(10, 10, 200, 20), "Game
over!");
        }

    }

    // Обработка смерти игрока
    void CheckPlayersDeath ()
    {
        if (deadPlayers == 1)
        { //Один игрок умер, второй - победитель

            if (deadPlayerNumber == 1)
            { //P1 умер, P2 - победитель
```

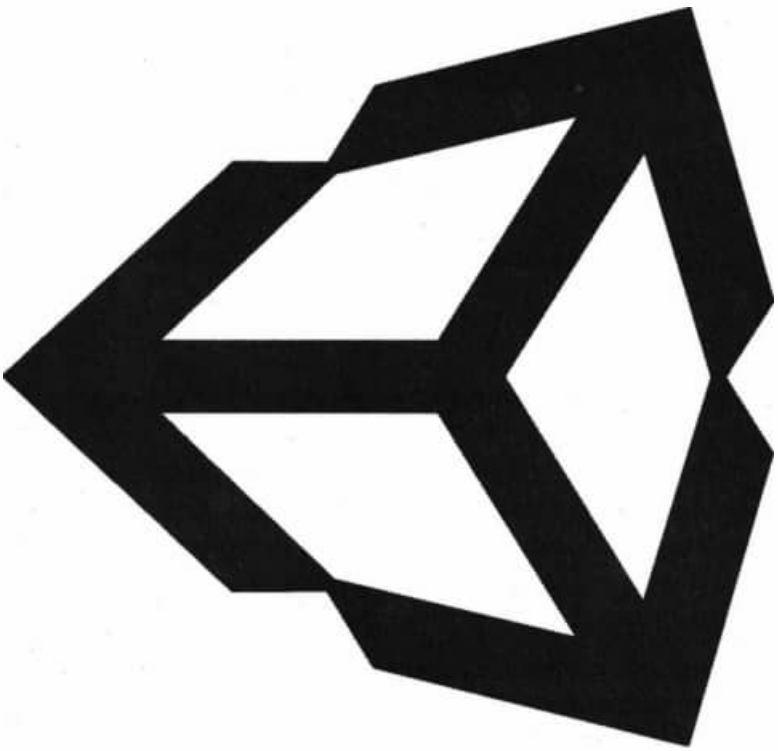
```
                Debug.Log ("Player 2 is the winner!");

            } else
            { //P2 умер, P1 - победитель
                Debug.Log ("Player 1 is the winner!");
            }
        } else
        {  //Оба умерли
            Debug.Log ("The game ended in a draw!");

        }
    }
}
```

На этом игра готова. При желании вы можете ее усовершенствовать, например:

- Улучшить интерфейс пользователя и создать красивое меню для нашей игры;
- Ограничить количество бомб, которые может поставить игрок;
- Подсчитывать число взорванных каждым игроком бомб.

Все зависит от той стратегии, которую вы выберете. Можно ограничить число бомб, доступное игроку, а можно, наоборот просто подсчитывать количество бомб, поставленных каждым игроком. Здесь уже на ваше усмотрение. Можно ввести дополнительных игроков, однако с управлением будет сложновато – за одним компьютером играть втроем не очень удобно.

Далее мы рассмотрим сборку игры. Наверняка вы хотите поделиться своей игрой с друзьями, а для ее запуска за пределами Unity необходимо «собрать» проект, после чего игру можно будет запускать на компьютере, где не установлена Unity.



Глава 14.

Сборка игры

Разумеется, вы создаете игру с целью поделиться с нею общественностью. Исключение составляют лишь редкие пользователи, создающие игру исключительно для себя. А чтобы игру можно было запустить на устройстве, на котором не установлен Unity, ее нужно откомпилировать. После этого можно будет поделиться игрой с друзьями. В этой небольшой главе мы поговорим об особенностях компиляции.

14.1. Создание исполнимого файла игры

Создать исполнимый (exe) файл созданной вам игры достаточно просто:

1. Выберите команду меню **File, Build Settings**. В появившемся окне (рис. 14.1) **Build Settings** вам предстоит сделать кое-какие настройки.
2. Нажмите кнопку **Add Open Scenes**, чтобы добавить имеющиеся сцены в область **Scenes In Build**.
3. Выберите платформы в списке **Platforms**. По умолчанию выбирается платформа **PC, Mac & Linux Standalone**.
4. Выберите целевую платформу в списке **Target Platform** (по умолчанию – **Windows**) и архитектуру (**Architecture**).
5. Список **Compression Method** позволяет выбрать метод сжатия. Помните, что чем выше сжатие, тем ниже производительность.

6. Нажмите кнопку **Build** для создания exe-файла и выберите расположение exe-файла (рис. 14.2)
7. Дождитесь завершения процесса сборки (рис. 14.3).
8. Просмотрите результирующую папку (рис. 14.4). В ней будет исполнимый файл игры, библиотека UnityPlayer.dll и папка с данными игры Game_data. Если вам интересно, то вся папка exe для нашего Bomberman заняла всего 68 Мб. По современным меркам это очень немного. Интересно, что при сборке этого же проекта в Unity 2017, его размер составляет всего 47 Мб. Разница в 21 Мб – не очень большая, но если считать в процентах, то прирост почти 50%, а это уже много.

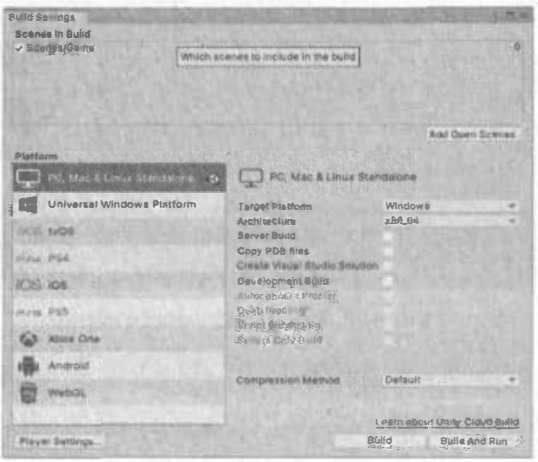


Рис. 14.1. Окно "Build Settings"

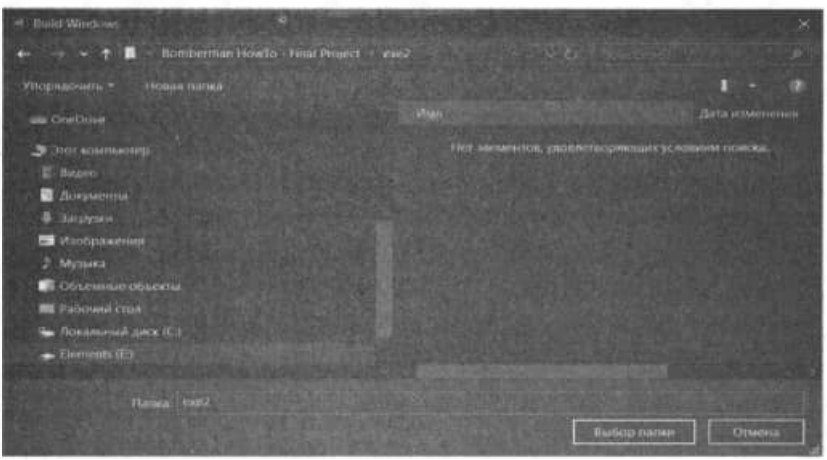


Рис. 14.2. Выберите расположение exe-файла

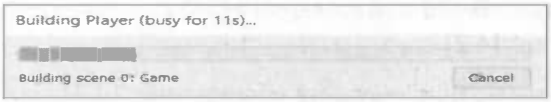


Рис. 14.3. Идет процесс сборки игры

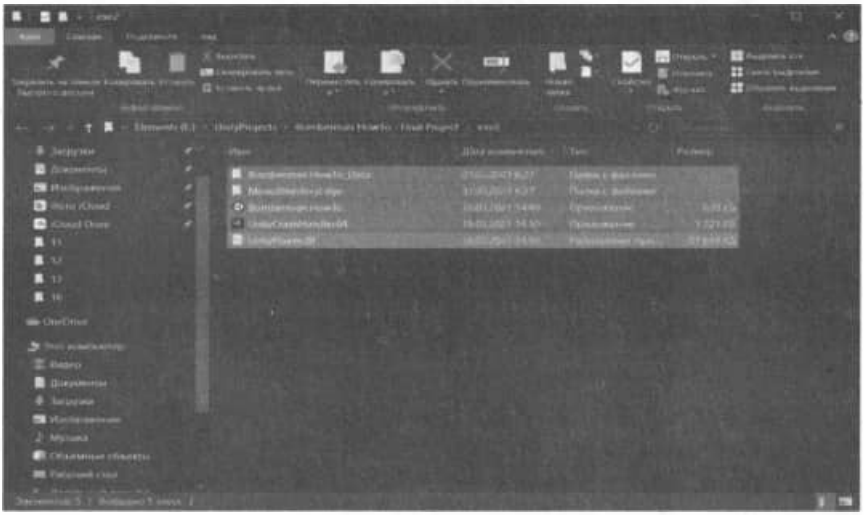
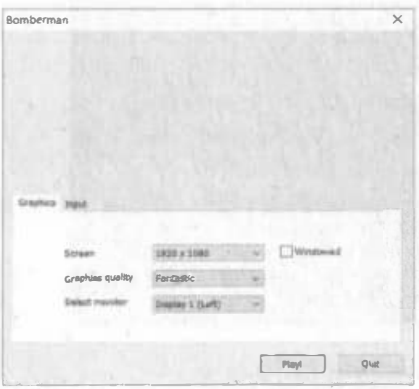


Рис. 14.4. Результирующая папка

14.2. Запуск игры

Просто запустите получившийся exe-файл. При запуске вы увидите окно с настройками игры. Выберите настройки игры – разрешение, качество графики, выберите монитор (если у вас их несколько). Параметр *Windowed* позволяет запустить игру в оконном режиме. На вкладке **Input** вы можете изменить параметры ввода.



Нажмите кнопку Play! для запуска непосредственно самого геймплея (рис. 14.6).

Рис. 14.5. Окно с параметрами запуска

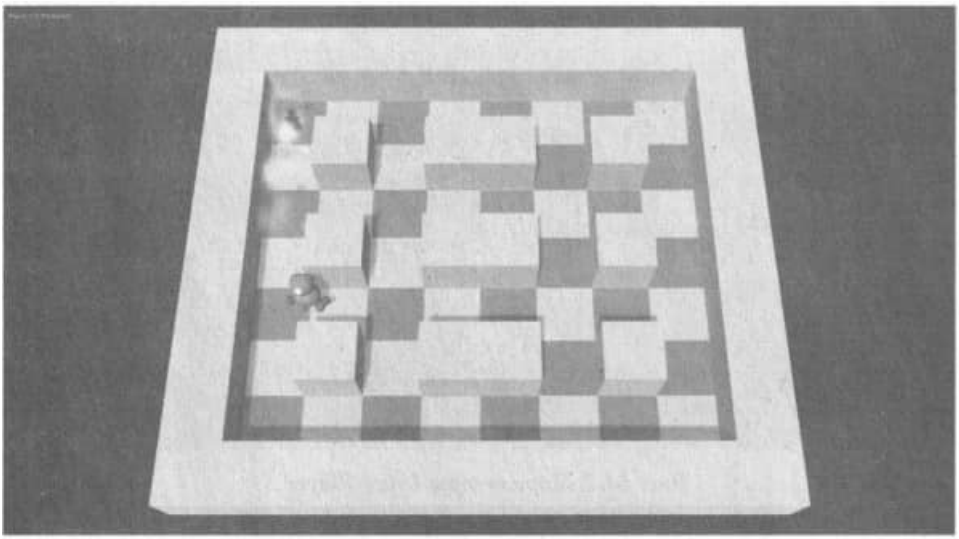


Рис. 14.6. Игра запущена

14.3. Настройка Unity Player

Скорее всего, после того, как вы собрали первый exe-файл, у вас скопилось несколько вопросов:

- Как изменить текст в заголовке Unity Player?
- Как вообще запускать игру сразу без выбора графического режима?
- Как изменить название компании в свойствах exe-файла?

Изменить настройки Unity Player можно, нажав кнопку **Player Settings** в окно **Build Settings**. Вы увидите окно настроек проекта, где можно изменить все необходимые вам параметры (рис. 14.7).

Параметр **Company Name** позволяет изменить имя компании-разработчика игры, **Product Name** – название игры, которое будет отображаться в заголовке окна (рис. 14.5) и в свойствах exe-файла. Параметр **Version** – задает номер версии игры, параметр Default Icon позволяет задать иконку по умолчанию для exe-файла. Если вы не укажете значок, Unity будет использовать свой значок по умолчанию.

Далее следует опции, характерные для каждой платформы. Мы рассматриваем только платформу **PC, Mac & Linux Standalone**, поэтому далее рассмотрим опции только для нее.

Рис. 14.7. Параметры Unity Player

Блок **Icon** позволяет изменить значок, причем указать не один значок на все случаи жизни, а указать значки разных размеров, которые будут использоваться системой при необходимости.

Теперь, что касается диалога выбора разрешения. Он появился, поскольку наша игра (из-за использования определенных ассетов) была создана в Unity 2017. В этой системе показом диалога выбора разрешения управляет параметр **Display Resolution Dialog** (рис. 14.8).

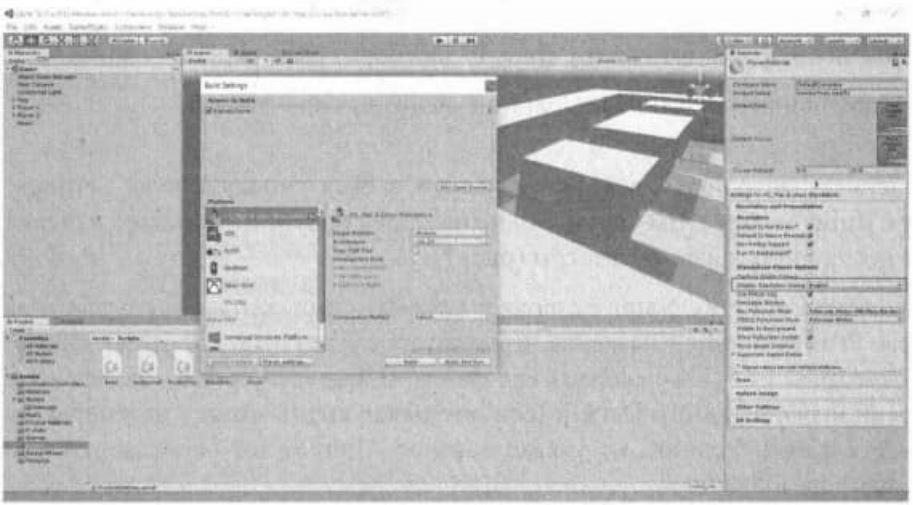


Рис. 14.8. Параметры Unity Player в Unity 2017

В Unity 2019 данный параметр и вообще диалог выбора разрешения упразднен. Согласно исследованиям, этот диалог оказался невостребованным, и

теперь управлять разрешением экрана придется из скриптов. Для полноценных игр ничего не поменялось, так как разработчики создавали собственные сцены с настройками графического режима и всегда отключали диалог выбора разрешения. А вот для таких маленьких, как наш Bomberman, это не очень удобно, поскольку подобный диалог теперь придется реализовывать вручную. Это еще одна из причин (кроме ассетов), по которой игра создавалась в Unity 2017.

> **Примечание.** Если вы по каким-то причинам пропустили предыдущие главы (особенно про интерфейс пользователя), то обязательно вернитесь и прочитайте их. В них показано, как создать UI пользователя – главное меню и окно с параметрами игры. По сути, если вы создадите такой UI, вам больше не нужен диалог выбора разрешения, изображенный на рис. 14.8. В Unity 2019 без подобного UI никак не обойтись, поскольку диалог выбора разрешения упразднен. А в более ранних версиях Unity можно сделать игу более привлекательной, реализовав настройки внутри игры, а не выводить безвкусный диалог при запуске.

Как же теперь управлять разрешением экрана? Для этого нужно использовать класс **Screen** (находите в Unity Engine):

- **Screen.fullscreen** – переключает между полноэкранным и оконным режимом
- **Screen.width, Screen.height** – позволяет узнать текущее разрешение
- **Screen.resolutions** – позволяет узнать все полноэкранные разрешения, поддерживаемые монитором
- **Screen.currentResolution** – свойство, поведение которого отличается в Windows, macOS и Linux:
  - » Windows – содержит текущее разрешение, но в некоторых случаях оно будет отличаться от значений Screen.width, Screen.height
  - » macOS – содержит текущее разрешение экрана, установленное операционной системой
  - » Linux (тестировано в Ubuntu 14.10) – данный параметр бесполезен в конфигурации с несколькими мониторами, поскольку всегда возвращает разрешение области экрана, покрываемой всеми мониторами, что в полноэкранном, что в оконном режиме

» Когда игра находится в оконном режиме, данный параметр вернет разрешение текущего экрана, что в Windows, что в macOS, а не размер окна

- **Screen.SetResolution**(int width, int height, bool fullscreen) – метод, устанавливающий разрешение экрана и режим его работы – если последний параметр будет равен true, игра будет выполняться в полноэкранном режиме

Сменить заставку при запуске игры (рис. 14.9) можно в разделе **Splash Image**. Помните, что если у вас бесплатная версия Unity, свою заставку вы установить сможете, но она будет отображаться после стандартной заставки Unity.

Рис. 14.9. Смена заставки игры

Остальные параметры Unity Player, как правило, изменять не приходится, поэтому оставим их без рассмотрения.

14.4. Создание инсталлятора

Unity позволяет создать только exe-файл, но вот создание инсталлятора выходит за рамки ее возможностей, как и за рамки этой книги. Но все же я не могу оставить читателей в неведении.

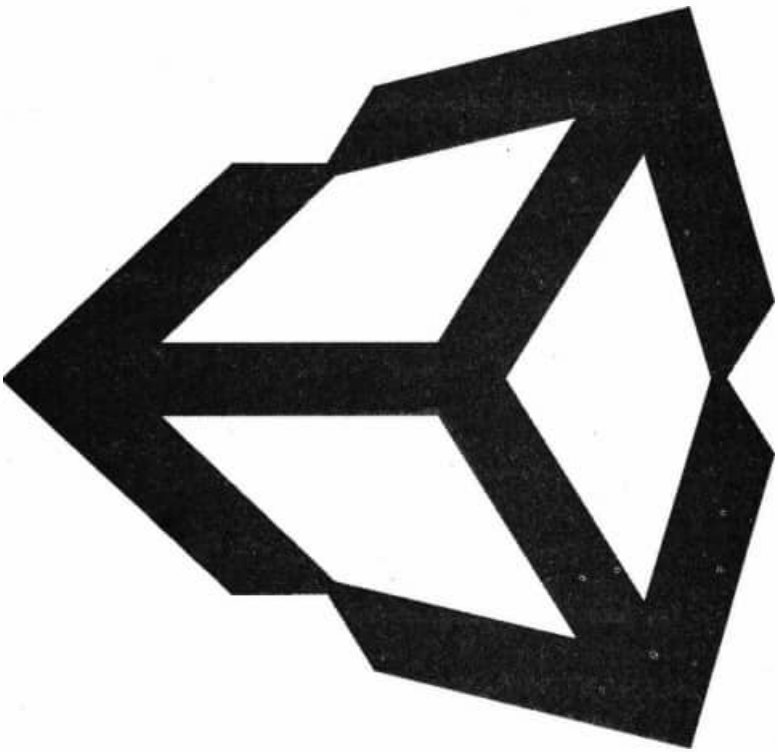
Для создания инсталлятора можно использовать различные программы, предназначенные для этого, например, Inno Setup. По адресу:

https://jazzteam.org/ru/technical-articles/manual-creating-an-installer-using-inno-setup/

вы сможете ознакомиться с руководством по созданию инсталлятора для любого приложения, в том числе и игрового.

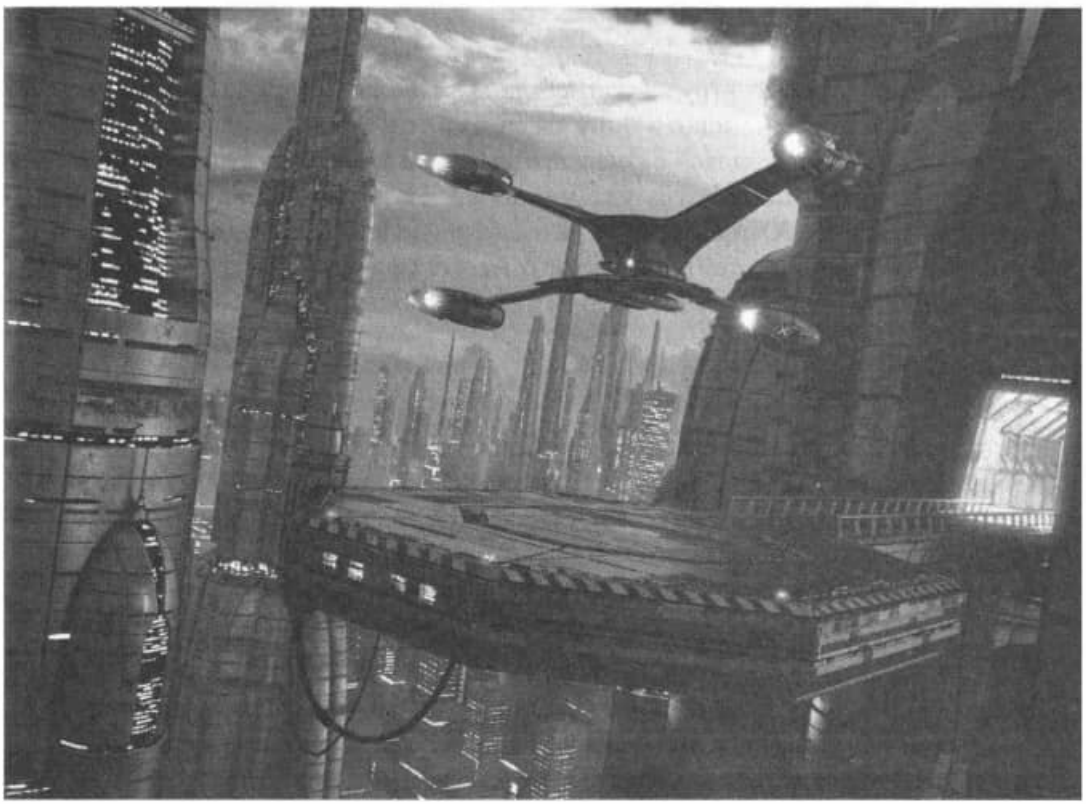
Для нашей простой игры можно создать самораспаковывающийся SFX-архив, если у вас нет времени или желания разбираться с Inno Setup. Такой архив можно создать с помощью любого современного архиватора, например, WinRAR или 7-Zip. Недостаток такого решения – архиватор при распаковке архива не сможет создать необходимые ярлыки, программные группы, отобразить ваш логотип при распаковке архива. Но есть случаи, когда с этим можно мириться, а когда наличие полноценного инсталлятора – необходимость.

В следующих главах мы поговорим о практических аспектах разработки игры, вы узнаете, с какими препятствиями вам придется столкнуться, и вы узнаете, как их обойти. Также будет затронута тема монетизации игры, то есть извлечения прибыли из вашего проекта.



Глава 15.

Практические аспекты разработки игры



Небольшую, простую игру, можно создать самостоятельно. Скорее всего, первую свою игру с помощью Unity вы создадите в гордом одиночестве. Но создание игр – это неплохой бизнес, поэтому рано или поздно вам захочется игру монетизировать. А здесь уже нужно создавать что-то нечто большее, чем простую игру. А чтобы создать увлекательную и эффектную игру, нужна команда – хотя бы по той причине, что вместе процесс пойдет быстрее, и быстрее можно будет получать деньги.

15.1. Искра

Огонь зарождается с искры, а игра – с идеи. Для хорошей игры нужна и хорошая идея. Но одной идеи, к сожалению, мало. Кроме обычной удачи, вам придется столкнуться с множеством подводных камней, о которых вы даже не подозреваете. Вам придется принимать решения очень быстро. Вы должны понимать, что время работает не на вашей стороне. Вполне вероятно, что кому-то в голову может придти та же идея и у него получится создать игру быстрее.

Если есть команда, то еще проще и меньше времени понадобится на создание игры. В целом, от первоначальной идеи до представления игры общественности, например, до публикации ее на App Store, может пройти от 3 до

6 месяцев. За это время может произойти все, что угодно, вплоть даже до распада самой команды. Но не будем драматизировать. Будем надеяться, что у вас все получится.

15.2. Выбор движка

Собственно, если вы читаете эту книгу, то ваш выбор вы уже сделали, особенно, если дочитали до последней главы. Вы можете разрабатывать игру и на C# (в MS Visual Studio), но времени на разработку уйдет в два раза больше. Unity позволяет многие операции сделать путем нескольких щелчков мыши, в то время как в Visual Studio вам придется писать «простыни» кода. Не говоря уже о возможности изменения множества объектов на лету – когда вы находитесь на вкладке Game, вкладки иерархии и инспектора все еще активны и вы можете полностью контролировать процесс путем изменения свойств любых объектов на сцене – изменяете свойства и сразу видите результат. Это очень экономит время!

По моему скромному мнению, **Unity** — это лучший игровой движок для инди-разработки в настоящий момент. Его программная архитектура хорошо продумана, редактор сделан добротно и становится лучше с каждой версией. Начать пользоваться Unity легко, но опытным разработчикам он может показаться поначалу спорным. Тем не менее, и новички, и профессионалы, скорее всего, столкнутся с одними и теми же подводными камнями.

Можно выделить следующие рекомендации относительно использования Unity:

- Необходимо с самого начала потратить время на изучение базовой архитектуры Unity, включая игровые объекты, скриптуемые объекты, сцены, префабы, ассеты, методы-события и порядок их вызова, сопрограммы, измерение времени, обработку ввода, сериализацию. Unity может показаться легким в использовании движком, но без понимания его основ можно провести бесконечные часы за отладкой и рефакторингом.
- С самого начала сформулируйте правила структуризации, а также именования ассетов и игровых объектов. Даже небольшая игра имеет тенденцию превращаться в хаос без должной организации. В Интернете вы без проблем найдете нужные рекомендации. Можно также ориентироваться на проекты, публикуемые разработчиками в Unity Assets Store.
- Программируйте как можно меньше. Вместо этого активно используйте WYSIWYG возможности редактора Unity. Благодаря легкой расши-

ряемости, редактор Unity позволяет превратить разрабатываемую игру в удобный конструктор, которым смогут пользоваться даже те, кто не умеет программировать. А это существенно упрощает и ускоряет создание игрового контента.

- Избегайте использования привычных практик, которые плохо сочетаются с идеологией Unity. Какой бы заманчивой не была возможность размещения всех объектов игры в рамках одной сцены или сохранение параметров игровой механики в XML-файлах, подобный подход существенно осложнит жизнь на более поздних этапах разработки.
- Будьте аккуратны и внимательны при использовании систем управления версиями. Unity имеет тенденцию непредсказуемо менять файлы. Например, внесение изменений в префаб влечет за собой модификацию всех файлов сцен, в которых он используется, но только после их последующего сохранения. Всегда используйте **force text** в качестве режима сериализации ассетов и внимательно следите за файлами, которые заливаете на сервер, чтобы не уничтожить результат работы коллег.
- Тестируйте игру на максимально возможном количестве платформ. При этом не забывайте, что настройки можно определить для каждой из платформ в отдельности. Без подобного тестирования, вы можете столкнуться с тем, что ваша игра ведет себя по-разному в web player-е под Windows и OS X.

15.3. Где брать ассеты?

Скорее всего, если вы создаете свою первую серьезную игру, то бюджет сильно ограничен, поскольку финансируется за счет личных средств самой команды. Не нужно тешить себя надеждами, что все ассеты вы будете создавать самостоятельно и не потратите на них ни копейки.

Создать большое количество качественных игровых ассетов с нуля практически невозможно. Приходится минимизировать количество необходимого. Но даже при условии минимизации, нужно оставить что-то, на что игроку было бы приятно посмотреть.

Где взять ассеты? Первым и самым очевидным ответом для вас будет Unity Assets Store. Какой бы заманчивой ни казалась эта возможность, она влечет за собой и ряд сложностей:

- Ассеты в Unity Assets Store доступны всем без исключения. Однако никто не хочет, чтобы их игра была похожа на другие. Скорее всего, вы тоже не хотите. Можно взять ассеты деревьев, некоторых предметов и т.д. Но вот что делать с ассетами персонажей?
- Практически невозможно купить один пак ассетов, который бы удовлетворил все нужды проекта, и возникает проблема стилистической совместимости. Вполне нормальная ситуация, когда вам придется покупать целые паки ассетов из-за нескольких моделей. В лучшем случае вы будете использовать не более половины набора.
- Несмотря на то, что общее количество ассетов в Unity Assets Store довольно велико, найти среди них что-то подходящее для конкретных нужд бывает весьма непросто. При этом качество зачастую оставляет желать лучшего. Мы потратили часы на поиски, при этом не всегда имея возможность оценить качество пака без его покупки. То есть сначала платим, а потом уже решаем, сгодится ассет или нет. Никаких демо-версий нет.

Второй способ – создать ассеты с нуля. По крайней мере, те ассеты, которые должны быть уникальными для вашей игры.

Третий – воспользоваться услугами аутсорсинга. Если вам повезет, то за небольшое время вы получите уникальные и качественные ассеты. Но придется потратиться. Ассеты на заказ стоят гораздо дороже, чем ассеты в составе готовых наборов.

Можно еще купить ассеты за пределами Unity Assets Store, но вам придется столкнуться с проблемой совместимости с Unity. Если ассеты бесплатные, то попробовать можно. Платить деньги и не иметь точного представления, как именно ассет поведет себя в Unity – удовольствие сомнительное.

Вот несколько простых правил, которым нужно следовать при покупке ассетов Unity Assets Store:

- Чтобы избежать стилистических расхождений, желательно покупать ассеты одного типа у одного автора. Это не мешает купить модели у одной студии, а партиклы у другой, но при этом нужно следить за совместимостью стилей.
- Избегайте использования ассетов в том виде, в котором они были куплены. Достаточно внести небольшие изменения (например, немного перерисовать текстуры, реструкурировать партиклы) или использовать ассеты нестандартным образом.

- Если вы планируете выпуск игры на мобильных платформах, убедитесь, что покупаемые ассеты оптимизированы под них.

Отдельного внимания заслуживает музыка. Вряд ли вы будете создавать музыку самостоятельно, поскольку для этого вам нужен собственный композитор, что является дорогим удовольствием для начинающей команды. К счастью, существует большое количество сервисов, поставляющих royalty-free (музыку, которую можно использовать без платы за ее использование) музыку и звуки, включая Unity Assets Store. И это совсем недорого.

С музыкой относительно несложно. А вот с озвучкой – проблема. Самое правильное решение – обратиться в студию звукозаписи, которая поможет за определенную сумму сделать профессиональную озвучку для вашей игры. Поверьте, результат будет стоить потраченных денег – вряд ли самостоятельно у вас получится сделать лучше.

15.4. Заботимся о монетизации

Just for fun создавался, наверное, только Linux и тот получилось монетизировать, правда, спустя многие годы. Вам же прибыль нужна здесь и сейчас, поэтому нужно подумать, как вы будете извлекать выгоду. Продажа лицензионных коробочных копий – это прошлый век. Рано или поздно ваша игра будет взломана и окажется в свободном доступе где-то на торренте с «таблэткой», как это происходит со многими другими продуктами.

Лучший способ монетизации – внутриигровые покупки. Это когда игрок покупает лучшие доспехи, дома, танки, автомобили и другие объекты за свои кровно заработанные. Понятно дело, что вам понадобится где-то хранить информацию о покупках игрока.

Даже если вы создаете однопользовательскую игру, вам понадобится место хранения данных о текущем прогрессе игрока, статистике его действий, сделанных покупках и прочей информации. Самым простым решением будет использовать для этого класс Unity PlayerPrefs[1]. Однако он сохраняет данные локально и, совершенно очевидно, не подходит для таких деликатных вещей, как внутриигровые покупки.

В поисках лучшего решения можно предложить сервис Parse. Parse — это кросс-платформенный сервис BaaS (не сочтите рекламой), позволяющий приложению сохранять данные в облаке, поддерживает авторизацию поль-

1 https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

зователей, серверные функции, push-оповещения и аналитику. Это далеко не все, но он дает общее представление о том, что есть Parse.

Использование Parse в рамках Unity не сопряжено с особыми трудностями и хорошо документировано. По сути, вам необходимо скачать Parse SDK, настроить вашу игру на сервере Parse и в проекте Unity, а также немного попрограммировать. Один очевидный нюанс: вы не сможете использовать Parse, если устройство, на которое установлена ваша игра, не имеет доступа к Интернету.

Если есть необходимость поддерживать оффлайн-режим и обновлять данные в Parse при наличии сетевого соединения, то вам придется написать для этого небольшую отдельную систему. Для этого можно использовать класс PlayerPrefs. Система сохраняет данные локально и заливает их в Parse, как только обнаруживает наличие подключения к Интернету.

15.5. Мобильная или настольная платформа

В самом начале нужно определиться с платформой: вы будете создавать игру для мобильных или настольных устройств? Это следует сделать в самом начале, поскольку, несмотря на постепенное сближение в течение последних лет, стационарные и мобильные платформы по-прежнему сильно отличаются друг от друга. Это становится очевидным при попытке заставить «красивые» шейдеры работать приемлемо (не говоря уже о том, чтобы заставить их работать быстро) на всех устройствах, на которых предполагается запуск игры. Unity существенно облегчает этот процесс, но, к сожалению, не решает всех проблем.

Некоторые размышления:

- Если вы планируете запуск на мобильных платформах и используете тени, то ограничьтесь режимом **forward lighting**. Несмотря на запекание освещения и сокращения числа объектов, отбрасывающих тени, в режиме **deferred lighting** вы не добьетесь приемлемой производительности.
- Минимизируйте количество draw call-ов. Большое количество полигонов в моделях не скажется так сильно на производительности, как дополнительные draw call-ы.
- Не злоупотребляйте текстурными выборками в шейдерах. Это может привести к существенному падению производительности. Скорее всего, вам придется использовать несколько специальных шейдеров вместо одного универсального – именно по этой причине.

Помимо различий в производительности и аппаратных ограничений, существует еще одно важное отличие между стационарными и мобильными платформами. И это отличие – режим ввода. Игра, созданная для управления с помощью клавиатуры и мыши, плохо переносится на мультитач и акселерометр. Прежде всего, в Unity разделена обработка мыши и мультитача. А потому было необходимо создать систему, унифицирующую этот аспект. Для этих целей можно использовать систему ввода из состава NGUI[2], которая, после небольших доработок, показала себя весьма хорошо. Она также позволяет решить проблему распределения ввода между пользовательским интерфейсом и игровым управлением.

15.6. Что делать с локализацией

По началу мало кто задумывается о локализации. Но если вы хотите достичь мирового господства, ой, то есть признания на мировом уровне, нужно позаботиться о локализации, причем сразу. Пусть игра будет всего на двух языках – русском и английском, но использование последнего даст вам прирост международной аудитории. Далее можно будет расширить список доступных языков.

Вы должны понимать, что локализация – это еще одна часть разработки, которую инди-команда практически никогда не может сделать самостоятельно. Это означает только одно: за локализацию придется заплатить. Та же озвучка в студии звукозаписи.

Локализовывать необходимо все, что имеет отношение к человеческому языку. То есть тексты, речь, надписи на текстурах — все это должно быть локализовано. Такой процесс очень быстро может стать весьма трудоемким и дорогим. Поэтому крайне важно свести количество локализуемого контента к минимуму.

Надписей на текстурах лучше избегать, и использовать, например, текстовые поля NGUI. Если в игре необходима речь, то, скорее всего, потребуются и субтитры, поскольку локализация речи не только дорогое удовольствие, но еще и требовательное с точки зрения места, занимаемого игрой.

Но минимизация локализуемого контента – это только начало. Следующий этап – подготовка самой игры к локализации. Плохая новость состоит в том, что Unity не имеет встроенных механизмов для этих целей. И хотя существует целый ряд специализированных решений в Unity Assets Store (напри-

2 http://www.tasharen.com/?page_id=140

мер, 12 Localization), можно использовать систему локализации, идущую с NGUI, особенно если вы будете использовать саму NGUI.

Система локализации NGUI проста и понятна в использовании. Она построена на основе одного CSV-файла, содержащего колонку для каждого языка. Наличие такого файла очень удобно для отправки строк на перевод (для этого существует большое количество специализированных сервисов) и последующей вставки переведенной версии. С текстовой локализацией проблем не будет. Проблема будет с озвучкой.

15.7. Не забываем о тестировании

После того, как все сделано, игру необходимо тщательно протестировать (стоит это делать и неоднократно в процессе разработки). Этим должно заниматься максимальное количество людей на максимальном количестве устройств. Ничто не портит хорошую игру так, как пропущенный баг. При этом нужно обязательно слушать жалобы игроков. Поскольку эти жалобы превратятся в неудовлетворенных клиентов сразу после выхода игры на рынок. А этого точно никто не хочет.

15.8. Сколько это стоит?

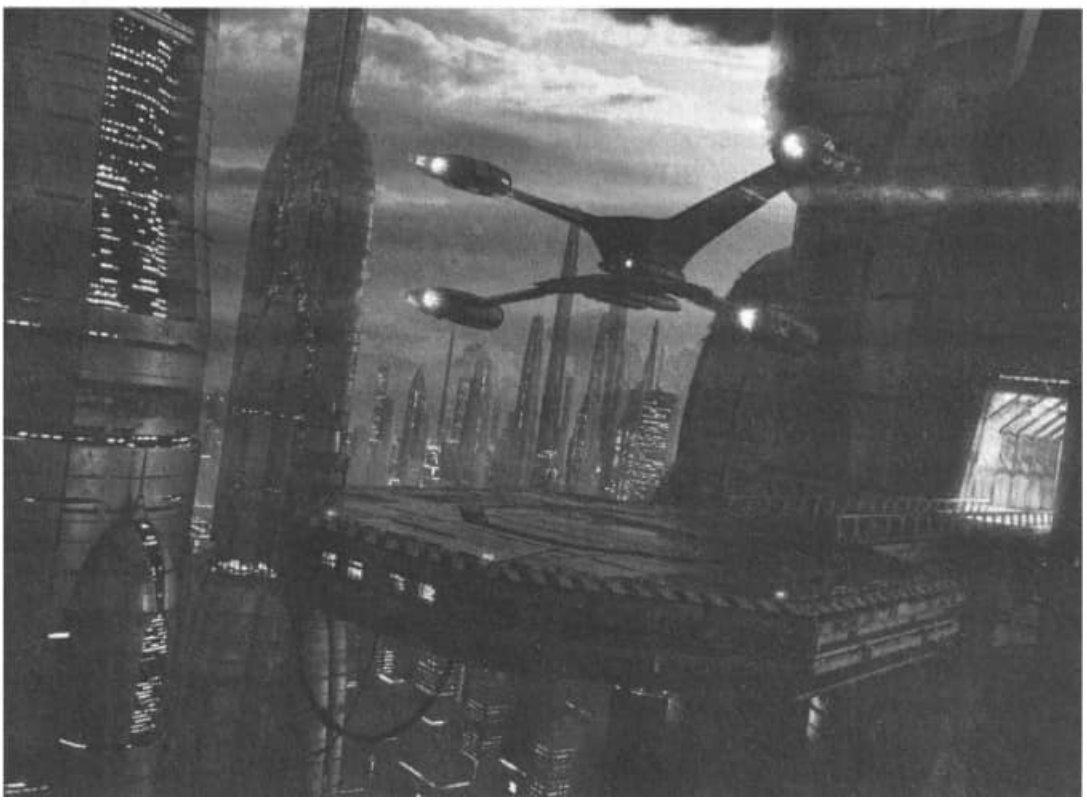
Наверное, самый интересный вопрос. Помните, я в начале главы писал об удаче. Все зависит от того, насколько вы удачливы. В качестве примера могу привести когда-то популярную игру City 2048, на создание которой автор потратил 10 тысяч рублей – на покупку музыки к игре и на покупку девелоперских аккаунтов в Google Play и AppStore (на тот момент – по 99$ за каждый аккаунт). Конечно, затраты на продвижение игры тоже были, всего с продвижением – порядка 20 тысяч рублей. Это очень мало. Но автор был очень удачен, поэтому данные затраты обернулись в 1.5 млн установок, что сулит очень неплохую прибыль.

Однако не всем так везет. Расходы на создание браузерной игры «Звездный воитель» поставили более 1.1 млн рублей по курсу 2015 года. Только на продвижение игры было потрачено более 300 тысяч рублей, на дизайн – более 500. При этом программирование осуществлялось силами команды, то есть затраты, собственно, на разработку не считаются. Сейчас цены выросли, и смело можно умножать данные суммы на 1.5. Готова ли ваша команда или вы лично к таким вложениям? Конечно, вам может улыбнуться удача и затраты будут в 10 раз меньше:)

Надеюсь, данная небольшая глава пролила свет на особенности разработки полноценной игры. По крайней мере, теперь вы знаете, с чем вы столкнетесь и знаете, какие решения можно использовать для обхода всех препятствий, стоящем на вашем пути.

Глава 16.

Обзор 3D Game



16.1. Что такое 3D Game?

Книга почти закончилась. Что делать дальше? Ответ очевиден – продолжать осваивать Unity. В качестве хорошей отправной точки можно посоветовать бесплатный проект игры 3D Game, скачать который можно с магазина ассетов. Это полноценная трехмерная игра с множеством моделей, которые вы можете использовать в своих проектах.

Да, игра состоит из всего двух уровней, но она достаточно занимательна, включает диалоги между персонажами и у нее есть сюжет. Объем проекта достаточно большой, что только подчеркивает его серьезность. Вы можете попробовать реализовать дополнительный, третий, уровень, а можете просто разобраться, что и к чему и использовать уже готовые модели в своей собственной игре. Решать только вам.

16.2. Как установить 3D Game

Игра устанавливается как обычный набор ассетов (процедура установки была описана ранее).

Если вкратце – нужно перейти по ссылке:

https://assetstore.unity.com/packages/templates/tutorials/3d-game-kit-115747

и нажать **большую синюю кнопку**. Дальше нужно следовать инструкциям Unity, которые понятны в большинстве случаев без каких-либо комментариев.

> **Примечание**. Мы заботимся о своих читателях и на всякий случай (если разработчики Unity удалят данный набор ассетов, что происходило и не раз), данный проект вы найдете на прилагаемом к книге виртуальном диске. Просто откройте проект 3dgame, который находится на прилагаемом диске.

Рис. 16.1. Проект 3dgame

16.3. Уровни игры

Как уже было отмечено, игра состоит из двух уровней. Уровни хранятся в папке Assets/ 3DGamekit/ Scenes/ GamePlay. В подпапках Level1 и Level2 хранятся текстуры (exr-файлы), используемые, соответственно, на первом и втором уровнях.

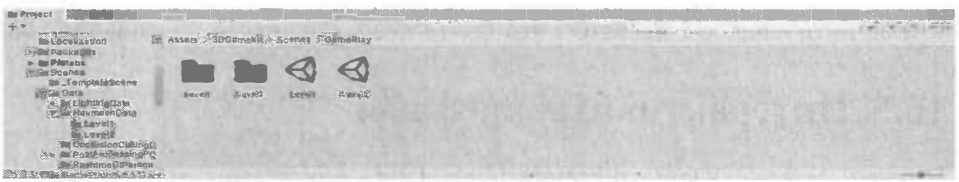


Рис. 16.2. Уровни игры

16.4. Основной персонаж

Основным персонажем является гуманоид по имени Ellen (рис. 16.3). Префаб игрока вы сможете найти в папке Assets/3DGamekit/Prefabs/Characters/Ellen.

Рис. 16.3. Основной персонаж

Для передвижения игрока по сцене используются следующие клавиши:

- **W**, **S**, **A**, **D** – стандартные клавиши перемещения игрока в пространстве
- **Пробел** – прыжок
- **Esc** – пауза
- **Левый клик мыши** – удар
- **Мышь** – управление камерой

Кроме того, в папке **Characters** находятся префабы других персонажей.

16.5. Интерактивные префабы

В папке Assets/3DGamekit/Prefabs/Interactables находятся префабы различных интерактивных элементов, например, перемещающихся платформ, открывающихся дверей и т.д.

Рис. 16.4. Перемещающаяся платформа

16.6. Используемые пакеты

По вполне понятным причинам игра не разрабатывалась на пустом месте. В ее основе лежат следующие пакеты (рис. 16.5):

- **2D Sprite** – содержит набор ассетов, предназначенных для прототипирования двухмерных игр
- **2D Tilemap Editor** – пакет, содержащий функционал для редактирования тайловых карт (Tilemaps)
- **Cinemachine** – набор инструментов для создания динамических, умных, не требующих программирования камер, создающих лучшие кадры на основе композиции сцены и взаимодействия объектов на ней, что позволяет разработчику игр настраивать, разрабатывать и задавать поведение камеры в реальном времени. Данный набор инструментов получил награду «Эмми»
- **Custom NUnit** – открытая среда для Unit-тестирования приложения, используется для создания модульных тестов игры
- **JetBrains Rider Editor** – быстрый и мощный редактор C# для Unity, работающий в Windows, Mac и Linux

- **Polybrush** – дает возможность смешивать текстуры и цвета, создавать сетки и размещать объекты прямо в редакторе Unity
- **Post Processing** – позволяет применять множество полноэкранных фильтров и эффектов к буферу камеры перед его отображением на экране. Позволяет кардинально улучшить внешний вид изображения с минимальными затратами времени на настройку
- **ProBuilder** – уникальный гибрид 3D моделирования и инструмента для дизайна уровней, оптимизированный для создания простых геометрий, но включающий в себе детальное редактирование. Позволяет создавать быстро прототипы строений, сложных ландшафтных возможностей, транспорт и оружие, или создавать произвольные геометрии коллизий, зоны триггеров или нав-мэшей
- **Settings Manager** – управляет пользовательскими данными и настройками
- **Test Framework** – еще один фреймворк для тестирования продукта
- **TextMesh Pro** – это простая в использовании система для высококачественного текста. Она имеет множество параметров внешнего вида и форматирования текста и является простым способом добавить профессиональный вид в пользовательский интерфейс любого проекта
- **Timeline** – это GameObject с компонентом timeline, который можно редактировать в окне Unity Timeline, управляющем ключевыми кадрами анимации и циклами жизни объектов
- **Unity Collaborate** – прощает сохранение, обмен и синхронизацию, обеспечивая доступ к проектам Unity для всех сотрудников независимо от их местоположения
- **Visual Studio Code Editor** – редактор кода (чтобы все было в одном месте)

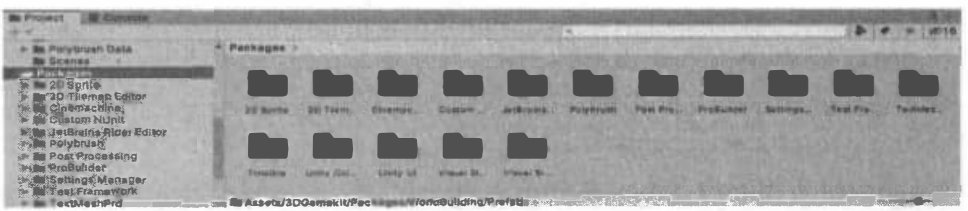


Рис. 16.5. Пакеты, лежащие в основе игры

16.7. Документация

Игра не поставляется сама по себе, в комплекте следует документация, помогающая разобраться что и к чему. Мы не видим особого смысла в переписывании этой документации, поэтому просто сообщаем, что она имеется. Загляните в папку Documentation (рис. 16.6). Даже если вы не знаете английский язык, современные переводчики позволят вам разобраться что и к чему.

Рис. 16.6. Имеющаяся документация

Вместо заключения

Напоследок один бесплатный совет. Исследуйте бесплатные ассеты магазина. Как показывает практика, совсем не обязательно иметь бюджет в несколько десятков тысяч долларов, чтобы создать хорошую игру. Собирайте собственные наборы ассетов. Бывает так, что из импортированного набора понадобится несколько материалов и 1-2 модели. Импортируйте набор в пустой проект, далее экспортируйте понравившиеся модели, импортируйте в рабочий проект игры. Так вы создадите свою коллекцию ассетов, необходимых для игры, и не будете увеличивать размер проекта, храня ненужные модели.

Корнилов Андрей Валентинович

Unity

Полное руководство

2-е издание

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *М. В. Финков*

Редактор: *Е. В. Финков*

Корректор: *А. В. Громова*

12+

ООО «Наука и Техника»
Лицензия №000350 от 23 декабря 1999 года.
192029, г. Санкт-Петербург, пр.Обуховской обороны, д. 107.
Подписано в печать 25.08.2021. Формат 70x100 1/16.
Бумага офсетная. Печать офсетная. Объем 31 п. л.
Тираж 1500. Заказ 8615.

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, Россия, г. Ульяновск, ул. Гончарова, 14

Unity

Полное руководство

2-е издание

+ виртуальный диск с примерами, проектами и ассетами

В этой книге мы расскажем, как с использованием Unity (популярной межплатформенной среды разработки компьютерных игр) вы сможете САМИ создавать свои игры и трехмерные миры, причем без лишних затрат и профессиональных навыков программирования.

Книга поделена на три части. Первая часть посвящена изучению интерфейса и основных возможностей Unity. Мы поговорим о двух- и трехмерных проектах; рассмотрим ключевые особенности Unity; узнаем, как использовать ассеты; подробном изучим интерфейс Unity; узнаем об игровых объектах, сценах, камерах, источниках света; создадим свои первые Unity-проекты.

Во второй части мы поговорим о других важных частях Unity – о графике; о физике; рассмотрим основы скриптинга (написания сценариев - скриптов); узнаем как работать со звуком; как настроить навигацию в игре; как использовать анимацию и многое другое.

Ну и в заключительной, третьей, части полученных знаний из первых двух частей нам хватит для создания полноценной игры. Рассмотрен весь цикл создания игры – от этапа планирования игрового мира до настройки игрового интерфейса. Все этапы сопровождаются примерами программных кодов и скриншотами. Если вы уже сейчас хотите создавать свои игры – то эта книга определенно для вас! Читайте и творите!

ISBN 978-5-94387-721-6

9 78- 5- 94387- 721- 6

Издательство "Наука и Техника"
г. Санкт-Петербург

Для заказа книг:
(812) 412-70-26
e-mail: nitmail@nit.com.ru
www.nit.com.ru

НиТ издательство

www.nit.com.ru