А. Адонин

Ren'Py

Создаем игры и приложения

- Основы языка Python
- Полный (с разбором кода) пошаговый цикл создания своей игры
- Практический кейс кроссплатформенной разработки
- Доступно любому уровню: от новичка до профессионала

НиТ издательство
nit.com.ru

Адонин А.

Ren'Py

Создаем игры и приложения

"Издательство Наука и Техника"

Санкт-Петербург

УДК 004.42
ББК 32.973

Адонин А.

Ren'Py. Создаем игры и приложения — СПб.: Издательство Наука и Техника, 2024. — 400 с., ил.

ISBN 978-5-907592-50-6

Игровой движок **Ren'Py** (РенПи) – это один из простейших способов познакомиться с основами программирования. Благодаря этой книге, шаг за шагом создавая свою первую игру, вы познакомитесь с основами разработки игр. Функционал движка настолько прост, что первую свою игру вы сможете запустить уже спустя несколько первых глав.

Несмотря на всю простоту, Ren'Py позволяет создавать игры во многих жанрах, таких, как: RPG, Симуляторы, Квесты, Стратегии, Новеллы и даже Шутеры. Также на базе движка можно разработать мобильное приложение или программу на ПК.

Ren'Py базируется на языке программирования Python, поэтому многие функции движка, структура, синтаксис и логика схожи с Python, все это будет рассмотрено в книге.

Подробные пошаговые примеры (с разбором кода) в данной книге помогут в создании игр и приложений любому уровню пользователей, от самых начинающих до профессионалов.

Читайте, программируйте и творите!

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

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

ISBN 978-5-907592-50-6

9 785907 592506

Контактные телефоны издательства:
(812) 412 70 26

Официальный сайт: www.nit.com.ru

© Адонин А.
© Издательство Наука и Техника

Содержание

Глава 1.

Знакомство с функционалом Ren'Py

1.1. Вступление

Игровой движок **Ren'Py** (РенПи) – это один из простейших способов погрузиться в сферу программирования. Постепенно, шаг за шагом, создавая свою первую игру в жанре визуальной новеллы, вы познакомитесь с основами разработки игр. Функционал движка настолько прост, что свою игру вы сможете запустить уже спустя несколько глав.

Однако, несмотря на всю простоту, Ren'Py позволяет создавать игры во многих жанрах. Это и RPG, и симуляторы, квесты, стратегии и даже 2D-шутеры. Кроме того, на базе движка можно разработать несложное мобильное приложение или программу на ПК. Но, конечно, для этого понадобится более глубокое знакомство с инструментарием. Этому отведена вторая часть книги.

Ren'Py базируется на языке программирования Python, поэтому многие функции движка, структура, синтаксис и логика схожи с этим ЯП (языком программирования). Но, не пугайтесь этих слов, все эти понятия будут вводиться постепенно. После изучения этой книги вы без труда сможете освоить Python. Более того, вы даже сможете считать себя начинающим программистом на этом языке.

1.2. Скачиваем и устанавливаем Ren'Py

Скачать и установить Ren'Py можно с официального сайта разработчика renpy.org. Всего существует две версии движка: Ren'Py 7.6, который осно-

вывается на Python второй версии, и Ren'Py 8, на базе Python третьей версии. В данном курсе будет разбираться последняя версия, но большинство примеров будут работать и для старой. Остальные скрипты потребуют незначительной доработки для внедрения их в предыдущую версию движка.

На главной странице сайта можно скачать актуальную сборку Ren'Py 8+. Нажав в верхнем меню кнопку **Download**, есть возможность скачать седьмую версию, а также дополнительный инструментарий, который на начальном этапе нам не понадобится.

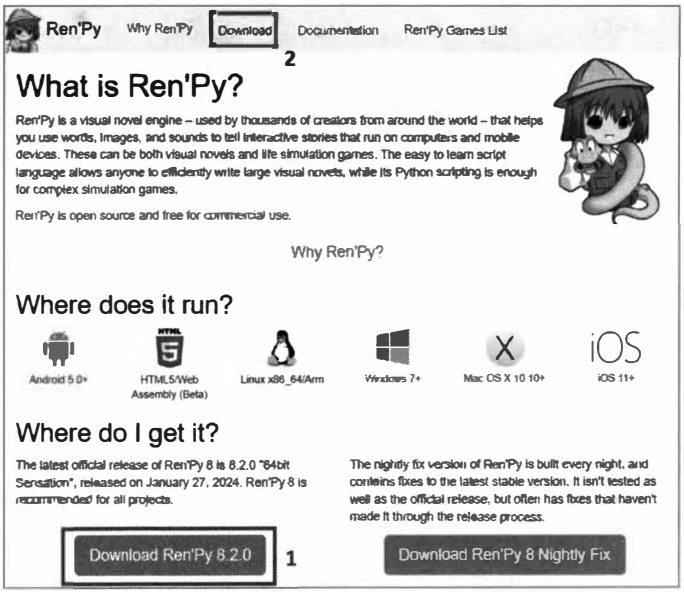


Рис. 1.1.

После скачивания установка происходит стандартным способом – двойным кликом по **exe**-файлу.

> **ВАЖНО**! При установке в пути не должно быть символов на кириллице. Это впоследствии может вызывать неочевидные ошибки скриптов. Оптимальным решением будет путь вроде этого: D:\RenPy.

1.3. Интерфейс движка

После установки и запуска Ren'Py открывается **лаунчер** (загрузчик). В первый раз он может быть на английском языке, поэтому меняем по необходимости на нужный. Жмём Preferences > General > Language.

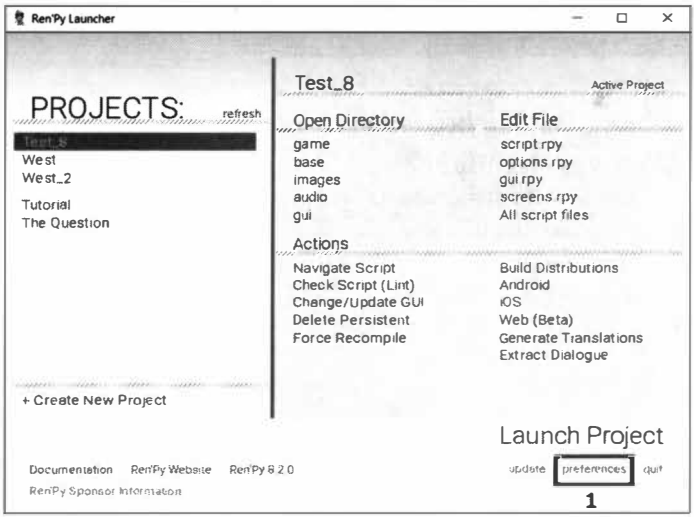


Рис. 1.2.

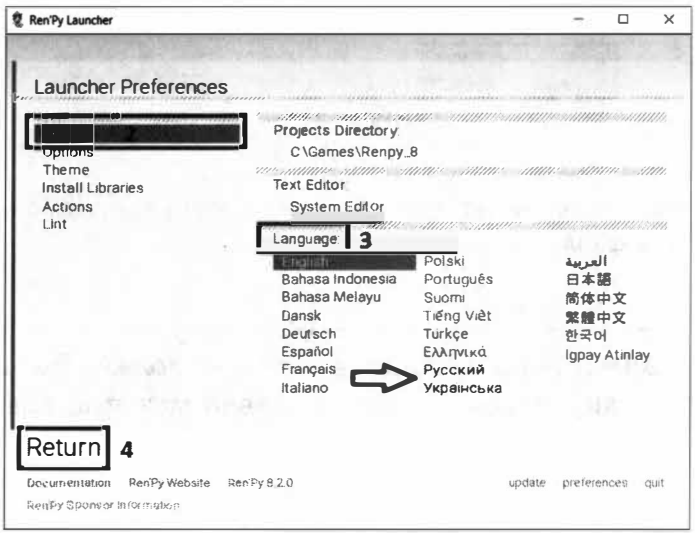


Рис. 1.3.

Далее возвращаемся на главный экран и начинаем знакомство с интерфейсом движка:

1. С левой стороны будет располагаться **список всех созданных вами проектов**.

2. Чуть ниже вы найдёте **встроенное обучение** и **демонстрационную новеллу "Вопрос"**.
Рекомендуется просмотреть обучение, так как оно даёт хорошее представление о том, какую игру можно сделать на Ren'Py.

3. Кнопка для создания нового проекта.

4. При клике на **документацию** мы попадаем на официальный сайт Ren'Py, в раздел с подробным описанием и возможностями движка. Документация написана на английском языке, но большинство браузеров позволяют переводить текст.

5. **Ссылка на главную страницу сайта Ren'Py**, где можно прочитать последние новости разработки, скачать дополнительные инструменты, оставить пожелания или баг-репорт, а также проверить актуальную версию игры.

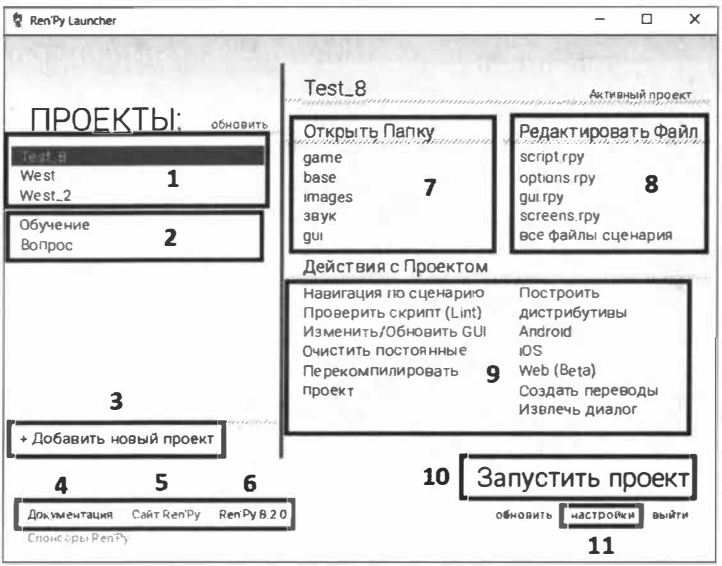


Рис. 1.4.

6. **Указатель текущей версии движка.** Если на официальном сайте опубликована новая версия, то по этой кнопке можно загрузить обновление.

7. В данном разделе располагаются **кнопки для открытия стандартных папок проекта**, выбранного в списке слева. Неудобство этого раздела заключается в том, что при разработке своей игры, как правило, количество папок растёт, и они не отображаются в этом списке. Поэтому проще управлять структурой файлов непосредственно из самого проекта. Подробнее об этом в следующей главе.

8. Раздел, аналогичный предыдущему, в котором отображаются **файлы скриптов текущего проекта**. Он также представляет только стартовые скрипты.

9. Раздел с **инструментами для создания переводов**, различных версий игры и прочего. На начальном этапе всё это нам не понадобится. Ознакомимся с ними в следующих главах, по мере необходимости.

.10. Запускает выбранный из списка **проект для тестирования**.

11. **Дополнительные настройки лаунчера**, на которых остановимся подробнее.

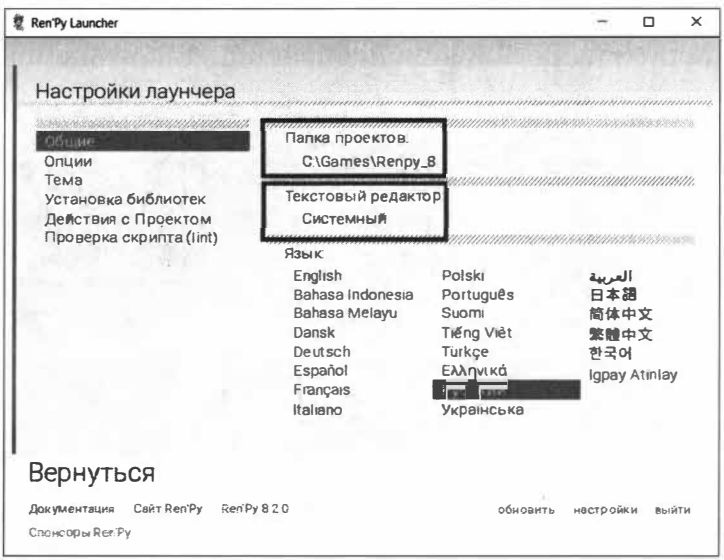


Рис. 1.5.

Раздел общих настроек нам уже знаком. Помимо выбора языка, здесь мы можем выбрать редактор для написания кода, а также установить директорию, где будут храниться наши проекты.

Разделы "Опции" и "Действия с проектом" интуитивно понятны. Они устанавливают или сбрасывают соответствующие настройки по умолчанию.

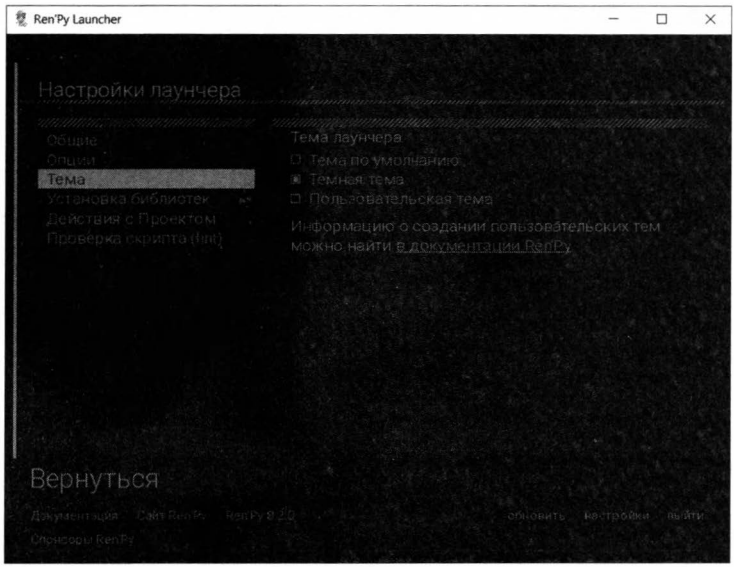


Рис. 1.6.

Раздел "Тема" позволяет изменить внешний вид лаунчера. Помимо тёмной версии, можно настроить свою собственную, изменив практически любой аспект. Для этого нужно зайти в папку, куда установлен Ren'Py, далее в папку launcher и skin. Здесь нас ждёт файл *skin.rpy*, который отвечает за внешний вид лаунчера Ren'Py.

Просмотреть его можно любым редактором кода. В открытом файле достаточно заменить любой код цвета на свой:

```
init -2 python:
    # Цвет неинтерактивного текста.
    custom_text = "#545454"
    # Цвета кнопок в различных состояниях.
    custom_idle = "#42637b"
    custom_hover = "#d86b45"
    custom_disable = "#808080"
```

Таким же образом здесь можно заменить и графические элементы. Например, открыть изображение по прописанному пути, и заменить своим с таким же названием. Или поместить в папку **image** новое изображение, а в коде прописать его название:

```
# Изображение, используемое в качестве разделительного шаблона.
custom_pattern = "images/pattern.png"
# Отображаемый объект, используемый в качестве общего фона.
custom_background = "skin/skin_background.jpg"
```

Раздел "Установка библиотек" позволяет подключить сторонние библиотеки, распространяющиеся по собственным лицензиям. Например, Steam или Live2D Cubism для анимации спрайтов.

Раздел "Проверки скриптов", как понятно из описания, проверяет весь код, и выдаёт список ошибок, при написании скриптов. Благодаря этому можно сразу исправить недочёты, без необходимости запускать проект, и проверять его работоспособность.

1.4. Структура папок и файлов

В предыдущей главе упоминалось о том, что не все файлы отображаются на главном экране лаунчера. Удобнее управлять ими непосредственно из той директории, где хранится проект. Делается это также просто, как и при создании любой папки на компьютере.

При создании новой игры в директории, где хранятся проекты, создаётся новая папка с названием вашего проекта.

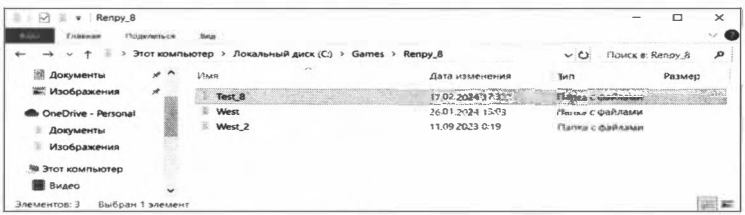


Рис. 1.7.

Внутри каждого из них автоматически генерируется папка **game**, в которой создаются дефолтные файлы. Именно они отображаются на главном экране лаунчера.

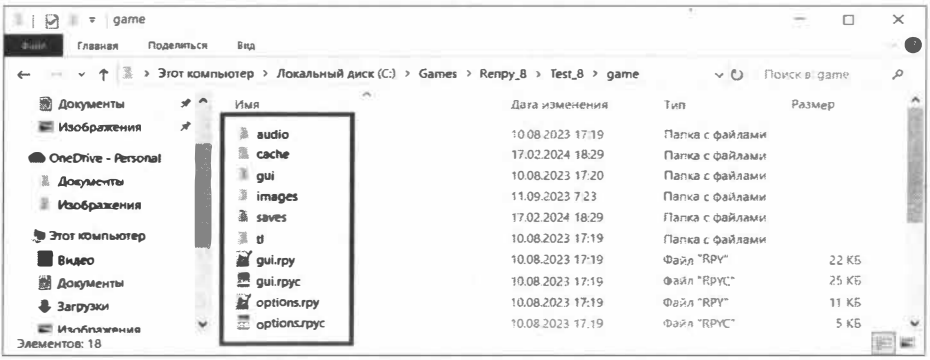


Рис. 1.8.

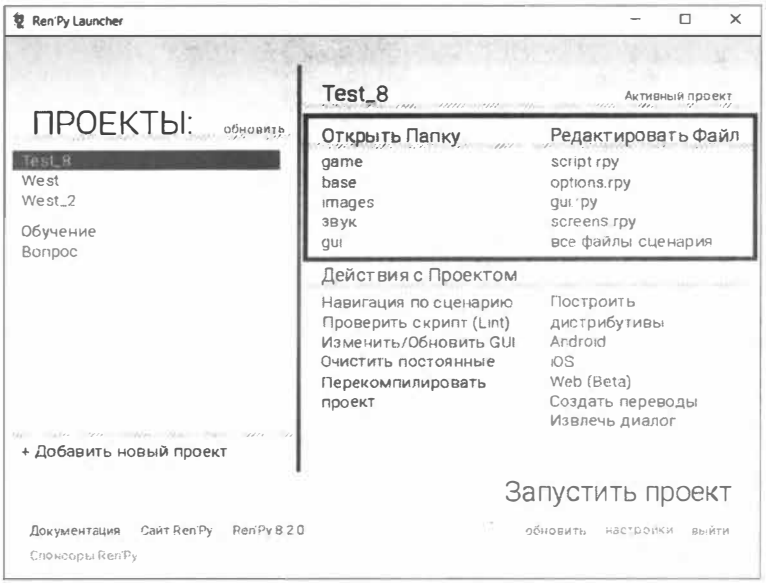


Рис. 1.9.

Помимо стандартных файлов и папок мы можем создать дополнительно любое их количество, для удобства и структурирования проекта. Например, создать папку **video**, куда помещать видеофайлы. Или создать новый файл скрипта **.rpy**, в котором прописать отдельные функции и сценарии.

Также не забываем, что все файлы должны именоваться латинскими буквами и цифрами, чтобы не допускать кириллицы в пути к файлам.

1.5. Установка и настройка редактора кода

Редактор кода – это основной наш инструмент, с помощью которого мы будем создавать игру.

В настоящее время существует множество различных редакторов. Однако большинство из них имеют схожий функционал, поэтому выбор стоит делать исходя из собственных предпочтений.

На сегодняшний день одними из самых популярных редакторов кода являются следующие: Visual Studio Code, Atom, Sublime Text, IntelliJ IDEA, Eclipse, Notepad++.

Все они бесплатны, а их внешний вид, для удобства, можно настроить под себя. В качестве примера будет представлен Notepad++, но в большинстве редакторов все настройки изменяются аналогичным образом.

Скачать любой из редакторов можно с официальных сайтов. Устанавливаются они также просто – двойным кликом по **exe**-дистрибутиву.
После установки запустите редактор. Если интерфейс на английском языке, выберите вкладку Settings > Preferences.

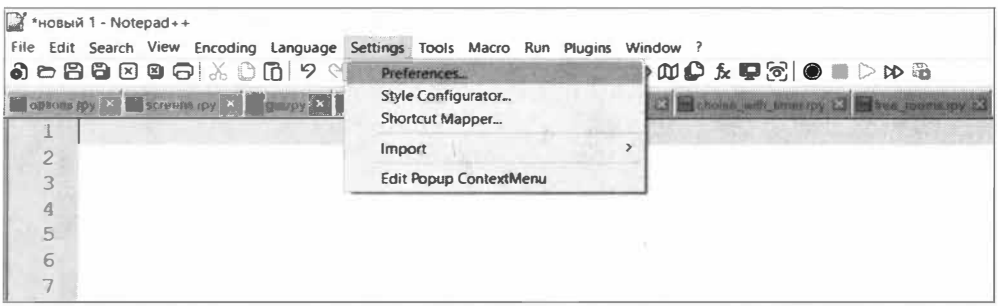


Рис. 1.10.

В открывшемся окне, в первой вкладке можно изменить язык на русский.

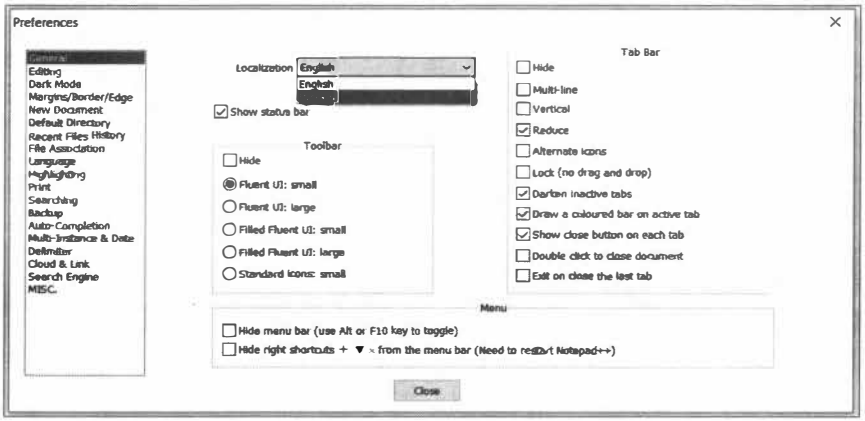


Рис. 1.11.

Здесь же, в разделе **"Новый документ"**, можно установить синтаксис по умолчанию. Как упоминалось ранее, синтаксис Ren'Py основан и во многом похож на язык программирования Python, поэтому с такой настройкой работать будет комфортнее. Подсветка синтаксиса (разных элементов кода) будет соответствующей.

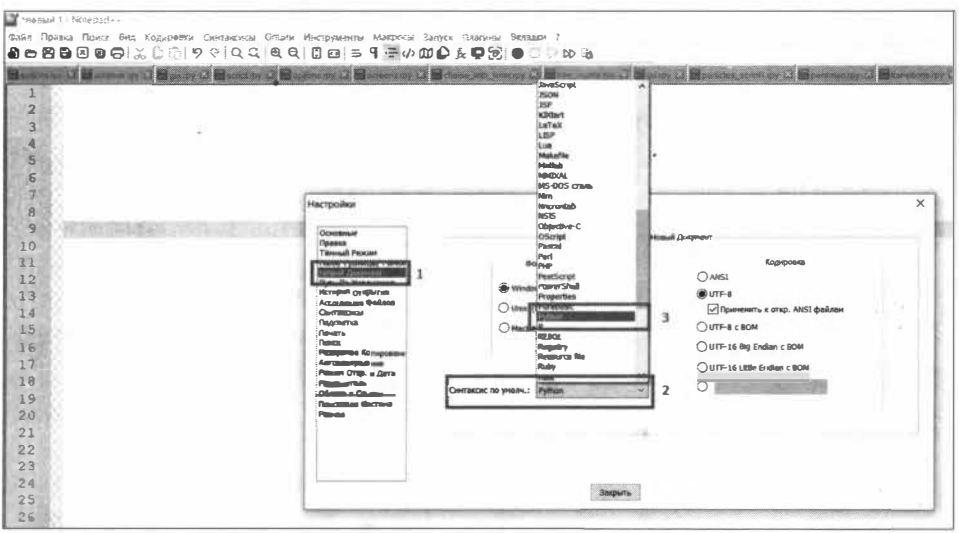


Рис. 1.12.

В разделе **"Синтаксисы"** убедитесь в правильной настройке табуляции. Она должна быть равна четырём пробелам. Если установлено другое значение, кликните по цифре и в открывшемся окне поставьте 4.

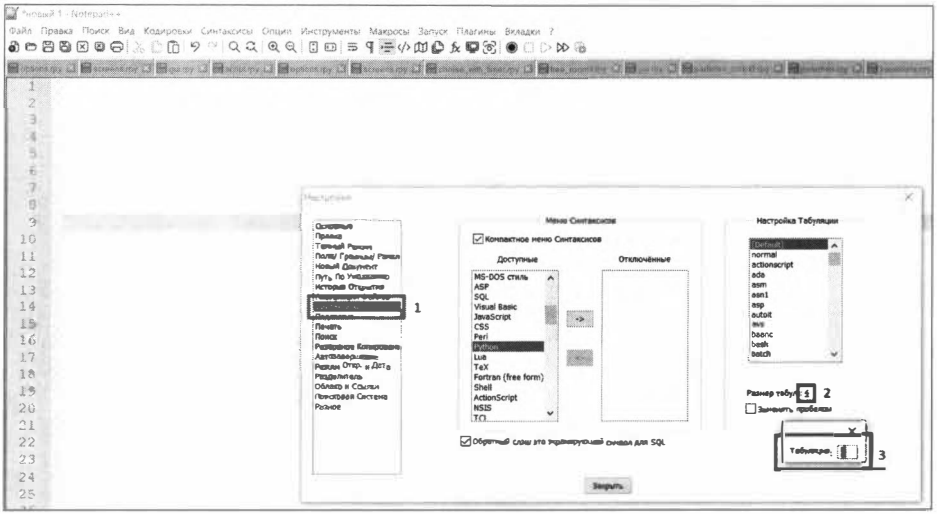


Рис. 1.13.

В некоторых версиях Notepad++ эта настройка может находиться в отдельной вкладке, которая так и называется "Настройка Табуляции".

В разделе **"Определение стилей"** можно настроить внешний вид редактора. Например, выбрать тёмный цвет фона, цвет шрифтов и пр.

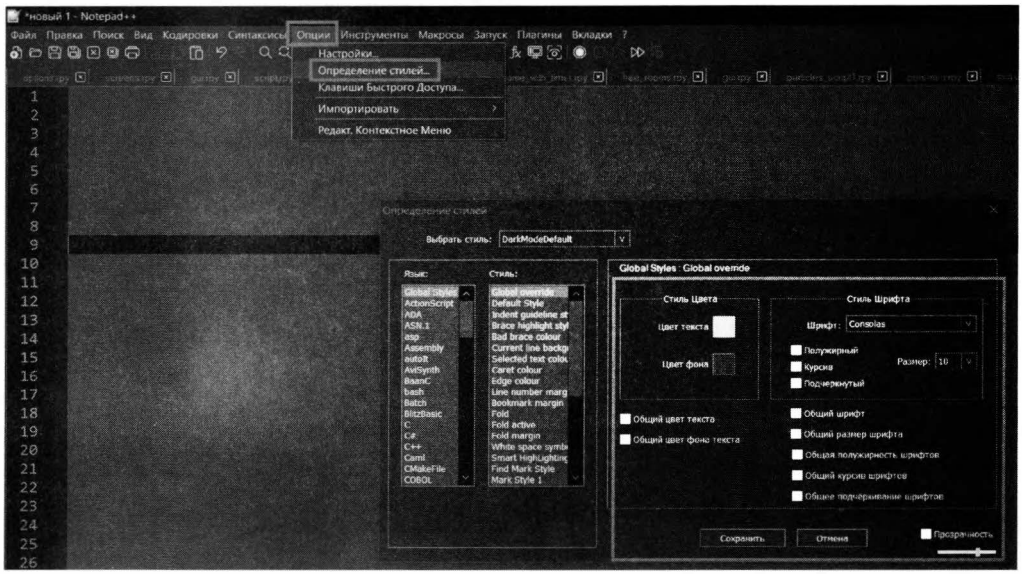


Рис. 1.14.

Это основные настройки, которые облегчат работу с кодом. Со всем остальным функционалом Notepad++ можно ознакомиться в официальной документации.

1.6. Основы синтаксиса в Ren'Py

Изначально Ren'Py разрабатывался для создания визуальных новелл. Его основные функции – это отображать изображения, спрайты и текст на экране. Для этого нужно дать движку определённые команды на выполнение. Например, покажи фон, покажи спрайт, напиши текст.

Рис. 1.15.

В коде это будет выглядеть следующим образом:

```
label start:
    scene background1
    show sylvie
    "Привет"
```

Команда *scene* вызывает изображение с названием background1, которое является фоновой картинкой. Этой же командой можно вызвать следующее изображение, которое заменит предыдущее, а также скроет все спрайты и текст на экране. То есть, создаст новую сцену.

Команда *show* также отвечает за вывод изображений на экран. Однако эти изображения выводятся на новом слое, поверх фонового изображения.

В нашем примере мы вызвали с помощью **show** изображение персонажа *sylvie*.

Необходимо учитывать регистр названий в коде и в названиях изображений. Они должны совпадать и не должны начинаться с цифр. Также, несмотря на то, что в названиях изображений можно использовать кириллицу, лучше избегать этого, и присваивать названия латинскими буквами и цифрами. Это связано с локализацией вашей игры. При создании версий на других языках, иностранные игроки могут столкнуться с ошибками из-за символов на кириллице.

Чтобы вывести текст на экран, достаточно поместить его в кавычки. Они могут быть как двойные "", так и одинарные ' '. Их действие равнозначно.

Аналогично языку Python, код в Ren'Py выполняется построчно, сверху вниз. Это значит, что команды нужно прописывать в правильном порядке. То есть, сначала мы объявляем фон, затем спрайт, и текст. Если спрайт персонажа вывести на экран перед фоном, то фон наложится поверх спрайта.

Одним из основных операторов в Ren'Py является **label** (метка). Он позволяет перемещаться к определённым местам в коде, с помощью команды **jump**.

Для простоты понимая его можно сравнить со ссылкой на новую страницу сайта, на которую мы попадём, "прыгнув" к метке.

В коде это выглядит следующим образом:

```
label start:
    scene background1
    show sylvie
    "Привет"
    jump part_1

label part_1:
    scene background2
    show lucy
    "Новая часть истории"
    "Разработка игр ещё никогда не была такой лёгкой"
    jump part_2
```

После того как мы перейдём дальше от реплики "Привет", команда **jump** отправит нас по адресу **part_1**. Он представляет собой новый блок, включающий в себя новый набор команд. Например, в нём мы показали новую сцену, которая автоматически очистила экран от предыдущих изображений и текста. Также мы отобразили новый спрайт с новым персонажем *lucy*, и новые диалоги.

Во время создания своей игры таких меток и блоков может быть любое количество. Это позволяет структурировать код и разбивать его на небольшие фрагменты, с которыми удобно работать. Кроме того, разные блоки можно помещать в разные файлы скриптов, чтобы не загромождать ими один файл.
Независимо от того, в каком файле находится следующая метка, Ren'Py самостоятельно найдёт к ней путь.

Следующий момент, на который нужно обратить внимание, это **отступы**. Они делаются клавишей **Tab** на клавиатуре.

> **ПРИМЕЧАНИЕ**. Стандартный отступ в Ren'Py равен **четырём пробелам**, это общепринятый промежуток.

С другими размерами отступа код тоже будет работать, но рекомендуется придерживаться стандарта, так как бывают исключения.

Один отступ – один шаг вправо делается в нескольких случаях. Сейчас мы не будем углубляться в этот аспект, а просто запомним, что после каждого двоеточия (:), всё, что относится к этому блоку, должно быть смещено на один шаг вправо.

В нашем примере каждый блок метки включает в себя набор команд после двоеточия. Все они имеют отступ. Это похоже на список, где после двоеточия мы пишем набор последовательных команд. Такая организация кода не только улучшает его читаемость, но и позволяет правильно структурировать весь скрипт игры.

1.7. Создаём свою первую новеллу за 10 минут

Теперь, когда мы познакомились с базовыми возможностями движка, мы можем создать свою первую небольшую новеллу. На это нам вполне хватит десяти минут. Итак, откройте лаунчер Ren'Py, и нажмите "+ Добавить новый проект".

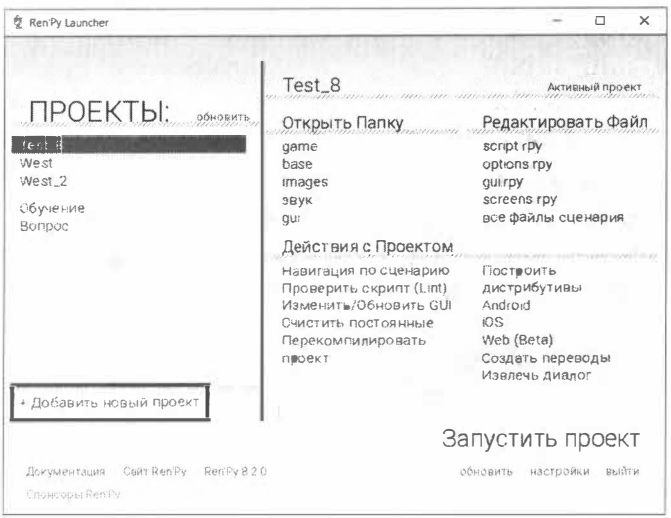


Рис. 1.16.

Следующее окно сообщит нам, на каком языке будет создан этот проект. Это значит, что при создании стартовых файлов игры, все подсказки и комментарии в скриптах, а также интерфейс будут на этом языке. Вместе с тем, сам язык будет установлен как оригинальный при создании переводов на другие языки.

Если нужно изменить язык игры, нажмите кнопку "Отмена", и в настройках лаунчера выберите другой. Как это сделать, подробно описывалось в пункте 1.3. "Интерфейс движка". После выбора необходимого языка жмём "Продолжить", и переходим к следующему этапу.

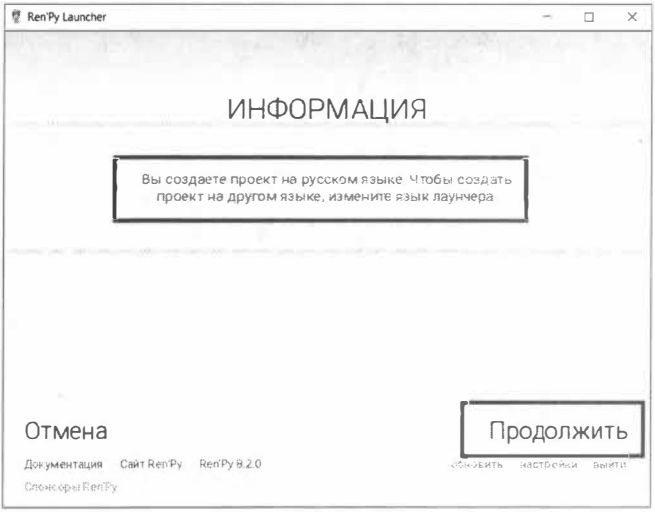


Рис. 1.17.

Здесь нам предложат дать имя своему проекту. Название игры должно быть написано латинскими буквами. Кириллицу ввести не получится.

Рис. 1.18.

В новом окне необходимо выбрать разрешение проекта. Несмотря на то, что Ren'Py автоматически адаптирует разрешение к экрану пользователя, нужно помнить, что при увеличении изображений их качество ухудшается. То есть, если ваша игра имеет разрешение 1920x1080, а игрок запустит её на мониторе 3840x2160, все изображения и спрайты будут пропорционально растянуты относительно экрана. Это, соответственно, скажется на их качестве.

Если произойдёт обратная ситуация, и пользователь откроет проект на планшете или телефоне, разрешение аналогичным образом будет подстроено под текущий экран. Однако при уменьшении изображений их качество не ухудшается.

Кроме того, при выборе оптимального формата, нужно учитывать, что чем больше изображения, тем больше их вес. Это может сказаться на общем объёме игры, а также на производительности, особенно на слабых ПК.

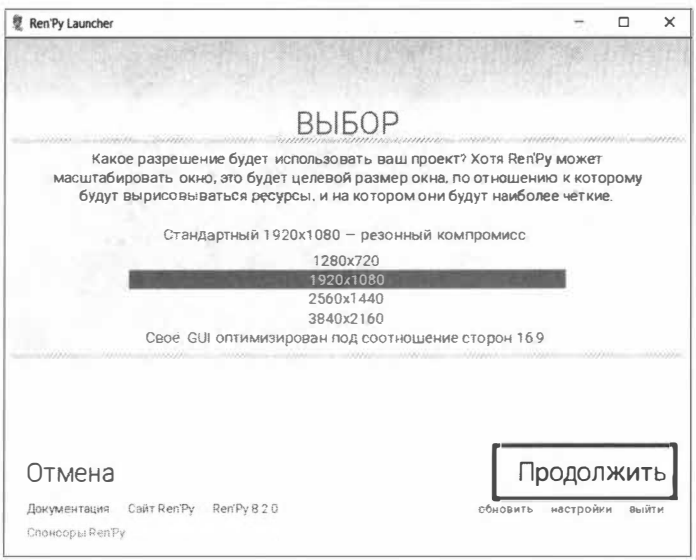


Рис. 1.19.

Следующий экран предлагает выбрать цветовую схему интерфейса. От этого будут зависеть цвета кнопок, надписей и других элементов. На данном этапе можно выбрать любой относительно подходящий **пресет** (набор предварительных настроек), так как в процессе разработки весь интерфейс можно изменить аналогично тому, как это было описано в разделе 1.3. "Интерфейс движка".

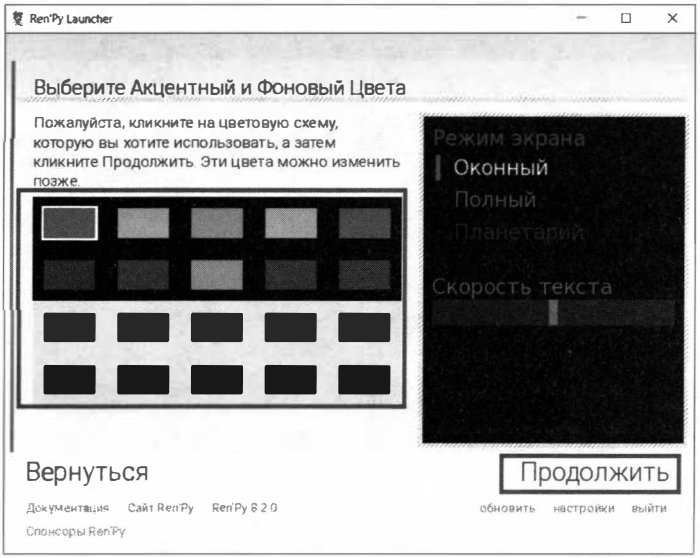


Рис. 1.20.

Далее нам остаётся немного подождать, пока Ren'Py создаст новый проект с предустановленными настройками.

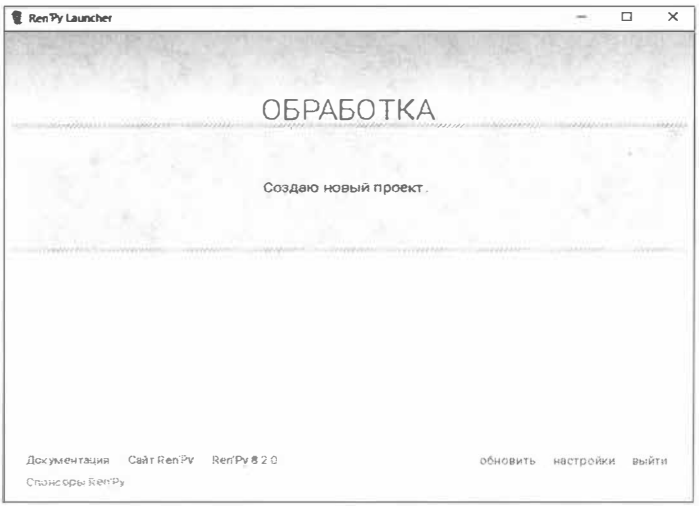


Рис. 1.21.

Спустя несколько мгновений наша новая игра появится в списке созданных проектов. Её уже можно запустить, и посмотреть, что у нас получилось.

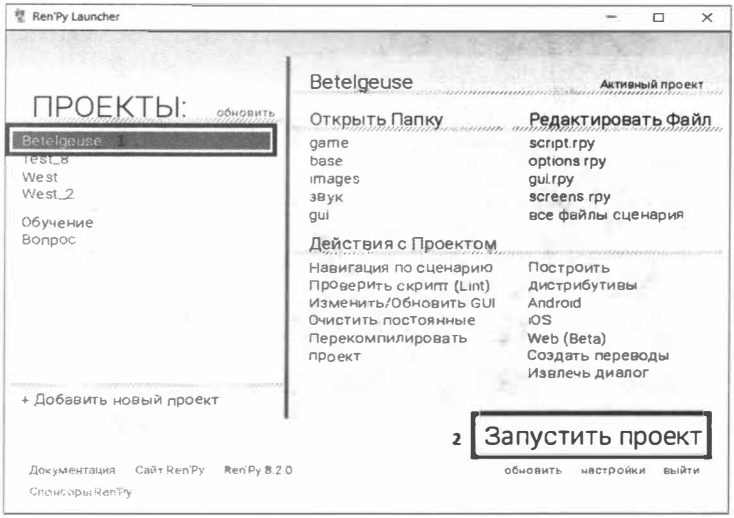


Рис. 1.22.

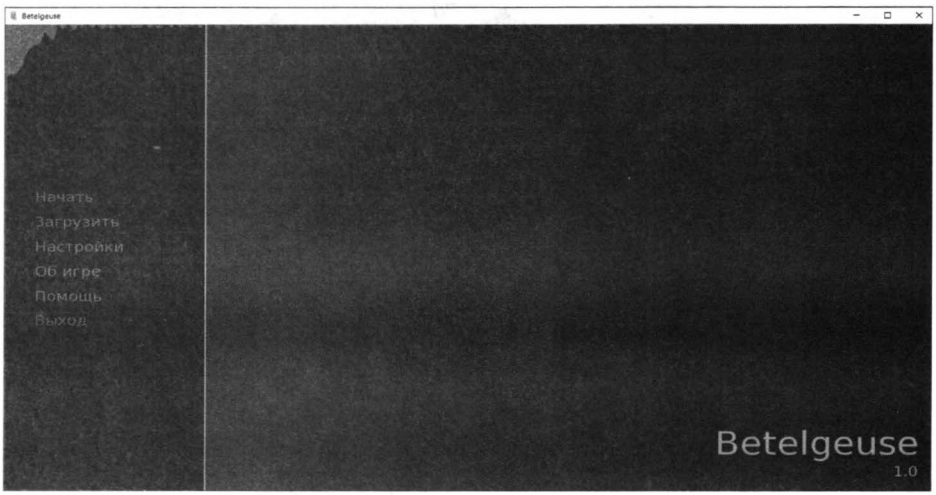


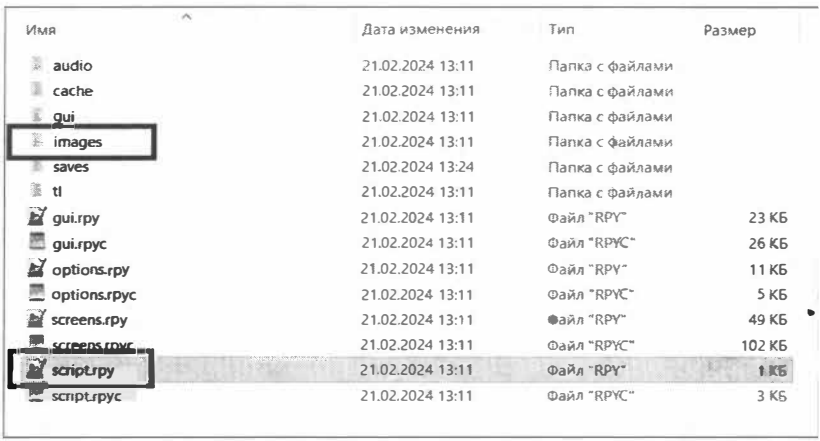
Рис. 1.23.

По умолчанию в скриптах прописывается базовое приветствие, которое предлагает наполнить нашу игру изображениями, музыкой и текстом. Так мы и поступим.

Чтобы долго не искать фоны и спрайты для обучения, можно позаимствовать их из демонстрационной новеллы "Вопрос". Ассеты из этой игры лежат по следующему пути: папка, где установлен Ren'Py \ the_question \ game \ images.

Скопируйте все изображения из данной папки и вставьте в аналогичную папку своего проекта: папка с вашим проектом \ game \ images.

После этого в папке **game**, с помощью редактора кода, откройте файл **script.rpy**.



Имя	Дата изменения	Тип	Размер
audio	21.02.2024 13:11	Папка с файлами	
cache	21.02.2024 13:11	Папка с файлами	
gui	21.02.2024 13:11	Папка с файлами	
images	21.02.2024 13:11	Папка с файлами	
saves	21.02.2024 13:24	Папка с файлами	
tl	21.02.2024 13:11	Папка с файлами	
gui.rpy	21.02.2024 13:11	Файл "RPY"	23 КБ
gui.rpyc	21.02.2024 13:11	Файл "RPYC"	26 КБ
options.rpy	21.02.2024 13:11	Файл "RPY"	11 КБ
options.rpyc	21.02.2024 13:11	Файл "RPYC"	5 КБ
screens.rpy	21.02.2024 13:11	Файл "RPY"	49 КБ
screens.rpyc	21.02.2024 13:11	Файл "RPYC"	102 КБ
script.rpy	21.02.2024 13:11	Файл "RPY"	1 КБ
script.rpyc	21.02.2024 13:11	Файл "RPYC"	3 КБ

Рис. 1.24.

Здесь мы сможем увидеть то самое приветствие при запуске нашего проекта. Его можно смело удалять, и прописывать вместо него сценарий нашей игры.

```
# Определение персонажей игры.
define e = Character('Эйлин', color="#c8ffc8")
# Игра начинается здесь:
label start:
    scene bg room
    show eileen happy
    e "Вы создали новую игру Ren'Py."
    e "Добавьте сюжет, изображения и музыку и отправьте её в мир!"
    return
```

Но для начала давайте познакомимся с новыми элементами в коде, о которых мы ещё не знаем.

Первый из них – это символ решётки **#** с описанием после него.

Символ # позволяет писать комментарии к коду, благодаря чему вы всегда можете оставить пояснение или напоминание для других разработчиков, которые могут работать вместе с вами.

Также это будет полезно вам самим, когда спустя долгое время вы вернётесь к своему скрипту и сможете быстро вспомнить, за что отвечает та или иная часть кода.

Вы можете писать любое количество комментариев. При воспроизведении проекта движок их не учитывает. Однако стоит иметь в виду, что каждая строка комментариев должна начинаться с решётки **#**. Для многострочного комментария можно воспользоваться **тройными одинарными кавычками** ''' '''.

Следующая команда **define**, она используется для определения новой переменной или константы.

Переменные могут быть использованы для хранения значений, таких как строки текста, числа или булевы значения, а константы – для задания постоянных значений, которые не могут быть изменены в процессе выполнения игры.

Это помогает в организации и структурировании кода.

В нашем случае создаётся персонаж с именем **e**, который содержит в себе функцию Character() с определёнными параметрами. Подобным образом создаются различные объекты в Ren'Py.

Теперь давайте удалим весь код из нашего скрипта, и самостоятельно создадим своего собственного персонажа:

```
define lia = Character('Лиара', color="#c8ffc8")
```

Мы создали переменную с именем *lia*, которая включает в себя функцию персонажа *Character* с параметрами имени и цветом имени. Помимо этих параметров, персонажу можно присвоить множество других, но об этом мы поговорим позже.

Чтобы Лиаре не было скучно, давайте создадим ещё одного героя, с которым она сможет поговорить.

Напишите в следующей строке:

```
define kay = Character('Кайден', color="#ccffff")
```

Исходя из примера, мы теперь знаем, как создавать персонажей, давать им имена и присваивать свой цвет для имени. Имейте в виду, что каждый персонаж или любой другой объект должен иметь своё уникальное имя – *lia*, *kay*. Они не должны повторяться, но могут быть любой длины и включать в себя цифры. Однако для удобства их обычно делают не слишком длинными.

Сейчас имя Лиары в игре будет окрашиваться зелёным оттенком, а Кайдена – голубым. Каждому персонажу можно присвоить свой цвет "#ccffff". Коды всех цветов можно найти в интернете, например по запросу "таблица цветов hex". В результате поиск выдаст подобные таблички:

Рис. 1.25.

Итак, после того, как мы создали персонажей, можно начинать писать скрипт самой игры. Отступите пару строк вниз и создайте стартовую метку **label start.** Этот лейбл всегда должен присутствовать в игре, так как при нажатии кнопки "Начать" в главном меню, Ren'Py обращается именно к ней.

Далее, согласно синтаксису, сделайте отступ вправо клавишей **Tab**, и напишите список из несколько команд, каждую с новой строки. Выведите на экран фоновое изображение, выведите спрайт первого персонажа, и пропишите его реплику.

```
# Определение персонажей игры.
define lia = Character('Лиара', color="#c8ffc8")
define kay = Character('Кайден', color="#ccffff")
# Игра начинается здесь:
label start:
    scene bridge # объявили фоновое изображение
    show liara # показали персонажа Лиару
    "Ого! Что это?!" # вывели реплику
    return
```

В игре это будет выглядеть следующим образом:

Рис. 1.26.

Первый недочёт, который мы сможем с вами заметить, это то, как резко отображаются фон и спрайт персонажа. Чтобы изображения появлялись более плавно, в Ren'Py используются специальные эффекты переходов. Их довольно много, но сейчас нам понадобятся два самых основных:

1. **fade** – эффект отображения из затемнения
2. **dissolve** – эффект плавного всплытия или растворения

> Чтобы применить эти эффекты к изображениям, используется оператор **with**.

Примеры:

- **scene bridge with fade #** изображение появляется из затемнения
- **show liara with dissolve #** спрайт отображается плавно

Второй недочёт заключается в том, что реплика персонажа не помечена его именем. Для этого перед текстом нужно просто указать имя говорящего персонажа:

```
lia "Ого! Что это?!"
```

В игре реплика будет помечена именем Лиары:

Рис. 1.27.

Исправленный код будет выглядеть так:

```
label start:
    scene bridge with fade # изображение появляется из затемнения
    show liara with dissolve # спрайт отображается плавно
    lia "Ого! Что это?!" # эту фразу говорит Лиара
    return
```

Теперь давайте введём в сюжет второго персонажа, чтобы он смог поговорить с Лиарой. Но сначала нам нужно познакомиться ещё с несколькими элементами. Если мы сейчас выведем спрайт второго персонажа командой **show kayden**, он отобразится по центру экрана, и наложится поверх предыдущего спрайта. Поэтому каждому персонажу нужно указать позицию, где он должен находиться.

В следующих главах мы научимся создавать множества разных позиций, а сейчас мы воспользуемся стандартными: **left**, **center**, **right**. Позиции указываются с помощью оператора **at**. Например: **show liara at left**.

Командой **hide** мы можем скрывать изображения и текст с экрана. Это может понадобиться, когда необходимо убрать спрайт или диалоговое окно. Например: *hide liara*, *window hide*. Чтобы изменения, вносимые в код, вступили в силу, нажмите комбинацию **Shift+R**, это перезагрузит Ren'Py и обновит скрипты.

Итак, давайте поместим спрайт Лиары в левой части экрана, а Кайдена справа, и продолжим нашу игру новой репликой.

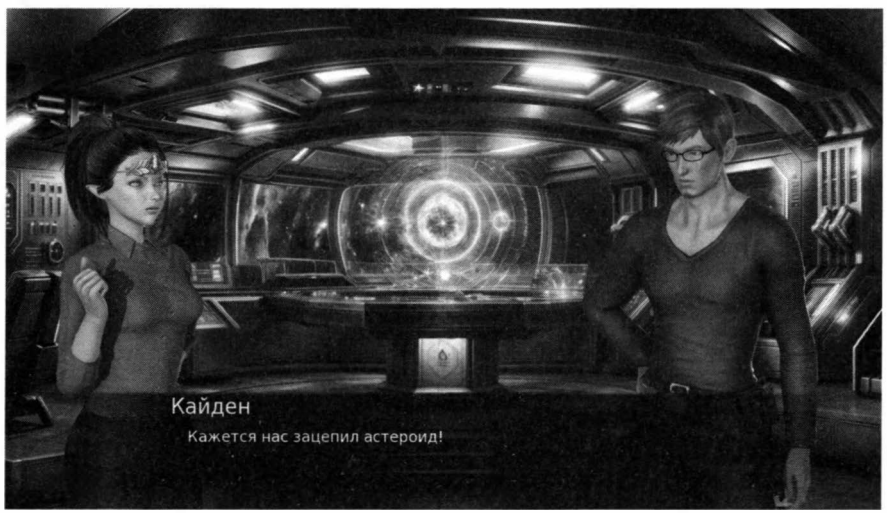


Рис. 1.28.

В коде это выглядит так:

```
label start:
    scene bridge with fade
    show liara at left with dissolve # отобразили слева
    lia "Ого! Что это?!"
    show kayden at right with dissolve # отобразили справа
    kay "Кажется, нас зацепил астероид!"
    return
```

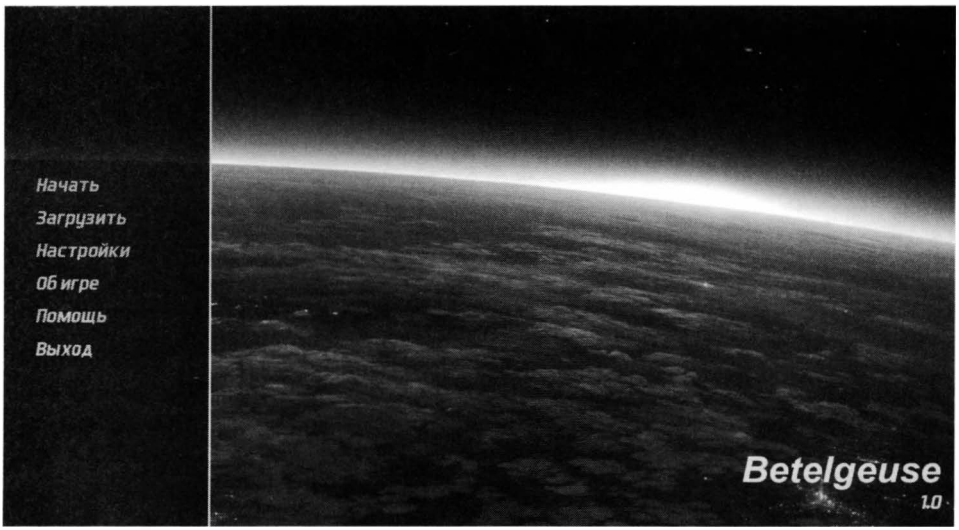
Для полноценного продолжения нашей истории, нам осталось научиться менять сцены. Ранее этот момент уже освещался. Достаточно просто вызвать новую сцену командой **scene**. Это автоматически скроет все спрайты на экране и заменит фон. Давайте ради тренировки сначала скроем персонажей вручную, а потом объявим новую сцену.

```
    kay "Кажется, нас зацепил астероид!"
    lia "Нужно проверить машинное отделение!"
    hide liara with dissolve # плавно скрыли спрайт Лиары
    hide kayden with dissolve # и Кайдена
    scene engine with dissolve # отобразили новый фон с эффектом
    pause # приостановили игру
    return # так как следующая команда вернёт в главное меню
```

Таким образом вы можете продолжать диалог персонажей, просто указывая говорящего, и его реплику. По необходимости меняя сцены, а также скрывая спрайты одних героев, и выводя на экран других.

Глава 2.

ДИЗАЙН ВИЗУАЛЬНОЙ НОВЕЛЛЫ



2.1. Загрузочный экран

После изучения предыдущей главы мы уже способны разработать полноценную визуальную новеллу. Однако для придания ей оригинальной стилистики необходимо поработать над дизайном. От того, как выглядит интерфейс и внешний вид вашей игры, зависит, какое первое впечатление о ней сложится у игрока.

Первый элемент – **это загрузочный экран**. Как правило, игры на Ren'Py запускаются довольно быстро, и в нём нет необходимости. Обычно его добавляют в большие проекты, которым требуется некоторое время на инициализацию. По умолчанию в движке уже присутствует нужная функция, и нам остаётся только добавить изображение.

Подготовьте подходящую картинку, назовите её **presplash** и сохраните в формате **.jpg** или **.png**, а затем положите её в папку **game** вашего проекта. Перед запуском игры она отобразится на рабочем столе.

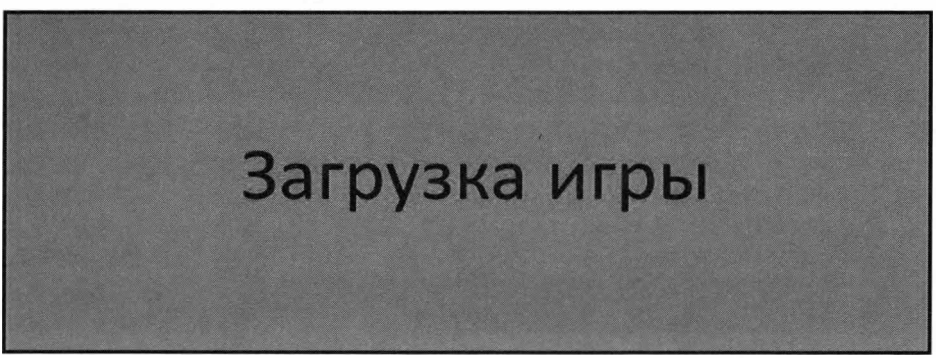


Рис. 2.1.

Этой заставке можно указать время отображения на случай, если у пользователя мощный компьютер, и игра загрузится моменталь-

но, не успев отобразить загрузочный экран. Откройте файл **options.rpy** и где-то вначале скрипта пропишите следующую команду:

```
define config.minimum_presplash_time = 5.0
```

5.0 – это значение в секундах, сколько картинка будет висеть на экране.

Второй способ реализации загрузочного экрана позволяет добавить полоску загрузки, которая будет отображать прогресс инициализации. Для этого подготовьте второе изображение с баром загрузки на прозрачном фоне и таким же разрешением, как у предыдущего изображения. Например, такое:

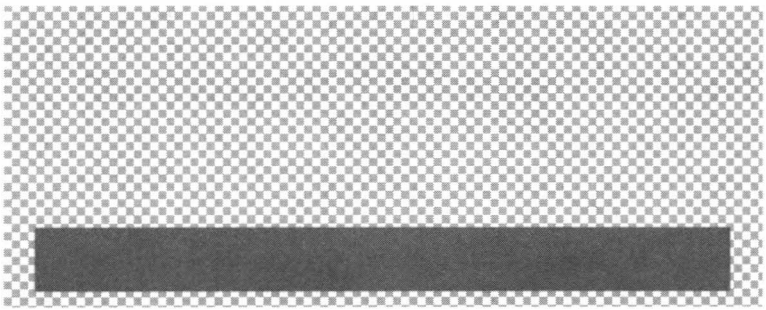


Рис. 2.2.

Переименуйте первое изображение на **presplash_background**, а второе на **presplash_foreground**, и поместите обе картинки в папку **game** вашего проекта.

Таким образом, при загрузке будет виден прогресс:

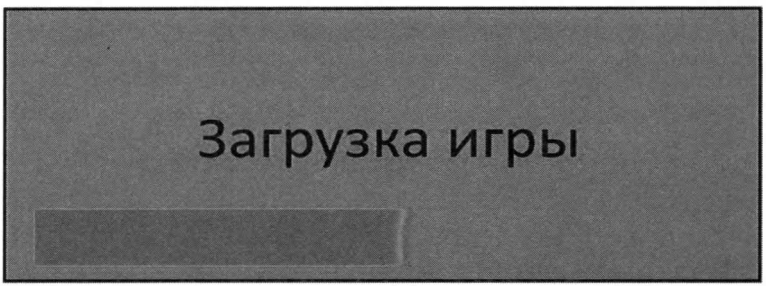


Рис. 2.3.

2.2. Вступительный логотип

Практически в каждой игре, при её загрузке, вначале отображается логотип команды разработчиков или вступительный трей-

лер. В этом разделе мы научимся делать подобное в своём проекте.

Рис. 2.4.

Для этого в Ren'Py также есть предустановленная функция, поэтому нам остаётся только создать изображение и соответствующую метку, которую движок отобразит на старте, перед главным меню. Картинку с логотипом положите в папку **images**, а затем откройте файл **script.rpy**, и напишите там следующий код:

```
label splashscreen:
    scene black with fade # показали чёрный экран
    pause 1 # сделали паузу на одну секунду
    scene logo with fade # показали логотип
    pause 3 # сделали ещё паузу
    scene black with fade # снова затемнение
    return # перешли в главное меню
```

Если в игре присутствует **label splashscreen,** движок автоматически воспроизведёт код из него перед загрузкой меню. Вместо примера с одной картинкой вы можете разместить в этой метке серию изображений аналогично тому, как прописываются изображения и спрайты в самой игре. Подобным образом можно сделать слайд-шоу или вставить видеоролик. Подробнее об этом будет рассказано в разделе, посвящённому анимации.

После проигрывания вступительного логотипа можно заметить, что экран главного меню появляется очень резко. Поэтому последним штрихом станет небольшая настройка, которая сделает отображение меню более плавным.

Откройте файл **options.rpy** и пропишите следующее:

```
define config.end_splash_transition = fade
```

2.3. Иконка игры

Иконка игры – это небольшое изображение, которое будет присвоено exe-файлу вашего проекта и отобразится на панели задач при его запуске.

По умолчанию в системе установлен стандартный логотип Ren'Py.

Рис. 2.5.

Вместо него создайте свою иконку в формате **png**. Можно использовать программы для графики, такие как Photoshop, GIMP или онлайн-редакторы. Затем назовите картинку **window_icon** и положите её по пути: *ваш_проект / game / gui*. Там вы сможете увидеть стандартный логотип с таким же названием. Замените его своим.

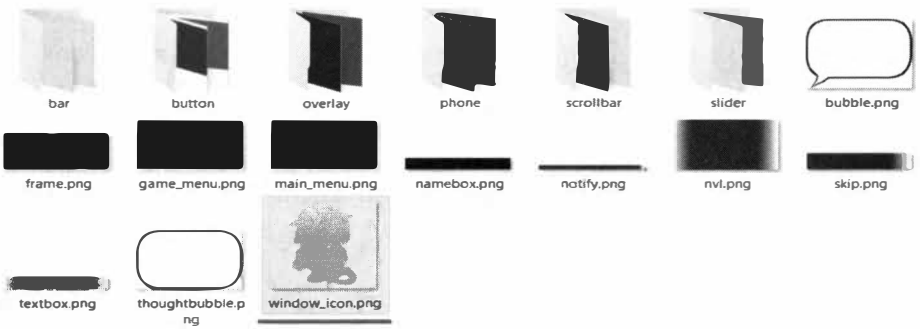


Рис. 2.6.

2.4. Простое оформление "Главного меню"

При запуске проекта нас встречает однотонный экран главного меню, цвет которого генерируется в зависимости от выбора стартового пресета. Настало время оформить его в стилистике игры. Откройте папку **gui**, в которую мы помещали иконку. Теперь нас интересуют файлы **main_menu.png** и **game_menu.png**.

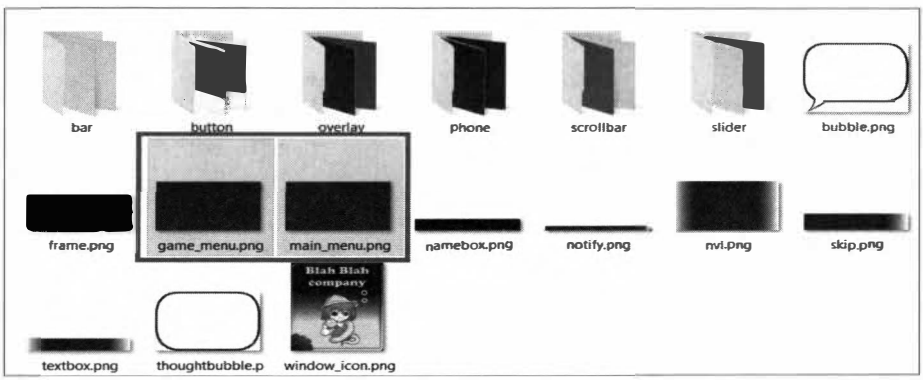


Рис. 2.7.

По аналогии с иконкой эти два изображения также необходимо заменить на свои, присвоив им те же названия и сохранив с тем же расширением – **png**.

> Файл **main_menu** отвечает за изображение в главном меню, а **game_menu** является фоном в игровом меню.

Его можно увидеть во время прохождения, если нажать правую кнопку мыши или **Esc**.

В игровом меню также имеется дополнительный полупрозрачный фон, который накладывается на наше изображение и создаёт эффект затемнения. Это изображение можно сделать полностью прозрачным в графическом редакторе или изменить его оттенок.

Находится это изображение в папке game / gui / overlay, и имеет такое же название – *game_menu.png*

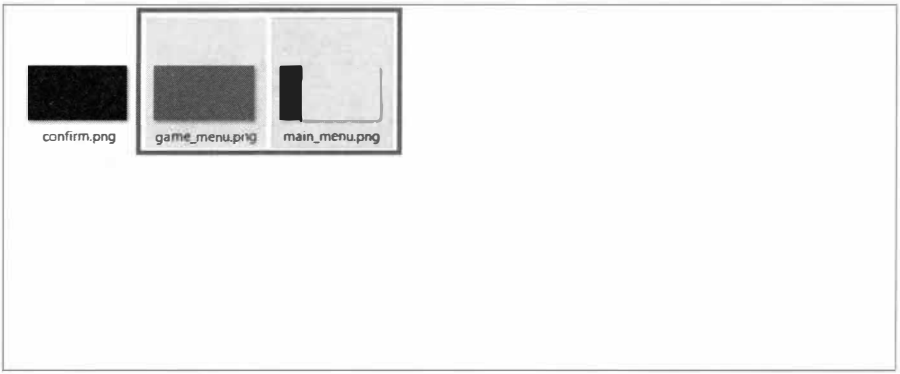


Рис. 2.8.

Здесь же лежит дополнительный слой для главного меню, который подкладывается под кнопки навигации – *main_menu.png*. Его также можно изменить под стиль проекта или сделать прозрачным.

Ещё одно изображение в этой папке – **confirm.png**, используется в окне подтверждения. Например, при выходе из игры. Оно также может быть стилизовано по необходимости.

2.5. Шрифты

Все шрифты, которые используются в проекте, можно заменить на свои и настроить их размер. Откройте файл **gui.rpy** в папке **game**. В самом начале скрипта вы найдёте соответствующий раздел. Если ваш проект был создан на русском языке, все комментарии будут на русском, и вы без труда сможете понять, какая настройка за что отвечает. Если проект создан на другом языке, используйте следующий шаблон:

```
# Акцентный цвет используется в заголовках и подчёркнутых текстах.
define gui.accent_color = '#99ccff'
# Цвет в текстовой кнопке, когда она не выбрана и не под наведением
курсора.
define gui.idle_color = '#888888'
#Small_color используется в тексте, который должен быть ярче/темнее
define gui.idle_small_color = '#aaaaaa'
# Цвет, используемый в кнопках и панелях, когда они под наведением
курсора.
define gui.hover_color = '#c1e0ff'
```

```
# Цвет текстовой кнопки, когда она выбрана, но не под наведением
курсора.
define gui.selected_color = '#ffffff'
# Цвет текстовой кнопки, когда она не может быть выбрана.
define gui.insensitive_color = '#8888887f'
# Цвета, используемые для частей панели, которые не заполняются.
define gui.muted_color = '#3d5166'
define gui.hover_muted_color = '#5b7a99'
# Цвета, используемые в тексте диалогов и выборов.
define gui.text_color = '#ffffff'
define gui.interface_text_color = '#ffffff'
```

Как упоминалось в разделе **1.3. Настройка интерфейса**, с помощью таблицы цветов можно настроить каждый элемент для соответствия стилю игры.

Следующий раздел в файле **gui.rpy** отвечает за игровые шрифты. Давайте попробуем заменить стандартные на свои. Для этого создайте новую папку в папке **game** и назовите её, к примеру, **fonts**. В неё мы будем складывать пользовательские шрифты. Сами шрифты можно найти в интернете и выбрать подходящие.

Стоит иметь в виду, что шрифты, изображения, музыка и прочие сторонние элементы, которые вы скачиваете из интернета, могут иметь авторские права. Поэтому убедитесь, что авторы разрешают использование их материалов. Как правило, они помечены определёнными символами: CC, CC-BY, и др.

Подготовленные шрифты обычно имеют расширение **.otf** или **.ttf**, на это нужно обратить внимание, так как в коде они должны быть прописаны также. Положите файлы во вновь созданную папку **fonts** и откройте *gui.rpy*. Ниже раздела с указанием цветов интерфейса располагается блок с настройками шрифтов. В кавычках необходимо указать путь к файлам. В нашем случае это: "fonts/название_шрифта.ttf"

Цифровые значения указывают на размер шрифта. Пробуйте разные варианты, экспериментируйте.

```
# Заголовок игры в главном меню.
define gui.title_text_font = "fonts/arialbi.ttf"
# Шрифт, используемый для внутриигрового текста.
define gui.text_font = "fonts/DejaVuSans.ttf"
# Шрифт, используемый для имён персонажей.
define gui.name_text_font = "fonts/DejaVuSans.ttf"
# Шрифт, используемый для текста вне игры.
```

```
define gui.interface_text_font = "fonts/AeroMaticsBoldItalic.ttf"
# Размер нормального текста диалога.
define gui.text_size = 35
# Размер имён персонажей.
define gui.name_text_size = 45
# Размер текста в пользовательском интерфейсе.
define gui.interface_text_size = 42
# Размер заголовков в пользовательском интерфейсе.
define gui.label_text_size = 36
# Размер текста на экране уведомлений.
define gui.notify_text_size = 35
# Размер заголовка игры.
define gui.title_text_size = 75
```

Если планируется перевод проекта на другие языки, нужно подбирать шрифты, поддерживающие соответствующие алфавиты. Проверить это можно открыв файл шрифта.

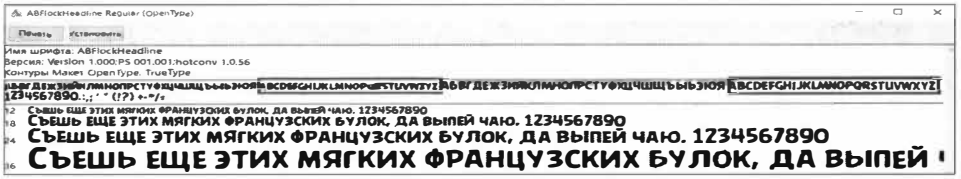


Рис. 2.9.

В примере шрифт, помимо русских букв, включает в себя латинские. Это позволяет переводить игру на языки, использующие латиницу. После всех манипуляций с главным меню оно должно выглядеть гораздо интереснее:

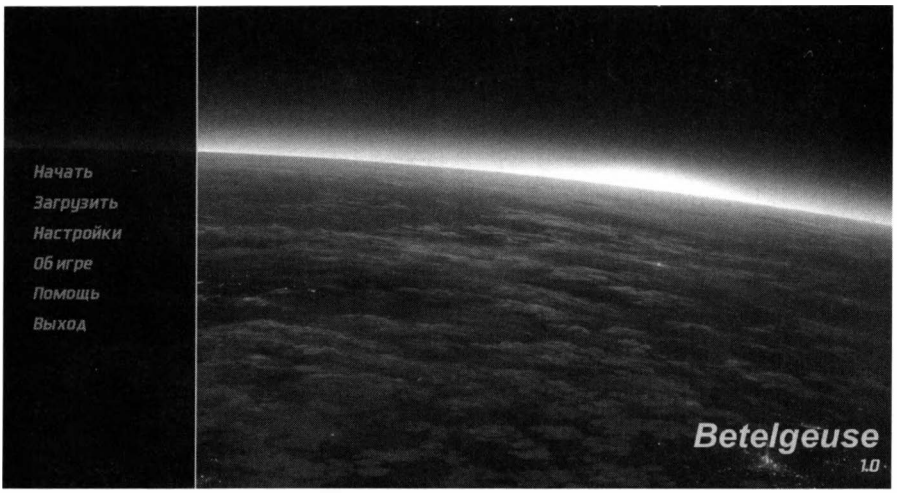


Рис. 2.10.

В правом нижнем углу экрана отображается название и версия проекта. Их также можно отредактировать или вовсе скрыть, например, если название игры будет присутствовать на самом изображении или выводиться отдельным слоем. Откройте файл *options.rpy*, и в начале скрипта найдите строку:

```
define config.name = _("Название_игры").
```

Поставьте перед ней символ решётки **#**, чтобы закомментировать (это скроет заголовок в главном меню):

```
#define config.name = _("Название_игры")
```

Для скрытия версии нужно провести аналогичные действия со строкой:

```
define config.version = "1.0"
```

Она располагается чуть ниже. Помимо этого здесь можно установить актуальную версию игры, написав, например, "0.1".

2.6. Музыкальное сопровождение

Музыка и звуки являются важными элементами любой игры. В визуальных новеллах они также играют не последнюю роль. В этом разделе мы разберёмся, как добавить музыкальное сопровождение в свой проект.

> По умолчанию Ren'Py предлагает **три звуковых канала**: для **музыки**, **звука** и **голоса**. Это означает, что все три канала могут работать одновременно.

На фоне может играть музыка, воспроизводиться звук и голос, читающий текст. Однако если мы включим ещё одну композицию, она заменит собой предыдущую.

Как правило, этих каналов хватает для воспроизведения всех необходимых фоновых эффектов. Для дополнительных звуков можно добавить новые. Давайте для этого создадим отдельный файл *rpy*, в котором будут располагаться наши каналы.

В редакторе кода создайте новый документ, кликнув на вкладку Файл > Новый, и сохраните его с именем **audio** и расширением *rpy*. Название можно указать любое, но с таким будет проще ориентироваться в проекте. Обратите внимание, что файл должен находится в папке **game** вашей игры. Если он сохранился где-то в другом месте, просто перенесите его.

Итак, для создания новых каналов пропишите в *audio.rpy* следующие строки:

```
init python:
    renpy.music.register_channel("music2", "music", loop=True)
```

В данном примере мы зарегистрировали новый канал с названием **music2**, так как просто **music** уже существует по умолчанию. Он входит в группу миксеров "music".

> Параметр **loop=True** отвечает за повторное воспроизведение, то есть композиция будет проигрываться до тех пор, пока мы не выключим её или не заменим другой.

Аналогичным образом регистрируются каналы для звука и голоса:

```
renpy.music.register_channel("sound2", "sound", loop=False)
renpy.music.register_channel("voice2", "voice", loop=False)
```

В отличие от музыкального канала здесь повторение отключено – *loop=False*, так как от них требуется только однократное воспроизведение.

Следующим шагом будет добавление музыкальных файлов в игру. Стоит помнить, что Ren'Py поддерживает форматы .mp3, .ogg, .wav и Opus. Поместите композиции по пути: *game / audio*. Затем в файле *audio.rpy* отступите пару строк от музыкальных каналов, и пропишите следующее:

```
define audio.kosmos = "audio/kosmos.mp3"
```

В данном примере мы присвоили имя первой композиции – *.kosmos* и указали по какому пути она находится. Пропишите аналогичным образом все композиции, которые были помещены в папку **audio**. Не забываем, что названия должны отличаться.

В зависимости от количества каналов и композиций, весь код будет выглядеть примерно так:

```
init python:
# Музыкальные каналы
    renpy.music.register_channel("music2", "music", loop=True)
    renpy.music.register_channel("sound2", "sound", loop=False)
    renpy.music.register_channel("voice2", "voice", loop=False)
# Музыка
define audio.kosmos = "audio/kosmos.mp3"
define audio.space = "audio/space.mp3"
# Звуки
define audio.hit = "audio/hit.mp3"
```

На следующем этапе нам предстоит добавить в код сценария музыкальное сопровождение. Откройте файл *script.rpy*, в котором мы прописывали сюжет игры. Перейдите к **label start**, и напишите в начале блока команду для воспроизведения музыки – **play music *kosmos***, то есть мы говорим движку играть музыку с названием *kosmos*.

Для включения звукового эффекта есть аналогичная команда **play sound *hit***, которая также сообщает, что нужно проиграть звук с названием *hit*.

```
label start:
    play music kosmos # включили фоновую музыку
    play sound hit # воспроизвели звук
    scene bridge with fade
    show liara serious at left with dissolve
    lia "Ого! Что это?!"
```

Как и в случае с отображением спрайтов, музыка и звук будут воспроизведены резко, что может быть не очень приятным на слух. Поэтому для них также есть эффекты для плавного запуска и затухания:

- **fadein** – команда отвечает за плавное нарастание музыки;
- **fadeout** – для плавного затухания композиций.

После этих команд необходимо прописать количество секунд, за которое звук будет возрастать или угасать от нуля до максимальной установленной громкости.

```
label start:
    # музыка нарастает в течение трёх секунд
    play music kosmos fadein 3
    # звук нарастает в течении одной секунды
    play sound hit fadein 1 # воспроизвели звук
```

Для плавного угасания используем следующие команды:

```
stop music fadeout 2
stop sound fadeout 1
```

То есть, теперь вы можете пройтись по всему сценарию, и в нужных местах прописать соответствующую музыку, погасив предыдущую композицию и запустив новую.

На данном этапе без музыкального сопровождения у нас осталось только главное меню. Давайте исправим этот момент. Откройте файл *options.rpy*, и найдите строку:

```
# define config.main_menu_music = "main-menu-theme.ogg"
```

Чтобы быстро находить в коде определённые строки, нажмите в редакторе кода комбинацию **Ctrl + F**, и в открывшееся окно поиска введите часть искомой фразы, а затем нажмите **Enter**.

Мы видим, что строка закомментирована, следовательно, Ren'Py не выполняет данную функцию. Если удалить символ решётки, движок будет воспроизводить композицию *main-menu-theme.ogg*. Однако, скорее всего у нас её нет, и файл предварительно нужно добавить в папку с игрой.

Чтобы соблюдать порядок, поместите композицию для главного меню в папку **audio**, а в конфигурации пропишите путь до файла. Например, так:

```
define config.main_menu_music = "audio/space.mp3"
```

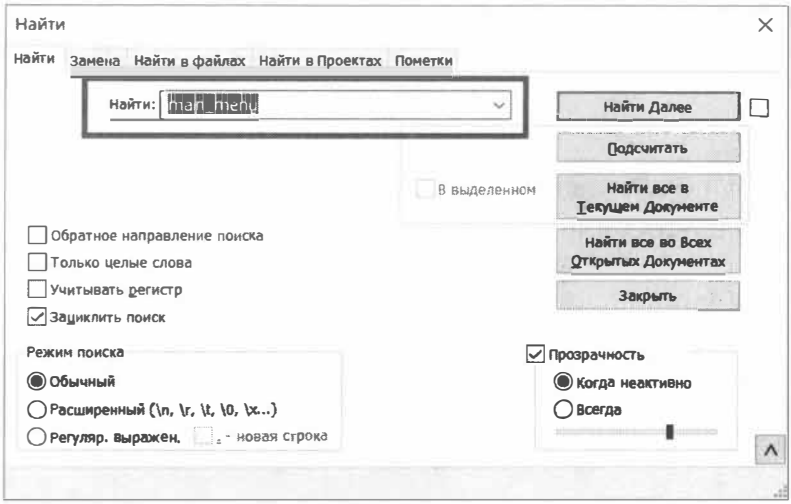


Рис. 2.11.

По умолчанию настройки громкости звука в игре выставлены на максимум. Это может шокировать пользователя, если он играет в наушниках, или к компьютеру подключены мощные колонки.

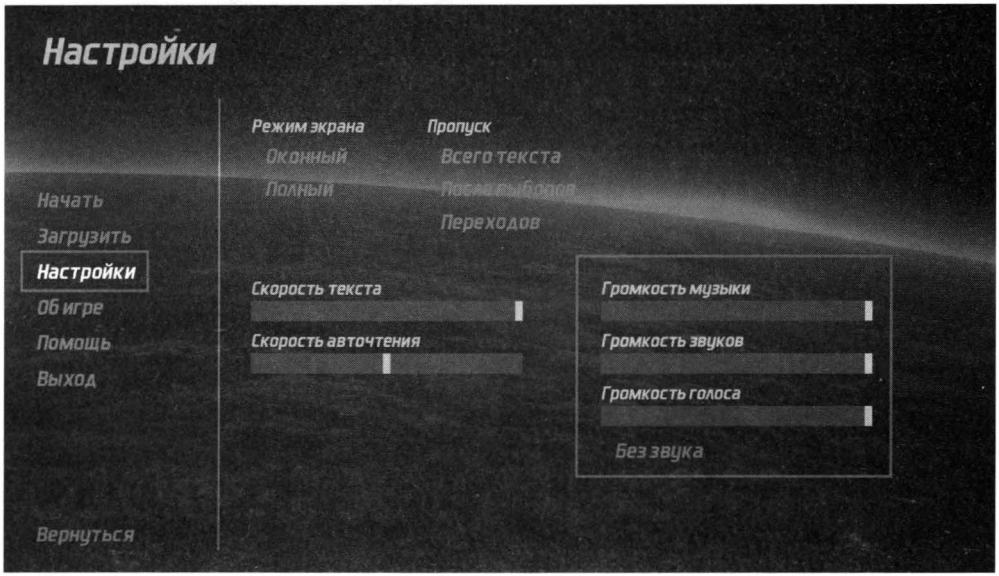


Рис. 2.12.

Давайте позаботимся об этом и допишем несколько команд.

Откройте файл *options.rpy* и вставьте следующее:

```
define config.default_music_volume = 0.3
define config.default_sfx_volume = 0.3
define config.default_voice_volume = 0.3
```

Числовые значения устанавливают громкость каналов, где 0.0 – это полная тишина, а 1.0 – максимальная громкость. В нашем случае мы установили все параметры на треть.

При запуске игры мы можем заметить, что ничего не изменилось. Дело в том, что в главном меню работают постоянные переменные, которые запоминают предыдущие настройки пользователя, чтобы ему не приходилось каждый раз устанавливать их снова, например, если он изменил параметр громкости на 0.5.

Чтобы проверить, устанавливается ли громкость по умолчанию, нам необходимо сбросить эти постоянные к дефолтному состоянию. Перейдите в лаунчер Ren'Py, выберите нужный проект из списка и нажмите "Очистить постоянные".

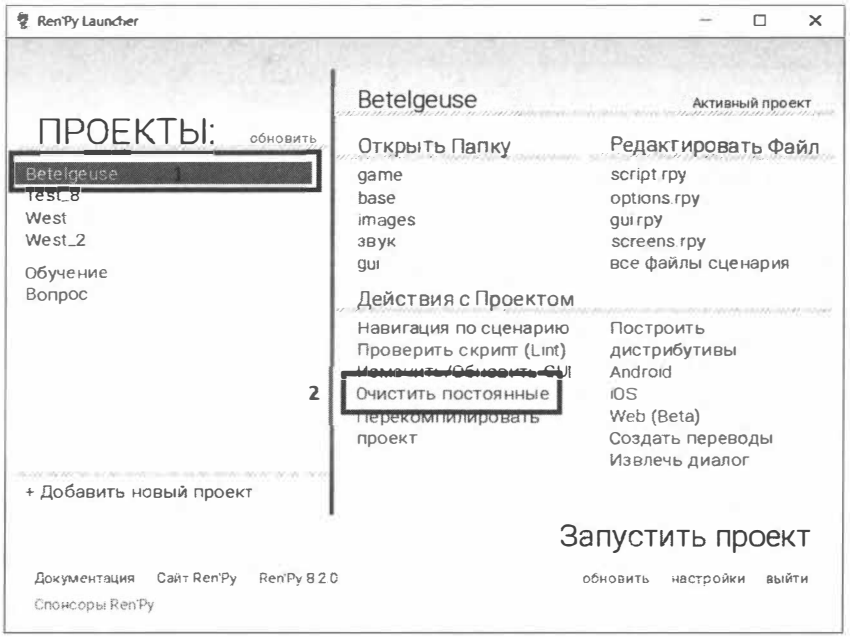


Рис. 2.13.

После этого можно зайти в настройки игры и убедиться, что изменения вступили в силу.

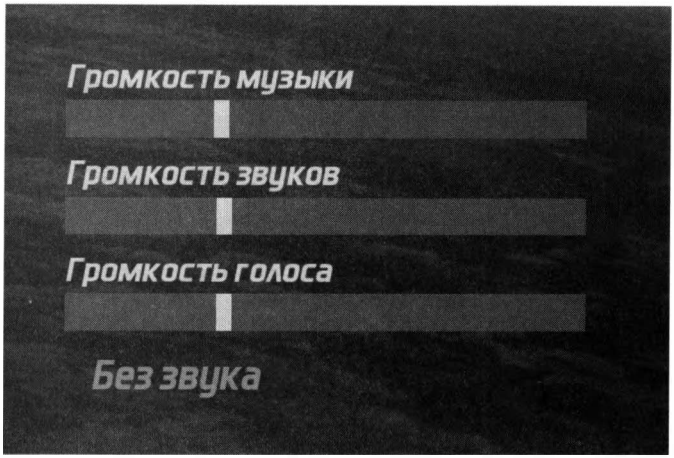


Рис. 2.14.

2.7. Скорость вывода текста

Ещё одна небольшая настройка – это **скорость вывода текста**, который произносят персонажи. По умолчанию параметр установлен на 0. Это значит, что при выводе реплики весть текст появляется на экране моментально. В большинстве визуальных новелл можно увидеть использование эффекта печатания, когда символы появляются поочерёдно с определённой скоростью.

По желанию можно сделать подобное в своей игре. Откройте файл *options.rpy*, и с помощью поиска (**Ctrl+F**), найдите аббревиатуру **CPS**.

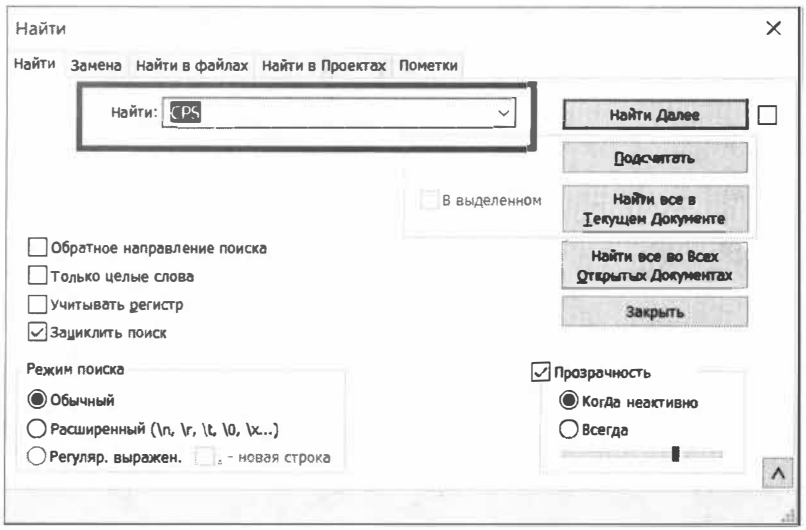


Рис. 2.15.

Результатом поиска станет строка:

```
default preferences.text_cps = 0
```

Она отвечает за скорость вывода текста на экран, где 0 – это количество выводимых символов в секунду. Можно установить нужное вам значение, например на 100.

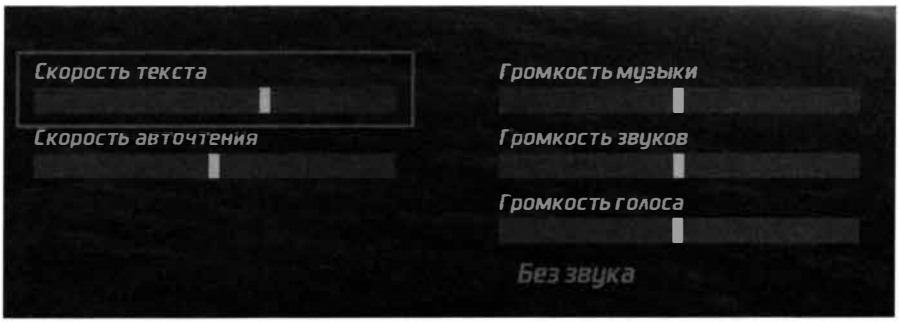
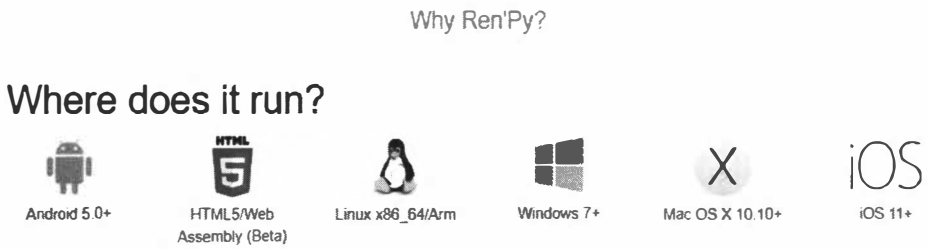


Рис. 2.16.

Глава 3.

КОМПИЛЯЦИЯ ГОТОВОГО ПРОЕКТА



На данном этапе вы уже способны разработать свою первую новеллу с простым функционалом. Нам осталось только скомпилировать её в исполняемый файл, чтобы игроки смогли скачать и запустить проект в пару кликов.

Откройте лаунчер Ren'Py, выберите проект из списка слева, а затем нажмите «Построить дистрибутивы».

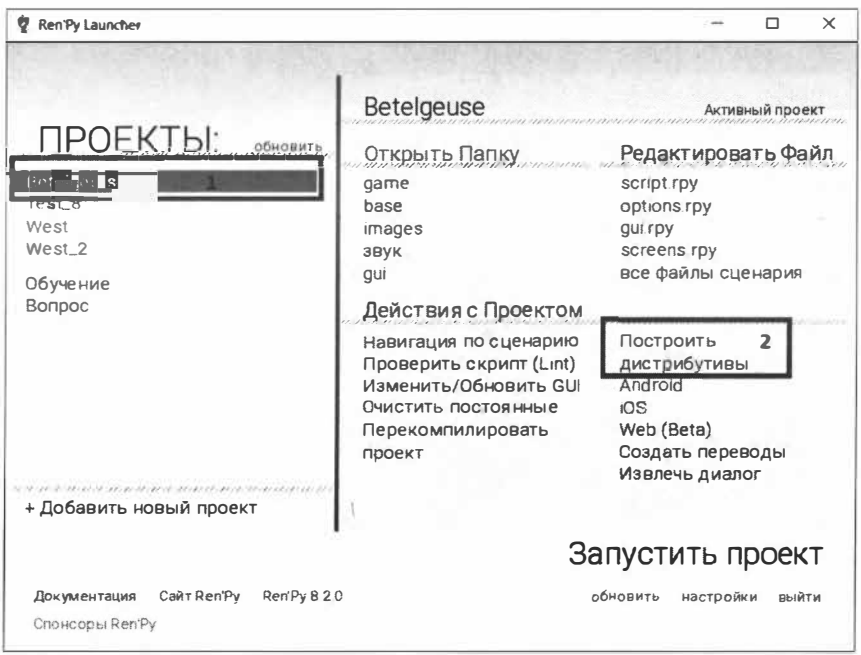


Рис. 3.1

Далее отметьте, какие версии вам нужны. В первом пункте создаётся общий проект для Windows и Linux, он включает в себя два файла **.exe** и **.sh**, для запуска на обеих системах. Третий пункт компилирует версию для Mac.

Остальные пункты опциональны.

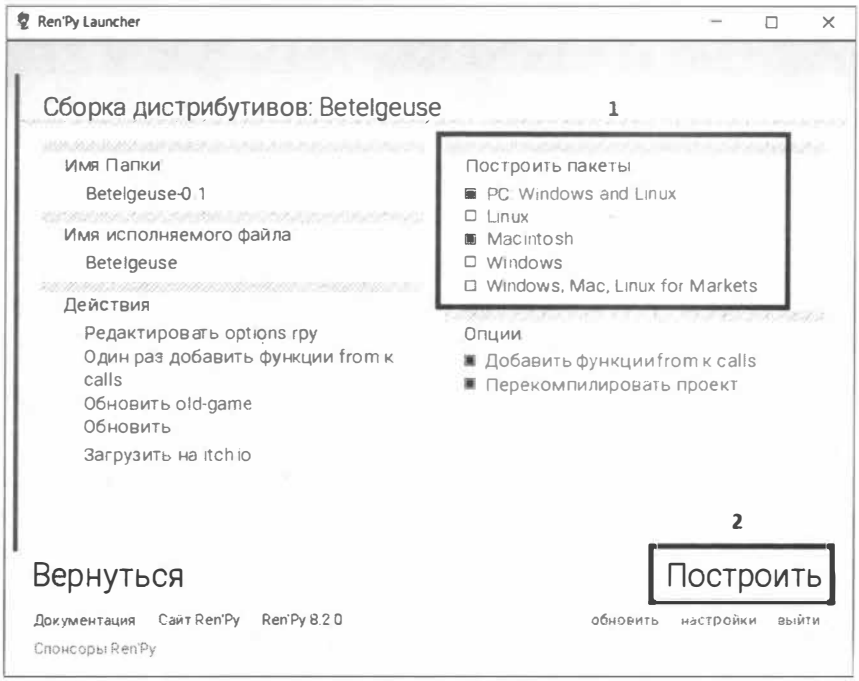


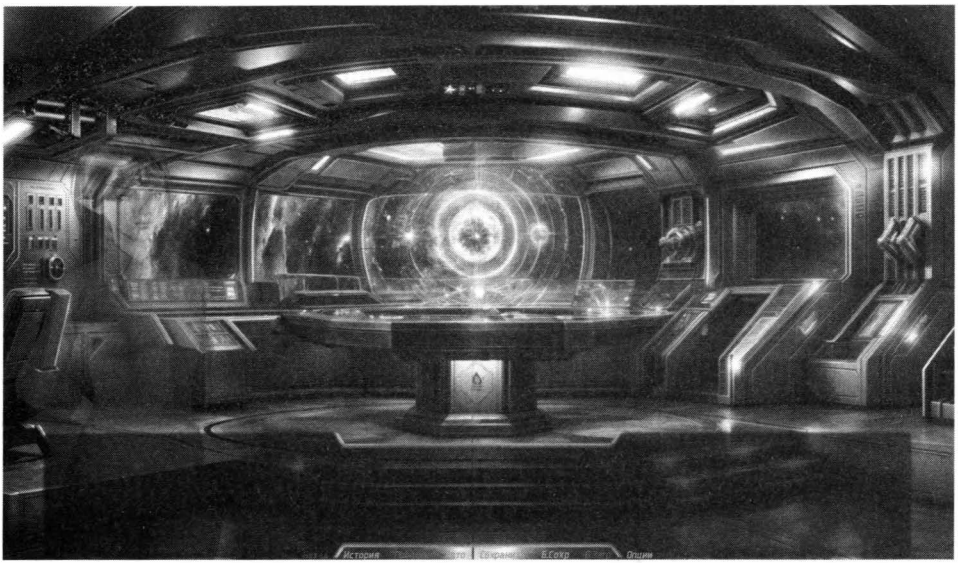
Рис. 3.2

После нажатия кнопки «Построить» Ren'Py приступит к сборке игры. Как правило, это занимает несколько минут, в зависимости от объёма проекта и мощности вашего ПК. По завершении откроется папка с готовыми дистрибутивами. Последним шагом будет публикация вашей игры на цифровых площадках.

В следующих главах мы познакомимся с расширенными возможностями движка, а также с дополнительными механиками, которые позволят внедрить в проект элементы, изначально не предусмотренные в Ren'Py.

Глава 4.

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ REN'PY



4.1. Настройки персонажа

В Ren'Py различные аргументы могут использоваться для настройки различных аспектов персонажей или текста. При создании персонажей *Character* в скобках мы прописывали два таких аргумента: имя героя и цвет имени. Однако помимо этого существует целый ряд параметров, которые можно передать.

Это позволяет устанавливать различные настройки или свойства для персонажей, текста, сцен и других элементов. Настройки для персонажей делятся на два типа. Одни отвечают за имена – **кто говорит** (who), другие относятся к тексту – **что говорит** (what).

В коде это выглядит так:

```
define lia = Character('Лиара', color="#c8ffc8", who_kerning = 5, what_kerning = 10)
```

В данном примере через запятую мы добавили новые параметры, в которых указали промежуток между буквами имени (who_kerning) и буквами в тексте (what_kerning). То есть, имя Лиары теперь будет иметь расстояние между буквами пять пикселей, а промежутки между буквами в её репликах будут равны десяти пикселам.

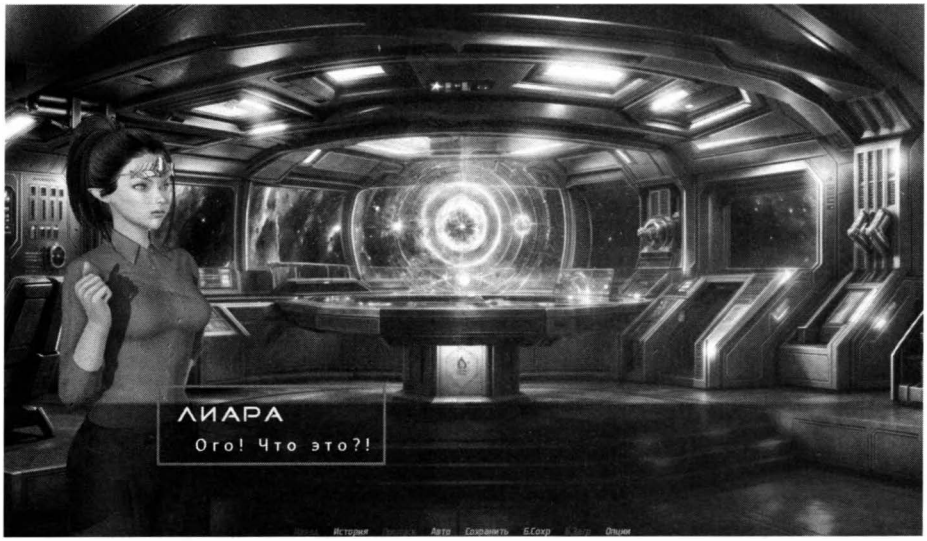


Рис. 4.1.

Таким образом можно индивидуально настроить каждого персонажа. Помимо межбуквенного интервала есть и другие настройки, которые прописываются через запятую.

name: Имя персонажа (используется для отображения имени в диалоговом окне).

image: Изображение персонажа (спрайт).

К каждому характеру можно привязать группу спрайтов, отображающих, к примеру, разные позы или эмоции персонажа. Благодаря этому, во время написания сценария с диалогами, очень просто менять одно изображение на другое.

Для начала прописываем общее название для группы спрайтов в характере:

```
define lia = Character('Лиара', color="#c8ffc8", image="liara")
```

Затем помещаем группу спрайтов персонажа в папку **images**:

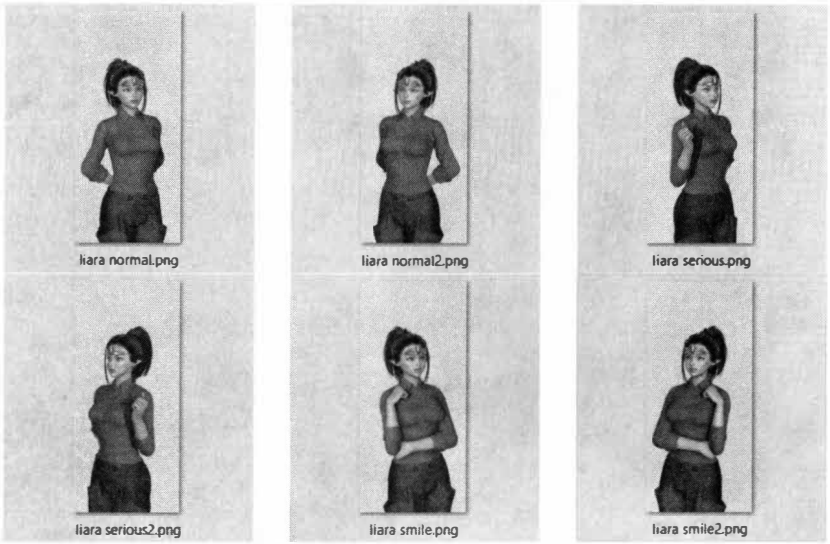


Рис. 4.2.

На изображении мы видим шесть спрайтов в папке, где через пробел дописано название каждого отдельного спрайта в группе. Важно соблюдать такой порядок именования изображений, чтобы Ren'Py правильно относил тот или иной спрайт к нужной группе и персонажу.

То есть вначале пишем название группы, для удобства оно совпадает с именем персонажа. И через пробел прописывается название позы или эмоции. Таких спрайтов с разными названиями для каждого персонажа можно сделать сколько угодно. Главное соблюдать структуру наименований.

Это позволяет значительно уменьшить количество кода, который нам предстоит написать. Раньше для смены позы нужно было:

```
label start:
    show liara serious at left # показали первый спрайт
    lia "Ого! Что это?!"
    hide liara # скрыли первый спрайт
    show liara normal at left # показали другой
    lia "Вы ощутили удар!?"
```

Теперь:

```
label start:
    show liara serious at left # показали первый спрайт
    lia "Ого! Что это?!"
    lia normal "Вы ощутили удар!?" # указали новый спрайт
```

Так как оба спрайта включены в одну группу и привязаны к персонажу, движок понимает, что необходимо изменить одно изображение на другое. При этом он запоминает позицию, на которой находился спрайт. Может показаться, что сокращение в пару строк незначительно, но представьте, если ваша игра состоит из нескольких тысяч строк диалога, где на каждые пять, две лишние. Это порядка 40% лишней работы.

side: Боковой портрет говорящего персонажа.

Помимо группы спрайтов каждому герою можно добавить мини-изображение возле строки диалога. Во время чтения это позволяет сразу понять, кто говорит, не отвлекаясь на имя персонажа, или если имя не указано вовсе.

Рис. 4.3.

Данная функция прописана по умолчанию, поэтому ничего дополнительно указывать не нужно. Нам остаётся подготовить соответствующее изображение и назвать его *side liara.* То есть привязать его к группе персонажа.

Рис. 4.4.

В результате во время реплики персонажа его портрет будет автоматически подставляться слева. По аналогии с обычными спрайтами героя их можно создать несколько с разными эмоциями. Например, *side liara normal*, *side liara serious* и тд.

who_kerning: Расстояние между буквами в имени персонажа.

what_kerning: Расстояние между буквами в реплике персонажа (примеры смотрите выше).

who_ypos: Устанавливает позицию имени в пикселях по координате **y**. Отсчёт идёт от верхнего края диалогового окна к нижнему.

Пример:

```
define lia = Character('Лиара', color="#c8ffc8", who_ypos= 200)
```

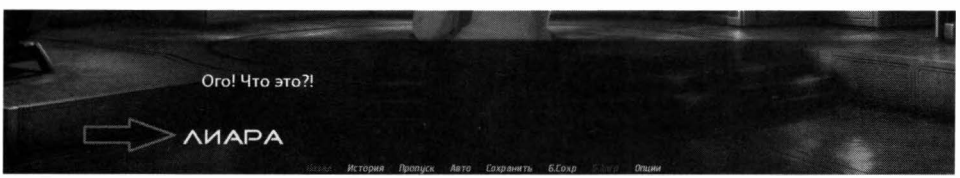


Рис. 4.5.

who_xpos: Устанавливает позицию имени в пикселях по координате **x**. Отсчёт идёт от левого края диалогового окна к правому.

Пример:

```
define lia = Character('Лиара', color="#c8ffc8", who_xpos= 900)
```

Рис. 4.6.

what_ypos и **what_xpos:** Устанавливают позиции реплики персонажа в пикселях. Работает аналогично предыдущим настройкам.

who_color: Устанавливает цвет для имени персонажа.

what_color: Устанавливает цвет текста в диалогах персонажа.

Пример:

```
define lia = Character('Лиара', who_color = "#fbe34f", what_color = "#fb73ff" )
```

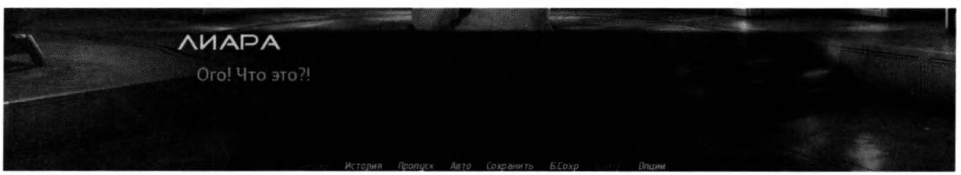


Рис. 4.7.

who_outlines: Используется для определения стиля обводки текста.

Пример:

```
define lia = Character('Лиара', who_outlines=[(1, "#ffffff",2,1) ])
```

- Первая цифра "1" отвечает за толщину обводки текста.
- "ffffff" – определяет цвет обводки.
- Вторая цифра "2" – используется для задания горизонтального смещения обводки текста.
- Третья цифра "1" – устанавливает вертикальное смещение обводки текста.

Все эти значения можно изменять для соответствия стилю персонажа.

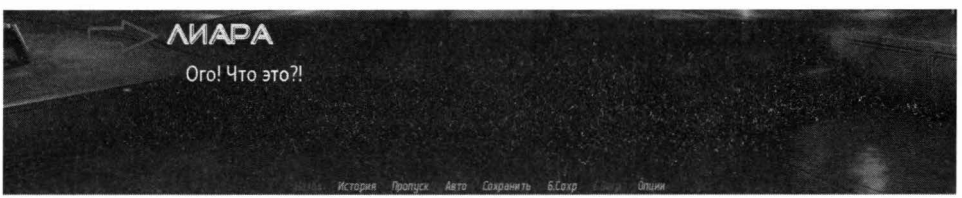


Рис. 4.8.

what_outlines: Настройка работает аналогично предыдущей, только для реплик персонажа.

who_size: Устанавливает размер имени персонажа.

what_size: Устанавливает размер текста.

Пример:

```
define lia = Character('Лиара', who_size = 45, what_size = 37)
```

who_font: Прописывает каждуму персонажу свой шрифт.

Ранее в настройках интерфейса мы устанавливали шрифты для определённых элементов игры. Теперь мы можем прописать каждому персонажу свой отдельный шрифт. Не забудьте положить сам файл шрифта в папку **fonts**, которую предварительно создали.

what_font: Устанавливает шрифт для реплики персонажа.

Пример:

```
define lia = Character('Лиара', what_font= "fonts/Goodtimes.otf" )
```

Рис. 4.9

window_background: Устанавливает индивидуальное диалоговое окно для каждого персонажа.

Пример:

```
define lia = Character('Лиара', window_background="gui/textbox_new.png" )
```

По умолчанию для всех диалогов используется стандартная фоновая подложка *textbox.png*, которая находится в папке **gui**. Как и большинство остальных элементов её можно заменить или стилизовать по необходимости. Но, кроме этого, для каждого персонажа можно определить отдельный текстбокс. Не забудьте положить новое изображение в папку **gui** и прописать правильную ссылку для него, как в примере выше.

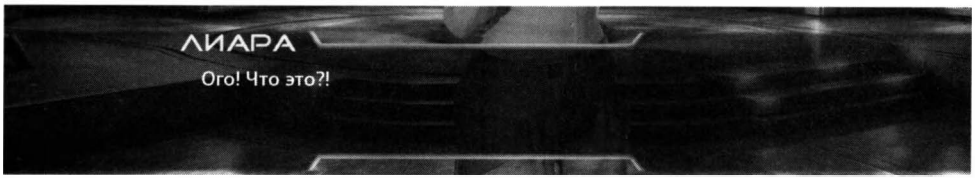


Рис. 4.10.

namebox_background: Аналогично фону диалогового окна, можно настроить окошко имени.

Это изображение также находится в папке **gui**, и по умолчанию является просто прозрачной png-картинкой **namebox**. Вы можете заменить её на свою или сделать для каждого персонажа отдельную.

Пример:

```
define lia = Character('Лиара', namebox_background="gui/namebox_new.png" )
```

Рис. 4.11.

ctc: Это небольшое изображение, как правило, в виде стрелочки или значка, оповещающего, что для продолжения нужно кликнуть по экрану или совершить определённое действие.

Пример:

```
define lia = Character('Лиара', ctc="gui/ctc.png" )
```

Изображение можно анимировать, указав соответствующее действие:

```
define lia = Character('Лиара', ctc=anim.Blink("gui/ctc.png") )
```

По умолчанию в системе нет такого изображения, поэтому вам придётся подготовить его самостоятельно.

Рис. 4.12.

slow_cps: Устанавливает скорость вывода имени персонажа на экран.

Пример:

```
define lia = Character('Лиара', slow_cps=10 )
```

Имя персонажа печатается очень медленно.

who_italic и **what_italic:** Прописывает курсивом имя и текст персонажа, соответственно.

Пример:

```
define lia = Character('Лиара', who_italic=True)
```

who_bold и **what_bold:** Выделяет жирным шрифтом имя и текст персонажа, соответственно.

Пример:

```
define lia = Character('Лиара', who_bold=True)
```

who_strikethrough и **what_strikethrough:** Зачёркивает имя и текст персонажа, соответственно.

Пример:

```
define lia = Character('Лиара', who_strikethrough=True)
```

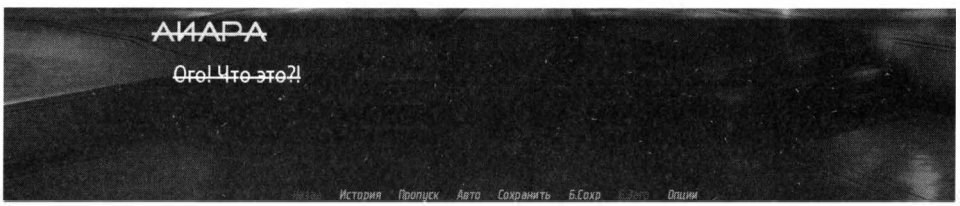


Рис. 4.13.

who_prefix: Позволяет подставить перед именем персонажа определённое значение.

Пример:

```
define lia = Character('Лиара', who_color = "#c8ffc8", who_prefix="Старпом: " )
```

Рис. 4.14.

В нашем примере мы указали должность Лиары – Старший помощник, которое будет указываться во всех её диалогах. Казалось бы, можно добавить должность в атрибут имени, и не создавать префикс. Однако вместо этого можно установить переменную, которая будет хранить должность Лиары, и динамически изменять её, когда она продвинется по службе. Подробнее с переменными мы познакомимся чуть позже.

what_prefix: Подставляет определённое значение перед репликой персонажа. Работает аналогично имени героя.

who_suffix и **what_suffix:** Подставляет определённое значение **после** имени и реплики персонажа, соответственно.

Работает аналогично предыдущим примерам.

```
define lia = Character('Лиара', who_color = "#c8ffc8", what_suffix=" (удивлённо)")
```

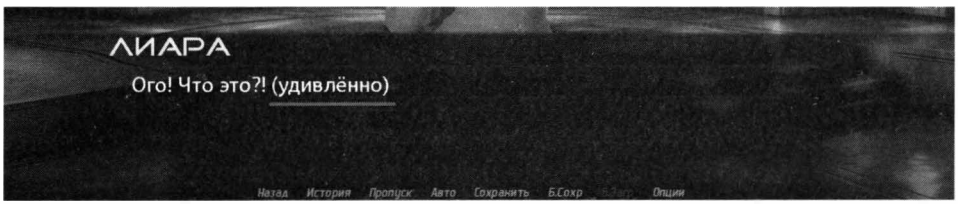


Рис. 4.15.

what_layout: Выравнивает все строки в реплике персонажа, чтобы они были примерно одинаковы по длине.

Пример:

```
define lia = Character('Лиара', who_color = "#c8ffc8", what_layout="subtitle")
```

Рис. 4.16.

Дополнительные вариации можно настроить с помощью стилей, о которых мы поговорим в следующих главах.

> **kind:** Позволяет продублировать настройки одного персонажа для остальных. То есть, всё то, что мы пропишем в атрибутах Лиары, можно добавить Кайдену, просто сославшись на неё.

Пример:

```
define lia = Character('Лиара', who_color = "#c8ffc8", who_prefix="Старпом ")
define kay = Character(kind=lia, name='Кайден', who_color="#ccffff")
```

В атрибутах Кайдена мы указали, что необходимо взять все параметры у Лиары – *kind=lia*. Но, так как персонажу передастся и имя Лиары, то после запятой мы снова установим нужное имя, и другие дополнительные атрибуты, которые должны отличаться. Таким образом, префикс "Старпом" и любые другие параметры, которые могли быть у Лиары, появились и у Кайдена.

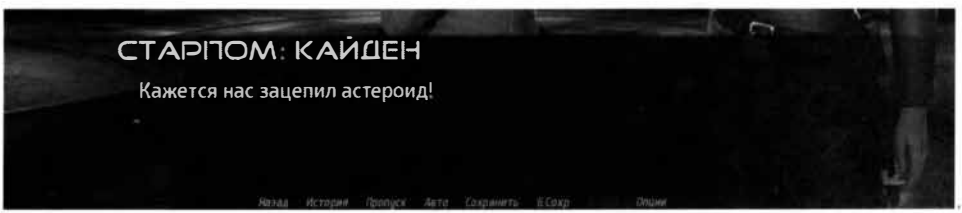


Рис. 4.17.

Если персонажу не указывать никаких параметров, то от имени этого героя можно писать без отображения в диалоговом окне каких-либо элементов. Это позволяет использовать его в качестве рассказчика. Стоит отметить, что в движке по умолчанию есть персонаж с именем narrator для таких случаев. Но, когда у вас много текста рассказчика, можно создать персонажа с более коротким именем.

Например:

```
define nar = Character(None)
```

Ещё более коротким вариантом будет вовсе не указывать имя персонажа, как было представлено в пункте **1.6. Основы синтаксиса в Ren'Py**, и просто писать текст в кавычках, не упоминая имени говорящего.

```
label start:
    narrator "Текст рассказчика"
    nar "Текст рассказчика"
    "Текст рассказчика"
    # Все три варианта равнозначны
```

4.2. Форматирование текста

Помимо настроек для каждого персонажа, можно форматировать отдельные строки диалога. Это может понадобиться, когда необходимо выделить только одно слово или фразу.

Рис. 4.18.

Для этого используются стандартные теги языка разметки BBCode, с которыми вы можете быть знакомы, если оформляли что-либо на сайтах или форумах в интернете. Такие теги помещаются в фигурные скобки **{}**, и делятся на **открывающий** и **закрывающий**.

Во фразе "Нужно проверить **{u}**машинное**{/u}** отделение!" мы подчеркнули слово "машинное" с помощью открывающего тега **{u}** и закрывающего тега **{/u},** таким образом, указав границы, в рамках которых нужно выделить слово или фразу.

В коде это выглядит так:

```
lia "Нужно проверить {u}машинное{/u} отделение!"
```

Кроме подчёркивания, можно использовать следующие тэги для форматирования:

- **{u}** – Подчёркивает текст {/u}
- **{i}** – Выводит текст курсивом {/i}
- **{s}** – Зачёркивает текст {/s}
- **{b}** – Выделяет текст жирным {/b}
- **{size=15}** – Указываем величину шрифта {/size}
- **{size=+15}** – Шрифт будет увеличен на пятнадцать от стандартного {/size}
- **{size=-15}** – Шрифт будет уменьшен {/size}
- **{color="#ffaa00"}** – Устанавливает цвет текста {/color}
- **{cps=15}** – Эффект печати, выводит 15 символов в секунду {/cps}
- **{font=fonts/KOMIKAX.ttf}** – Задаёт выбранный шрифт {/font}
- **{k=2}** – Расстояние между буквами
- **{/k}** – Знакомая нами функция **kerning**, которая отвечает за межбуквенный интервал в пикселах.
- **{alpha=0.5}** Устанавливает прозрачность текста {/alpha} – где 0.0 это полностью прозрачный, а 1.0 – не прозрачный. В нашем примере прозрачность установлена на половину – 0.5.
- **{a=https://renpy.org/}Ren'Py{/a}** – Создаём ссылку, например на сайт разработчика, или на свой сайт, где игроки смогут оставить отзывы и пожелания к вашей игре. Между открывающим и закрывающим тегами пишется слово или предложение, которое станет кликабельной ссылкой.
- **{image=images/smile.png}** – Выводит изображение.
  Обратите внимание, что для изображения не нужен закрывающий тег, а само изображение должно быть размером с текущий шрифт, иначе оно может сместить весь ваш текст и нарушить форматирование.

Рис. 4.19.

- **{w}** – Устанавливает паузу до клика игрока.

Пример:

```
lia "Нужно проверить {w}машинное отделение!"
```

- **{w=2}** – Устанавливает паузу на определённое количество секунд
- **{nw}** – Запускает следующую строку диалога, без клика игрока
- **{nw=1}** – Запускает следующую строку через определённое количество секунду.

  Пример:

```
lia "Нужно проверить {w=2}машинное отделение!{nw=1}"
kay "Я уверен, что двигатель не пострадал."
```

  Таким образом можно создать эффект, как будто один персонаж перебил другого, вставив свою реплику. Если вставлять тег **nw=1** после каждой реплики, создаётся эффект авточтения, где игроку нет необходимости кликать мышью для продолжения.

- **{vert}** – Данный тег выводит текст вертикально. Как правило, используется в экранах (screen), о которых речь пойдёт в следующих главах.
- **{horiz}** – Аналогичный тег для горизонтального вывода текста. Он используется по умолчанию.
- **{space=200}** – Делает отступ от левого края, например, чтобы начать текст с "Красной строки". Расстояние указывается в пикселах.
- **{vspace=200}** – Аналогично для вертикального отступа.
- **{fast}** – Независимо от настроек скорости вывода текста, выводит его моментально.

  Также можно ставить один тег внутри другого:

```
"Нужно проверить {size=+10}{u}машинное{/u}{/size} отделение!"
```

  Текст будет увеличен и подчёркнут.

Рис. 4.20.

Помимо фигурных скобок, в Ren'Py используются и другие элементы форматирования:

\n – Обратный слэш + **n** переводят следующий текст на новую строку:

```
"Нужно проверить\n машинное отделение!"
```

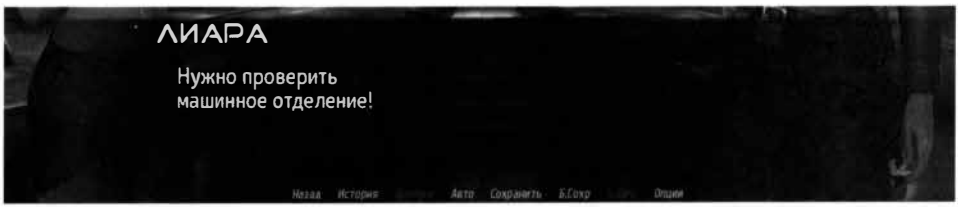


Рис. 4.21

\"Текст\" – Слэш с кавычками указывает, что данный текст нужно поместить в кавычки.

Пример:

```
lia "Нужно проверить \"машинное\" отделение!"
```

Так как стандартные кавычки используются движком при выводе всего текста, если мы внутри простых кавычек поставим ещё одни, это приведёт к ошибке.

%% – Двойной символ процента выводит на экран один символ **%**.

Пример:

```
kay "Я на 100%% уверен, что двигатель не пострадал."
```

Так как сам символ зарезервирован системой для вычисления математических операций, использование одиночного символа **%** в тексте также приведёт к ошибке.

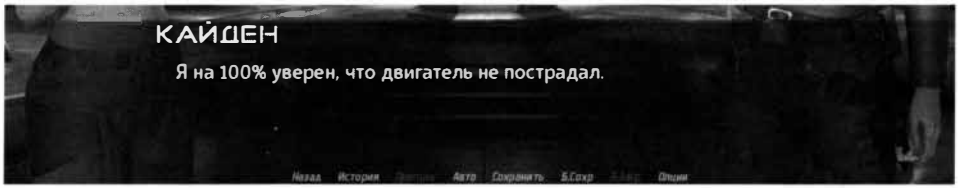


Рис. 4.22.

4.3. Стандартные эффекты переходов

В разделе "1.7. Создаём свою первую новеллу за 10 минут" мы познакомились с базовыми эффектами переходов, которые применяются для более плавного отображения фонов и спрайтов на экране. Помимо этих эффектов, Ren'Py предлагает и множество других переходов.

Давайте вспомним, как они используются:

```
label start:
    scene bridge with fade
    show liara serious with dissolve
    lia "Ого! Что это?!"
```

С помощью оператора with мы можем выводить изображения с разными эффектами.

Некоторые из них больше подходят для бэкграундов, а некоторые для спрайтов. Так как анимации в текстовом формате невозможно отобразить в полной мере, рекомендуется попробовать каждый из них, как для фона, так и для спрайта. Это позволит сформировать представление, какие из них применять в той или иной ситуации.

- **with fade** – выводит изображения из затемнения, а также скрывает его, если мы пропишем соответствующую команду *hide liara with fade*
- **with dissolve** – плавно выводит или растворяет изображение
- **with zoomin** – выводит изображение с эффектом увеличения от нуля до оригинального размера

- **with zoomout** – противоположно предыдущей команде, убирает изображение с экрана, с эффектом уменьшения
- **with moveinright** – выводит изображение или спрайт с правой части экрана
- **with moveinleft** – аналогичный эффект, с выводом слева
- **with moveintop** – выводит изображение сверху вниз
- **with moveinbottom** – снизу вверх
- **with moveoutright** – изображение уезжает в правую часть экрана
- **with moveoutleft** – уезжает влево
- **with moveouttop** – уезжает вверх
- **with moveoutbottom** – вниз
- **with wiperight** – эффект похож на предыдущий, только в данном случае новое изображение наезжает поверх предыдущего
- **with wipeleft** – аналогично в левую сторону
- **with wipeup** – вверх
- **with wipedown** – вниз
- **with slideright** – слайд-переход в правую сторону
- **with slideleft** – влево
- **with slideup** – вверх
- **with slidedown** – вниз
- **with slideawayright** – эффект противоположный предыдущему
- **with slideawayleft** – на лево
- **with slideawayup** – вверх
- **with slideawaydown** – вниз
- **with irisout** – отображает изображение из центра экрана, которое замещает собой предыдущее
- **with irisin** – аналогичный эффект, работающий в обратную сторону
- **with pushright** – создаёт эффект склейки двух изображений, где одно смещает другое слева направо
- **with pushleft** – смещение в левую сторону

- **with pushdown** – вниз
- **with pushup** – вверх
- **with blinds** – замена одного изображения на другое с эффектом вертикальных жалюзи
- **with squares –** весь экран разделяется на множество квадратных фрагментов, каждый из которых плавно замещается на фрагмент нового изображения
- **with pixellate** – изображение разбивается на крупные пиксели, незаметно подменяя старое на новое
- **with easeinright** – новое изображение появляется с правой стороны, с эффектом плавного торможения. Эффект лучше заметен при использовании на спрайтах персонажей
- **with easeinleft** – тот же эффект слева
- **with easeintop** – сверху
- **with easeinbottom** – снизу
- **with easeoutright** – эффект, похожий на предыдущий, но работающий в обратной последовательности. Исчезая с экрана, спрайт постепенно ускоряется, создавая эффект "убегания" персонажа за пределы экрана
- **with easeoutleft** – тот же эффект в левую сторону
- **with easeouttop** – вверх
- **with easeoutbottom** – вниз
- **with vpunch** – отображает изображение с эффектом вертикальной встряски
- **with hpunch** – эффект горизонтальной встряски

Каждому эффекту дополнительно можно задать время срабатывания.

Пример:

```
show liara with Dissolve (3.0)
```

Спрайт Лиары будет медленно проявляться на экране в течение трёх секунд. Обратите внимание, что название эффекта в данном случае пишется с **большой буквы**!

Вместе с тем эффекты переходов можно комбинировать с позиционированием персонажей на экране, благодаря чему они будут перемещаться с одного места на другое, освобождая пространство для новых героев.

Пример:

```
label start:
    scene bridge with fade
    show liara serious with dissolve
    lia "Ого! Что это?!"
    show liara normal at left with moveinleft
    lia "Вы слышали удар?!"
    show kayden normal at right with moveinright
    kay "Кажется, нас зацепил астероид!"
```

В строке *show liara serious with dissolve* спрайт по умолчанию отображается по центру, так как точная позиция не указана.

В строке *show liara normal at left with moveinleft* мы указываем, что персонаж должен отобразится слева (at left), применив к нему эффект перехода **moveinleft**, который отвечает за сдвиг влево.

Далее, в строке *show kayden normal at right with moveinright* мы отображаем Кайдена с правой стороны (at right), применив для него **эффект движения** (with moveinright), чтобы он не просто появился из ниоткуда, а выехал из-за правой части экрана.

Скопируйте вышеуказанный код в свой проект, чтобы увидеть, как это работает. Попробуйте комбинировать другие эффекты, чтобы создать новые варианты движений. Подобные эффекты и анимированные перемещения позволяют оживить происходящее на экране, придавая динамики, чтобы игра не выглядела статичной.

4.4. Создаём свои эффекты переходов

Для создания своих эффектов перехода обычно применяются два способа. В первом мы редактируем стандартные эффекты под свои нужды. Во втором добавляем собственный переход с использованием своих изображений.

Чтобы соблюдать порядок в файлах, давайте создадим отдельный документ в редакторе кода для добавления в него своих эффектов. Назовите его, к примеру, *effects.rpy*, и напишите в нём следующий скрипт:

```
init python:
    ld2 = Dissolve(2.0)
    ld3 = Dissolve(3.0)
    ld4 = Dissolve(4.0)
```

Строка **init python** указывает движку, что всё перечисленное в этом блоке является кодом на языке программирования Python, и его нужно инициализировать при запуске проекта. Таким образом, наш код будет работать во всех частях игры. Например, если мы захотим использовать эффекты в **label splashscreen**.

Следующие три строки в блоке являются переменными, которые дублируют эффект **Dissolve**. Но, с разным количеством времени в секундах, для воспроизведения эффекта. Из предыдущего раздела мы знаем, что с помощью with Dissolve (3.0) можно установить любое необходимое время для перехода. Однако сокращение **ld2** позволяет записывать команду в более короткой форме.

По умолчанию стандартный эффект **dissolve** длится 1,5 секунды, но в некоторых ситуациях может понадобиться более долгий эффект. Именно для этого можно создать несколько похожих переходов. Вместо названий **ld2**, **ld3**, **ld4** можно выбрать любые удобные. В данном случае это сокращение от **long dissolve** (долгий эффект растворения), а цифра в конце указывает продолжительность эффекта.

Это самый простой пример модификации стандартного перехода. Во время игры он используется также, как и базовые:

- **scene engine with ld4 #** – фон медленно проявляется в течение четырёх секунд
- **show liara with ld2 #** – спрайт Лиары проявится в течение двух секунд

Теперь в том же блоке, с новой строки и соблюдая **отступ** (Tab), напишите следующий эффект:

```
init python:
    lfade = Fade(2.0, 0.0, 1.0)
```

По аналогии с предыдущим переходом мы модифицировали стандартный эффект затемнения, сделав его более долгим. Это может понадобиться, например, для плавного ухода в темноту.

Первый атрибут в скобках отвечает за время затемнения экрана в секундах (2.0)
Второй – указывает время паузы, которое продлиться темнота. В нашем случае это ноль.
Третий – определяет время просветления экрана (1.0).

Следующий пример:

```
init python:
    flash = Fade(.25, 0, .25, color="#8B0000")
```

Здесь мы ещё больше изменили стандартный переход, добавив свой цвет (красный). В данном случае время затемнения и просветления длится четверть секунды, что создаёт эффект красной вспышки, которая может использоваться в качестве визуализации какого-нибудь столкновения или ушиба персонажа.

Чтобы создавать более сложные эффекты, подобные **pixellate**, **squares** и **blinds**, потребуются соответствующие изображения. Как правило, это разнообразные узоры и орнаменты, состоящие из двух или нескольких цветов. Например, таких:

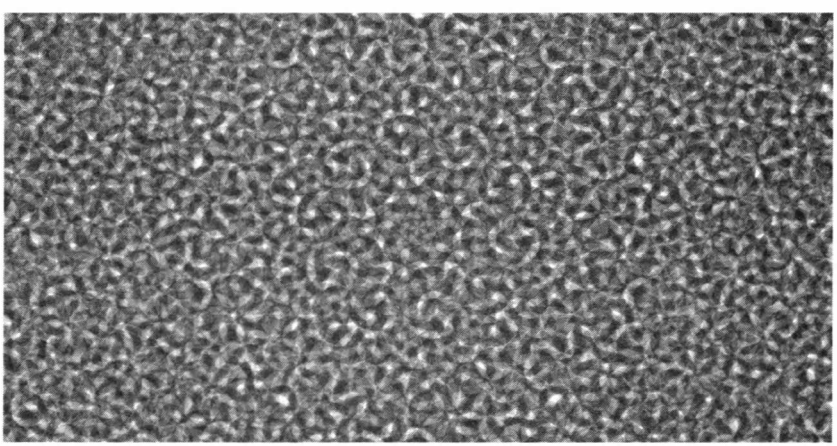


Рис. 4.23.

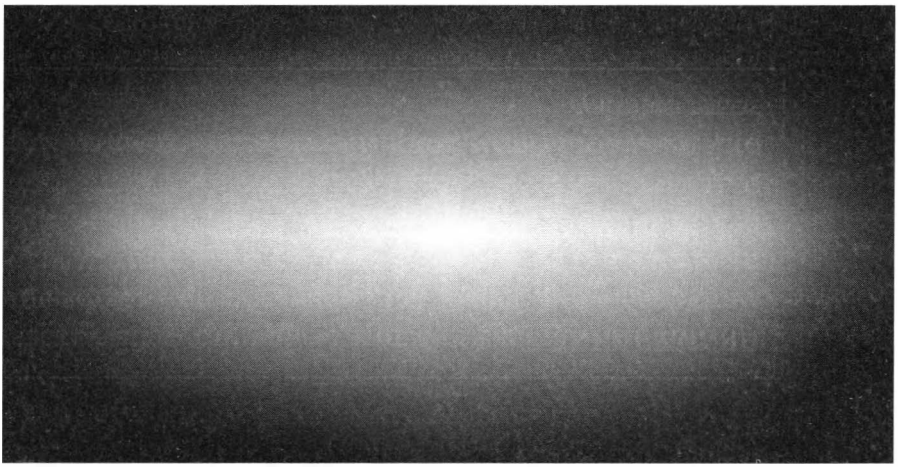


Рис. 4.24.

Суть таких переходов заключается в том, что одно изображение проявляется сквозь другое в виде соответствующего узора. Проявление происходит постепенно от тёмных участков изображений к светлым или наоборот. Изображения не обязательно могут быть чёрно-белыми.

```
init python:
    wakeup = ImageDissolve("ornament.jpg", 2.0, 50, reverse=False)
    sleep = ImageDissolve("ornament.jpg", 2.0, 50, reverse=True)
```

В примере мы создали два практически одинаковых перехода, которые имитируют открытие и закрытие глаз.

- **Первый атрибут** – это название изображения *ornament.jpg*, которое используется в качестве шаблона эффекта.
- **2.0** – второе число отвечает за длительность перехода. В нашем случае две секунды.
- **50** – следующий атрибут устанавливает коэффициент сглаживания краёв между тёмными и светлыми частями, где
- **0** – без сглаживания, а **100** – максимальное сглаживание.
- **reverse** – разворачивает просветление в обратную сторону. Если он выключен (*False*) происходит переход от светлого к тёмному. Когда включен (*True*) – от тёмного к светлому.

В коде и в игре это будет выглядеть следующим образом:

```
scene engine with wakeup # - эффект открытия глаз
```

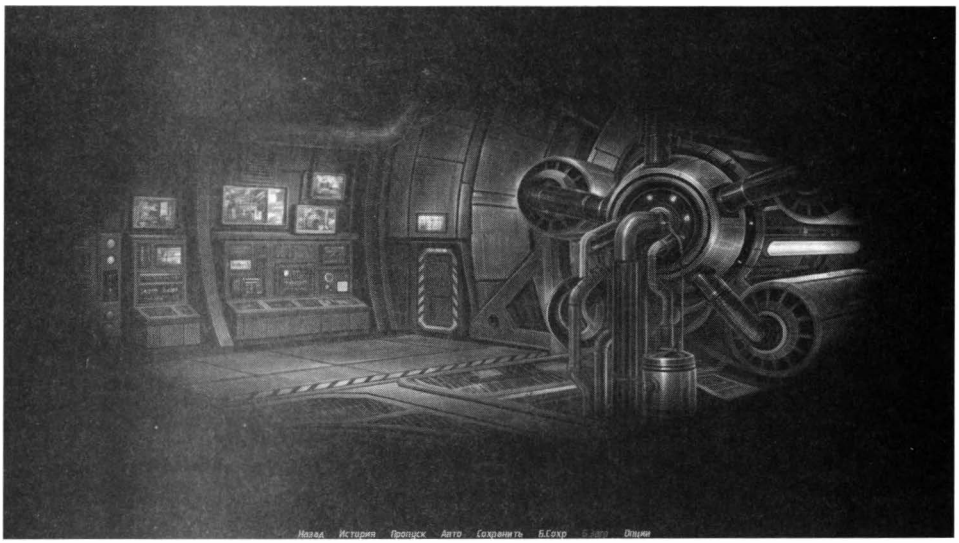


Рис. 4.25.

Таким образом, подставляя разные изображения и присваивая каждому эффекту своё уникальное имя переменной, вы можете создать целый пул новых переходов, которые придадут вашей игре оригинальности и выделят её на фоне других новелл.

4.5. Создаём позиции для персонажей

Ранее мы познакомились с базовыми позициями спрайтов, на которые можно вывести персонажей. Это **left**, **center** и **right**. Однако иногда этих позиций может оказаться недостаточно. Например, когда в кадре находится больше героев, или, если сразу несколько героев нужно вывести по центру.

Аналогично тому, как мы модифицировали эффекты переходов, можно изменить расстановку спрайтов на экране, указав им точную позицию. Но, для начала, давайте разберёмся с системой координат.

В Ren'Py отсчёт по горизонтали (координата **X**) начинается от левого края экрана к правому. То есть, указав позицию спрайта **xpos 300**, слева будет отсчитано триста пикселей, и в этой точке установится наше изображение.

Рис. 4.26.

То же самое по вертикали (координата **Y**) отсчёт идёт сверху вниз. То есть, указав позицию **ypos 400**, отступ будут считаться сверху. Стоит отметить, что при выводе спрайтов на экран, они автоматически "приклеиваются" к нижней его части. Поэтому можно настраивать только координату **X**.

```
label start:
    scene medcenter with fade
    show liara serious:
        xpos 400
```

> **ВАЖНО**! Обратите внимание, что наш спрайт **liara serious** получил дополнительный параметр, поэтому мы ставим двоеточие. Затем с новой строки, и с отступом вправо, прописываем его. Таким образом, Ren'Py понимает, что позиция относится именно к спрайту.

Как видим, изображение Лиары сместилось на 400 пикселей от левого края, и находится в промежуточной позиции, ближе к центру. Далее в коде можно прописать позицию для второго персонажа.

```
label start:
    scene medcenter with fade
    show liara serious:
        xpos 400
    show kayden normal:
        xpos 800
```

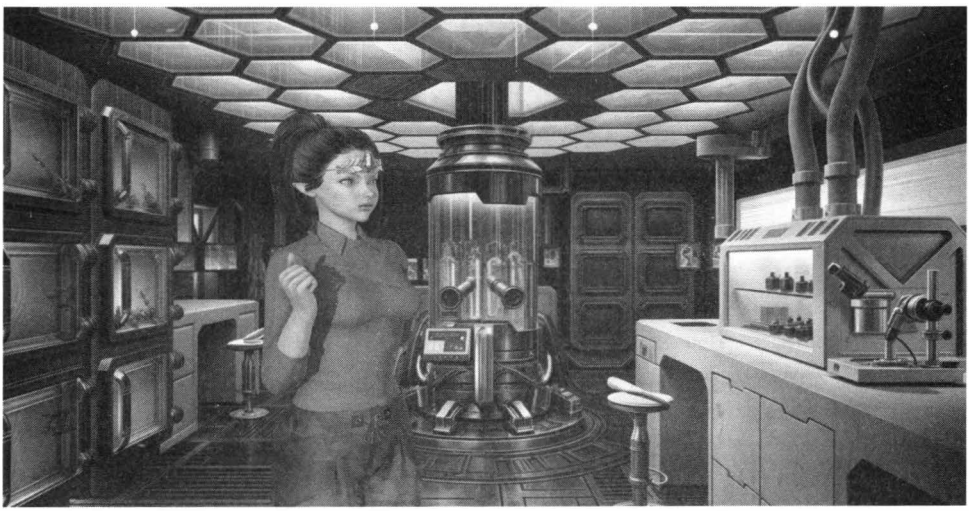


Рис. 4.27.

Теперь наши герои стоят ближе друг к другу, относительно стандартных позиций **left** и **right**. Заметьте, что спрайт Кайдена также имеет двоеточие и отступ, после которого прописываются дополнительные параметры. Благодаря этому в коде соблюдается структура, что помогает сделать его более читаемым, понятным, и проще поддающимся изменениям и доработкам.

В случаях, когда нам нужно вывести спрайт где-то в центре экрана или сверху, потребуется дополнительно указать координату по **Y**.

```
label start:
    scene medcenter with fade
    show kolobok:
        xpos 500
        ypos 300
```

Рис. 4.28.

Теперь к строке с позицией по **X** добавилась ещё одна с позицией по **Y**. Эти строки имеют одинаковый отступ, так как относятся к одному и тому же блоку **show kolobok**. Помимо этого, допустимо прописывать обе позиции в одной строке.

Следующий момент, который нужно учитывать, касается того, что отсчёт от края экрана ведётся не к центру спрайта, а к его верхнему левому углу. В некоторых случаях это требует корректировки позиции.

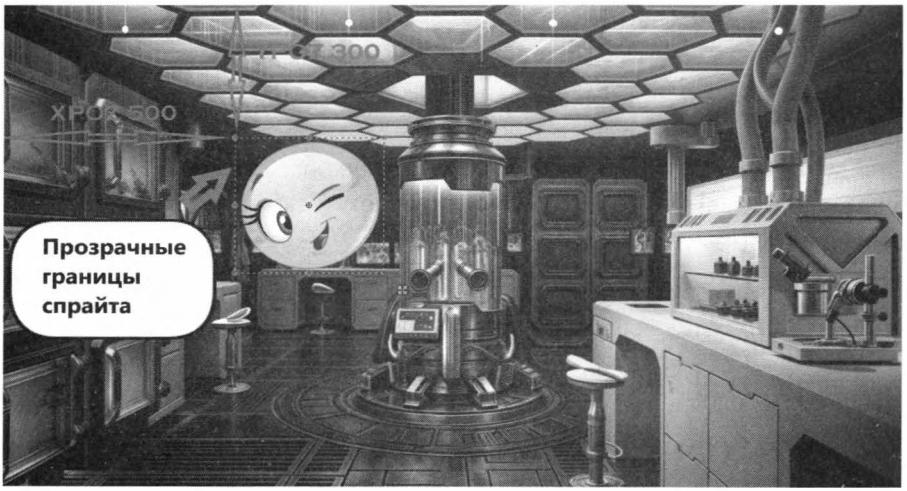


Рис. 4.29.

Чтобы отсчёт вёлся до центра спрайта, ему необходимо установить **якорь** (anchor), точку на изображении, к которой будет привязываться координата.

```
label start:
    scene medcenter with fade
    show kolobok:
        anchor (0.5, 0.5)
         xpos 500 ypos 300
```

Якорь можно установить к любой точке изображения от 0.0 до 1.0. Вместе с тем, расположение спрайта можно указывать не только по точным координатам в пикселях (xypos), но и в процентном соотношении **xyalign**.

```
label start:
    scene medcenter with fade
    show kolobok:
        align (0.5, 0.5) # спрайт в центре экрана
```

Или:

```
label start:
    scene medcenter with fade
    show kolobok:
        xalign 0.4  # 40% от левого края
        yalign 0.75 # 75% отступ сверху
```

Приведённые выше примеры нужны для понимания, как работает вся система. Однако, такое позиционирование на протяжении всей игры, выглядит довольно громоздко. Поэтому далее рассмотрим более короткий и удобный вариант. В котором мы, по аналогии с базовыми позициями, сохраним новые координаты.

Создайте новый документ в редакторе кода, или пропишите в самом верху файла *script.rpy* следующие строки:

```
init python:
    lcenter = Position(xalign=0.25) # слева от центра
    rcenter = Position(xalign=0.75) # справа от центра
```

В данном примере мы создали две новые позиции с координатой по оси **X**. При необходимости, через запятую можно указать расположение по **Y**.

```
init python:
    newpos = Position(xalign=0.5, yalign=0.1)
```

Теперь, показывая спрайты персонажей, мы можем их отображать также, как и в стандартном варианте:

```
label start:
    scene medcenter with fade
    show liara serious at lcenter
    show kayden serious at rcenter
```

Лиару мы показали левее от центра, а Кайдена правее:

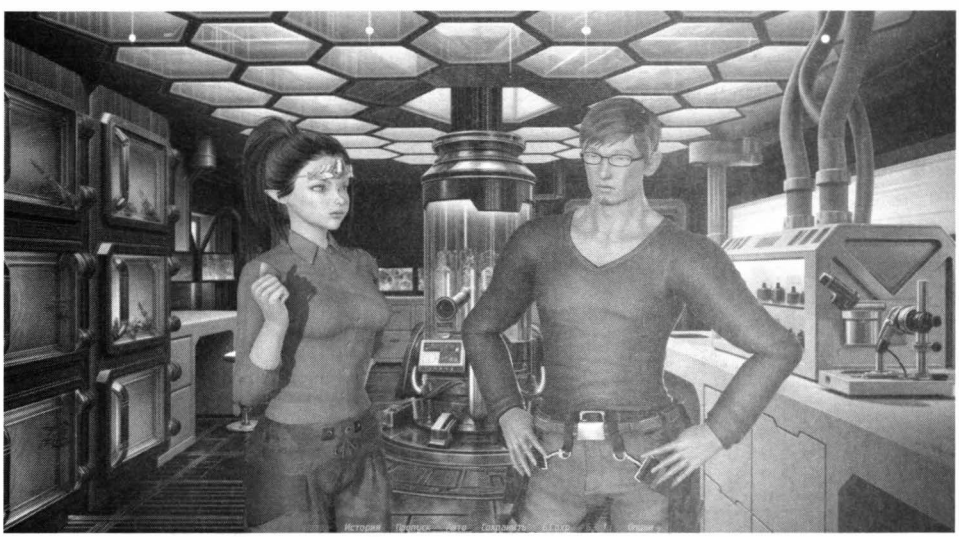


Рис. 4.30.

При подготовке спрайтов для своего проекта, не забывайте, что прозрачные края тоже учитываются при вычислении координат. Если изображения будут иметь разрешение как у игры, то установленные позиции будут сильно отличаться от задуманных.

Рис. 4.31.

Поэтому лишнюю часть следует обрезать в фотошопе или в подобных графических редакторах. Поместите все изображения персонажа на разные слои, центрируйте их, и обрежьте все сразу одной операцией. Затем сохраните каждый со своим названием в формате **png**. Таким образом, у каждого из спрайтов будет одинаковый размер, и при смене в игре один на другой, они не будут смещаться из-за разности ширины.

Рис. 4.32.

4.6. Меню выборов и развилки

Одним из отличительных элементов визуальных новелл является возможность делать развилки в сюжете, где игроку предоставляется выбор действий, от которых будет зависеть дальнейший ход повествования. Благодаря этому можно разнообразить прохождение, сделать несколько концовок, и повысить реиграбельность.

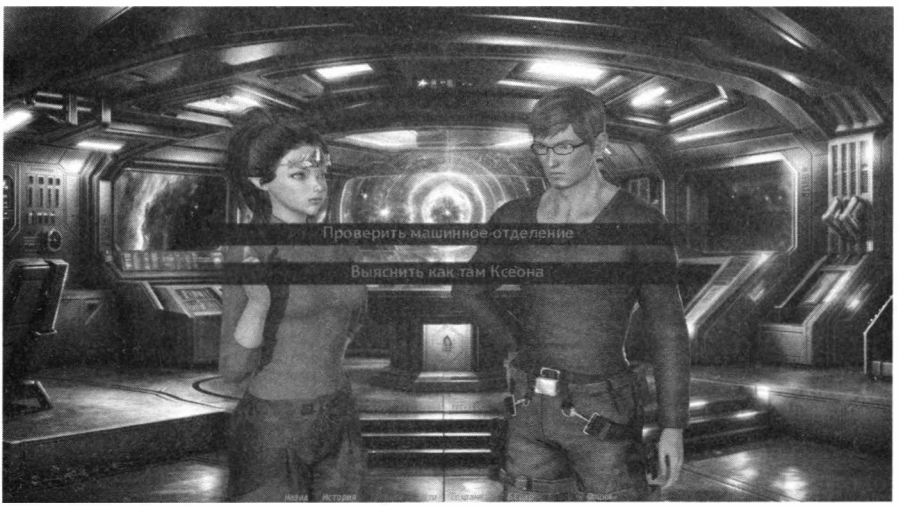


Рис. 4.33.

В коде это выглядит так:

```
lia "Нужно проверить машинное отделение!"
kay "Мне кажется, лучше выяснить, как там Ксеона"
menu:
    "Проверить машинное отделение":
        jump engine
    "Выяснить, как там Ксеона":
        jump medcenter
```

В третьей строке примера мы открываем блок **menu**, который включает в себя несколько вариантов выбора, следовательно, нам необходимо сделать отступ вправо для всех строк этого блока.

Каждый из вариантов тоже является отдельным блоком, который может включать в себя список событий, поэтому они также заканчиваются двоето-

чием, и, с отступом вправо прописываются события, которые произойдут, если мы выберем этот вариант ответа.

В данном случае, в зависимости от выбора, мы прыгнем **jump** к новым меткам. Первый вариант отправит нас к метке *engine*, второй к метке *medcenter*.

```
label engine: #Перешли к метке, если выбрали первый ответ
    scene engine with fade # фон
    show liara serious at lcenter # Спрайт Лиары
    show kayden normal at rcenter # Спрайт Кайдена
    kay "Кажется, с двигателем всё в порядке"
    # здесь дальнейшее развитие этого ответвления
    jump part_2 # прыгаем к новой метке

label medcenter: #Перешли к метке, если выбрали второй ответ
    scene medcenter with fade # фон
    show xeona serious at center # Спрайт Ксеоны
    show liara serious at left # Спрайт Лиары
    show kayden normal at right # Спрайт Кайдена
    lia "Ксеона, как ты себя чувствуешь?"
    # здесь дальнейшее развитие этого ответвления
    jump part_2 # прыгаем к новой метке
```

Визуально схему развилки можно отобразить так:

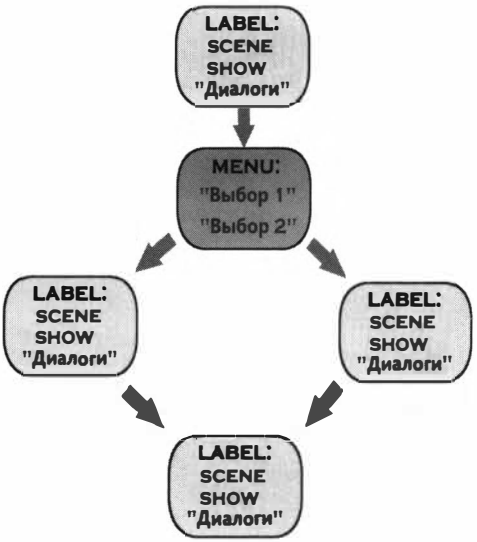


Рис. 4.34.

После развилки два отдельных события могут возвращать в одну общую сюжетную линию, или расходиться на две отдельные ветки, которые, в свою очередь, могут ещё много раз расходиться на дополнительные подветки и второстепенные сюжеты, приводя игрока к разным концовкам. Главное, самому не запутаться в подобных хитросплетениях.

В блоке **menu** можно предлагать любое количество вариантов ответа, но стоит иметь в виду, что большое их количество не поместится на экране. Вместе с тем, при выборе ответа можно выводить последнюю реплику персонажа, или вопрос. На случай, если игрок отвлёкся и забыл контекст, или загрузил сохранение после долгого перерыва.

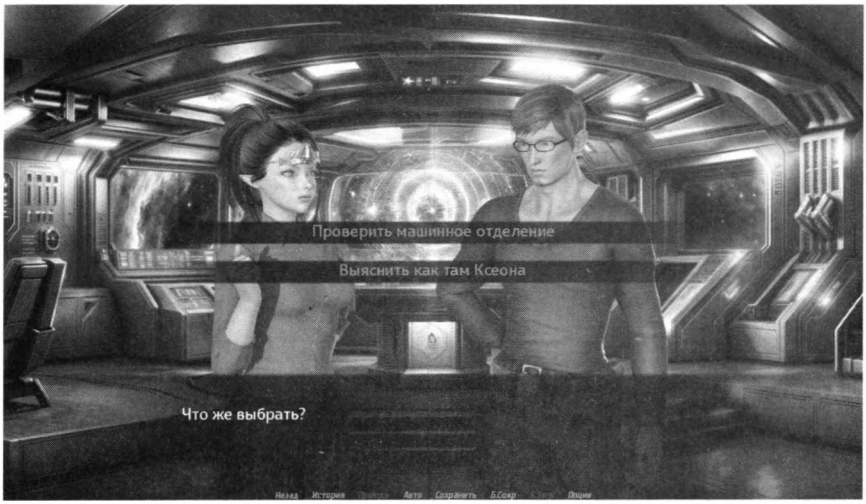


Рис. 4.35.

Код:

```
menu:
    "Что же выбрать?" # Вопрос или реплика
    "Проверить машинное отделение":
        jump engine
    "Выяснить, как там Ксеона":
        jump medcenter
```

Обратите внимание, что вопрос или реплика не имеют двоеточия, так как не подразумевают дополнительных действий, но являются частью блока **menu**, поэтому сдвинуты вправо относительно него.

Блокам меню, как и меткам, можно присваивать собственное имя, чтобы "прыгать" к ним по необходимости.

```
menu first_choice:
    "Что же выбрать?" # Вопрос или реплика
    "Проверить машинное отделение":
        jump engine
    "Выяснить, как там Ксеона":
        jump medcenter
```

Это можно использовать для перехода к выбору из разных сюжетных ответвлений с помощью всё той же команды **jump first_choice**. Или, если один из ответов предполагает отказ персонажа, который предложит выбрать что-то другое.

```
menu first_choice:
    "Что же выбрать?"
    "Проверить машинное отделение":
        jump engine
    "Выяснить, как там Ксеона":
        jump medcenter
    "Давай останемся здесь": # Третий ответ
        lia "Нет, мы должны что-то предпринять"
        jump first_choice # Возвращаем к выбору
```

> **ПРИМЕЧАНИЕ**. Также имейте в виду, что любые названия меток, меню выборов, переменных и пр., состоящие из двух и более слов, должны писаться либо **слитно**, либо через **нижнее подчёркивание** (пример: first_choice). Если вы разделите название на два слова, второе будет считаться отдельным объектом, который неизвестен движку. Это вызовет ошибку.

Если вместо подчёркивания использовать символ **минуса** или **тире** (first-choice), это будет восприниматься движком, как математическая операция, где вы предлагаете ему вычесть из первого слова второе, что, естественно, невозможно.

4.7. Переменные: запоминаем выбор игрока

Переменные являются фундаментальным понятием, поскольку они позволяют хранить и обрабатывать данные. В Ren'Py переменные представляют собой контейнеры, хранящие определённые значения, которые могут быть изменены в процессе игры.

Чтобы создать переменную, нужно придумать ей название, и присвоить необходимое значение. Например, **var = 1**. Мы создали переменную с именем **var**, которой присвоили (=) значение 1. Помимо чисел, переменные могут хранить в себе строки, списки, булевы значения, и другие объекты.

Необходимо запомнить, что символ "=" не является знаком равенства. Начинающие разработчики часто путают эти понятия. В Ren'Py, Python и многих других языках, это оператор присвоения. С его помощью мы присваиваем переменной определённое значение. Чтобы поставить знак равенства, используется два символа равно "==".

Как можно использовать переменные? Проводить математические операции для подсчёта каких-либо событий в игре, например: плюсовать и минусовать отношения персонажа к главному герою, подсчитывать очки навыков персонажа (сила, ловкость и пр.), запоминать выбор игрока в блоках **menu**, открывать или скрывать дополнительные варианты ответов, и многое другое.

Создать переменные в Ren'Py можно несколькими способами. Основным является установка через ключевое слово *default*, так как это внутренний инструмент движка.

```
# Примеры создания переменных
default name = "Лиара"
default liara_rank = "Старпом "
default liara_age = 112
default liara_ratio = 5.0
default liara_roster []
```

Второй и третий варианты создают переменные непосредственно с помощью Python.

Для создания или изменения переменных внутри меток (label) используется символ $.

Этот знак сообщает движку, что всё, написанное в этой строке, является кодом на языке Python.

```
label start:
    $ hero_name = "Кайден"
    $ kayden_rank = "Инженер "
    $ kayden_age = "35"
```

Данный способ менее предпочтителен для создания переменных, так как они не будут существовать в игре до тех пор, пока игрок не дойдёт до определённой метки, в которой эти переменные объявляются. Такое может произойти, если мы в одной метке создадим переменную, а игрок пройдёт мимо неё по второстепенному ответвлению, через другие метки.

В этом случае, все возможные действия с переменной будут приводить к ошибке, так как движок о ней не знает, а вы пытаетесь сделать проверку или провести с ней математические операции. Как правило, переменные через **$** внутри меток создаются, если они используются один раз, здесь и сейчас, и больше нигде не понадобятся.

То есть, если вы решили ввести переменные этим способом, нужно показать их движку на старте игры, в **label start**, или в другом лейбле, который точно будет пройден игроком. Этот способ нужен в основном для **изменения значений** уже существующих переменных по ходу игры.

Третий вариант создания переменных – прописать их в блоке *init python*, как в примерах при создании эффектов и позиций персонажей.

```
init python:
    hero2_name = "Ксеона"
    xeona_rank = "Медик"
    xeona_age = "324"
```

В вышеприведённом примере отсутствует знак **$**, так как слово **python** указывает движку, что весь код в этом блоке написан на этом ЯП.

Помимо обычных переменных, существуют постоянные переменные. Они отличаются от обычных тем, что не сбрасываются к дефолту (по умолчанию) при старте новой игры, а постоянно сохраняют свои значения. С постоянными переменными мы с вами уже сталкивались ранее, когда настраивали громкость звуковых каналов.

Благодаря их работе игрокам не нужно при запуске игры снова настраивать громкость, постоянные переменные запоминают установленное значение. Но, всё же в любой момент могут быть изменены. Постоянные переменные используются не только для настройки звуков, но и для других параметров, в рамках всего проекта.

Например, мы можем сохранять в них различные **ачивки** (достижения) пользователя в игре. Если игрок получил ачивку за определённое событие, то при повторном прохождении она будет считаться уже полученной. Постоянные переменные прописываются в коде также как и обычные, но только с соответствующим указанием.

```
default persistent.part = 1
default persistent.spaceship = "Firefly"
default persistent.achievement = False
```

Префикс persistent сообщает движку, что переменная является *постоянной*.

Также стоит иметь в виду, что переменные нельзя называть ключевыми словами, такими как *True*, *False*. Эти слова зарезервированы системой, и используются по другому назначению. Кроме того, не стоит называть переменную словом, если оно подсвечивается в редакторе кода цветом оператора.

Все ключевые слова: **True**, **False**, **None**, **and**, **as**, **at**, **assert**, **break**, **class**, **continue**, **def**, **del**, **elif**, **else**, **except**, **finally**, **for**, **from**, **global**, **if**, **in**, **import**, **is**, **lambda**, **nonlocal**, **not**, **or**, **pass**, **raise**, **return**, **try**, **while**, **with**, **yield**.

После того как мы научились создавать переменные, с их помощью можно сохранять разные данные в игре. Например, давайте запомним выбор игрока.

```
# По дефолту переменная fork1 хранит пустую строку
default fork1 = ""
# Игра начинается здесь:
label start:
    menu first_choice:
        "Что же выбрать?"
        "Проверить машинное отделение":
            $ fork1 = "машинное" # Запомнили выбор
            jump engine
        "Выяснить как там Ксеона":
            $ fork1 = "медцентр" # Запомнили выбор
            jump medcenter
```

При выборе игрока мы присваиваем переменной **fork1** определённое значение. Когда пользователь выберет первый вариант, переменная сохранит слово "машинное", если будет выбран второй вариант, соответственно, слово "медцентр". То есть, переменная вместо пустой строки " ", теперь хранит слово, по которому мы определим, куда направился игрок.

Обратите внимание, что операция с переменной является частью блока выбора, поэтому должен быть соответствующий отступ вправо. Также стоит иметь в виду, что команда **jump** всегда должна выполняться последней в списке задач. Помним, что движок спускается построчно сверху в низ. Сначала он производит первую операцию после выбора игрока. В нашем случае это присвоение значения переменной. Затем Ren'Py выполняет следующую задачу **jump** – перейти к указанной метке.

Если в начале списка будет стоять команда **jump**, а после неё любая другая задача, то движок прыгнет по указанной метке и не увидит нижеследующих операций этого списка.

Ещё одним вариантом сохранения выбора является изменение переменной в самой метке перехода.

```
label engine: #Перешли к метке, если выбрали первый ответ
    $ fork1 = "машинное" # Запомнили выбор
    scene engine with fade

label medcenter: #Перешли к метке, если выбрали второй ответ
    $ fork1 = "медцентр" # Запомнили выбор
    scene medcenter with fade
```

Таким образом, переменная получает новое значение только при попадании игрока в определённое место.

Теперь давайте рассмотрим пример с числовыми значениями, где мы можем изменять отношения персонажей, навыки и прочие параметры, которые можно посчитать.

```
default fork1 = ""
default liara_relat = 1 # Отношения с Лиарой
default xeona_relat = 1 # Отношения с Ксеоной
default kayden_engin = 5 # Навык инженерного дела у Кайдена
# Игра начинается здесь:
label start:
    menu first_choice:
        "Что же выбрать?"
        "Проверить машинное отделение":
            $ fork1 = "машинное" # Запомнили выбор
            $ liara_relat = liara_relat + 1 # Повысилось отношение
            $ kayden_engin = kayden_engin + 1 # Улучшился навык
            jump engine
        "Выяснить как там Ксеона":
            $ fork1 = "медцентр" # Запомнили выбор
            $ xeona_relat = xeona_relat + 1 # Повысилось отношение
            jump medcenter
```

При выборе машинного отделения мы запомнили выбор игрока, увеличили отношение с Лиарой, а Кайден повысил свой навык инженерного дела. При выборе второго варианта мы только улучшили отношения с Ксеоной.

Как можно увидеть, принцип изменения всех трёх переменных одинаков. Каждая переменная по умолчанию хранит **определённое значение** (default). Во время выбора игрока им присваивается новое значение. Мы указываем движку, что к текущему значению переменной *liara_relat* нужно прибавить одно очко. С примерами Кайдена и Ксеоны ситуация аналогичная.

Ещё один тип переменных – булевы переменные. Они могут принимать только два возможных значения: *истину* (*true*) или *ложь* (*false*). Часто они используются для оценки условий.

Например, если условие истинно, выполняются определённые действия.

```
default persistent.achievement1 = False
default persistent.achievement2 = False
# Игра начинается здесь:
label start:
    menu first_choice:
        "Что же выбрать?"
        "Проверить машинное отделение":
            $ fork1 = "машинное"
            $ liara_relat = liara_relat + 1
            $ kayden_engin = kayden_engin + 1
            $ persistent.achievement1 = True # Достижение 1
            jump engine
        "Выяснить как там Ксеона":
            $ fork1 = "медцентр"
            $ xeona_relat = xeona_relat + 1
            $ persistent.achievement2 = True # Достижение 2
            jump medcenter
```

По умолчанию у нас заявлены две переменные со значением *False*, каждая из которых открывает игровую ачивку (достижение). При выборе первого варианта разблокируется первое достижение, значение переменной изменяется на *True*. При выборе второго, соответственно, открывается второе достижение.

Аналогично языку Python, Ren'Py предлагает тот же набор операторов для работы с переменными.

Арифметические операторы

+ Плюс

- Минус

***** Умножить

/ Разделить

// Целочисленное деление. Двойной слэш при делении выводит только целое число, и отбрасывает число после запятой. Пример: 5//2=2 (от 2,5 была отброшена дробная часть).

% Остаток деления. При делении получаем только остаток. Пример 5%2=1. Четыре на два делится без остатка, остаётся единица.

****** Возведение в степень. 5**3=125 – пять в третьей степени.

Операторы сравнения

== Равно. 2==2
!= Не равно. 5!=2
> Больше. 5>2
< Меньше. 2<5
>= Больше или равно. 5>=2
<= Меньше или равно. 2<=5

Логические операторы

and оператор "и". Числа 5 and 2
or оператор "или". Число 5 или 2
not оператор "нет".

Операторы присваивания

= Присваивание. *liara_age = 112*, переменной присвоили число 112.
+= Прибавить к переменной. *liara_age +=1*, прибавляет единицу к числу, которое уже храниться в переменной.
-= Отнимает указанное число от переменной.
***=** Умножает число в переменной на указанное.
/= Делит число в переменной на указанное.

Согласно вышеописанным операторам, мы можем немного сократить код во время присвоения значений.

Вместо:

```
$ liara_relat = liara_relat + 1
```

Мы можем записать то же самое в виде:

```
$ liara_relat += 1
```

Вместе с тем, можно проводить различные подсчёты переменных:

```
kayden_relat = liara_relat + xeona_relat
```

Отношение Кайдена будет являться суммой переменных Лиары и Ксеоны.

Чтобы увидеть значение переменной в игре, используется тег квадратных скобок **[]**, внутри которых нужно написать название переменной, значение которой нужно показать.

```
label engine:
    scene engine with fade
    show liara serious at lcenter
    lia "Твоё мастерство достигло уровня [kayden_engin]" #
    lia "Скоро ты сможешь чинить антигравитационные двигатели."
```

Вместо переменной в квадратных скобках [kayden_engin] будет показано значение, которое она хранит. Таким образом, можно увидеть, что при выборе переменная действительно изменилась.

Рис. 4.36.

В следующих главах этот тег пригодится нам при создании элементов интерфейса, в которых можно отображать скилы (навыки) персонажей, их взаимоотношения и прочую информацию, которая меняется во время прохождения. А в настоящий момент его уже можно применить, например, в характеристиках персонажей.

В разделе "4.1. Настройка персонажа" мы познакомились с понятием префикса, благодаря которому можем подставлять к имени дополнительное значение.

```
define lia = Character('Лиара', who_color = "#c8ffc8", who_prefix="Старпом " )
```

Создайте переменные с должностями персонажей, если они ещё не созданы:

```
default liara_rank = "Старпом"
default kayden_rank = "Инженер"
default xeona_rank = "Медик"
```

Теперь мы можем добавить их должности на корабле в виде префиксов имён.

```
define lia = Character('Лиара', who_prefix="[liara_rank] ")
define kay = Character('Кайден', who_prefix="[kayden_rank] ")
define xeo = Character('Ксеона', who_prefix="[xeona_rank] ")
```

Далее, во время прохождения их должности могут меняться, и эти изменения будут автоматически отображены в имени.

```
label engine:
    lia "Твоё мастерство достигло уровня [kayden_engin]"
    lia "Скоро ты сможешь чинить антигравитационные двигатели."
    $ kayden_rank = "Ст. инженер" # Изменили должность
    kay "Кажется, с двигателем всё в порядке"
```

В строке *$ kayden_rank = "Ст. инженер"* мы присвоили персонажу новую должность, и теперь в игре это отражено:

Рис. 4.37.

4.8. Проверки и условия

В предыдущем разделе мы узнали как сохранять различную информацию с помощью переменных. Но это только половина дела. Теперь давайте разберемся, как этой информацией пользоваться.

В Ren'Py аналогично языку Python существует система проверок, которая может включать в себя различные условия. Условия создаются с помощью условных операторов.

- **if** – используется для проверки определённого условия и выполнения блока кода, если это условие истинно.
- **elif** – создаёт дополнительное условие для проверки, если первое условие не является истинным.
- **else** – определяет блок кода, который будет выполнен, если ранее заданные условия не выполнились.

```
label engine:
    scene engine with fade
    if kayden_engin > 6: # Проверяем навык Кайдена
        show kayden serious
        kay "Кажется, в боковом блоке трещина"
        kay "Я исправлю это за пять минут"
    show kayden normal
    kay "Кажется, с двигателем всё в порядке"
```

Строку "if kayden_engin > 6:" по-русски можно прочитать как:

```
если (if) навык Кайдена (kayden_engin) больше (>) 6, то (:)
```

То есть, происходит проверка навыка Кайдена, в зависимости от которого будут показаны новые строки диалога или нет. Мы говорим движку: сравни навык Кайдена с определённым числом, и если навык больше, то выполни следующее. Ниже с отступом вправо мы пишем, что нужно показать спрайт *kayden serious* и его реплики.

Если навык Кайдена будет меньше или равен этому числу, то условие не будет выполнено, и все строки, относящиеся к условию (смещённые вправо),

будут пропущены. Нам сразу покажется спрайт *kayden normal* и следующие за ним строки кода.

Второй пример:

```
label engine:
    scene engine with fade
    if kayden_engin == 6: # Проверяем навык Кайдена
        show kayden serious
        kay "Кажется, в боковом блоке трещина"
        kay "Я исправлю это за пять минут"
    elif kayden_engin >= 7: # Проверяем навык Кайдена
        show kayden sad
        kay "Хм… здесь очень серьёзная поломка"
        kay "Мне понадобится полчаса"
    show kayden normal
    kay "Кажется, с двигателем всё в порядке"
```

Теперь в нашей проверке появилось ещё одно условие **elif**. Его можно понимать как "в противном случае". Мы указываем движку проверить первое условие: если навык Кайдена равен **6**, выполни условия после двоеточия; в противном случае сделай следующую проверку, и выполни другие условия после двоеточия.

Получается, если навык равен 6, выполнится этот блок. А блок **elif** будет пропущен.
Если навык не равен 6, произойдёт вторая проверка.
Если навык больше или равен 7, выполнится код после двоеточия.
Если навык окажется меньше, этот блок тоже будет пропущен.

Подобных проверок можно сделать любое необходимое количество:

```
label engine:
    scene engine with fade
    if kayden_engin == 6: # 1-я проверка
        show kayden serious
    elif kayden_engin == 7: # 2-я проверка
        show kayden sad
    elif kayden_engin == 8: # 3-я проверка
        show kayden sad
    elif kayden_engin == 9: # N-я проверка
        show kayden sad
    show kayden normal
    kay "Кажется, с двигателем всё в порядке"
```

Оператор **else** выполняет условие во всех остальных случаях, если не совпало ни одно из событий выше.

```
label engine:
    scene engine with fade
    if kayden_engin == 6: #
        show kayden serious
    elif kayden_engin >= 7: #
        show kayden sad
    else:
        show kayden normal
        kay "Кажется, с двигателем всё в порядке"
```

> **ПРИМЕЧАНИЕ.** Важно помнить, что оператор **else** никогда не делает никаких сравнений. После его указания сразу идёт *двоеточие*, и выполняется код, относящийся к его блоку.

Else может быть полезен в тех случаях, когда у вас много разных условий, и вы не можете предусмотреть все варианты событий. Тогда во всех остальных случаях, не прописанных в **if/elif**, укажите, что должно произойти.

Помимо простых проверок, с одним сравнением, можно прописывать целый ряд условий, которые должны совпасть, чтобы событие состоялось.

```
label medcenter:
    if xeona_relat > 1 and part == 1:
        xeo "Кайден, как ты себя чувствуешь?"
    else:
        xeo "Наконец-то ты пришёл"
```

В данном примере, если отношения с Ксеоной больше единицы **и** игрок находится в первой части игры, то она спросит Кайдена, как он себя чувствует. Иначе, реплика будет другая. То есть, первая реплика будет произнесена, если оба условия будут выполнены.

```
label medcenter:
    if xeona_relat > 1 or part == 1:
        xeo "Кайден, как ты себя чувствуешь?"
    else:
        xeo "Наконец-то ты пришёл"
```

Теперь Ксеона скажет первую реплику, если одно из условий будет выполнено. Если уровень отношений достаточный **или** первая часть игры.

```
label medcenter:
    if not xeona_relat > 1 and part != 1:
        xeo "Кайден, как ты себя чувствуешь?"
    else:
        xeo "Наконец-то ты пришёл"
```

В этом примере событие состоится, если отношение Ксеоны **не** больше единицы **и** переменная *part* не равна одному.

```
label medcenter:
    if (xeona_relat > 1 and part == 1) or (fork1 == "медцентр"):
        xeo "Кайден, как ты себя чувствуешь?"
    else:
        xeo "Наконец-то ты пришёл"
```

Здесь мы предусматриваем несколько блоков, где событие произойдёт, если совпадут условия в первых или во вторых скобках. Таким образом, можно проверять большое количество переменных в одной строке.

Ещё один вариант проверки нескольких переменных выглядит так:

```
label medcenter:
    if xeona_relat > 1:
        if part == 1:
          xeo "Кайден, как ты себя чувствуешь?"
    else:
        xeo "Наконец-то ты пришёл"
```

Сначала мы проверяем отношения с Ксеоной, и только в случае успеха, происходят следующие проверки. Однако избегайте большого количества вложений, это ухудшает читаемость кода.

Стоит иметь в виду, что условия в блоках **меню выбора** прописываются иначе:

```
label start:
    menu:
        "Проверить машинное отделение":
            $ fork1 = "машинное"
            jump engine
        "Выяснить, как там Ксеона":
            $ fork1 = "медцентр"
            jump medcenter
        "Пойти в каюту капитана" if kapitan==True:
            $ fork1 = "каюта_капитана"
            jump cabin
```

Как видим, в данном случае проверка условий находится в той же строке, что и вариант ответа. Это просто нужно запомнить. Таким образом, можно создавать секретные варианты ответов, которые будут отображаться, только если игрок выполнил определённое условие.

В Ren'Py также есть возможность показывать скрытые ответы, но без возможности нажать на них, пока условия не будут выполнены.

Рис. 4.38.

Неактивный вариант отображается на экране, подсказывая игроку, что есть альтернативное прохождение этого эпизода. По умолчанию эта функция

отключена в движке, но её можно активировать, дописав соответствующую настройку конфигурации.

Переменную можно поместить в любой файл, но для соблюдения порядка пропишите команду в *options.rpy*:

```
define config.menu_include_disabled = True
```

4.9. Ввод текста, изменение имени персонажа

Главному герою и любым другим персонажам в игре можно присваивать и менять личные имена. Более того, можно предложить сделать это игроку, чтобы он назвал персонажа, к примеру, своим именем.

Ранее, с помощью переменных, мы подставляли героям префиксы их должностей на корабле, которые можно изменять во время прохождения. Так, например, Кайден был повышен из инженеров до старшего инженера. По аналогии с этим, замените имя персонажа переменной, которая позже сохранит введённое имя.

```
define kay = Character('[name_mc]', who_color="#ccffff", who_prefix="[kayden_rank] ")
```

Вместо имени Кайдена прописана переменная **name_mc**. Теперь, в нужном месте игры, мы предложим игроку ввести имя для главного героя:

```
label start:
    scene bridge with fade
    lia "Ты потерял сознание после взрыва!"
    lia "Помнишь, как тебя зовут?"
    $ name_mc = renpy.input ("Введите имя или оставьте текущее")
    if name_mc == "":
        $ name_mc = "Кайден"
    kay "Меня зовут [name_mc]."
```

В строке *$ name_mc = renpy.input()* мы запускаем функцию ввода с клавиатуры. Введённый текст будет присвоен переменной **name_mc**.
Далее запускается проверка: если переменная хранит пустую строку (игрок ничего не ввёл и нажал **Enter**), то присваивается стандартное имя (Кайден).

В последней строке мы отображаем значение переменной с помощью тега **[]**.

Рис. 4.39.

После этого во всех диалогах от имени персонажа будет подставляться новое имя.

Рис. 4.40.

Также мы можем ограничить длину введённого имени, чтобы оно оставалось в разумных пределах, и указать допустимые символы, чтобы не было возможности назвать персонажа, к примеру, сотней пробелов и символами: *@$#&.

```
$ name_mc = renpy.input("Введите имя", length=10, allow="abcdefg")
```

length=10 – указывает максимальное количество символов.
allow="abcdefg" – в кавычках прописываются допустимые символы.

Для краткости, в примере указано несколько букв английского алфавита, но можно перечислить весь алфавит в нижнем и верхнем регистре, а также алфавиты других языков.

Подобную конструкцию ввода имени можно сделать для каждого персонажа, и в любой момент игры давать изменять имя на новое, если это необходимо. Таким же образом можно вводить любую информацию и сохранять её в переменных.

4.10. Расширенные настройки звуков и музыки

Помимо базовых команд для воспроизведения звуков, музыки и голоса, с которыми мы познакомились в разделе "2.6. Музыкальное сопровождение", существует целый пул дополнительных опций. Многие из них, как правило, не используются большинством разработчиков, однако, благодаря этим настройкам можно тщательно срежиссировать каждый эпизод игры.

Следующие команды отвечают за направление звука:

- **$ renpy.music.set_pan(-1, 1) #** Проигрывание музыки только с левой стороны, будь то колонка или наушник.
- **$ renpy.music.set_pan(1, 2) #** Проигрывание музыки только с правой стороны.
- **$ renpy.music.set_pan(0, 1) #** Проигрывание музыки стандартно, с обеих сторон.

В коде это выглядит так:

```
label start:
    play music kosmos fadein 3 # Воспроизвели композицию
    scene bridge with fade
```

```
    lia "Ты потерял сознание после взрыва!"
    $ renpy.music.set_pan(-1, 1) # Сместили звук влево
```

То есть, в определённый момент звук можно сместить в одну сторону, затем к центру, и в конце в противоположную. Это может создать эффект, как будто что-то пролетает с одной стороны в другую.

Следующая опция – это возможность создания альбомов. Она может понадобиться, если фоновая музыка не так важна, и, чтобы не прописывать по ходу сценария включение и выключение композиций (**play music**, **stop music**), можно собрать их все в альбом, и запустить одной командой.

```
label start:
    queue music ["audio/music1.mp3", "audio/music2.mp3"]
```

В квадратных скобках через запятую пропишите все необходимые композиции, и они будут проигрываться по очереди. Когда список подойдёт к концу, он начнётся сначала.

Чтобы плейлист воспроизвёлся один раз, вместо *queue music* пропишите *queue sound*. Для случайного воспроизведения, список можно перемешать:

```
init python:
    playlist = ["music1.mp3", "music2.mp3", "music3.mp3"]
    renpy.random.shuffle(playlist)
label start:
    play music playlist fadeout 2.0 fadein 2.0
```

В примере выше мы создали переменную **playlist** со списком композиций, а в следующей строке с помощью метода **shuffle** перемешали указанный в скобках список. Затем в любом нужном месте игры запускаем наш плейлист с плавным воспроизведением и затуханием композиций в нём.

Для добавления озвучки в вашу игру, потребуются непосредственно звуковые файлы с записанными фразами персонажей, которые потом вызываются аналогично музыке и звукам.

```
label start:
    voice "audio/liara_replica1.mp3" # Голос персонажа
    lia "Ты потерял сознание после взрыва!"
```

Ещё один элемент звукового дизайна – это управление громкостью музыки во время игры:

```
label start:
    play music kosmos fadein 3 # Воспроизвели композицию
    $ renpy.music.set_volume(0.4) # Установили громкость
    lia "Ты слышишь?"
    $ renpy.music.set_volume(0.7) # Звук нарастает
    lia "Что это за звук?"
    $ renpy.music.set_volume(1.0) # Нарастает
    kay "Кажется, с двигателем что-то не так!"
    $ renpy.music.set_volume(0.7) # Затихает
    kay "Какие-то странные вибрации…"
    $ renpy.music.set_volume(0.4) # Ещё тише
    kay "Хм… теперь всё в порядке."
```

С помощью команды *renpy.music.set_volume* можно также управлять громкостью звуков и голоса, заменив соответствующий миксер звука на sound и voice, соответственно.

Также можно воспроизводить композиции не полностью, а лишь необходимый фрагмент.

```
label start:
    play music "<from 34 to 56>audio/kosmos.mp3"
```

В примере выше музыка начнёт проигрываться с 34-ой секунды и продлится до 56-ой.

- Команда **play music "<from 34>audio/kosmos.mp3"** просто запустит композицию с 34-ой секунды.

- Команда **play music "<loop 34>audio/kosmos.mp3"** воспроизведёт песню полностью, а затем повторит её с 34-ой секунды
- Команда **play music "<from " +str(renpy.random.randint (34, 56)) + ">audio/kosmos.mp3"** воспроизведёт трек из случайного места в пределах 34-ой и 56-ой секунды. Таким же образом можно указать случайное место завершения композиции.

4.11. Система *Multiple* и диалоговые пузыри

В Ren'Py система **multiple** используется для отображения мульти-диалога – ситуации, когда несколько персонажей говорят одновременно или друг за другом с быстрой сменой реплик.

Рис. 4.41.

Код:

```
label medcenter:
    scene medcenter with fade
    lia "Вы не видели капитана?"
    xeo "Он в своей каюте" (multiple=2)
    kay "Он в машинном отделении" (multiple=2)
```

В текущем примере Ксеона и Кайден одновременно отвечают на вопрос Лиары. Команда **multiple** в конце реплики указывает, какие диалоги нужно выводить вместе. Однако в стандартном варианте они будут наложены друг на друга, и, чтобы вывести один выше другого, нужно создать дополнительный стиль для дублирующего диалога.

Со стилями мы познакомимся немного позже, а сейчас давайте откроем файл *screens.rpy*, и где-нибудь в самом низу напишем стиль отображения для нового окна. Стиль представляет собой копию оригинального, за исключением позиции по "y" и названия.

```
style block2_multiple2_say_window:
    xalign 0.5
    xfill True
    yalign 0.65 # здесь меняем расположения окна по "y"
    ysize gui.textbox_height
```

В таком виде это всё равно выглядит не лучшим образом, так как диалоговые окна закрывают пол экрана. Поэтому можно перенастроить стили обоих окон, указав новые позиции, и заменить изображения для диалогов. Напомню, что оригинальный бэкграунд текстбокса (элемент графического интерфейса пользователя, предназначенный для ввода небольшого объёма текста без переноса строк) находится в папке **gui/textbox.png**.

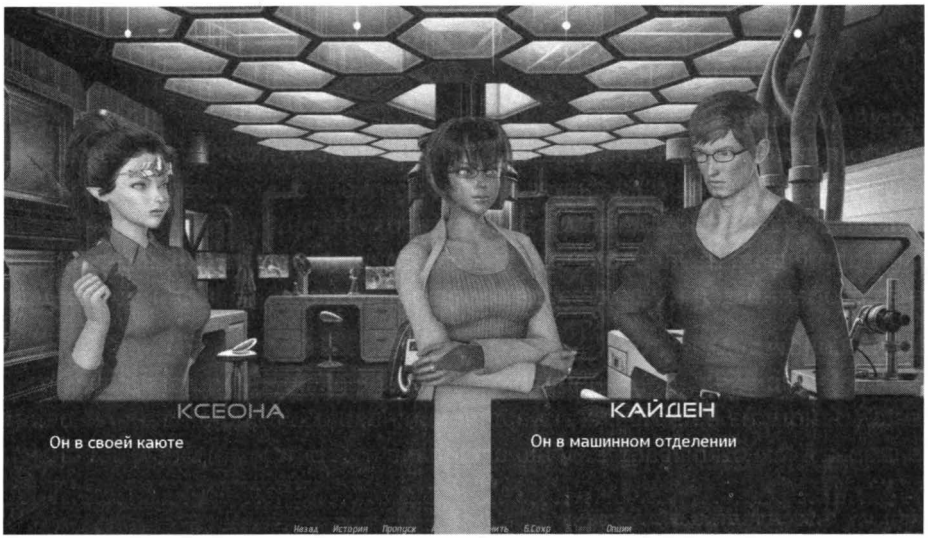


Рис. 4.42.

Теперь персонажи имеют два небольших окна для своих реплик. По аналогии с этим, можно придать обоим текстбоксам вид диалоговых пузырей, которые обычно используются в комиксах. И таким образом можно стилизовать игру в подобном формате.

Рис. 4.43.

Однако в 8-й версии Ren'Py разработчик движка решил облегчить жизнь начинающим создателям игр и внедрил функционал диалоговых пузырей. Теперь, помимо системы **multiple**, можно создавать диалоги с их помощью. В папке **gui** вы можете найти стандартные шаблоны этих изображений, и, по необходимости, переделать под стиль вашей игры. Например, перекрасить их в другой цвет и сделать полупрозрачными.

Чтобы переключить режим диалогов с простого окна на речевые пузыри, нужно прописать в характеристиках персонажей соответствующий параметр: *kind=bubble*. Также для корректной работы необходимо указать группу изображений, которые будут связаны с этими персонажами: image="liara".

```
define lia = Character('Лиара', kind=bubble, image="liara")
define kay = Character('Кайден', kind=bubble, image="kayden")
define xeo = Character('Ксеона', kind=bubble, image="xeona")
```

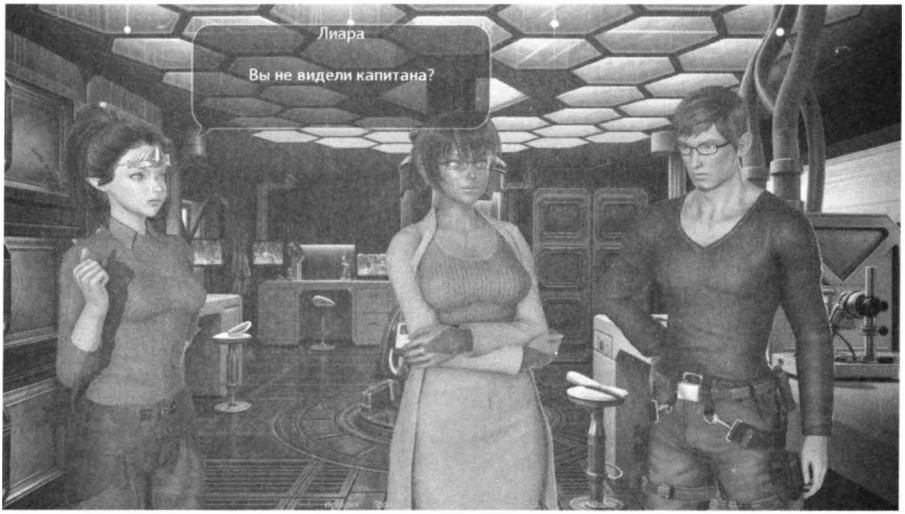


Рис. 4.44.

По умолчанию диалоговый пузырь настроен для спрайта, который будет выводиться по центру экрана. Поэтому каждое отдельное окно нужно подстраивать вручную для каждого персонажа. Для этого есть инструмент, позволяющий отрисовывать мышкой позицию пузыря на экране. Вызывается он комбинацией клавиш **Shift + B**, после чего в верхнем левом углу отобразится окошко с настройками.

Рис. 4.45.

Первая настройка сообщает координаты, по которым располагается речевой пузырь, а вторая отвечает за расположение хвостика диалогового окошка. Щёлкните мышкой несколько раз по этой настройке, чтобы увидеть, как это работает.

После этого для выбранного персонажа необходимо указать место отображения его пузыря. Зажмите левую клавишу мыши и выделите область, на которую переместиться окно. Если попытка оказалась не удачной, и есть необходимость немного исправить расположение или размер, снова нажмите на координаты персонажа (первая настройка), и выделите новую область.

Рис. 4.46.

Установленные позиции окошек будут сохранены для каждого персонажа для всех следующих диалогов в этой сцене, однако в каждом конкретном моменте есть возможность перенести пузырь на новое место.

> **ПРИМЕЧАНИЕ**. Стоит иметь в виду, что с диалоговыми пузырями не получится добавить сайд изображения, так как этот инструмент не поддерживает данную функцию.

По завершении настройки новых диалоговых окон, в папке **game** будет создан файл *bubble.json*, он хранит позиции всех пузырей. Его можно открыть тем же редактором кода, которым вы пользуетесь, и, если необходимо, отредактировать каждую отдельную позицию вплоть до пикселя.

```
    "medcenter_97863cdf": {
      "area": [
          1040,
          450,
          320,
          225
      ],
      "properties": "top_left"
```

Каждый такой блок кода имеет строку идентификатор, определяющую, к какой реплике принадлежит этот пузырь. Его не стоит изменять, чтобы не нарушать привязку к диалогу. Далее идут позиции и размеры (координаты **x**, **y**, **ширина**, **высота**), их можно отредактировать для более точного расположения. Нижняя строка *"properties": "top_left"* отвечает за расположение хвостика. Его позицию также можно изменить по необходимости.

Вместе с тем есть возможность изменить ячейки сетки при выделении области на экране. Сделать их меньше, чтобы устанавливать более точные позиции без необходимости редактировать код. По умолчанию сетка состоит из 24-х столбцов и 24-х строк, которые равномерно занимают всю область экрана. За эти размеры отвечает переменные *"bubble.rows"* и *"bubble.cols"*. Давайте увеличим количество столбцов и строк в сетке, сделав, тем самым, каждую отдельную ячейку меньше.

```
init python:
    bubble.rows = 48
    bubble.cols = 48
```

4.12. Режим *NVL*

Режим **NVL** (Novel Visual Language) представляет собой систему отображения текста. Он имитирует стиль оригинальных японских визуальных новелл и отличается от стандартного режима **ADV** тем, что текст отображается на весь экран, а не в виде блоков диалога.

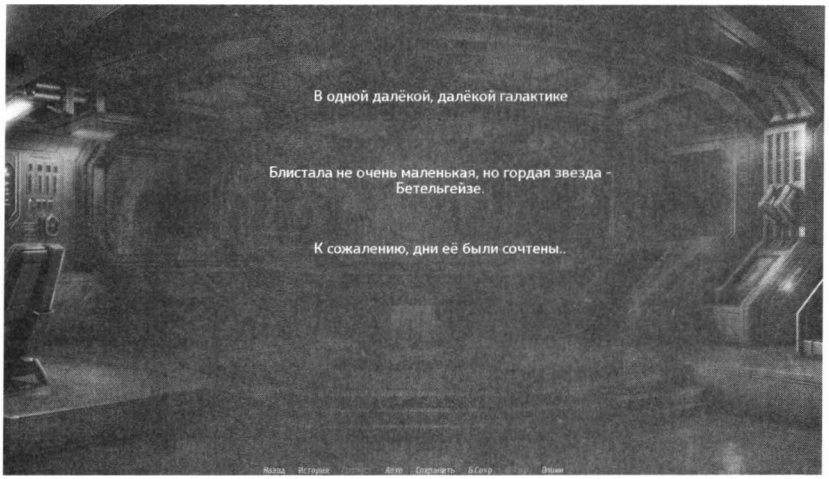


Рис. 4.47.

Режим NVL позволяет сосредоточить внимание игрока на повествовании, когда нужно вывести большой объём текста, не разбивая его на короткие фразы, как в стандартном варианте. NVL можно использовать как слова автора или финальные титры. Кроме того, задний фон диалогового окна можно оформить в виде развёрнутого свитка, имитируя текст письма, или экрана планшета.

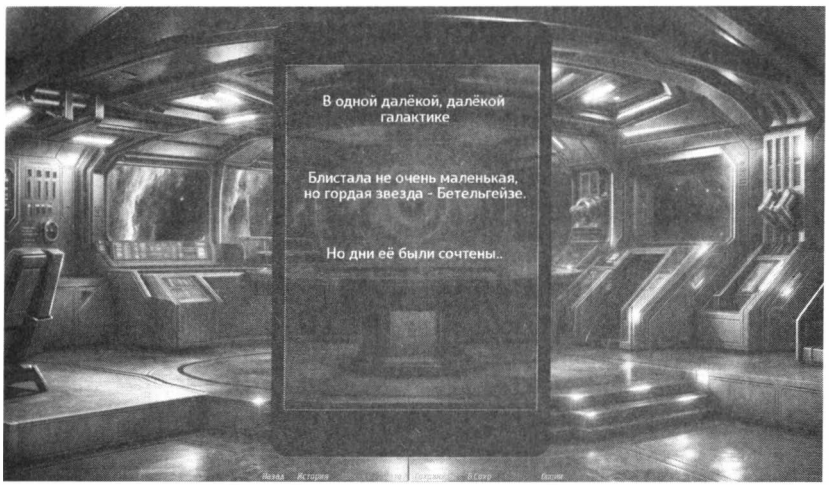


Рис. 4.48

Чтобы выводить текст в этом режиме, необходимо создать отдельных персонажей, которые будут иметь соответствующий атрибут. Для простоты, можно

скопировать характеры наших, уже созданных героев, и внести изменения.

```
define lianvl = Character ('Лиара', who_color = "#c8ffc8", kind=nvl)
define kaynvl = Character ('Кайден', who_color="#ccffff", kind=nvl)
```

> Атрибут **kind=nvl** сообщает движку, что реплики этих персонажей нужно выводить в режиме NVL.

Как и обычным характерам, им можно присвоить дополнительные атрибуты, которые были описаны в разделе "4.1. Настройки персонажа".

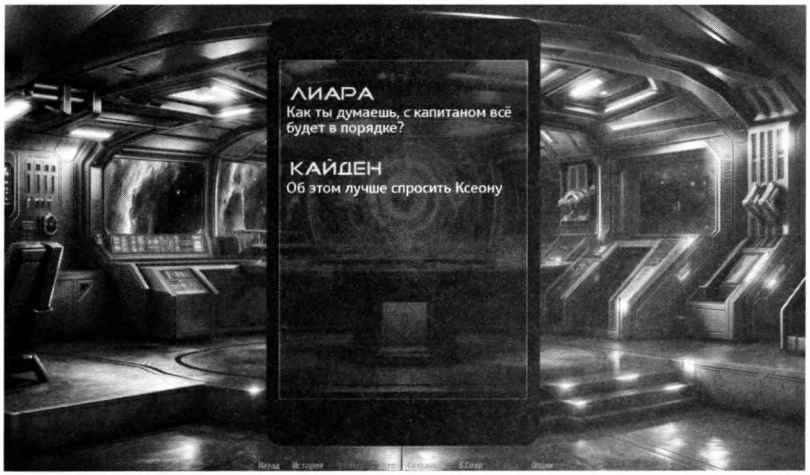


Рис. 4.49.

Чтобы перейти в режим NVL, достаточно просто продолжить диалог от имени новых персонажей **lianvl** и **kaynvl**, и Ren'Py сам переключит стандартный экран ADV на NVL.

```
label start:
    lianvl "Как ты думаешь, с капитаном всё будет в порядке?"
    kaynvl "Об этом лучше спросить Ксеону"
    lia "Мне кажется, она что-то замышляет"
    kay "Просто ты не переносишь Минбари"
```

Когда экран потребуется скрыть, нужно опять продолжить диалог от лица стандартных персонажей.

В отличие от обычного режима, в NVL реплики героев не исчезают сразу, а постепенно поднимаются выше, уступая место новым.

Если есть необходимость очистить экран, и вывести новый текст, используется команда **nvl clear**

```
label start:
    lianvl "Как ты думаешь, с капитаном всё будет в порядке?"
    nvl clear # Предыдущий текст будет удалён с экрана
    kaynvl "Об этом лучше спросить Ксеону"
```

4.13. Работа над ошибками

В процессе разработки каждый из нас время от времени допускает ошибки. Это могут быть как *синтаксические*, *логические*, так и ошибки связанные с отсутствием файла или чем-то ещё.

Логические ошибки возникают, когда программа компилируется и выполняется без проблем, но не производит ожидаемых результатов из-за ошибок в её логике.

Логические ошибки могут быть вызваны неправильным расчётом, неверным порядком операций, неправильным использованием операторов и т.д.

Синтаксические ошибки – это ошибки, возникающие из-за неправильного синтаксиса кода.

Например, незавершённые инструкции, неправильное использование ключевых слов, отсутствие двоеточия в конструкциях, непарные скобки и другие нарушения. Код с синтаксическими ошибками не может быть выполнен движком. Об этом он сообщит при попытке запустить проект.

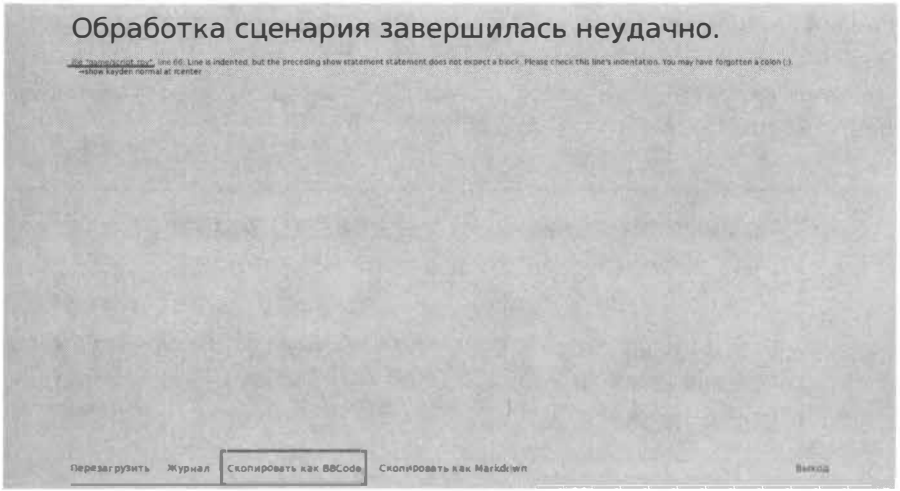


Рис. 4.50.

В окне с ошибкой будет указано название файла и номер строки, в которой она допущена. Здесь же можно прочитать описание проблемы. При клике на ссылку с ошибкой будет автоматически открыт нужный файл и выделена строка. Однако проблема не всегда может быть связана именно с этой строкой. Поэтому просмотрите весь блок кода, к которому она относится. Вероятно, где-то выше допущена ошибка, не позволяющая этой строке выполнить свою задачу.

Если вы не владеете английским языком, и самостоятельно не получается выяснить в чём же всё-таки проблема, можно скопировать текст ошибки, нажав на ссылку внизу. Текст добавится в буфер обмена. После этого откройте любой онлайн переводчик текста, и комбинацией клавиш **Ctrl+V** вставьте его в окно для перевода.

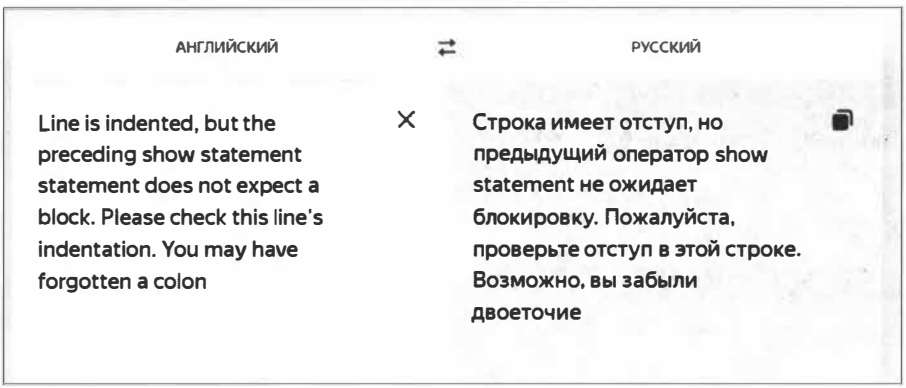


Рис. 4.51.

Как правило, большинство синтаксических ошибок решаются этим путём. Если описание ошибки ни о чём вам не говорит, остаётся вставить оригинальный текст (на английском) в строку поиска, и попытаться найти решение в интернете.

> Помимо этого, в Ren'Py есть **режим разработчика**, который активируется комбинацией клавиш **Shift+D**. В нём можно увидеть список команд, часть из них нам уже известна. При нажатии горячих клавиш **Shift+O** поверх игры открывается консоль, позволяющая управлять большинством процессов.

Например, при вводе команды **help** вы увидеть список инструкций, которые могут помочь в тестировании и отладке игры. При вводе в консоль **jump** и название метки, можно моментально перенестись к ней. Это полезно, когда проект становится очень большим, и нужно протестировать какой-то фрагмент игры, находящийся далеко от её начала.

Если прописать в консоли **watch** и название переменной, её значение отобразится в правом верхнем углу. Это также может пригодиться для контроля во время теста. Для скрытия переменной нужно ввести *unwatch* и её название.

Все команды консоли:

- **clear:** очищает историю консоли
- **escape:** включает экранирование символов юникода в строках юникода
- **exit:** выход из консоли
- **help:** справка по всем возможностям консоли
- **help expr:** показывает сигнатуру и документацию для указанного выражения
- **jump label:** переходит к метке
- **load slot:** загружает игру из указанного слота
- **long:** выводит полное представление объектов на консоль
- **reload:** перезагружает игру, обновляя скрипты

- **save slot:** сохраняет игру в указанный слот
- **short:** сокращает представление объектов в консоли
- **stack:** отображает стек вызовов функций
- **unescape:** отключает экранирование символов юникода
- **watch expression:** отображает указанное выражение
- **unwatch expression:** скрывает указанное выражение
- **unwatchall:** скрывает все выражения, выведенные ранее на экран

Кроме того, помимо инструментария движка, многие игровые элементы можно тестировать непосредственно вызвав их на экран в любом удобном месте, или прыгнув к ним командой **jump**.

Так, например, для теста определённой метки, которая находится очень далёко от начала игры, и у нас ещё нет сохранения рядом с этим моментом, можно просто прописать команду в **label start**.

```
label start:
    jump part_99 # переход к 99-му эпизоду игры со старта
```

Такой вариант выглядит проще, чем открытие консоли и пропись той же команды в ней. А после теста эта строка просто удаляется из скрипта.

Аналогичным образом можно сделать целую чит-панель на старте.

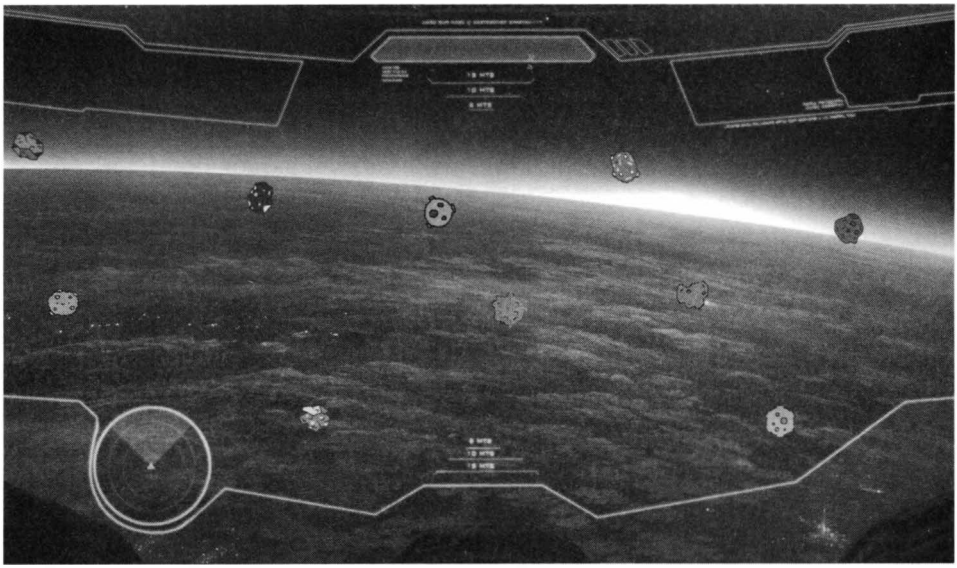
```
label start:
    menu cheat_panel:
        "99-ый эпизод":
            jump part_99 # переход к 99-му эпизоду игры со старта
        "64-ый эпизод":
            jump part_64 # переход к 64-му эпизоду игры со старта
        "Лиара +10":
            $ liara_relat += 10 # Накрутка отношений с персонажем
            jump cheat_panel # возвращаемся в чит-меню
        "Ксеона +10":
            $ xeona_relat += 10 # Накрутка отношений с персонажем
            jump cheat_panel # возвращаемся в чит-меню
        "Начало игры":
            jump part_1 # переходим к началу игры
```

На старте игры открывается меню выбора, в котором мы можем перейти к тесту одного из нескольких эпизодов или накрутить определённую переменную, например, чтобы проверить, как себя будут вести персонажи в том или ином случае.

Из-за особенностей меню выбора большое количество элементов для теста в нём прописывать неудобно. Так как без дополнительной настройки блока меню на экране может уместиться 9-10 строк выбора. В следующем разделе мы познакомимся с экранами в Ren'Py, благодаря которым можно сделать большое чит-меню для тестов и множество других полезных панелей, таблиц и кнопок.

Глава 5.

СИСТЕМА ЭКРАНОВ



5.1. Что такое экраны в Ren'Py

Система экранов в Ren'Py – это мощный инструмент, который позволяет создавать сложные и интерактивные интерфейсы в ваших проектах.

Практически всё, с чем мы взаимодействовали ранее, или будем взаимодействовать далее, работает благодаря этой системе.

Диалоговые окна режимов ADV, NVL, текстовых пузырей, главное меню, и остальной интерфейс игры состоят из экранов, включающих в себя настройки для вывода и отображения фонов, спрайтов, текста и прочих объектов. Благодаря экранам мы можем добавлять в проект собственные интерфейсы, такие как: таблицы, списки, инвентари, карты локаций, журнал заданий и пр.

Создание своих экранов происходит аналогично меткам. Без каких-либо отступов от края строки нужно прописать оператор **screen** и присвоить ему название:

```
screen example():
    add "deck"
    add "liara serious"
    text "Пример простого экрана"
```

Чтобы соблюдать порядок в файлах игры, для своих экранов, как правило, создаётся отдельный документ .rpy. Обратите внимание, что по умолчанию в папке **game** проекта уже существует *screens.rpy*, он включает в себя стандартный интерфейс. Желательно пока не трогать его, и создавать свои экраны в новых файлах.

В примере выше мы создали экран с названием *example*, скобками для параметров и двоеточием. На данном этапе дополнительных параметров у нас не будет, поэтому они остаются пустыми. А двоеточие сообщает, что это начало блока со списком инструкций. То есть, следующие команды мы прописываем с отступом, как в случае с лейбами (метками), чтобы движок понимал, что всё нижеследующее относится к этому блоку.

Внутри экрана мы указываем движку добавить (add) объект "deck", который является фоновой картинкой, затем объект "liara serious", являющийся спрайтом персонажа, и в конце просим вывести текст. В таком виде экран будет существовать в коде, но не использоваться игрой. Так как Ren'Py не знает, где и когда он должен быть отображён.

Для этого мы должны прописать соответствующую команду в одной из меток игры. В том месте, где нам нужно вывести этот экран.

```
label start:
    show screen example with fade
    pause
    hide screen example with dissolve
```

В данном примере мы вывели экран прямо в стартовом лейбе стандартной командой **show**. Для плавности появления и скрытия экрана можно использовать стандартные эффекты. В текущем варианте такое отображение картинок и текста мало чем отличается от обычного сценария в метках.

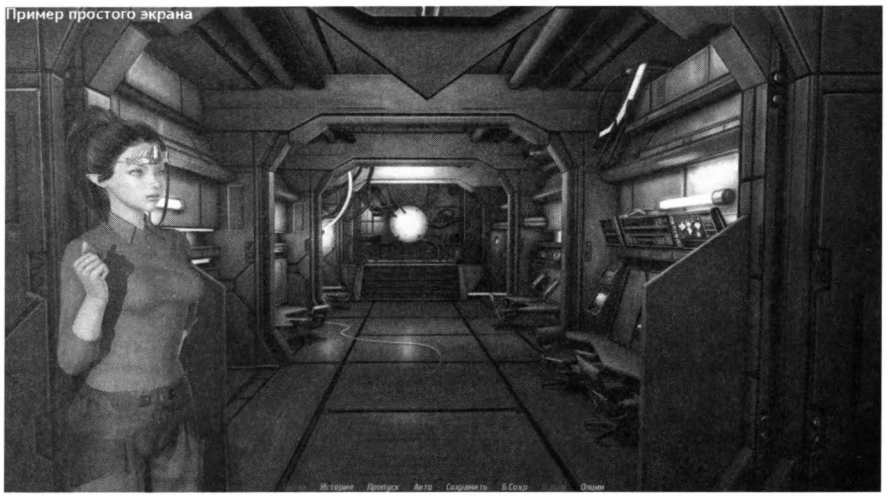


Рис. 5.1.

Однако, благодаря дополнительным свойствам и операторам, экраны можно переделывать под абсолютно разные задачи. Например, вывести на экран дополнительную информацию.

Рис. 5.2.

В данном случае вверху экрана отображается текст и кнопка для открытия журнала заданий. Сам же экран остаётся прозрачным, и мы можем продолжать двигаться по сюжету. Такой экран с интерфейсом можно отобразить в самом начале, и он будет висеть на протяжении всей игры, чтобы игрок мог отслеживать параметры персонажа и задачи, которые предстоит выполнить.

```
screen interface(): # экран интерфейса
    text _("Пример простого интерфейса или текущий квест")
    textbutton _("Журнал заданий"): # текстовая кнопка
        action Hide("interface"), Show("quest_log")
        # события при нажатии на кнопку
```

И первое, с чем мы познакомимся в данном примере, это тег нижнего подчёркивания с круглыми скобками **_()**. Он нужен, чтобы указать движку на текст, который следует учитывать при генерации файлов перевода.

В сценарии, который прописывается в **метках** (label), весь текст автоматически подхватывается движком для создания файлов локализации. Однако текст, расположенный в **экранах** (screen) не учитывается. Чтобы движок понимал, какие именно элементы экрана необходимо перевести, их обрамляют конструкцией _().

В примере этим тегом окружены _("Журнал заданий") и _("Пример простого интерфейса или текущий квест"). Эти строки будут добавлены для перевода. О самой системе локализации на другие языки мы поговорим в соответствующем разделе. Сейчас просто нужно завести себе привычку обрамлять весть текст в экранах скобками и нижним подчёркиванием.

Это стоит делать, даже если вы не планируете переводить игру. Ваши планы могут измениться в будущем. И тогда вам придётся редактировать все ваши файлы и экраны, добавляя в них этот тег. Кроме того, ваш проект может найти фанатов в других странах, которые сами захотят сделать локализацию на свой язык. И, проставляя тег для перевода, вы значительно облегчите им работу.

Следующее свойство экрана – **модальность**. Предыдущий пример интерфейса может отображаться на протяжении всей игры, никак не мешая пользователю взаимодействовать с другими элементами экрана. Однако в некоторых случаях это взаимодействие необходимо блокировать, чтобы при работе с экраном (screen) диалоги и другие события не продвигались дальше.

Рис. 5.3.

В этом случае помогает свойство **modal True**. По умолчанию, если не прописывать его, оно находится в выключенном состоянии (False). При открытии экрана с этим параметром нам доступны интерактивные элементы только в нём. То есть реплики персонажей и все остальные экраны будут не активны.

```
screen quest_log():
    modal True
```

Свойство **tag** позволяет собрать несколько экранов в одну группу, где при открытии одного экрана будет закрываться другой из этой группы, если он сейчас открыт.

По такому принципу работает главное меню, где при выборе одного из экранов, он замещает предыдущий.

На примере наших экранов интерфейса можно увидеть, что при клике на кнопку действия происходят два события: закрытие текущего экрана и открытие нового.

```
screen interface(): # экран интерфейса
    text _("Пример простого интерфейса или текущий квест")
    textbutton _("Журнал заданий"): # текстовая кнопка
        action  Hide("interface"), Show("quest_log") #
        # Закрыли interface / Открыли quest_log
```

```
screen quest_log(): # журнал заданий
    text _("Нужно найти капитана")
    textbutton _("Закрыть"): # текстовая кнопка
        action Hide("quest_log"), Show("interface")
    # Закрыли quest_log / Открыли interface
```

Подобную конструкцию можно заменить, присвоив обоим экранам общий тег, назвав его, например, **interface**. Таким образом, при открытии одного из них, он заменит другой.

```
screen interface():
    tag interface # общий тег
```

```
screen quest_log():
    modal True
    tag interface # общий тег
```

Это удобно, когда у вас много разных экранов. К примеру, инвентарь, журнал событий, таблица со статистикой персонажа, что-то ещё. При закрытии каждого из них приходилось бы прописывать лишние команды, перегружая код дополнительными элементами и усложняя общую картину.

> Свойство **zorder** отвечает за расположение экранов в пространстве относительно друг друга.

То есть при выводе нескольких экранов можно установить, какой из них будет перекрывать другой.

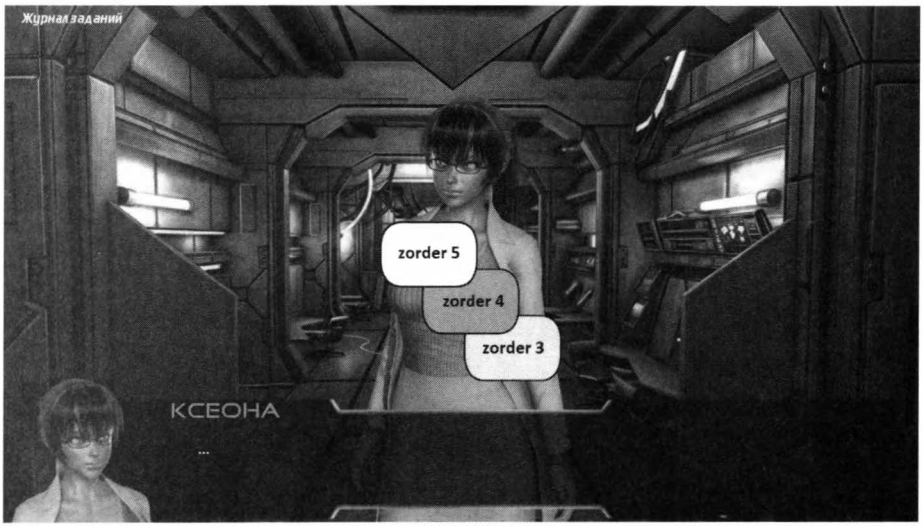


Рис. 5.4.

Каждый из речевых пузырей выводится на отдельном экране, однако, независимо от очередности вывода, мы устанавливаем каждому из них порядок наложения по оси **Z**. Чем больше значение, тем ближе слой к пользователю относительно других.

```
screen example_1():
    zorder 5
    add "gui/thoughtbubble.png"

screen example_2():
    zorder 4
    add "gui/v1.png"

screen example_3():
    zorder 3
    add "gui/l1.png"
```

Оператор **on** позволяет задать определённые действия при открытии или закрытии экрана.

```
screen quest_log():
    modal True
    tag interface
    on "show" action Hide("say")
```

В примере в момент открытия экрана (**on "show"**), автоматически будет скрыто диалоговое окно. Помимо этого оператор **on** поддерживает действия на закрытие экрана и смену экрана на другой, если они связаны общим тегом (tag).

```
screen quest_log():
    modal True
    tag interface
    on "hide" action ... # действие при закрытии этого экрана
    on "replace" action ... # текущий экран заменяет предыдущий
    on "replaced" action ... # текущий экран заменяется следующим
```

Свойство *style_prefix* устанавливает для всех объектов на экране определённый стиль, если объекты не имеют собственного стиля.

О стилях мы поговорим в соответствующем разделе, а здесь будет приведён только пример использования.

```
screen say(who, what):
    style_prefix "say"
```

Переменная *variant* отвечает за тип экрана, на котором должен быть отображён объект.

При разработке игры нужно учитывать разрешение экрана, на котором будет играть пользователь (ПК, планшет, телефон). Например, на

больших мониторах можно использовать среднего размера элементы интерфейса, тогда как на экранах телефонов эти элементы могут быть слишком маленькими и не удобными для взаимодействия. Поэтому в **экранах** (screen) можно прописать элементы интерфейса разного размера.

```
screen quest_log():
    if renpy.variant("large"): # Если большой ПК монитор
        textbutton _("Закрыть"): # Кнопка среднего размера
            action Show("interface")
    elif renpy.variant("medium"): # Если экран планшета
        textbutton _("Закрыть"): # Кнопка среднего размера
            action Show("interface")
    elif renpy.variant("small"): # Если экран телефона
        textbutton _("Закрыть"): # Большая кнопка
            action Show("interface")
```

Вместе с тем, как и в метках сценария, в экранах можно писать код на Python, указав соответствующий символ **$** для одной строки или слово **Python:** – в случае, если нужно добавить целый блок кода.

```
screen quest_log():
    $ kapitan_relat = 5
    text "[kapitan_relat]"

    python:
        kapitan_rank = "Капитан"
        kapitan_ratio = 67
        kapitan_namber = 1473
    text "[kapitan_rank]"
    text "[kapitan_ratio]"
    text "[kapitan_number]"
```

Стоит иметь в виду, что такой код выполняется только в рамках экрана.

5.2. Основные контейнеры в экранах

Контейнеры в экранах позволяют группировать разные объекты в нём определённым образом. То есть, это такие "вместилища", в которые можно положить несколько изображений или строк текста, и они будут разложены

в нужном порядке. Тогда как при простом выводе на экран этих объектов, они будут накладываться друг на друга.

Рис. 5.5

В примере мы отобразили текст, кнопку и изображение без указания позиции. Поэтому, по умолчанию, всё отобразилось в нулевой точке отсчёта координат, в верхнем левом углу. Эту проблему можно решить, указав каждому объекту свою позицию на экране. Однако, когда таких объектов очень много, проще поместить их в определённый контейнер.

> Контейнер **Frame** является базовым. Поместив в него группу объектов, мы можем указать сразу для всех новое расположение.

```
screen interface(): # экран интерфейса
    frame: # контейнер с объектами
        xysize(800,500) # размер фрейма (области загрузки)
        align(0.5,0.5) # расположение фрейма (области загрузки)
        xpadding 50 # отступы от края фрейма
        ypadding 50
        add "side liara"
        text _("Пример простого интерфейса или текущий квест")
        textbutton _("Журнал заданий"): # текстовая кнопка
            action Hide("interface"), Show("quest_log")
```

Рис. 5.6

Обратите внимание, что всё, относящееся к фрейму (области загрузки), должно быть смещено на один шаг вправо. Таким образом движок понимает, что относится к этому блоку, что именно должно лежать в нашем контейнере.

По умолчанию для фона фрейма используется стандартное изображение, которое находится в папке *gui/frame.png*. Оно автоматически подкладывается в виде подложки. Картинку можно заменить на свою, либо создать новую в графическом редакторе. Кроме того, можно прописать свой цвет фона или название картинки, которая станет фоном.

```
screen interface():
    frame:
        # Прописали картинку для фона
        background "gui/tablet.png"


screen interface():
    # полупрозрачный чёрный фон
    frame:
        background "#00000080"
```

Во втором примере мы указываем шестизначный код цвета, который должен стать фоном, а последние две цифры означают уровень прозрачности фона от 0 до 100. Чем меньше число, тем прозрачнее фон.

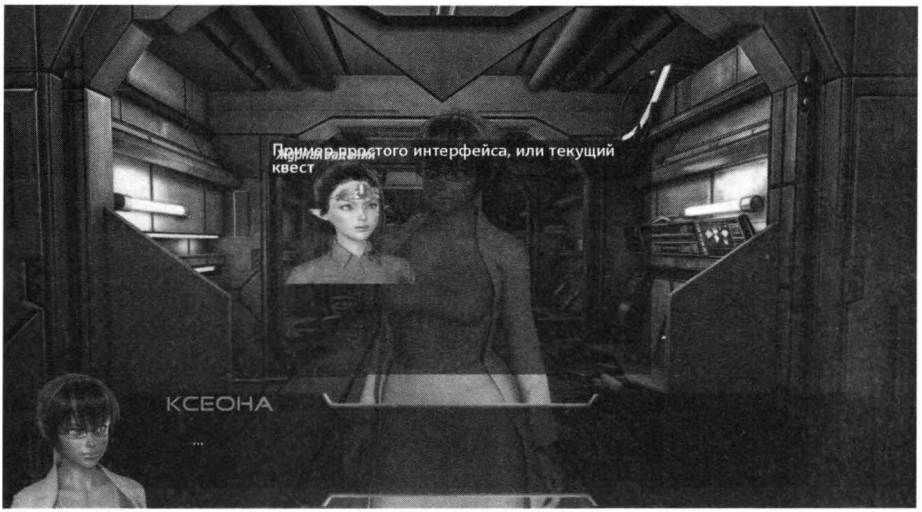


Рис. 5.7

Теперь, указав позиции для каждого объекта во фрейме, фрейм можно перемещать по экрану, меняя только его координаты. Позиции объектов внутри высчитываются в рамках фрейма, и остаются неизменными при его перемещении.

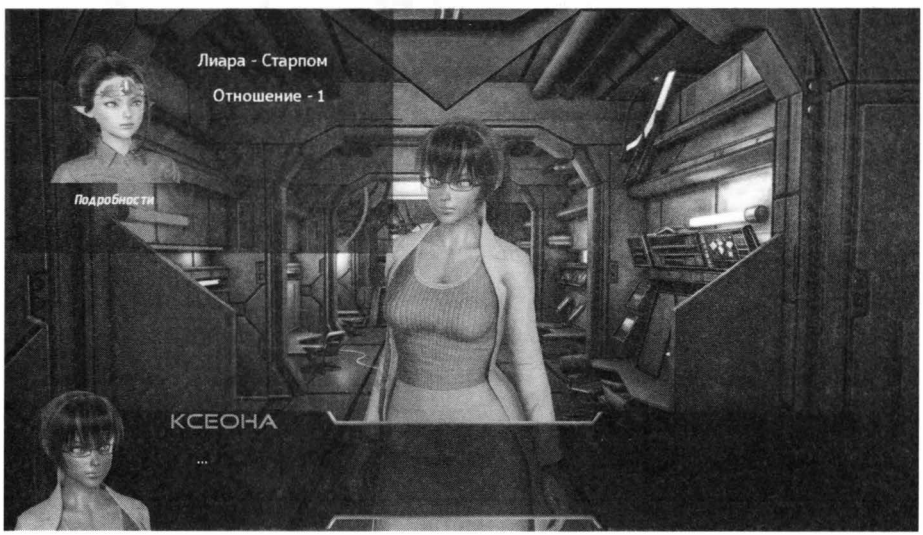


Рис. 5.8

```
screen interface():
    frame:
        background "#00000080" # полупрозрачный чёрный фон
        xysize(800,500) # размер фрейма
        align(0.01,0.01) # расположение фрейма
        xpadding 50 # Отступы от краёв фрейма
        ypadding 50
        add "side liara" xpos 5 ypos 5 # или align(0.1,0.5)
        text _("Лиара - [liara_rank]") align(0.8,0.1)
        text _("Отношение - [liara_relat]") align(0.8,0.3)
        textbutton _("Подробности"): # текстовая кнопка
            action Show("details")
            align(0.15,0.9)
            text_size 30 # размер текста кнопки
            text_color "#88ffff" # цвет текста
```

Контейнеры vbox и hbox полезны при необходимости разместить несколько объектов в ряд. Vbox размещает вертикально один под другим, hbox – горизонтально в строку, один за другим.

Это главное отличие от **frame**, который по умолчанию складывает объекты один поверх другого. Таким образом, можно выводить большое количество элементов интерфейса, не указывая для каждого свою позицию, а только позицию самого контейнера.

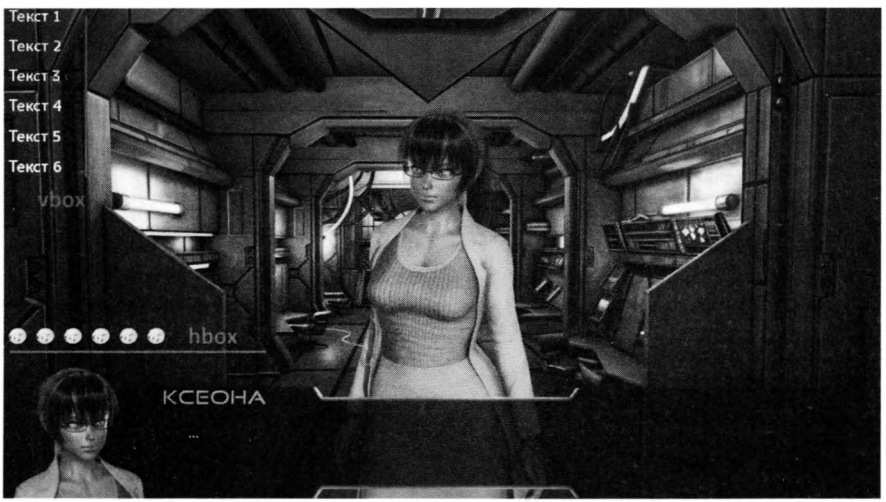


Рис. 5.9

```
screen interface():
    vbox: # вертикальный бокс
        xysize(800,800) # размер бокса
        align(0.01,0.01) # расположение бокса
        box_wrap True # Перенос объектов в новый столбец
        spacing 30 # Расстояние между объектами
        text "Текст 1" # Объекты
        text "Текст 2"
        text "Текст 3"
        text "Текст 4"
        text "Текст 5"
        text "Текст 6"

    hbox: # горизонтальный бокс
        xysize(800,800) # размер бокса
        xpos 10 ypos 700 # расположение бокса
        box_wrap True # Перенос объектов на новую строку
        spacing 15 # Расстояние между объектами
        add "smile" # Объекты
        add "smile"
        add "smile"
        add "smile"
        add "smile"
        add "smile"
```

Фреймы и коробки (boxes) можно совмещать, складывая одни в другие. Главное, не забывать про отступы и правильную последовательность контейнеров.

Рис. 5.10

```
screen interface():
    frame:
        background "#00000080" # полупрозрачный чёрный фон
        vbox: # вертикальный бокс
            xysize(500,800) # размер бокса
            align(0.02,0.01) # расположение бокса
            box_wrap True # Перенос объектов в новый столбец
            spacing 30 # Расстояние между объектами
            text "Текст 1" # Объекты
            text "Текст 2"
            text "Текст 3"
    hbox: # горизонтальный бокс
        xysize(500,800) # размер бокса
        xpos 20 ypos 700 # расположение бокса
        box_wrap True # Перенос объектов на новую строку
        spacing 15 # Расстояние между объектами
        frame:
            add "smile"
        frame:
            add "smile"
        frame:
            add "smile"
```

Ещё один вариант контейнера – **Grid**.

Он позволяет создавать таблицы. Стоит учитывать, что все ячейки получают одинаковый размер. А сам размер определяется по самому большому объекту в таблице.

Пример работы этого контейнера можно увидеть, если открыть экран сохранения или загрузки. Помимо этого, таблицы могут использоваться для создания галерей изображений или структурирования какой-либо информации.

Рис. 5.11

```
screen interface():
    frame:
        background "#00000080"
        xysize(300,220)
        align(0.02,0.01)
        grid 2 3: # Число столбцов и рядов
            spacing 50 # Расстояние между ячейками
            text "Текст 1" # Объекты
            text "Текст 2"
            text "Текст 3"
            text "Текст 4"
            text "Текст 5"
            text "Текст 6"
```

Viewport создаёт контейнер, который можно прокручивать мышью.

Полезно, когда область интерфейса имеет ограниченные размеры, и часть информации остаётся за пределами видимости.

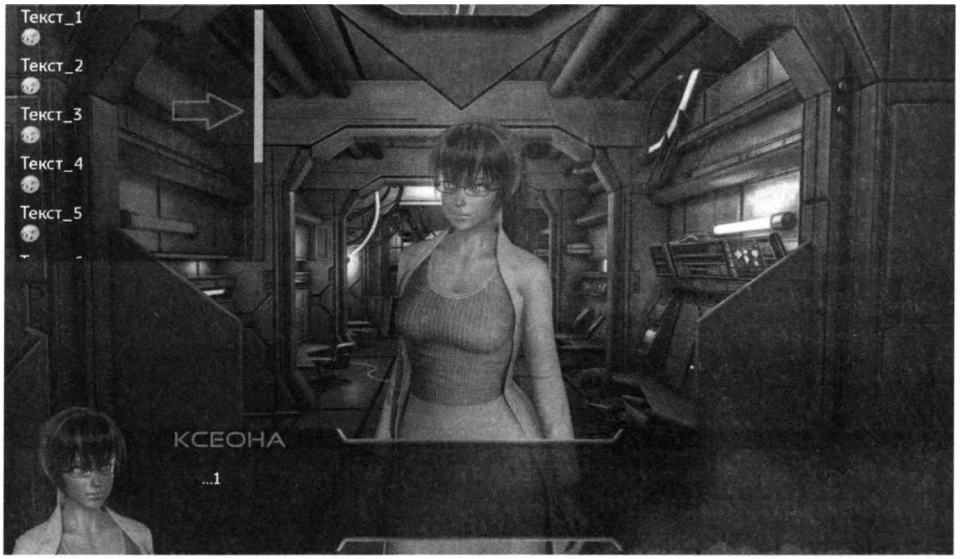


Рис. 5.12

```
screen interface():
    frame:
        background "#00000080"
        xysize(500,500)
        align(0.02,0.01)
        viewport: # Область прокрутки
            scrollbars "vertical" #
            mousewheel True  #
            draggable True #
            edgescroll (100, 100)  #
            yinitial 50 #
            vbox: # Вертикальный бокс
                text "Текст_1" # Объекты
                add "smile"
                text "Текст_2"
                add "smile"
                text "Текст_3"
                add "smile"
                text "Текст_4"
                add "smile"
                text "Текст_5"
                add "smile"
                text "Текст_6"
                add "smile"
```

Viewport может иметь дополнительные параметры, отвечающие за следующие настройки:

- **scrollbars "vertical"** – указывает на вертикальное расположение каретки и возможность прокрутки вверх-вниз
- **scrollbars "horizontal"** – добавляет горизонтальную каретку для прокрутки влево-вправо
- **mousewheel True** – разрешает прокручивать список колёсиком мыши. Если настройка отсутствует или установлена на *False*, список можно прокрутить, только потянув за каретку.
- **draggable True** – позволяет прокручивать список, зажав мышкой любую область контейнера и перетягивая его по направлению прокрутки. Если настройка отсутствует или установлена на *False*, то она отключает эту возможность.
- **edgescroll (100, 100)** – позволяет прокручивать список, подведя курсор мыши к краю области прокрутки. Первое число устанавливает расстояние в пикселях от края области, в пределах которой начинается автоматическая прокрутка. Второе число отвечает за скорость прокрутки. В данном примере 100 пикселей в секунду.
- **yinitial 50** – устанавливает начальную позицию каретки по вертикали. В данном примере список по умолчанию будет прокручен на 50 пикселей вниз.
- **xinitial** – аналогичная настройка для горизонтальной каретки

5.3. Экранные кнопки

В систему экранов можно встраивать кнопки для взаимодействия с интерфейсом игры. Ранее в примерах мы уже неоднократно встречались с текстовой кнопкой. Давайте подробнее рассмотрим её возможности и способы применения.

Текстовая кнопка (textbutton) считается самым простым вариантом кнопки, так как не имеет графического оформления и представляет собой простой текст, при клике на который запускается определённое событие.

```
screen interface():
    textbutton _("Журнал заданий"): # текстовая кнопка
        action Show("quest_log")
        align(0.05,0.05)
```

Для создания текстовой кнопки необходимо указать, что это **textbutton**, имеющая название в кавычках " ", и выполняющая действие **action**. Это основные элементы, которые должна включать в себя кнопка. Также, по необходимости, обрамляйте текст в тег _() для перевода на другие языки.

Кнопку можно записать двумя способами: в **одну строку** или **стандартным блоком после двоеточия**.

```
textbutton _("Журнал заданий") action Show("quest_log")
```

В таком виде можно создать текстовую кнопку, если она имеет минимальный набор дополнительных параметров. Так как, если у кнопки множество разных настроек и целый список действий, длина строки может уходить за край экрана, что не очень удобно для чтения и общего восприятия кода.

Что касается дополнительных атрибутов, кнопке можно передать целый ряд параметров для оформления её в стилистике вашей игры.

- **align (0.5,0.5)** – Устанавливает позицию кнопки по **x** и **y** в пределах от 0.0 до 1.0, где 0.5 – середина контейнера или экрана.
- **xpos 10 ypos 20** – Для установки координат точно в пикселях.
- **text_size 30** – Устанавливает размер текста кнопки.
- **text_font "fonts/arialbi.ttf"** – Устанавливает шрифт для текста кнопки.
- **activate_sound "audio/hit.mp3"** – Звук при клике на кнопку.
- **hover_sound "audio/hit2.mp3"** – Звук при наведении курсора на кнопку.
- **action** – Действие при клике на кнопку.
- **hovered** – Действие при наведении курсора на кнопку.
- **unhovered** – Действие при снятии курсора с кнопки.

- **alternate** – Действие при нажатии ПКМ или при удержании (на планшетах).
- **sensitive True** – Если установлена на *False*, то кнопка не кликабельна.
- **keysym** – При нажатии указанной кнопки на клавиатуре, запускает действие, прописанное в **action**.
- **alternate_keysym** – Альтернативное событие, происходит при нажатии указанной клавиши на клавиатуре.
- **text_color "#88ffff"** – Устанавливает цвет кнопки.
- **text_hover_color "#88aaff"** – Цвет кнопки при наведении курсора.

Стоит иметь в виду, что по умолчанию цвет кнопки задаётся в зависимости от выбранного стиля интерфейса при создании проекта. Если заменить цвет кнопки на свой (text_color), то стандартный цвет при наведении перестанет работать. Поэтому его также необходимо установить (text_hover_color).

Когда большому количеству кнопок требуется присвоить множество одинаковых параметров, используется общий стиль, в котором все необходимые настройки прописываются один раз. Это удобно, чтобы не проставлять одни и те же параметры отдельно для каждой кнопки. Как работать со стилями, будет разобрано в соответствующем разделе.

Кнопка-картинка – imagebutton, большинством функций похожа на *textbutton*, разница лишь в том, что вместо текста кнопка представляет собой картинку.

Это позволяет сделать интерфейс игры более привлекательным.

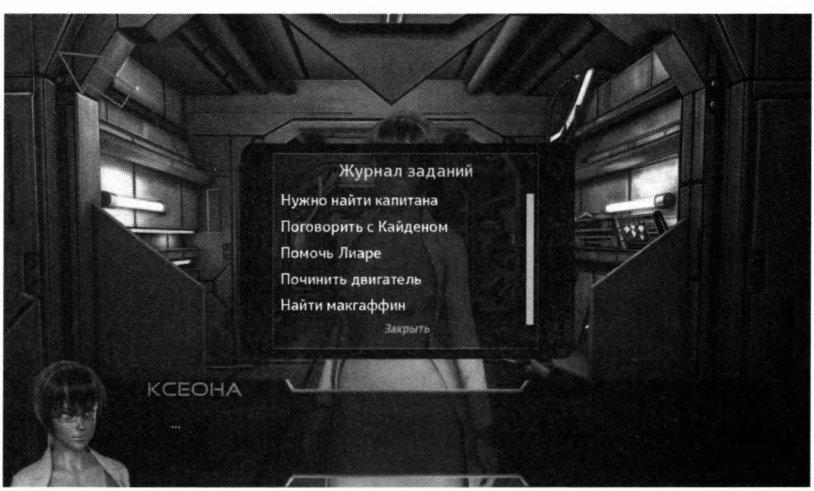


Рис. 5.13

```
screen interface():
    imagebutton: # кнопка-картинка
        action Show("quest_log")
        align(0.01,0.01)
        idle "images/infoi.png"
        hover "images/infoh.png"
        insensitive "images/clos.png"
        sensitive True
        focus_mask True
```

- **idle** – Указывает путь до картинки, которая отображается если кнопка в спокойном состоянии.
- **hover** – Картинка при наведении курсора.
- **insensitive** – Отображает картинку, если кнопка не кликабельна.
- **sensitive True** – Если *False* – кнопка не кликабельна, и отображается картинка, прописанная в **insensitive**.
- **focus_mask True** – Игнорирует прозрачные элементы кнопки. Это полезно, когда у изображения есть прозрачные поля по краям, и при наведении мыши на эти места кнопка не считается нажатой.

Кнопка button – включает в себя возможности *textbutton* и *imagebutton*. По сути, является универсальным инструментом, так как может отображать вместе картинку и текст.

Для примера, на обычной **imagebutton** тоже можно написать текст в графическом редакторе. Но, при переводе игры и интерфейса на другой язык, такие кнопки потребуется перерисовывать, с соответствующими надписями на другом языке.

Тогда как в **button** можно просто заключить текст в тег _(), и названия кнопок будут добавлены в файлы локализации.

Рис. 5.14

```
button: # кнопка
    action Show("interface")
    align(0.5,0.9)
    xsize 120 ysize 35 # Размер кнопки
    idle_background "images/but_s.png"
    hover_background "images/but_h.png"
    text _("Закрыть") yalign .5 xalign .5 size 20 color "#000000"
```

Обратите внимание, что параметры для текста кнопки пишутся в одну строку, иначе движок не поймёт, для какого элемента они указаны. Также необходимо указать размер кнопки, иначе она будет растянута на весть экран. Размер кнопки обычно устанавливается по размеру картинки.

Mousearea – область, при взаимодействии с которой можно задать определённое действие. Это, своего рода, *прозрачная кнопка*, **срабатывающая, если в указанную область попадает курсор мыши.**

Её можно использовать, например, для альтернативного открытия каких-либо интерфейсов.

```
screen interface():
    mousearea:
        area (0, 140, 150, 580)
        hovered Show("quest_log")
        unhovered Hide("quest_log")
    mousearea:
        area (140, 0, 1640, 140)
        hovered Show("statistics")
        unhovered Hide("statistics")
    mousearea:
        area (1770, 140, 150, 580)
        hovered Show("inventory")
        unhovered Hide("inventory")
```

В данном примере, если курсор подвести к левому краю экрана, будет открыт журнал заданий. Если к правому – статистика персонажа. К верхнему краю – инвентарь. Когда курсор покидает указанную область, окно интерфейса скрывается.

Третье и четвёртое число в скобках указывают ширину и высоту активной области.
Первое и второе – координаты верхнего левого угла области по **x**, **y**.
Если не указывать размеры и координаты, **mousearea** заполнит весь экран или контейнер (frame, box, fixed), в котором находится.

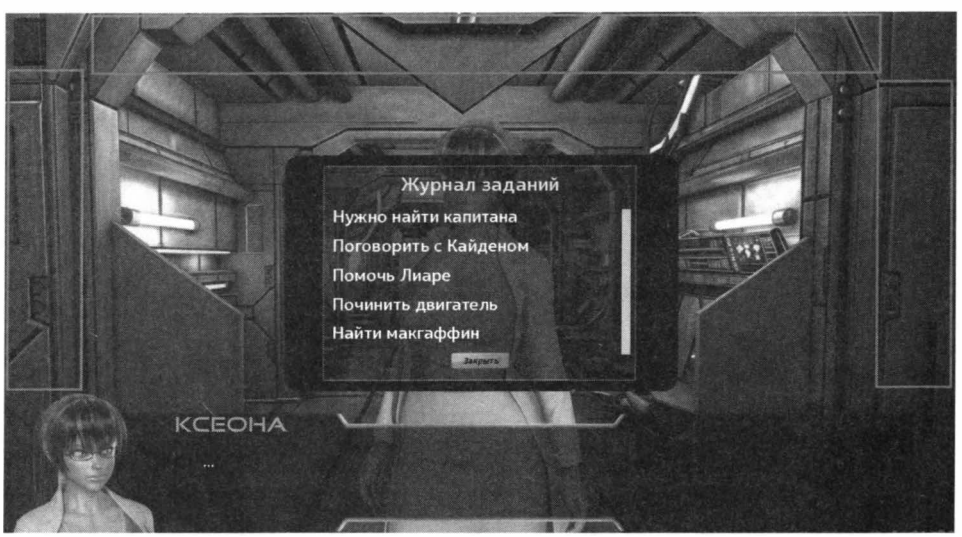


Рис. 5.15

Действия, которые можно установить на *action* (при нажатии кнопки)

Каждая кнопка в обязательном порядке должна иметь атрибут **action**, в котором указываются действия, происходящие при нажатии. Все команды действий записываются с большой буквы.

- **Show ("Название_экрана")** – При клике на кнопку будет показан (Show) экран, название которого написано в кавычках и скобках.
- **Hide ("Название_экрана")** – Закроет экран, указанный в скобках.
- **Jump ("Название_метки")** – При клике на кнопку происходит переход к указанной метке. Например, **Jump** ("start").
- **Call ("Название_метки")** – Вызывает указанную метку.
- **NullAction()** – Бездействие. Может пригодиться во время теста кнопок, чтобы они были кликабельны, но не выполняли каких-либо действий.
- **SetVariable ("achievement", False)** – Присваивает переменной в кавычках булево значение False или True.
- **SetVariable ("fork", "медцентр")** – Присваивает переменной "fork" строчное значение "медцентр".
- **SetVariable ("part", 5)** – Присваивает переменной "part" числовое значение 5.
- **SetVariable("part", part+2)** – К текущему значению переменной "part" будет прибавлено нужное число. Аналогично можно вычесть "-", разделить "/", умножить "*" и выполнить другие математические операции.
- **ToggleVariable ("achievement", True, False)** – При нажатии на кнопку переменная будет поочерёдно менять своё значение с *True* на *False* и обратно.
- **ToggleVariable ("part", 0, 1)** – Аналогично можно изменять два числовых значения.

> **ПРИМЕЧАНИЕ**. Обратите внимание, если кнопка имеет несколько действий, они записываются через запятую, и обрамляются квадратными скобками. Если одной из команд является **Jump**, она должна находиться в самом конце списка.

```
imagebutton: # кнопка-картинка
    action [ToggleVariable("liara_relat", 0,1),Jump("engine")]
```

Движок выполняет действия последовательно, одно за другим. Сначала будет изменено значение переменной, а затем произойдёт прыжок в указанную точку. Если вначале списка будет стоять Jump("engine"), Ren'Py выполнит это событие и "перепрыгнет" к новой метке, не увидев следующие команды.

В случаях, когда список действий получается очень длинным, его можно переносить на новую строку.

```
imagebutton: # кнопка картинка
    action [Show("quest_log"), ToggleVariable("liara_relat", 0,1),
    SetVariable ("xeona_relat", 5), # Несколько действий
    SetVariable ("fork", "медцентр"), # перенесены
    Jump("engine")] # на новые строки
```

5.4. Всплывающие подсказки и уведомления

В Ren'Py при наведении курсора на определённый интерактивный объект можно выводить всплывающую подсказку. Это помогает обеспечить лучшее понимание назначения определённой кнопки интерфейса.

Рис. 5.16

За всплывающую подсказку отвечает функция **tooltip**, внешний вид которой можно настроить аналогично кнопкам интерфейса.

```
screen interface():
    tag interface
    imagebutton: # Журнал заданий
        action Show("quest_log")
        align(0.01,0.01)
        idle "images/infoi.png"
        hover "images/infoh.png"
        focus_mask True
        tooltip _("Журнал заданий") #
    imagebutton: # Статистика
        action Show("statistics")
        align(0.06,0.01)
        idle "images/stats.png"
        hover "images/stats.png"
        focus_mask True
        tooltip _("Статистика") #
    imagebutton: # Инвентарь
        action Show("inventory")
        align(0.11,0.01)
        idle "images/inventory.png"
        hover "images/inventory.png"
        focus_mask True
        tooltip _("Инвентарь") #
    $ tooltip = GetTooltip() # функция Tooltip
    if tooltip:
        frame:
            pos renpy.get_mouse_pos()
            anchor (-0.07, -0.4)
            xsize 350 ysize 60
            background "#00000080"
            text __("[tooltip]") align (.5,.5)
```

В примере каждая кнопка интерфейса получила атрибут **tooltip**, эти строки помечены решётками. В кавычках необходимо написать текст всплывающей подсказки для каждого отдельного элемента.

В нижней части экрана можно увидеть переменную *$ tooltip = GetTooltip()*, запускающую соответствующую функцию. Далее установлено условие *if tooltip*, которое проверяет, наведён ли курсор на интерактивный объект

интерфейса. И если условие верно (True), отображается фрейм со следующими настройками:

- Строка **pos renpy.get_mouse_pos()** сообщает движку, что уведомление должно появляться в том же месте, где находится курсор.
- Строка **anchor (-0.07, -0.4)** немного смещает это окошко, чтобы оно было ниже и правее указателя. Вы можете настроить любую необходимую позицию.
- Строка **xsize 350 ysize 60** устанавливает размеры окошка.
- Строка **background "#00000080"** подкладывает нужный фон. Вместо цвета можно указать название картинки, которая станет фоном.
- Строка **text __("[tooltip]") align (.5,.5)** вставляет текст по центру окна. В данном случае текст берётся из переменной *$ tooltip*, в которую передаётся описание от наведённой кнопки.

Обратите внимание, что строка в кавычках "[tooltip]" обрамлена двумя нижними подчёркиваниями и скобками. Аналогично другим строкам в экранах **screen** это нужно для добавления текста в файлы локализации. Однако, в отличие от простого текста, здесь перевода требует переменная. Поэтому она отмечается вторым нижним подчёркиванием. Это нужно запомнить.

Вторым видом всплывающих уведомлений является **Notify**. Оно обычно используется для оповещения об уже случившемся действии или событии и устанавливается на клик по интерактивному объекту.

Рис. 5.17

```
screen interface():
    tag interface
    imagebutton: # Журнал заданий
        action [Show("quest_log"),Notify(_("Журнал заданий"))]
```

Такое оповещение плавно всплывает на несколько секунд и автоматически исчезает. Вместе с тем, **notify** можно использовать внутри меток, по ходу сценария для дополнительных подсказок игроку.

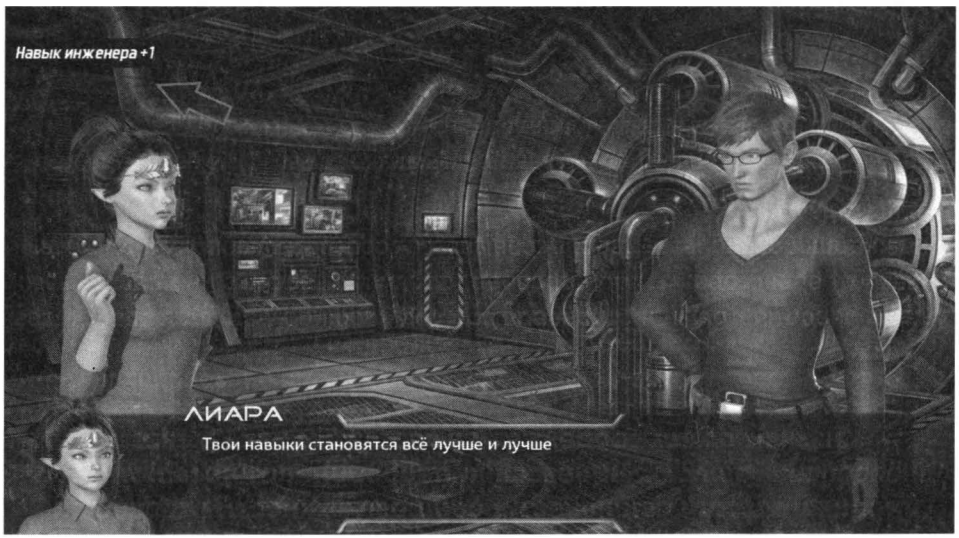


Рис. 5.18

```
label engine:
    scene engine with fade
    show liara serious at left
    show kayden normal at right
    kay "Кажется, с двигателем всё в порядке"
    $ kayden_engin +=1 # Повысили навык
    $ renpy.notify(_("Навык инженера +1")) # Сообщили об этом
    lia "Твои навыки становятся всё лучше и лучше"
```

Код notify-уведомления находится в файле *screens.rpy*. Там можно изменить его внешний вид, время отображения и расположение на экране. Откройте *screens.rpy*, нажмите **Ctrl + F** и напишите в поисковой строке **notify**, а затем нажмите **Enter**. Нас интересует экран **screen notify**(message).

```
screen notify(message):
    zorder 100
    style_prefix "notify"
    frame at notify_appear:
        text "[message!tq]"
    timer 3.25 action Hide('notify')
```

Многие элементы этого экрана нам уже известны, с остальными мы познакомимся чуть позже.

- **zorder 100** – Указывает, что сообщение должно отобразиться поверх всех экранов.
- **style_prefix "notify"** – Отвечает за стиль текста.
- **frame at notify_appear** – показывает сообщение с плавной анимацией появления и растворения окна уведомления.
- **timer 3.25 action Hide('notify')** – Таймер закрывает этот экран через три с небольшим секунды после того, как мы его открыли. Таким образом, вы можете установить любое время отображения.
- В строке **"[message!tq]"** можно увидеть странный тег **!tq**, он работает также как и __(), и нужен для перевода переменной на другой язык.

Чтобы изменить позицию всплывающего сообщения, допишите нужные координаты. Например, нижний правый угол:

```
screen notify(message):
    zorder 100
    style_prefix "notify"
    frame at notify_appear:
        xalign 0.9 yalign 0.9 # Новые координаты
        text "[message!tq]"
    timer 3.25 action Hide('notify')
```

Чтобы изменить изображение бэкграунда (фона), прокрутите страницу *screens.rpy* чуть ниже, до настройки стиля **style notify_frame**:

```
style notify_frame:
    ypos gui.notify_ypos
    background Frame("gui/notify.png", gui.notify_frame_borders)
    padding gui.notify_frame_borders.padding
```

В строке *background Frame("gui/notify.png", ...)* замените изображение на своё. Также вы можете открыть папку **gui** и заменить в ней изображение *notify.png* на любое другое.

Вместе с тем, познакомившись с функцией таймера, который выполняет определённое действие через назначенное время, можно создавать свои экраны уведомлений.

```
screen msg(): # Свой экран уведомлений
    timer 5.0 action Hide("msg") # Экран закроется через 5 секунд
    fixed at notify_appear: # Используем стандартную анимацию
        xalign 0.2 yalign 0.2 # Координаты
        add "изображение.png" # Любое изображение
        text _("[variable]") # Переменная, хранящая сообщение
```

Далее, в нужном месте сценария вызываем наш экран с уведомлением:

```
label engine:
    scene engine with fade
    kay "Кажется, с двигателем всё в порядке"
    # Добавляем в переменную текст, который нужно отобразить
    $ variable = "Всплывающее сообщение"
    show screen msg # Показали экран, который сам закроется
```

5.5. Оригинальный курсор

Стилизация курсора в игре под общий дизайн может значительно улучшить визуальное восприятие. Подбор соответствующего курсора помогает усилить общую тему и стиль. Например, в фэнтезийной игре он может быть выполнен в виде посоха, а в проекте о космосе – в виде космического корабля. Красиво оформленный курсор помогает игроку глубже погрузиться в виртуальный мир.

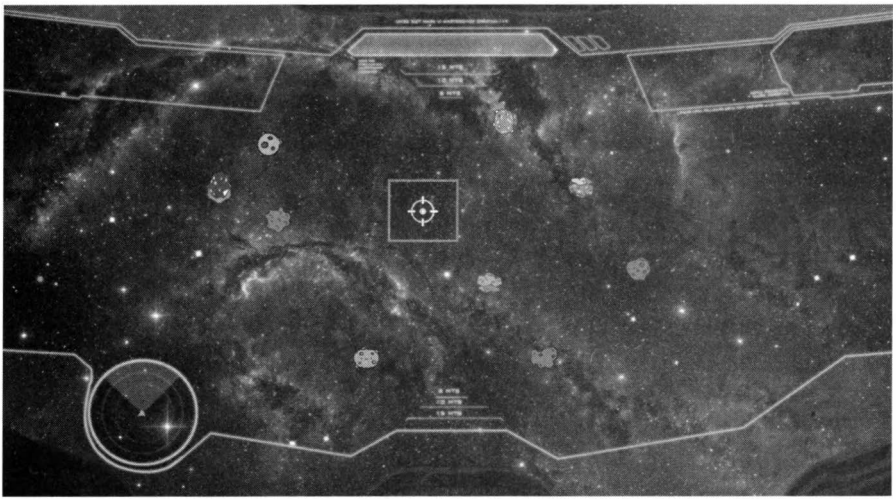


Рис. 5.19

Подбор уникального курсора позволяет выделить его на экране и облегчить навигацию для игрока. Кроме того, разные варианты курсора могут указывать на различные типы взаимодействия в игре, такие как перемещение, возможность манипулировать объектами, запуск действий и т. д.

Изменить внешний вид курсора в Ren'Py не сложно. Необходимо подготовить изображения с прозрачным фоном и нужным разрешением. Стандартный размер курсора в Windows 32x32 пиксела, но можно использовать и большего размера.

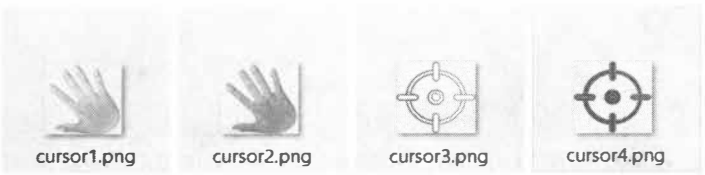


Рис. 5.20

Вторым шагом, после создания изображений, будет дополнительная настройка конфигурации.

Откройте файл *options.rpy*, и добавьте следующие строки в любое удобное место:

```
define config.mouse = {}
define config.mouse["cursor1"] = [("images/cursor1.png", 30, 30)]
define config.mouse["cursor2"] = [("images/cursor2.png", 30, 30)]
define config.mouse["cursor3"] = [("images/cursor3.png", 30, 30)]
define config.mouse["cursor4"] = [("images/cursor4.png", 30, 30)]
```

Здесь мы добавили словарь с указателями для мыши, присвоили каждому своё название ["cursor1"] и прописали пути к изображениям. Если в вашей игре запланировано больше вариантов курсора, просто добавьте для каждого аналогичную настройку.

Числа в конце указывают координаты по **x** и **y**, в котором будет установлен "активный пиксель" курсора. У стандартной стрелочки в Windows – это верхний левый угол курсора с координатами 0,0. Если ваш курсор видоизменён, но сохраняет общий вид такой стрелочки, можете оставить такие координаты. В примере выше используются изображения с разрешением 60x60 пикселей, с активным пикселем по центру. То есть координата x=30, y=30.

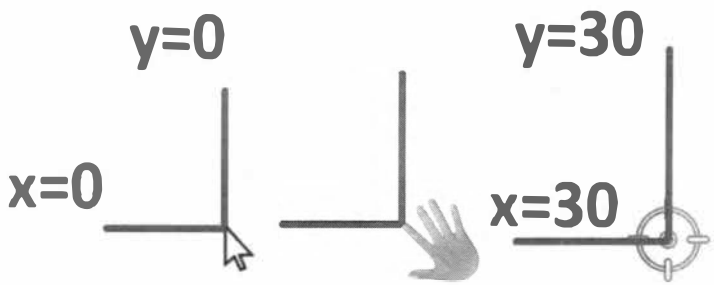


Рис. 5.21

Теперь, в нужном месте игры нам остаётся только переключать курсор по необходимости.

```
label engine:
    scene engine with fade
    $ default_mouse = "cursor1" # Курсор изменился на ладонь
```

В экранах:

```
screen fmg():
    imagebutton:
        idle "bbb"
        hovered SetVariable("default_mouse", "cursor2")
        unhovered SetVariable("default_mouse", "cursor1")
        action NullAction()
```

В примере выше, при наведении курсора на кнопку, срабатывает функция **hovered** и переключает нашу белую ладонь на красную, сообщая игроку, что с этим объектом можно взаимодействовать. При отведении курсора, функция **unhovered** возвращает изначальный вид курсора.

Таким образом для каждого интерактивного элемента в игре можно настроить свои указатели. Чтобы вернуть курсор к стандартному виду, нужно прописать переменной базовое значение "default".

```
label engine:
    scene engine with fade
    $ default_mouse = "default" # Возвращаем стандартный курсор
```

5.6. Бары и таймер

Бары представляют собой *анимированные полосы*, которые могут отображать различные параметры в игре и изменяться в зависимости от их значений.

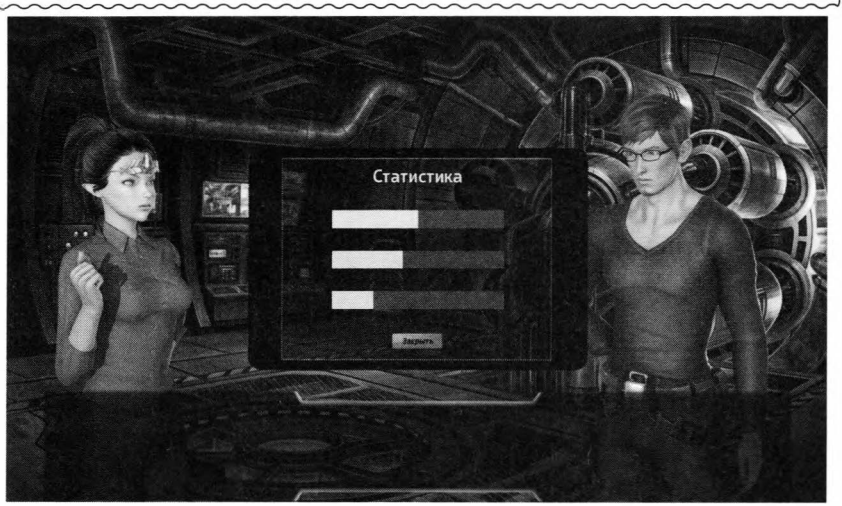


Рис. 5.22

```
screen statistics():
    tag interface
    frame:
        background "gui/tablet.png" # Фон
        xysize(800,500) # Размер фрейма
        align(0.5,0.5) # Расположение фрейма
        text _("Статистика") align(0.5,0.1) size 40 color "#88ffff"
        vbox: # Вертикальный бокс с горизонтальными боксами внутри
            align(0.5,0.5) # Расположение в-бокса внутри фрейма
            spacing 50 # Промежутки между горизонтальными боксами
            hbox:
                bar value AnimatedValue(value=kayden_engin, range=10,
delay=1.0) xsize 400 ysize 40
            hbox:
                bar value AnimatedValue(value=liara_relat, range=100,
delay=1.0) xsize 400 ysize 40
            hbox:
                bar value AnimatedValue(value=xeona_relat, range=100,
delay=1.0) xsize 400 ysize 40
        button: # Кнопка закрытия интерфейса
            action Show("interface"), Hide("quest_log")
            align(0.5,0.9)
            xsize 120 ysize 35 # Размер картинки
            idle_background "images/but_s.png"
            hover_background "images/but_h.png"
            text _("Закрыть") yalign .5 xalign .5 size
```

В данном примере внутри фрейма мы создали вертикальный бокс, который будет складывать объекты внутри себя вертикально сверху вниз. Этими объектами являются горизонтальные боксы. Объединение их в один **vbox** позволяет перемещать всю эту конструкцию одной настройкой координат внутри фрейма.

Горизонтальные контейнеры внутри **vbox** тоже будут складывать объекты внутри себя поочерёдно слева на право. Сейчас в каждом из них хранится только по одному объекту – бару, но в дальнейшем в hbox-ы можно добавить изображения, текст, и пр.

Что же собой представляет **полоска бара**?

```
bar value AnimatedValue(value=xeona_relat, range=100, delay=1.0) xsize 400 ysize 40
```

- **value** – Изменяемый параметр, переменная, которая будет визуально отражена в виде полоски бара. Сами переменные *xeona_relat*, *liara_relat* и *kayden_engin* мы создали в одной из предыдущих глав.

- **range=100** – Устанавливает диапазон значения (0-100), в пределах которого будет отображаться изменение.
- **delay** – Скорость анимации, с которой будет происходить визуальное изменение. По умолчанию установлено на 1.0 секунды. Можно изменить по необходимости.
- **xsize 400 ysize 40** – Размеры полоски бара по ширине и высоте.

В примере *value=xeona_relat* – отображает переменную Ксеоны, отвечающую за её отношение к главному герою, в пределах от 0 до 100. Картинка полоски бара по умолчанию берётся стандартная. Найти и изменить её можно в папке gui/bar/...png.

Здесь хранятся четыре изображения, по два для вертикального и горизонтального бара. **left.png** представляет собой картинку заполненного бара, а **right.png** – пустого. Для вертикальной полоски всё аналогично – **bottom.png** и **top.png**.

Чтобы создать вертикальный бар, достаточно добавить английскую букву **V** в начале – **vbar** value AnimatedValue().

> **bar_invert True** – С помощью этой команды можно перевернуть полоску отображения, чтобы анимация бара происходила в противоположную сторону.

В текущем виде наши полосы в статистике малоинформативные, поэтому в каждый **hbox** можно добавить иконки и текст.

```
vbox: # Вертикальный бокс с горизонтальными боксами внутри
    align(0.5,0.5)
    spacing 50
    hbox:
        add "ic_kayden" # Картинка
        bar value AnimatedValue(value=kayden_engin, range=10, delay=1.0)
xsize 400 ysize 40
        text _(" [kayden_engin]") # Текст
    hbox:
        add "ic_liara" # Картинка
        bar value AnimatedValue(value=liara_relat, range=100, delay=1.0)
xsize 400 ysize 40
        text _(" [liara_relat]%") # Текст
```

```
    hbox:
        add "ic_xeona" # Картинка
        bar value AnimatedValue(value=xeona_relat, range=100, delay=1.0)
xsize 400 ysize 40
        text _(" [xeona_relat]%") # Текст
```

Каждая горизонтальная коробка теперь включает в себя по три объекта, которые выстраиваются в линию. Дополнительно каждому из них можно указать координаты или промежутки для более корректного отображения.

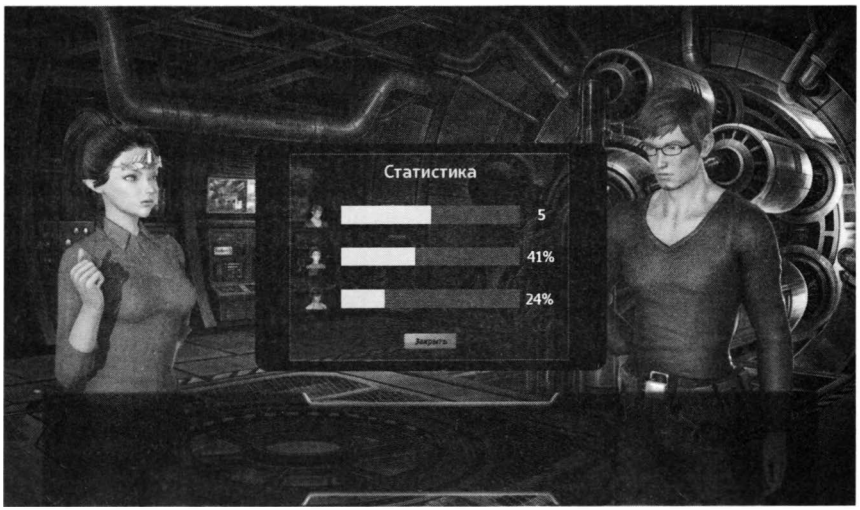


Рис. 5.23

Помимо этого, для каждого персонажа можно создать свои собственные полоски и прописать их в коде.

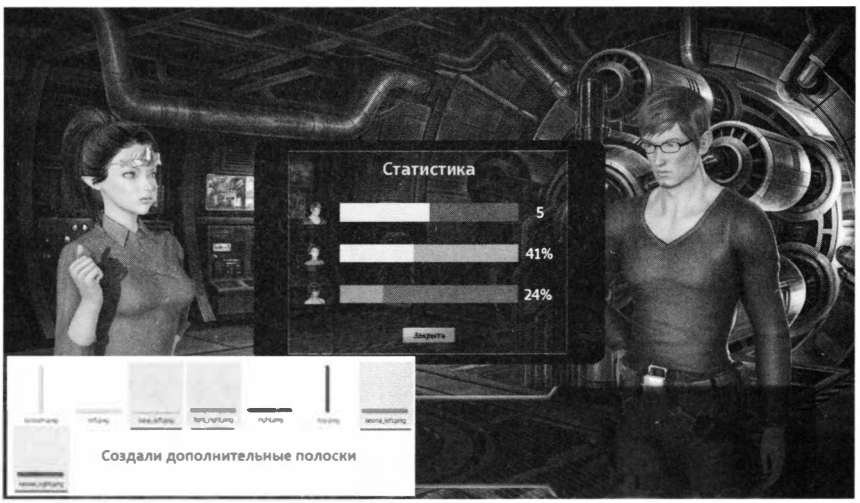


Рис. 5.24

```
vbox: # Вертикальный бокс
    align(0.5,0.5)
    spacing 50
    hbox:
        add "ic_kayden" # Картинка
        bar value AnimatedValue(value=kayden_engin, range=10, delay=1.0)
xsize 400 ysize 40
        text _(" [kayden_engin]") # Текст
    hbox:
        add "ic_liara" # Картинка
        bar value AnimatedValue(value=liara_relat, range=100, delay=1.0):
# Двоеточие и отступ
            xsize 400 ysize 40
            left_bar Frame("gui/bar/liara_left.png",5,5)
            right_bar Frame("gui/bar/liara_right.png",5,5)
        text _(" [liara_relat]%") # Текст
    hbox:
        add "ic_xeona" # Картинка
        bar value AnimatedValue(value=xeona_relat, range=100, delay=1.0):
# Двоеточие и отступ
            xsize 400 ysize 40
            left_bar Frame("gui/bar/xeona_left.png",5,5)
            right_bar Frame("gui/bar/xeona_right.png",5,5)
        text _(" [xeona_relat]%") # Текст
```

В скрипте для левой и правой полоски Лиары и Ксеоны мы прописали свои изображения. Числа в конце указывают отступы от краёв бара. Обратите внимание, что бар получил дополнительные параметры, поэтому, чтобы не растягивать код в длину, мы прописываем его ниже после двоеточия и с отступом вправо относительно бара.

Подобным образом можно отображать различные параметры в игре, которые будут изменяться в режиме реального времени. Но, помимо статистики, полоску бара можно использовать для визуализации таймера, с которым мы познакомились чуть выше. Например, с его помощью можно сделать небольшую мини-игру с поиском предмета или выбора действия за ограниченное время.

На скриншоте можно увидеть уменьшающийся бар, отображающий количество времени на поиск, а также полускрытый предмет, который необходимо найти. При наведении на него курсора в виде ладони, он краснеет, подсказывая игроку, что это интерактивный предмет.

Рис. 5.25

Для реализации подобного функционала в нужном месте сценария создаём переменную, которая будет хранить количество времени, и вызываем экран с мини-игрой.

```
default vremya = 30 # переменная времени с любым числом

label part_2:
    scene cabin with fade
    $ default_mouse = "cursor1" # Меняем курсор на ладонь
    $ vremya = 15 # Установили время 15 секунд
    $ config.rollback_enabled = False # Отключаем откат назад
    show kayden normal2 with dissolve
    show screen timing # Открываем экран с мини-игрой
```

Непосредственно перед открытием экрана мы включаем курсор для игры, устанавливаем нужное количество времени и отключаем возможность откатится назад колёсиком мыши, чтобы игрок не мог повторить попытку. Сам экран выглядит следующим образом:

```
screen timing(): # Экран таймера
    modal True # Блокируем нижние слои на время игры
    timer vremya action [Hide ("timing"),SetVariable("default_mouse",
"default"),Jump("part_4")]
    vbar: # Вертикальный анимированный бар
        align (0.9,0.6) # Расположение бара
```

```
        xysize (40,400) # Размер бара
        value AnimatedValue(old_value=vremya, value=0.0, range=vremya,
delay=vremya)
        bottom_bar Frame("gui/bar/new_bottom.png",1,1)
        top_bar Frame("gui/bar/new_top.png",1,1)

    button: # Искомый предмет в виде кнопки
        action [Hide("timing"), SetVariable("default_mouse",
"default"),Jump("part_3")]
        hovered SetVariable("default_mouse", "cursor2")
        unhovered SetVariable("default_mouse", "cursor1")
        xpos 203 ypos 672 # Позиция кнопки на экране
        xsize 137 ysize 111 # Размер кнопки
        idle_background "images/plns.png" # Изображения кнопки
        hover_background "images/plns.png"
```

- В строке: **timer** vremya **action** [...] запускается таймер, время которого указано в переменной **vremya**. В метке label part_2 мы указали, что это 15 секунд. По истечении этого времени функция **action** выполнит следующие действия: закроет экран с мини-игрой **Hide ("timing"),** вернёт курсору стандартный вид **SetVariable("default_mouse", "default"),** и отправит игрока к метке **Jump("part_4"),** в которой произойдут события после проигрыша в мини-игре. Так как истечение таймера означает, что игрок не успел найти предмет.
- Следующий блок **vbar** отвечает за визуализацию и анимацию таймера. Строка **value AnimatedValue()** также использует значение, хранящееся в переменной **vremya** для расчёта анимации.
- Далее две строки **bottom_bar Frame** и **top_bar Frame** отвечают за внешний вид бара. В них прописаны оригинальные картинки, аналогичные тем, которые создавались для статистики Лиары и Ксеоны. Если удалить эти две строки, будет отображена стандартная полоска.
- Последний блок **button** представляет собой предмет, который необходимо найти. При клике на него в **action** также будет закрыт экран с мини-игрой, курсор примет стандартный вид, а игрок будет переправлен к метке с победным исходом.
- Строки **hovered** и **unhovered** будут менять внешний вид курсора при наведении и снятии с кнопки, для подсказки игроку. Если удалить эти строки, курсор не будет подсвечиваться, и мини-игра станет немного сложнее, так как пользователь не будет знать интерактивен предмет или нет.

Вместе с тем, сам предмет тоже может подсвечиваться. Для этого изображение в строке **hover_background** должно цветом немного отличаться от оригинального.

В обеих метках с победой и поражением не забудьте снова включить возможность отката назад колёсиком мыши *$ config.rollback_enabled = **True***.

```
label part_3: # Сюда попадаем в случае победы
    $ config.rollback_enabled = True # Включили откат
    kay "О, вот оно!"
    jump part_5

label part_4: # Сюда при поражении
    $ config.rollback_enabled = True # Включили откат
    kay "Кто-то идёт! Нужно срочно спрятаться!"
    jump part_6
```

Подобный таймер можно также включать во время меню выборов (menu:), чтобы ограничить время игрока на принятие решений.

5.7. Привязка действий на кнопки клавиатуры

К кнопкам клавиатуры можно привязывать различные действия, аналогично тому, как мы делали это с кнопками в экранах **screen**. То есть, при нажатии на горячую клавишу можно открыть журнал заданий, статистику, инвентарь, вызвать определённое событие или что-то ещё.

Однако, следует помнить, что по умолчанию в Ren'Py уже существуют стандартные привязки к некоторым клавишам, и если их изменить на свои это может вызвать неудобство для игрока. Например, при нажатии клавиши "**S**" создаётся скриншот, кнопка "**H**" скрывает диалоговые окна и интерфейс, "**Shift+A**" – открывает настройки отображения текста и громкость звука. Подробнее с привязкой клавиш можно ознакомиться во вкладке "Помощь" главного меню.

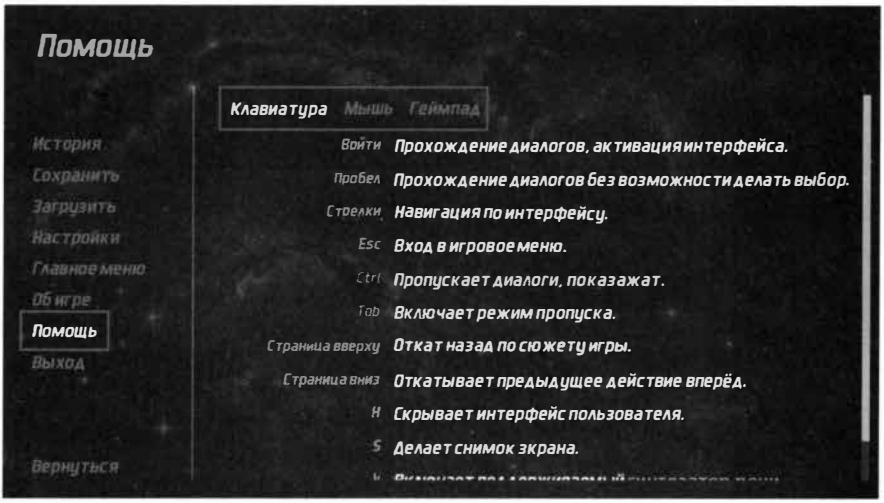


Рис. 5.26

Для привязки действия к нажатию клавиши, в движке предусмотрен оператор **key**, который работает в пределах экрана (screen). То есть, кнопка будет выполнять установленную функцию, пока экран, в котором она прописана, остаётся открытым. Также стоит иметь в виду, что Ren'Py чувствителен к регистру и раскладке клавиатуры, поэтому необходимо прописать все возможные варианты, при которых клавиша должна выполнить действие.

```
screen interface():
    tag interface
    # При нажатии открывается экран и проигрывается звуковой эффект
    key "q" action [Show('quest_log'),Play("sound", "zvuk.mp3")]
    key "Q" action [Show('quest_log'),Play("sound", "zvuk.mp3")]
    key "й" action [Show('quest_log'),Play("sound", "zvuk.mp3")]
    key "Й" action [Show('quest_log'),Play("sound", "zvuk.mp3")]
```

Вместе с тем, если вы планируете выпустить игру на мобильные устройства, у которых нет клавиатуры и мыши, нужно учесть, что эта функция будет недоступна пользователю. В версиях для телефонов и планшетов подобный функционал нужно дублировать иконками в интерфейсе. То есть, стандартными кнопками, которые мы разбирали в соответствующем разделе.

Для привязки действий к клавишам мыши используется команды вида: "mousedown_№" и "mouseup_№" – событие на нажатие и отпуск клавиши, где № – это номер клавиши мыши.

Нумерация кнопок идёт в следующем порядке:

1 – ЛКМ
2 – СКМ
3 – ПКМ
4 – Колесо мыши вверх
5 – Колесо мыши вниз

В коде это будет выглядеть так:

```
screen interface():
    key "mousedown_2" action ...
    # При нажатии СКМ срабатывает определённое действие
```

Исходя из вышеописанного, можно создать игру-платформер с управлением на кнопки клавиатуры. Или, например, небольшую мини-игру в рамках нашей новеллы.

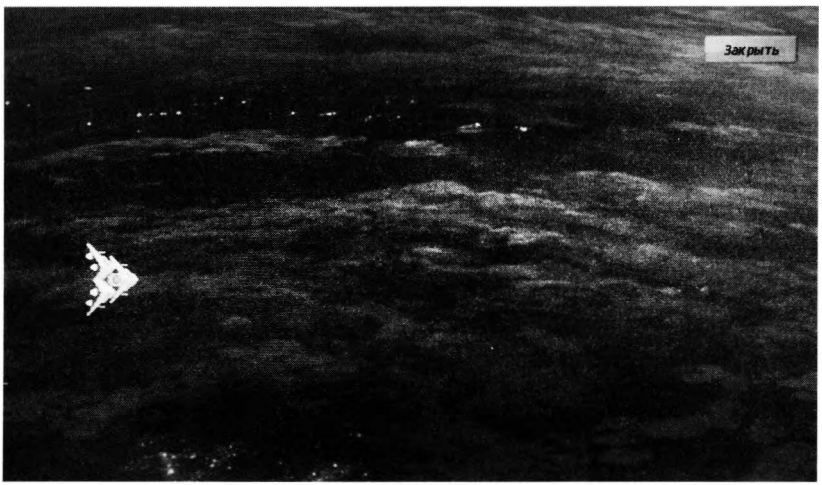


Рис. 5.27

Для начала необходимо создать три переменные, чтобы определять координаты управляемого объекта и величину шага при перемещении.

```
init python:
    move_x = 0.2 # Стартовая координата x
    move_y = 0.5 # Стартовая координата y
    step = 0.05 # Величина шага при перемещении
```

Затем, в нужно месте сценария открываем экран с мини-игрой, в котором прописываем действия при нажатии клавиш.

```
label platformer:
    scene kosmos1 with fade
    show screen game_platformer with dissolve
    pause
    jump start

screen game_platformer():
    modal True
    add "kosmolet" xpos move_x ypos move_y
    key "K_LEFT" action SetVariable('move_x', move_x - step)
    key "K_RIGHT" action SetVariable('move_x', move_x + step)
    key "K_UP" action SetVariable('move_y', move_y - step)
    key "K_DOWN" action SetVariable('move_y', move_y + step)
    button: # Кнопка закрытия экрана
        action [Hide("game_platformer"), Jump("part_7")]
        align(0.9,0.1)
        xsize 120 ysize 35 # Размер картинки
        idle_background "images/but_s.png"
        hover_background "images/but_h.png"
        text _("Закрыть") yalign .5 xalign .5
```

В экран **game_platformer** мы добавляем изображение **add "kosmolet"**, которое будет находиться на позициях, указанных в переменных **move_x** и **move_y**. В свою очередь, эти переменные меняют значения, при нажатии указанных клавиш.

При нажатии **key "K_LEFT"** запускается действие **action SetVariable**, изменяющее переменную **"move_x"**, которая теперь получает новое значение, равное **move_x - step**. То есть, из текущего значения будет вычтен шаг 0.05, и полученный результат передастся переменной, а космолёт сменит свою позицию на экране.

В текущем варианте это выглядит как демо (пробная версия), но в следующих главах мы познакомимся с функциями и трансформациями, благодаря которым можно будет задать дополнительные действия и условия, что значительно улучшит нашу мини-игру.

5.8. Оформление главного меню

В разделе 2.4 мы поверхностно настроили главное меню. Теперь, когда мы познакомились с основными возможностями системы экранов, нам по силам полностью переделать стандартный интерфейс игры.

Все экраны интерфейса находятся в *screens.rpy*. Откройте этот файл, и с помощью **поиска** (Ctrl+F) найдите экран **screen navigation**. В нём прописаны боковые кнопки, которые отображаются на каждой из вкладок меню.

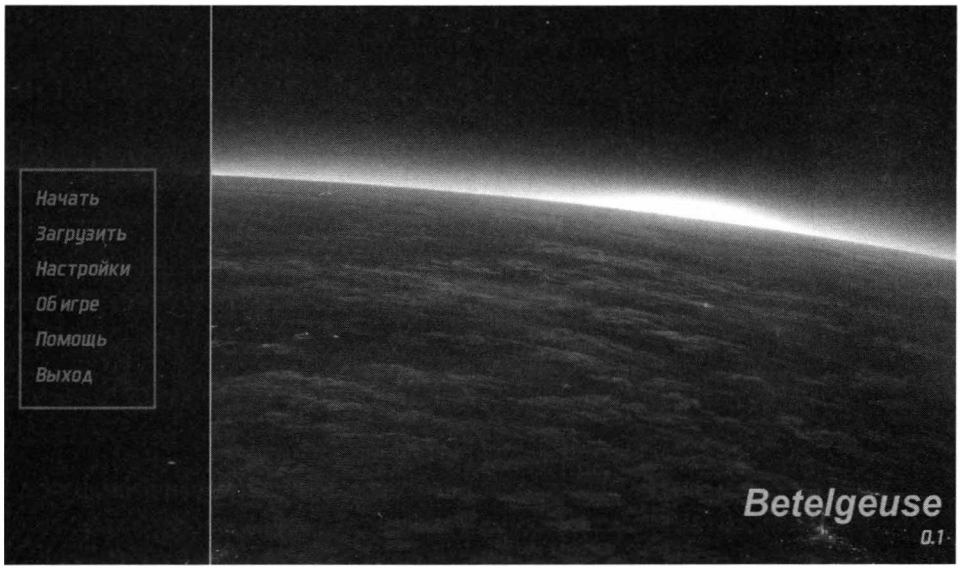


Рис. 5.28

Screen navigation является отдельным экраном, который открывается в связке с каждым из остальных экранов меню. При его изменении это отразится на каждом из экранов.

То есть, если нам потребуется изменить внешний вид и расположение кнопок только в одном экране, то эту привязку следует отключить, и в нужном экране прописать отдельные кнопки меню. Например, если их нужно расположить не слева, а справа.

Прокрутите *screens.rpy* чуть ниже, до экрана *screen main_menu*. Этот экран непосредственно отвечает за главное меню. В нём мы можем увидеть нашу привязку **use navigation**. То есть, при открытии главного экрана, вместе с ним открывается экран навигации.

```
screen main_menu():
    tag menu
    add gui.main_menu_background
    frame:
        style "main_menu_frame"
    use navigation
    if gui.show_name:
        vbox:
            style "main_menu_vbox"
            text "[config.name!t]":
                style "main_menu_title"
            text "[config.version]":
                style "main_menu_version"
```

Давайте разберёмся с остальными элементами главного меню.

- **tag menu** – Добавляет этот экран в группу **menu**, где при открытии любого экрана из этой группы, предыдущий автоматически закрывается.
- **add gui.main_menu_background** – Добавляет фоновое изображение из соответствующей настройки в gui. Вместо неё вы можете прописать адрес любой другой картинки или видеоролика. Например, add "images/engine.jpg".
- **frame:** – Следующее окошко фрейма отвечает за полупрозрачную рамку под кнопками меню. Её можно переделать на свою, заменив, например, соответствующее изображение в папке gui/overlay/main_menu.png. Или закомментировать, если рамка не требуется.

```
#frame:
    #style "main_menu_frame"
```

  Если поставить перед строками символ решётки, этот код становится комментарием, который не учитывается движком при обработке скрипта. Ненужные элементы в коде всегда предпочтительней помечать #, вместо простого удаления, так как в будущем вы можете передумать, и код придётся писать с нуля.

- **use navigation** – Закомментировав эту строку, мы отключаем привязку навигации, и теперь кнопки не будут отображаться на главном экране. Это будет полезно, если мы планируем переделать внешний вид нашего меню.

- **if gui.show_name:** – Следующий блок отображает название игры и номер версии. Сами эти параметры также находятся в файле *options.rpy*. Но в качестве альтернативного варианта этот блок можно закомментировать здесь.

Таким образом, убрав всё ненужное, мы получили пустой главный экран, который можем оформить в собственном стиле.

Рис. 5.29

В коде это выглядит так:

```
screen main_menu():
    tag menu
    add "gui/planeta.png" # Фон
    add "images/kosmolet.png" align(0.48,0.5) # спрайт космолёта
    text _("Betelgeuse"): # Название игры с настройками стиля
        align(0.5,0.1) # Расположение текста
        size 150 # Размер текста
        color "#ccccff" # Цвет текста
        font "AeroMaticsBoldItalic.ttf" # Шрифт текста
    frame: # Новый блок навигации
        background "#00000070" # Полупрозрачный фон
        xysize(1600,200) # Размер фрейма
        align(0.5,0.8) # Расположение фрейма
        hbox: # Горизонтальный бокс с текстовыми кнопками
            align(0.5,0.5) # Центрировали кнопки внутри фрейма
```

```
        spacing 50 # Расстояние между кнопками
        textbutton _("Загрузить") action ShowMenu("load")
        textbutton _("Настройки") action ShowMenu("preferences")
        textbutton _("Начать") action Start()
        textbutton _("Об игре") action ShowMenu("about")
        textbutton _("Помощь") action ShowMenu("help")
        textbutton _("Выход") action Quit(confirm=not main_menu)
```

Блок названия игры выглядит немного громоздко, так как мы ещё не разбирали стили, но в дальнейшем подобные конструкции можно существенно сократить.

Шаблоны кнопок были взяты из экрана навигации, но вместо них можно использовать свои. Давайте рассмотрим ещё один вариант с кнопками картинками.

Рис. 5.30

Код:

```
screen main_menu():
    tag menu
    add "gui/planeta.png" # Фон
    add "images/kosmolet.png" align(0.5,0.4) # спрайт космолёта
    text _("Betelgeuse"): # Название игры с настройками стиля
        align(0.5,0.1) # Расположение текста
        size 150 # Размер текста
        color "#ccccff" # Цвет текста
```

```
    font "AeroMaticsBoldItalic.ttf" # Шрифт текста
button: # Кнопка "Загрузить"
    action ShowMenu("load")
    align(0.15,0.4)
    xsize 250 ysize 150
    idle_background "gui/menu_butt.png"
    hover_background "gui/menu_butt2.png"
    text _("Загрузить") align(0.5,0.5) size 50 color "#ccccff"
button: # Кнопка "Настройки"
    action ShowMenu("preferences")
    align(0.325,0.6)
    xsize 250 ysize 150
    idle_background "gui/menu_butt.png"
    hover_background "gui/menu_butt2.png"
    text _("Настройки") align(0.5,0.5) size 50 color "#ccccff"
button: # Кнопка "Начать"
    action Start()
    align(0.5,0.8)
    xsize 250 ysize 150
    idle_background "gui/menu_butt.png"
    hover_background "gui/menu_butt2.png"
    text _("Начать") align(0.5,0.5) size 50 color "#ccccff"
button: # Кнопка "Об игре"
    action ShowMenu("about")
    align(0.675,0.6)
    xsize 250 ysize 150
    idle_background "gui/menu_butt.png"
    hover_background "gui/menu_butt2.png"
    text _("Об игре") align(0.5,0.5) size 50 color "#ccccff"
button: # Кнопка "Помощь"
    action ShowMenu("help")
    align(0.85,0.4)
    xsize 250 ysize 150
    idle_background "gui/menu_butt.png"
    hover_background "gui/menu_butt2.png"
    text _("Помощь") align(0.5,0.5) size 50 color "#ccccff"
```

В данном случае вместо текстовых кнопок мы создали универсальные кнопки, которые при наведении меняют цвет фона и текста. Их код во многом похож и может быть сокращён с помощью стилей.

Кроме того, каждую кнопку можно сделать полностью оригинальной, изменяя параметры в ней по-своему.

5.9. Оформление экрана навигации

Аналогично главному меню, мы можем изменить внешний вид всех остальных экранов. Однако, нужно учитывать, что экран навигации (**screen navigation**) привязан к каждому из них. Его можно отвязать в каждом разделе, закомментировав (#), или привести к тому виду, который будет вписываться в новый дизайн. Но, для начала давайте разберёмся, из чего он состоит.

```
screen navigation():
    vbox:
        style_prefix "navigation"
        xpos gui.navigation_xpos
        yalign 0.5
        spacing gui.navigation_spacing
        if main_menu:
            textbutton _("Начать") action Start()
        else:
            textbutton _("История") action ShowMenu("history")
            textbutton _("Сохранить") action ShowMenu("save")
        textbutton _("Загрузить") action ShowMenu("load")
        textbutton _("Настройки") action ShowMenu("preferences")
        if _in_replay:
            textbutton _("Завершить повтор") action
EndReplay(confirm=True)
        elif not main_menu:
            textbutton _("Главное меню") action MainMenu()
        textbutton _("Об игре") action ShowMenu("about")
        if renpy.variant("pc") or (renpy.variant("web") and not
renpy.variant("mobile")):
            textbutton _("Помощь") action ShowMenu("help")
        if renpy.variant("pc"):
            textbutton _("Выход") action Quit(confirm=not main_menu)
```

Как видим, все элементы помещаются в вертикальный контейнер, который раскладывает кнопки по порядку. Многие кнопки имеют особые условия, при которых они будут отображаться. Некоторые можно увидеть, только открыв меню во время игры.

Например, экран сохранения, так как в главном меню он без надобности.

- **style_prefix "navigation"** – Устанавливает стиль для блока навигации. Сам стиль прописан чуть ниже экрана навигации, а некоторые элементы в файле *gui.rpy*.
- **xpos gui.navigation_xpos и yalign 0.5** – Устанавливают позицию бокса по "x" и "y". Позиция "x" также ссылается на настройку в файле *gui*. На самом деле, такой разброс настроек по разным файлам не очень удобен. Поэтому некоторые лучше прописывать непосредственно в экране.
- **spacing gui.navigation_spacing** – Устанавливает расстояние между кнопками. Само расстояние также можно найти в *gui.rpy*.
- **if main_menu:** – Кнопки, относящиеся к этому условию, отображаются только в главном меню. Например, кнопка "Начать". Если выйти в меню во время игры с помощью кнопок "Esc" или ПКМ, она будет скрыта.
- **else:** – Следующее условие предусматривает противоположное действие, при котором доступны две кнопки во время игры: "Сохранить" и "История".
  Кнопки "**Загрузить**" и "**Настроить**" не относятся ни к одному из предыдущих условий, поэтому доступны всегда.
- **if _in_replay:** – Условие, проверяющее, находится ли игрок на экране повтора. В этом случае доступна кнопка завершения повтора.
- **elif not main_menu:** – Альтернативное условие, если игрок не находится в главном меню, и, соответственно, не находится на экране повтора. То есть, кнопка будет доступна во всех остальных случаях.
- **if renpy.variant("pc") or (renpy.variant("web") and not renpy.variant("mobile")):** – Если пользователь играет на ПК или в онлайн-версию, но не с мобильного устройства, будет доступна кнопка "Помощь". Дело в том, что для мобильной версии не предусмотрено каких-либо вспомогательных материалов, поэтому и нет смысла отображать этот экран в такой версии.
- **if renpy.variant("pc"):** – Похожее условие, отображающее кнопку выхода только в версии для ПК. На телефонах и планшетах разработчик предлагает выходить из игры средствами самих устройств. Например, свернув игру и выключив её в проводнике телефона. Это, вероятно, обусловлено нехваткой места на экранах мобильных телефонов, поэтому дополнительные элементы интерфейса скрываются.

Однако, по желанию, в своём проекте мы можем оставить доступ к дополнительным элементам. Для этого нужно удалить соответствующее условие (if renpy.variant), и настроить кнопки таким образом, чтобы они органично размещались на небольшом пространстве экранов мобильных устройств.

Теперь можно приступить к настройке навигации. В экране **screen navigation** все объекты, как и в главном меню, можно разложить в любом расположении и порядке. Но, не стоит забывать, что он связан с другими экранами, и его элементы могут накладываться поверх элементов других экранов.

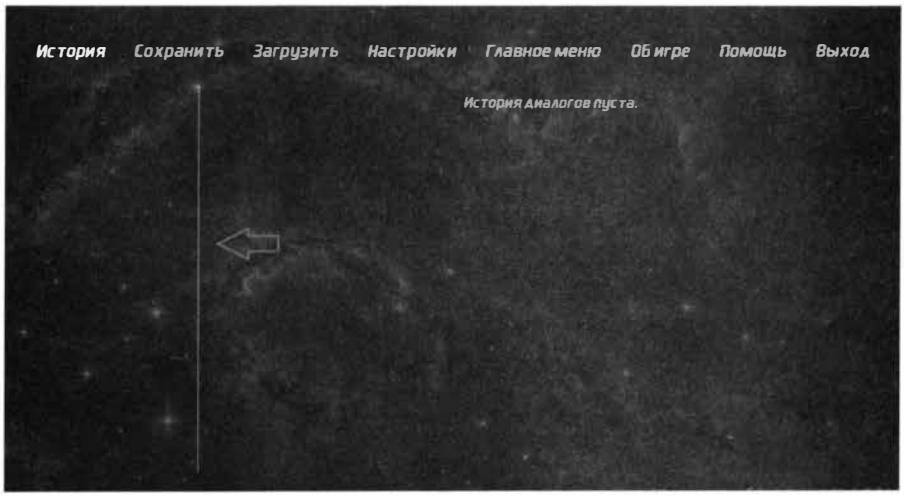


Рис. 5.31

При настройке экрана навигации можно заметить вертикальную полосу старого интерфейса. Это изображение находится по адресу gui/overlay/game_menu.png. Его можно заменить на своё или закомментировать соответствующую строку в коде.

Для этого с помощью **поиска по файлу** (Ctrl+F), найдите game_menu.png. Результатом должна стать строка: *background "gui/overlay/game_menu.png"*. Поставьте символ решётки перед ней, и проблема будет решена.

Код из примера выше выглядит следующим образом:

```
screen navigation():
    hbox:
        align(0.5,0.07)# Расположение бокса
        spacing 50 # Промежуток между кнопками
```

Изменений не много. Вертикальный бокс заменён на горизонтальный. Это позволяет освободить больше места на экране. Настройки стиля, расположения и промежутка, которые ссылались на файл **gui**, были удалены, а вместо них прописаны непосредственно в экране. Все остальные условия и кнопки остались без изменений.

5.10. Оформление экрана "Об игре"

По умолчанию здесь располагается текст, сообщающий версию движка, на котором создан проект, и лицензии распространения. Данный текст вы можете изменить или удалить по собственному усмотрению, а вместо него добавить свою информацию. Например, написать имена команды разработчиков, благодарности, спонсоров, подписчиков, лицензии, и всё то, что посчитаете необходимым.

Рис. 5.32

```
screen about():
    tag menu
    use game_menu(_("Об игре"), scroll="viewport"):
        style_prefix "about"
        vbox:
            label "[config.name!t]"
            text _("Версия [config.version!t]\n")
            if gui.about:
                text "[gui.about!t]\n"
            text _("Сделано с помощью Ren'Py...")
```

Чтобы изменить экран "об игре", найдите в *screens.rpy* **screen about**.

- **tag menu** – Включает этот экран в группу экранов меню.
- **use game_menu(_("Об игре"), scroll="viewport")** – Привязывает к этому экрану экран game_menu, который будет иметь возможность скролла (пролистывания) при необходимости, а также стиль и контейнер с информацией.
- **label "[config.name!t]"** – Отображает переменную с названием игры, которая хранится в файле options.rpy (define config.name). Тег **!t** отвечает за перевод названия на другие языки.
- **text _("Версия [config.version!t]\n")** – Строка, отображающая переменную с версией игры. Сама переменная также находится в *options.rpy* (define config.version). Там же можно установить номер текущей версии.
- **if gui.about:** – Это условие отобразит текст в нижеследующей переменной, если она не пустая.
- **text "[gui.about!t]\n"** – строка с переменной, в которую вы можете поместить свой текст. Сама переменная находится в *options.rpy* и представляет собой следующую конструкцию: define gui.about = _p(""" """). Внутри тройных кавычек можно написать свой текст, который отобразится над текстом разработчика движка.
- **text _("Сделано с помощью Ren'Py ...")** – Чтобы скрыть данный блок, достаточно закомментировать эту строку.

Пример своего экрана "Об игре":

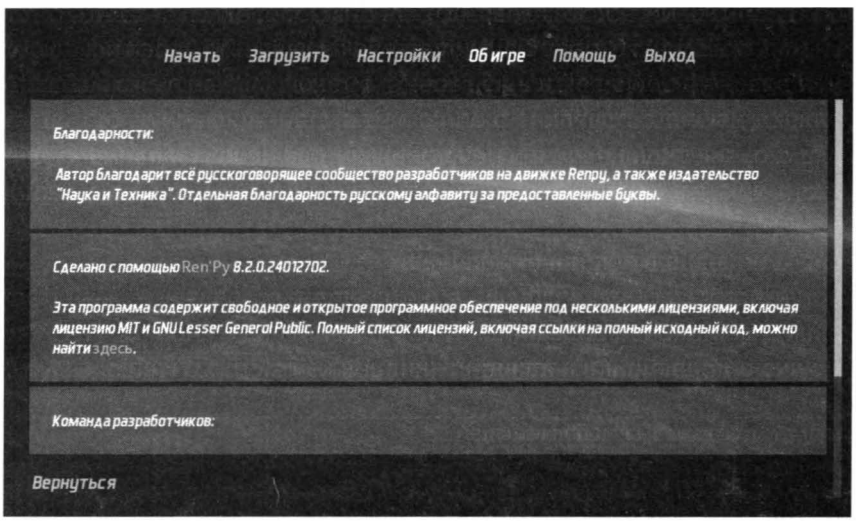


Рис. 5.33

```
screen about():
    tag menu
    use game_menu(_("Об игре"), scroll="viewport"):
        vbox:
            xysize(1800,600)
            align(0.5,0.5)
            spacing 10
            frame:
                xsize 1800
                padding (50,50)
                background "#99c5ff70"
                text _("Благодарности:\n\n Автор благодарит ...")
            frame:
                xsize 1800
                padding (50,50)
                background "#99c5ff70"
                text _("Сделано с помощью Ren'Py...")
            frame:
                xsize 1800
                padding (50,50)
                background "#99c5ff70"
                text _("Команда разработчиков:\n ...")
            frame:
                xsize 1800
                padding (50,50)
                background "#99c5ff70"
                text _("Подписчики:\n ...")
```

В примере мы удалили упоминания о версии и названии игры, но оставили стандартную привязку к **game_menu**. В вертикальный бокс были добавлены несколько фреймов с одинаковыми параметрами, для разделения текста на блоки. После ознакомления со стилями каждому фрейму и тексту можно присвоить общий стиль, чтобы уменьшить количество кода.

5.11. Оформление экрана настроек

За экран настроек отвечает **screen preferences**, который также можно найти в файле *screens.rpy*.

Аналогично другим экранам меню он работает в связке с **game_menu** и имеет несколько контейнеров, вложенных друг в друга. Каждый из которых можно перемещать, изменять и удалять.

По опыту предыдущих экранов с большинством элементов мы уже знакомы. Давайте разберёмся с оставшимися.

- **box_wrap True** – Настройка отвечает за перенос объектов горизонтального бокса на новую строку, если они достигли края бокса.
- **null height (4 * gui.pref_spacing)** – Устанавливает отступ для следующего объекта. В нашем случае из настроек конфигурации берётся определённое значение и умножается на 4. Вместо этого можно просто указать необходимый промежуток в пикселях: null height 40 – для отступа по высоте, и null width 30 – для отступа по ширине.
- **if config.has_music / sound / voice** – Условия, при которых отображаются полоски баров для громкости звуков. По умолчанию в *options.rpy* значения этих переменных установлены на *True*, и они доступны в настройках. Если переключить их в положение *False*, они будут скрыты из меню. Это может быть полезно, например, в случае, когда в проекте нет озвучки или фоновой музыки. Тогда эти настройки выключаются, чтобы они не занимали место и не вводили пользователей в заблуждение, так как не будут ни на что влиять.

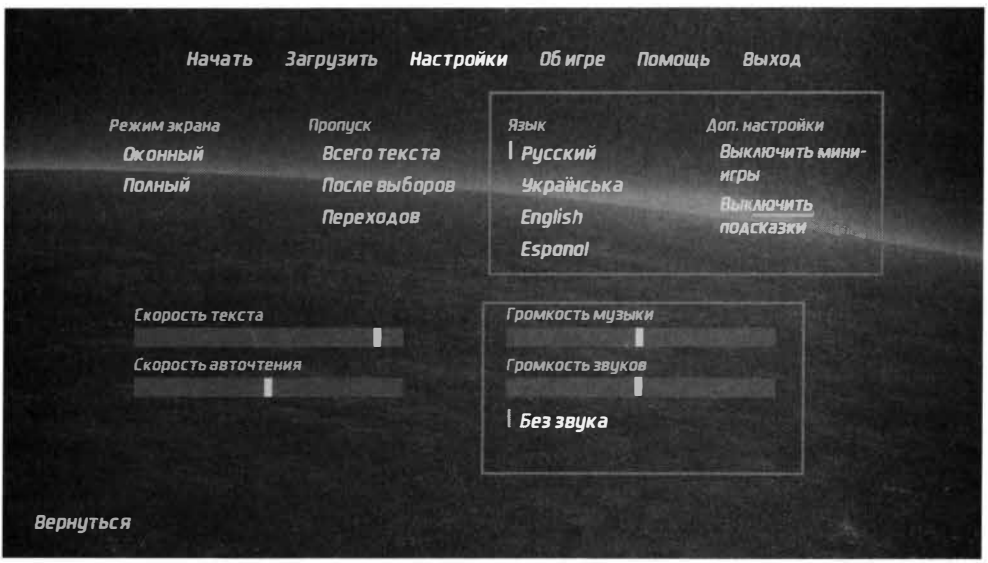


Рис. 5.34

```
vbox: # Переключение языков
    style_prefix "radio"
    label _("Язык")
    textbutton _("Русский") action Language (None)
    textbutton _("Українська") action Language ("ukrainian")
    textbutton _("English") action Language ("english")
    textbutton _("Espanol") action Language ("spanish")
vbox:
    style_prefix "radio"
    label _("Доп. настройки")
    # Вкл/Выкл мини-игр
    if persistent.mini_game:
        textbutton _("Выключить мини-игры") action
SetVariable("persistent.mini_game", False)
    else:
        textbutton _("Включить мини-игры") action
SetVariable("persistent.mini_game", True)
    # Вкл/Выкл подсказок
    if persistent.hints:
        textbutton _("Выключить подсказки") action
SetVariable("persistent.hints", False)
    else:
        textbutton _("Включить подсказки") action
SetVariable("persistent.hints", True)
```

В примере выше большая часть оригинального кода осталась без изменений. Поэтому приведены только новые блоки **vbox**, в которые были добавлены дополнительные настройки. Вместе с тем, была скрыта настройка громкости голоса.

Для добавления своих настроек мы создали два контейнера с сохранением общего стиля. Блок с переключением языков имеет стандартную команду действия **action Language ("")**, с указанием в кавычках языка. То есть, при нажатии на текстовую кнопку переменная меняет своё значение на выбранный язык, и к тексту в игре подставляются соответствующие файлы локализации.

> **ПРИМЕЧАНИЕ**. Значение *None* по умолчанию присваивается тому языку, на котором был создан проект.

Например, при создании игры на английском текстовая кнопка этого языка имела бы значение *None*, а кнопка русского языка **Language** ("russian"). Так как перевод ещё не создан, кнопки в настройках пока ни на что не влияют. Второй блок включает в себя дополнительные настройки геймплея. К примеру, игроку может не понравится проходить мини-игры. В таком случае он волен отключить их и сосредоточиться только на сюжете. Аналогичным образом работают подсказки, которые могут помогать в прохождении.

Условия для кнопок позволяют отобразить только один вариант из двух: включить или выключить. Если кнопка "выключить" имеет значение *True* (if persistent.hints), то отображается только кнопка с отключением этой опции. При её выключении условие становится ложным, кнопка перестаёт отображаться, но вместо неё начинает работать противоположное условие, которое отображает кнопку включения.

В текущем виде это всего лишь демонстрация добавления своих настроек. Чтобы они полноценно работали, мини-играм и подсказкам в игре нужно также прописать соответствующие условия их доступности.

```
label start:
    if persistent.mini_game: # Если переменная True
        jump mini_game # Переходим в лейбл с мини-игрой
    else: # иначе
        jump part_1 # Переходим в сюжетную часть
```

5.12. Оформление экранов сохранения/загрузки

Если открыть экраны **screen save** и **screen load**, можно увидеть, что они состоят всего из двух строк: *tag menu* и *use file_slots()*. То есть, по сути, являются пустыми экранами, использующими один общий экран со слотами сохранений.

Чтобы внести какие-либо изменения в сохранения и загрузку, предстоит работать именно с этим экраном. Однако при желании, можно в качестве альтернативы убрать привязку и настроить каждый экран отдельно.

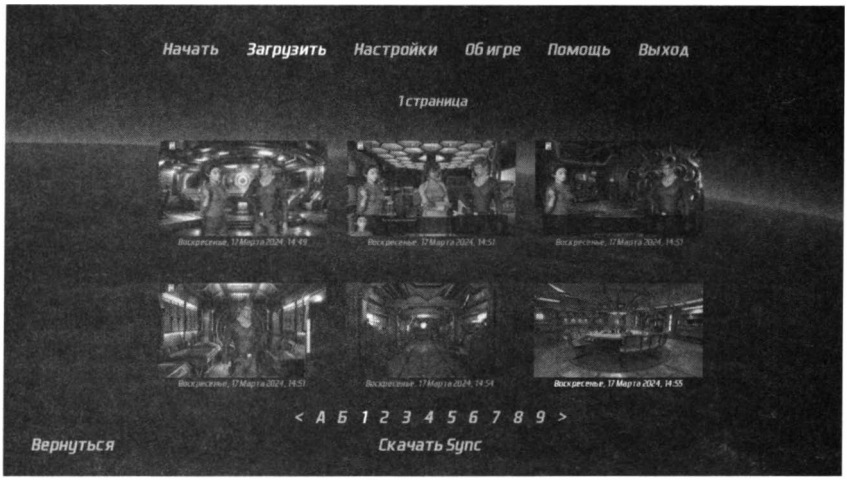


Рис. 5.35

Первая строка в **screen file_slots** записывает в переменную шаблон построения страницы (*default page_name_value = FilePageNameInputValue*), который будет использоваться для всех созданных вкладок сохранения и загрузки. Следующая кнопка позволяет переименовывать заголовок страницы. Введённый текст будет сохранён в вышеуказанной переменной.

```
button:
    style "page_label"
    key_events True
    xalign 0.5
    action page_name_value.Toggle()
    input:
        style "page_label_text"
        value page_name_value
```

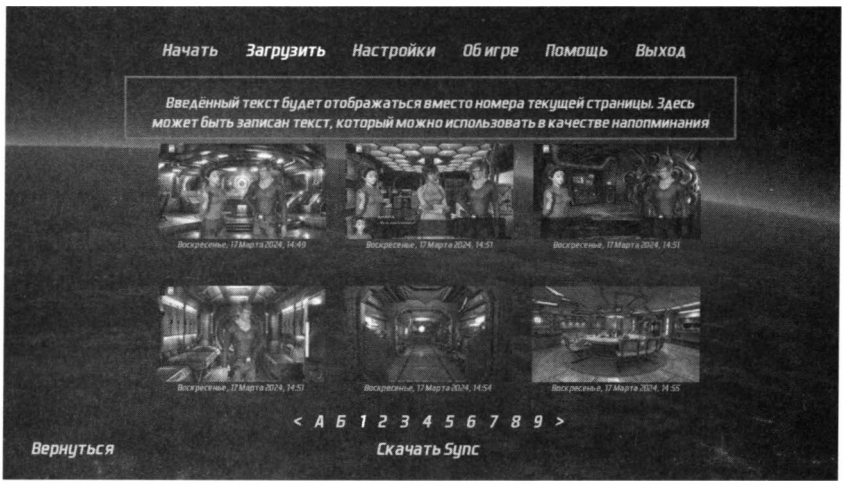


Рис. 5.36

Следующий блок отвечает за расположение слотов сохранения на экране:

```
grid gui.file_slot_cols gui.file_slot_rows:
    style_prefix "slot" # Стиль оформления таблицы
    xalign 0.5 # Расположение на экране
    yalign 0.5
    spacing gui.slot_spacing # Расстояние между слотами
    for i in range(gui.file_slot_cols * gui.file_slot_rows):
        $ slot = i + 1 # Цикл, раскладывающий слоты по порядку
        button: # Кнопка, представляющая собой слот
            action FileAction(slot)
            has vbox
            add FileScreenshot(slot) xalign 0.5
            text FileTime(slot, format=_("{#file_time}%A, %d %B %Y,
%H:%M"), empty=_("Пустой слот")):
                style "slot_time_text"
            text FileSaveName(slot):
                style "slot_name_text"
            key "save_delete" action FileDelete(slot)
```

Из раздела 5.2 мы знаем, что контейнер **grid** создаёт таблицы с указанным количеством столбцов и строк. В данном случае эти значения лежат в соответствующих переменных в файле *gui.rpy* (gui.file_slot_cols / gui.file_slot_rows). Быстро найти и изменить их можно **поиском** (Ctrl+F).

Если изменить эти значения на свои, то таблица сохранений может иметь другое количество слотов в рамках одной страницы.

Рис. 5.37

```
define gui.file_slot_cols = 4 # Изменили количество столбцов
define gui.file_slot_rows = 2 # В файле gui.rpy
```

Для добавления большого количества слотов на один экран нужно иметь в виду, что многие размеры придётся рассчитать с нуля. Соответствующие настройки можно найти там же в *gui.rpy*, чуть выше переменных с колонками и строками.

Рис. 5.38

```
define gui.slot_button_width = 200 # 414 стандартные значения
define gui.slot_button_height = 149 # 309
define gui.slot_button_borders = Borders(7, 7, 5, 5)
define gui.slot_button_text_size = 21 # Размер текста
define gui.slot_button_text_xalign = 0.5 # Расположение текста
define gui.slot_button_text_idle_color = gui.idle_small_color # Цвет
define gui.slot_button_text_selected_idle_color = gui.selected_color
define gui.slot_button_text_selected_hover_color = gui.hover_color

define config.thumbnail_width = 187 # 384 стандартные значения
define config.thumbnail_height = 106 # 216

define gui.file_slot_cols = 8 # Количество столбцов
define gui.file_slot_rows = 3 # Количество рядов
```

ВАЖНО! Перед настройкой новой таблицы, необходимо удалить все старые сохранения, так как старые скриншоты сохранений останутся прежних размеров и не будут вмещаться в новые слоты. Для удаления сохранений наведите курсор на слот, чтобы он подсветился, и нажмите клавишу "Delete".

5.13. Оформление кнопок выбора

За кнопки выбора отвечает отдельный экран **screen choice**, он имеет несколько настроек, часть из которых записана в файле *screen.rpy* и в *gui.rpy*.

Кроме того, полупрозрачные изображения кнопок можно найти в папке **gui/button**, они имеют названия *choice_idle_background.png* и *choice_hover_background.png*. Первая отображает кнопки в спокойном состоянии, вторая – при наведении курсора.

Изображения кнопок выбора можно заменить на свои, чтобы они соответствовали стилю игры. А остальные настройки необходимо сделать непосредственно в коде.

Рис. 5.39

В файле *gui.rpy*:

```
define gui.choice_button_width = 800 # Длина кнопки
define gui.choice_button_height = None # Ширина кнопки
define gui.choice_button_tile = False # Масштабирование картинки
define gui.choice_button_borders = Borders(50, 8, 50, 8) # Отступы
текста от краёв
```

```
define gui.choice_button_text_font = gui.text_font # Размер шрифта
define gui.choice_button_text_size = gui.text_size # Размер текста
define gui.choice_button_text_xalign = 0.5 # Расположение текста
внутри кнопки
define gui.choice_button_text_idle_color = "00fff4"# цвет текста
кнопки, когда курсор не наведён на неё
define gui.choice_button_text_hover_color = "#ffffff" # цвет текста
кнопки, когда курсор наведён на неё
define gui.choice_button_text_insensitive_color = '#8888887f' #
цвет текста неактивной кнопки
```

В файле *screens.rpy*:

```
screen choice(items):
    style_prefix "choice"
    vbox: # Все объекты раскладываются сверху вниз
        for i in items: # Цикл выводит ответы по порядку
            textbutton i.caption action i.action

style choice_vbox:
    xalign 0.5 # Расположение ответов на экране по "x"
    yalign 0.5 # Расположение ответов по "y"
    yanchor 0.5
    spacing gui.choice_spacing # Расстояние между ответами
    # Значение берётся из переменной в gui.rpy
```

5.14. Оформление быстрого меню

Быстрое меню (quick menu) представляет собой небольшой раздел интерфейса, который по умолчанию отображается внизу экрана.

Его можно выключить, изменить или добавить новые элементы.

Рис. 5.40

Для сокрытия этого экрана во время игры, необходимо выключить соответствующую переменную в нужном месте кода. Это может понадобиться, например, в мини-играх или во время работы таймера с ограничением времени на выполнение действия, чтобы игрок не мог сохраниться в этот момент.

```
label start:
    # При переключении переменной в значение False быстрое меню скрывается
    $ quick_menu = False
    jump mini_game # Перешли в мини-игру

label part_2:
    $ quick_menu = True # Включили быстрое меню
    lia "Что-то говорит"
```

Чтобы изменить кнопки быстрого меню, найдите в файле *screens.rpy* экран **screen quick_menu**.

```
screen quick_menu():
    zorder 100 # Открывается поверх экранов, имеющих меньшее
значение
    if quick_menu: # Если переменная в значении True
        hbox: # Горизонтальный бокс раскладывает текстовые кнопки
            style_prefix "quick" # Общий стиль кнопок
            xalign 0.5 # Расположение бокса на экране
            yalign 1.0
            textbutton _("Назад") action Rollback()
            textbutton _("История") action ShowMenu('history')
            textbutton _("Пропуск") action Skip()
            textbutton _("Авто") action Preference()
            textbutton _("Сохранить") action ShowMenu('save')
            textbutton _("Б.Сохранение") action QuickSave()
            textbutton _("Б.Загрузка") action QuickLoad()
            textbutton _("Опции") action ShowMenu('preferences')
            textbutton _("Скрыть диалог") action Hide('say')
            # Добавили свою кнопку
```

Помимо добавления и удаления кнопок из быстрого меню, ему можно добавить фоновое изображение или опциональную кнопку, благодаря которой игрок сам сможет скрывать и открывать этот элемент интерфейса. Это будет полезно в мобильных версиях.

Стоит иметь в виду, что на небольших экранах смартфонов эти кнопки будут очень маленькими и не удобными для использования. Обычно, их делают немного больше, но они, соответственно, занимают больше места или вовсе не помещаются на экране. В таком случае можно создать одну кнопку доступа, которая по умолчанию скрывает интерфейс, а когда игроку понадобится им воспользоваться, он без проблем сможет его открыть.

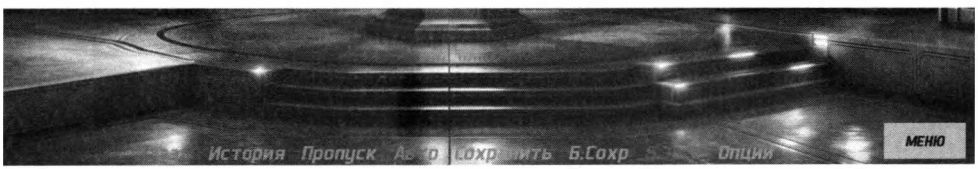


Рис. 5.41

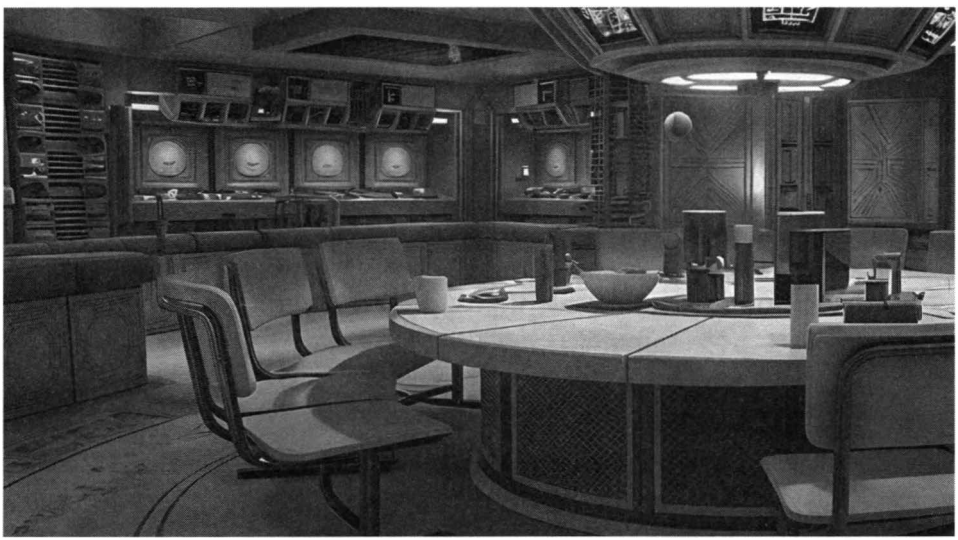
```
screen interface():
    tag interface
    button:
        action ToggleVariable("quick_menu", False, True)
        align(0.99,0.99)
        xsize 150 ysize 70
        idle_background "#ccccff"
        hover_background "#ddffff"
        text _("МЕНЮ") yalign .5 xalign .5 size 30 color "#000"
```

При нажатии на кнопку, она поочерёдно будет менять значение переменной **quick_menu**, таким образом, скрывая или открывая доступ к остальным кнопкам. В свою очередь, для скрытия самой кнопки "меню" во время мини-игры, нам просто нужно скрыть экран интерфейса стандартной командой **hide screen interface**.

Помните, что на мобильных телефонах нет возможности нажать правую клавишу мыши или "Esc", чтобы открыть меню. Поэтому полное скрытие или удаление быстрого меню сделает невозможным сохранение и загрузку во время игры.

Глава 6.

СТИЛИ В REN'PY



6.1. Применение стилей

В структуре движка Ren'Py стили используются для определения внешнего вида текста, кнопок и других элементов интерфейса. Они позволяют задавать различные параметры, такие как шрифт, цвет, размер, выравнивание и многое другое.

С помощью стилей можно легко изменить оформление текста во всей игре, просто изменив его в одном месте. Новые параметры будут применены к соответствующим элементам во всём скрипте.

Стиль (style) должен иметь уникальное название, как в случае *меток* (label) и *экранов* (screen). Он должен начинаться с буквы и может содержать буквы, цифры и подчеркивание. Стиль, также как метки и экраны, представляет собой отдельный блок со списком инструкций, определяющих визуальные характеристики стиля.

Ранее для разных кнопок и текста мы прописывали определённые параметры, такие как: расположение на экране, цвет, фон, шрифт и др.

```
text _("МЕНЮ") yalign .5 xalign .5 size 30 color "#000" font
"AeroMaticsBoldItalic.ttf.ttf"
```

Или:

```
text _("МЕНЮ"):
    yalign .5
    xalign .5
    size 30
    color "#000"
    font "AeroMaticsBoldItalic.ttf"
```

Такие варианты подходят, когда нужно настроить один определённый элемент должным образом. Однако когда подобные элементы повторяются в разных местах кода, чтобы не прописывать одни и те же параметры каждой кнопке или тексту, можно задать им общий стиль.

```
style fortext: # Создали стиль и дали ему название (fortext)
    yalign .5 # Расположение текста
    xalign .5
    size 30 # Размер текста
    color "#000" # Цвет текста
    font "AeroMaticsBoldItalic.ttf" # Шрифт текста
```

И теперь каждому тексту с такими параметрами мы можем указать стиль, в котором он должен быть отображён.

```
screen interface():
    text _("МЕНЮ") style "fortext"
```

Аналогичным образом можно сократить и настроить другие блоки кода, которые мы создавали ранее. Например, кнопки главного меню также имели схожие настройки.

```
    button: # Кнопка "Загрузить"
        action ShowMenu("load")
        align(0.15,0.4)
        xsize 250 ysize 150
        idle_background "gui/menu_butt.png"
        hover_background "gui/menu_butt2.png"
        text _("Загрузить") align(0.5,0.5) size 50 color "#ccccff"
font "AeroMaticsBoldItalic.ttf" hover_color "#ffffff"
    button: # Кнопка "Настройки"
        action ShowMenu("preferences")
        align(0.325,0.6)
        xsize 250 ysize 150
        idle_background "gui/menu_butt.png"
        hover_background "gui/menu_butt2.png"
        text _("Настройки") align(0.5,0.5) size 50 color "#ccccff"
font "AeroMaticsBoldItalic.ttf" hover_color "#ffffff"
```

Создаём новые стили, и заносим в них все повторяющиеся параметры наших кнопок и текста внутри них:

```
style buttonmenu: # Стиль кнопки меню
    xsize 250 ysize 150 # Размер кнопки
    idle_background "gui/menu_butt.png" # Изображение
    hover_background "gui/menu_butt2.png" # Изображение при наведении курсора

style buttontext: # Стиль текста в кнопке
    yalign .5 # Расположение текста
    xalign .5
    size 50 # Размер текста
    color "#fffff" # Цвет текста
    font "AeroMaticsBoldItalic.ttf" # Шрифт текста
```

Теперь, наш достаточно объёмный экран главного меню стал гораздо компактнее:

```
screen main_menu():
    tag menu
    button: # Кнопка "Загрузить"
        action ShowMenu("load")
        align(0.15,0.4) # Расположение на экране
        style "buttonmenu" # Стиль кнопки
        text _("Загрузить") style "buttontext"
    button: # Кнопка "Настройки"
        action ShowMenu("preferences")
        align(0.325,0.6)
        style "buttonmenu"
        text _("Настройки") style "buttontext"
```

Вместе с тем, применяя общий стиль ко множеству элементов, по необходимости можно изменять любой параметр для одной отдельной кнопки.

```
button: # Кнопка "Загрузить"
    action ShowMenu("load")
    align(0.15,0.4)
    style "buttonmenu"
    text _("Загрузить") style "buttontext" font "arialbi.ttf" size 70
    # Изменили шрифт и размер, остальные настройки берутся из стиля
```

Стоит иметь в виду, что некоторые стандартные названия стилей уже используются системой, и если вы назовёте свой стиль также, это может отразиться на внешнем виде стандартного интерфейса. Если такое произошло, просто немного измените название своего стиля.

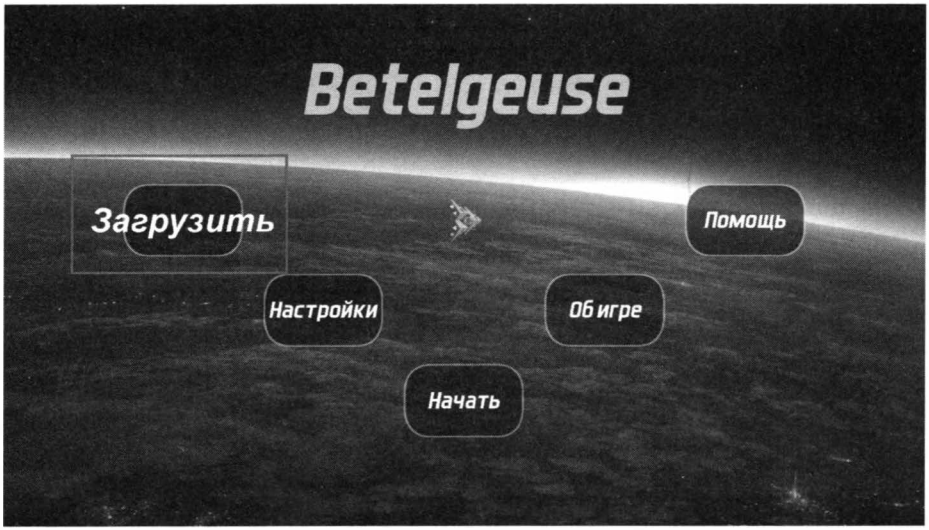


Рис. 6.1

Стилей в проекте может быть довольно большое количество для разных случаев и элементов. Поэтому рекомендуется создать для них отдельный файл, например *styles.rpy*.

6.2. Подробный разбор стилей

Некоторые элементы интерфейса в разное время могут находиться в разных состояниях, например кнопка, изображение или полоска бара. В каждом из этих состояний элемент может выглядеть по-разному: активная кнопка может иметь другой цвет или изображение, увеличенный размер текста и пр. Чтобы обозначить конкретное состояние, свойства могут иметь специальные префиксы, которые уточняют их значение.

В общей сложности существует **шесть визуальных состояний**, в которых объект может находиться:

1. **idle** – Состояние по умолчанию, объект не выбран, и на нем нет курсора.
2. **hover** – Состояние, когда курсор наведён на объект.
3. **selected_idle** – Выбранное состояние, элемент активен или выбран. Например, при выборе определённой вкладки в главном меню, кнопка этого раздела будет окрашена другим цветом.

4. **selected_hover** – Активное состояние, элемент активен и на нём находится курсор.
5. **insensitive** – Неактивное состояние, игрок не может взаимодействовать с объектом. Это может произойти, если для объекта не указана какая-либо функция или действие.
6. **selected_insensitive** – С элементом нельзя взаимодействовать, но он может иметь дополнительное свойство.

```
style buttonmenu: # Стиль кнопки меню
    idle_background "gui/menu_butt.png" # Изображение в спокойном состоянии
    hover_background "gui/menu_butt2.png" # Изображение при наведении курсора
    insensitive_background "gui/menu_butt3.png" # Изображение при
невозможности клика
```

Разные объекты имеют как общие свойства, так и конкретно стилевые, которые могут быть применены только к определённым типам объектов. Например, позиционирование на экране может быть применено как к изображениям, так и к тексту. Это общее свойство. Если указать в стиле шрифт, то он может быть применён к тексту, но не может к изображению, так как применим только к определённому типу.

Свойства для позиционирования на экране

- **alt** – Альтернативный текст, использующийся в режиме *self-voicing*.

```
style new_style:
    alt "Текст"
```

- **xpos** – Устанавливает позицию в пикселях по координате "x".
- **ypos** – Устанавливает позицию в пикселях по координате "y"
- **pos()** – Задаёт обе координаты.

```
style new_style:
    xpos 50
    ypos 250
    # Или
     pos (50,250)
```

- **xanchor** – Устанавливает якорь от левого края объекта.
- **yanchor** – Якорь от верхнего края объекта.
- **anchor()** – Устанавливает обе координаты.

```
style new_style:
   xanchor 50
   yanchor 250
    # Или
   anchor (50,250)
```

В разделе 4.5 было описано, в чём разница позиционирования с якорем и без него. Напомню, что все расчёты координат в Ren'Py ведутся от левого верхнего края экрана и объекта (изображения / текста). Чтобы отсчёт шёл от центра изображения или текста, им необходимо указать якорь – точку отсчёта.

- **xalign** – Работает аналогично **xpos**, однако устанавливает координаты в процентном соотношении от 0.0 до 1.0. Если установить координаты меньше нуля, объект будет расположен за левым краем экрана. Если установить больше единицы – произойдёт смещение за правую часть экрана.
- **yalign** – Также устанавливает координаты объекта в диапазоне от 0.0 до 1.0, но в данном случае нулевым значением является верхний край экрана. А единица – нижний край.
- **align()** – Устанавливает оба значения.

```
style new_style:
    xalign 0.5
    yalign 0.5
    # Или
    align (0.5,0.5)
```

- **xcenter** – Устанавливает позицию по центру. Работает аналогично xalign 0.5.
- **ycenter** – Устанавливает центр по координате "y" (yalign 0.5).
- **xoffset()** – После установки позиции сдвинет объект ещё на определённое количество пикселей.

- **yoffset** – Аналогично по координате "y".
- **offset()** – Одновременное смещение по обеим координатам.

```
style new_style:
    xoffset 100
    yoffset 55
   # Или
   offset (100,55)
```

- **xmaximum** – Устанавливает максимальную ширину элемента в пикселях. Может быть полезно для окон или полосок баров.
- **ymaximum** – Устанавливает максимальную высоту элемента.
- **maximum()** – Устанавливает оба значения.
- **xminimum** – Устанавливает минимальную ширину.
- **yminimum** – Минимальная высота.
- **minimum()** – Устанавливает оба значения.

```
style new_style:
    xmaximum 500
    ymaximum 300
    # Или
    maximum (500,300)
```

- **xsize** – Устанавливает ширину объекта в пикселях. Также может быть полезно для окон и кнопок.
- **ysize** – Устанавливает высоту в пикселях.
- **xysize()** – Устанавливает оба размера.

```
style new_style:
    xsize 450
    ysize 199
    # Или
    xysize (450,199)
```

- **xfill** – Может иметь значения *True* или *False*. Если *True* – элемент, например, фон, растянется на всю длину экрана или контейнера, в котором находится. При установке значения *False* – будет занимать минимально возможную ширину, ориентируясь на объекты, находящиеся внутри этого фона / контейнера.

- **yfill** – Аналогичная настройка, работающая по высоте.

```
style new_style:
    xfill True
    yfill False
```

- **area** – Область, имеющая заданные координаты и размеры. Включает в себя четыре значения – позицию по **x**, **y**, **ширину** и **высоту**.

```
style new_style:
    area (150,35,650,400)
```

Свойства, применимые к тексту

- **antialias** – Может принимать значения *True* / *False*. Если установлено на *True*, к тексту применяется сглаживание.

```
style new_style:
    antialias True
```

- **black_color** – Заменяет чёрный цвет шрифта указанным.

```
style new_style:
    black_color "#cceeff"
```

- **adjust_spacing True** – Масштабирует текст под размер экрана.
- **bold True** – Выделяет текст жирным
- **italic True** – Выделяет текст курсивом. Необходимо, чтобы шрифт его поддерживал.
- **underline True** – Весь текст будет подчёркнутым.
- **strikethrough True** – Весь текст зачёркнут.
- **vertical True** – Текст будет развёрнут вертикально. Сверху вниз.
- **color # "000000"** – Задаёт указанный цвет тексту.
- **first_indent 60** – Делает отступ на указанное количество пикселей для первой строки ("красная строка").

- **rest_indent 50** – Делает отступ для всех строк кроме первой (эффект вложенного списка).
- **font "DejaVuSans.ttf"** – Устанавливает шрифт для текста.
- **size 45** – Задаёт размер тексту.
- **justify True** – Выравнивает весь текст, кроме последней строки, например, для отступа с указанием автора.
- **kerning -0.5** – Расстояние между буквами. Отрицательные значения сближают, положительные – отдаляют символы друг от друга.
- **line_leading 15** – Отступ перед каждой строкой, например, общий отступ от края для всего текста.
- **line_spacing 9** – Расстояние между строками.
- **text_align 0.5** – Выравнивает текст по центру. 0.0 – по левому краю. 1.0 – По правому краю.
- **outlines [(1, "#408040", 0, 0)]** – Обводка текста. (Толщина обводки, "# Цвет текста", смещение обводки по **x**, **y**)
- **layout "subtitle"** – Делает все строки текста примерно одинаковой длины.
- **caret None** – отображает мигающую вертикальную черту в конце текста. Вместо *None* можно указать необходимое изображение.
- **emoji_font** – Указывает шрифт для эмодзи (смайликов).
- **language "unicode"** – Устанавливает правила переноса на новую строку. По умолчанию это значение установлено в системе. Для переноса только по пробелам, используется "western".
- **slow_cps 100** – Задаёт скорость вывода текста на экран. В данном примере 100 символов в секунду. Если значение установить на *True*, скорость вывода текста будет соответствовать той, что указана в настройках игры.
- **slow_abortable True** – При клике выводит медленно печатающийся текст мгновенно.

Свойства, применимые к окнам, фреймам и кнопкам

- **background "image.jpg"** – Устанавливает задний фон, который может быть залит цветом или изображением.

```
style new_style:
    background "image.jpg"
```

- **foreground "#000000"** – Верхний слой, который накладывается поверх контейнера. Также может быть в виде цвета или изображения.
- **left_padding 20** – Устанавливает отступ от левого края контейнера, после которого начнёт отрисовываться фон.
- **right_padding 20** – Аналогичный отступ с правой стороны.
- **xpadding 50** – Устанавливает одинаковый отступ слева и справа.
- **top_padding 20** – Отступ от верхнего края контейнера.
- **bottom_padding 20** – Отступ от нижнего края.
- **ypadding 40** – Одинаковый отступ сверху и снизу.
- **padding (15,15)** – Если установлены два числа, соответствует отступам слева и справа. Если четыре числа (15,15,30,25) – оставляет отступы со всех сторон окна.
- **modal True** – Текущий экран или контейнер перекрывает собой остальные, не позволяя взаимодействовать с чем-либо, не относящемуся к этому экрану.

Свойства, применимые к кнопкам (button)

- **child "image.jpg"** – Заменяет объект дочерним элементом, например, меняет кнопку на изображение, если она установлена в режим *insensitive* и не может быть активирована.

```
style new_style:
    child "image.jpg"
```

- **hover_sound "sound.mp3" –** Воспроизводит звук при наведении курсора на кнопку.
- **activate_sound "sound.mp3"** – Звук при клике на кнопку.
- **mouse "cursor1"** – Отображает определённый вид курсора при наведении на кнопку. Это курсор предварительно должен быть создан в настройках. Подробнее смотрите раздел "5.5. Оригинальный курсор".

- **focus_mask True** – Если изображение кнопки имеет прозрачные элементы, то при наведении на них курсора кнопка не будет реагировать.
- **keyboard_focus True** – Позволяет выбирать экранную кнопку стрелками на клавиатуре. Если некоторые из кнопок на экране не имеют этого значения, они будут пропущены при переключении стрелками.

Свойства, применимые к полоскам бара (bar)

- **bar_vertical True** – Переворачивает стандартный горизонтальный бар вертикально.

```
style new_style:
    bar_vertical True
```

- **bar_invert True** – Инвертирует отображение бара. Стандартная полоса заполняется слева на право, и снизу вверх, в случае вертикальной полосы. При инвертировании заполнение происходит в противоположную сторону.
- **bar_resizing True** – Отвечает за растягивание и отрисовку полоски бара. По умолчанию, во время анимации правая полоска становится прозрачной с лева на право. В то время как левая полоса начинает отрисовываться на освободившемся пространстве.

Рис. 6.2

- Если **bar_resizing** установлен в значении *True*, левая полоса воспроизводит анимацию вытягивания, а правая – сжимания.

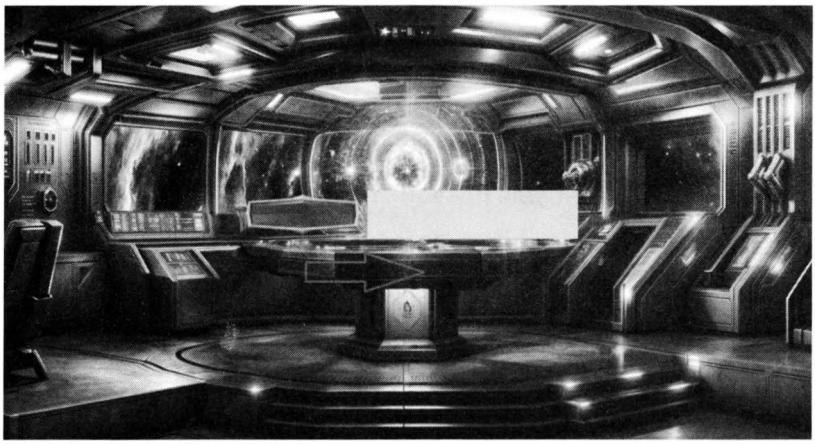


Рис. 6.3

- **left_gutter 15** – Устанавливает ширину левого буфера полоски. По умолчанию 0.
- **right_gutter 15** – Ширина буфера с правой стороны. По умолчанию 0.
- **top_gutter 15** – Аналогичная настройка для вертикального бара. Создаёт буферную часть верхнего края.
- **bottom_gutter 15** – Буферная часть для нижнего.

Для наглядности, бар состоит из пяти элементов.

Рис. 6.4

1 – Левый буфер
2 – Левая сторона
3 – Ползунок
4 – Правая сторона
5 – Правый буфер

- **left_bar** – Устанавливает изображение или цвет для левой полоски бара.
- **right_bar** – Аналогично для правой стороны.
- **top_bar** – Изображение верхней полосы для вертикального бара.
- **bottom_bar** – Изображение нижней полосы для вертикального бара.

```
style new_style:
    left_gutter 50
    left_bar "#000000"
    right_bar "image.png"
```

- **base_bar** – Устанавливает один общий цвет или изображение для обеих полос (левой, правой / верхней, нижней).
- **thumb** – Устанавливает фон для ползунка
- **mouse "cursor1"** – Устанавливает внешний вид курсора при наведении на ползунок.
- **keyboard_focus True** – Позволяет выделить и перемещать ползунок бара стрелками на клавиатуре.

Свойства, применимые к боксам (vbox, hbox)

- **spacing 20** – Устанавливает промежутки в пикселях между объектами бокса.

```
style new_style:
    spacing 20
```

- **first_spacing 30** – Устанавливает промежуток после первого элемента контейнера. Если до этого всем объектам был установлен **общий отступ** (spacing), то он не действует для первого интервала.
- **box_reverse True** – Объекты в коробке раскладываются в обратном порядке. По умолчанию **vbox** раскладывает элементы сверху вниз, а **hbox** – слева на право.

- **box_wrap True** – переносит элементы бокса на новую строку или столбец внутри контейнера, если они достигли его края. По умолчанию объекты выходят за рамки контейнера, продолжая складываться по порядку.
- **box_wrap_spacing 50** – Определяет интервал между столбцами и строками, если **box_wrap** установлен в значение *True*.

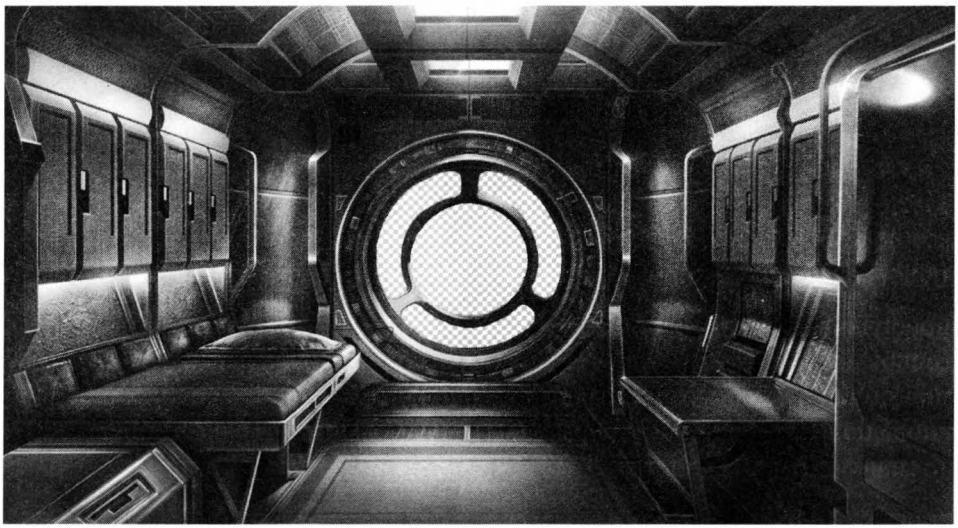
Свойства, применимые к таблицам (Grid)

- **xspacing 20** – Устанавливает расстояние между горизонтальными ячейками таблицы.
- **yspacing 20** – Расстояние между вертикальными ячейками таблицы.
- **spacing -20** – Общий промежуток для всех ячеек.

```
style new_style:
    xspacing 20
    yspacing 35
```

Глава 7.

АНИМАЦИЯ



7.1. Покадровая анимация

В разделах 4.3 и 4.4 мы поверхностно познакомились с анимациями в Ren'Py, и научились двигать спрайты с помощью встроенных инструментов. Однако, это лишь малая часть тех возможностей, которые может предложить движок. В этой главе мы разберём покадровую анимацию.

Из названия понятно, что подобная анимация состоит из поочерёдно сменяющихся кадров, которые в динамике создают "живое" изображение. Самый простой способ сделать это – быстро сменять кадры в коде игры.

```
label part_11:
    scene slide_1 with fade # Первый слайд
    pause 0.5 # Пауза пол секунды
    scene slide_2 with dissolve # Второй слайд заменяет предыдущий
    pause 0.5
    scene slide_3 with dissolve
    pause 0.5
    scene slide_4 with dissolve
    pause 0.5
```

Таким образом, несколько кадров с небольшим изменением в каждом из них, при смене создают эффект движения.

slide_1 **slide_2** **slide_3** **slide_4**

Рис. 7.1

Но этот вариант довольно громоздкий в плане количества кода, учитывая, что таких кадров может быть значительно больше четырёх. Для этого в движке можно создать виртуальное изображение, которое не будет существовать физически в файлах игры, но состоять из нескольких разных слайдов.

Давайте создадим для таких изображений отдельный файл, который назовём, к примеру, *animations.rpy*, и будем прописывать в нём все наши анимированные кадры. Создаются виртуальные картинки аналогично экранам, меткам и стилям. Нужно указать, что это изображение (image), и дать ему подходящее название.

```
image liara_moving: # Изображение с названием liara_moving
    "slide_1" with dissolve # Смена кадров, как в примере выше
    pause 0.3
    "slide_2" with dissolve
    pause 0.3
    "slide_3" with dissolve
    pause 0.3
    "slide_4" with dissolve
    pause 0.3
```

На первый взгляд может показаться, что разницы нет. Но теперь мы можем вызывать анимированное изображение одной командой в разных эпизодах игры.

```
label part_11:
    show liara_moving # Показали анимацию Лиары
    lia "Хм.."

label part_29:
    show liara_moving # Показали ещё раз
    lia "Кхм..."
```

Обратите внимание, что если помещать все изображения в папку **game/images**, то нет необходимости прописывать полный путь к каждому из них. Достаточно указать название в кавычках. Это также будет работать, если внутри папки **images** делать подпапки, в которых можно сортировать изображения.

Подобные анимированные картинки можно делать не только из полноценных кадров, но также из спрайтов. Например, таким образом можно

реализовать моргание глаз и дыхание героев, благодаря чему, из простых статичных изображений они превратятся в живых персонажей.

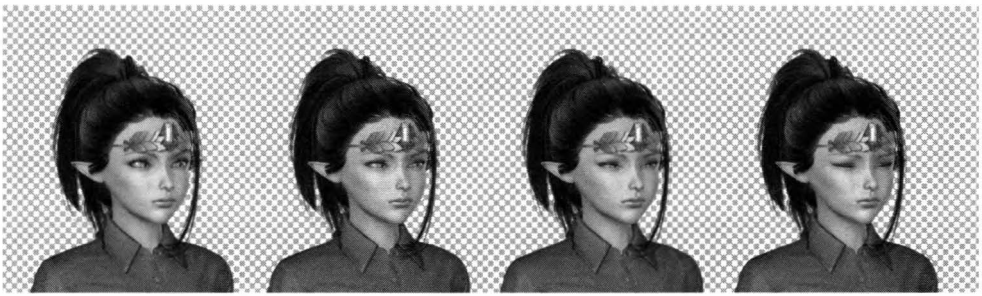


Рис. 7.2

```
image liara_blink: # Изображение моргающей Лиары
    "liara_blink1" with dissolve # Закрывает глаза
    pause 0.1
    "liara_blink2" with dissolve
    pause 0.1
    "liara_blink3" with dissolve
    pause 0.1
    "liara_blink4" with dissolve # Закрыла
    pause 0.1
    "liara_blink3" with dissolve # Открывает глаза
    pause 0.1
    "liara_blink2" with dissolve
    pause 0.1
    "liara_blink1" with dissolve
    pause 0.1
    choice: #
        pause 9
    choice:
        pause 12
    choice:
        pause 15
    repeat # Повторяет анимацию

label part_12:
    scene deck with fade
    show liara_blink with dissolve # Моргающая Лиара
```

Оператор **choice** предлагает движку выбрать один из нескольких вариантов.

В результате Лиара моргает через разные промежутки времени, чтобы не выглядеть как робот. Теперь такой анимированный спрайт можно использовать в разных местах на протяжении всей игры.

Таким же образом можно сделать анимацию ног, чтобы создать эффект походки. А затем анимированный спрайт перемещать с одной позиции на другую, как это было представлено в разделе 4.3. Стандартные эффекты переходов.

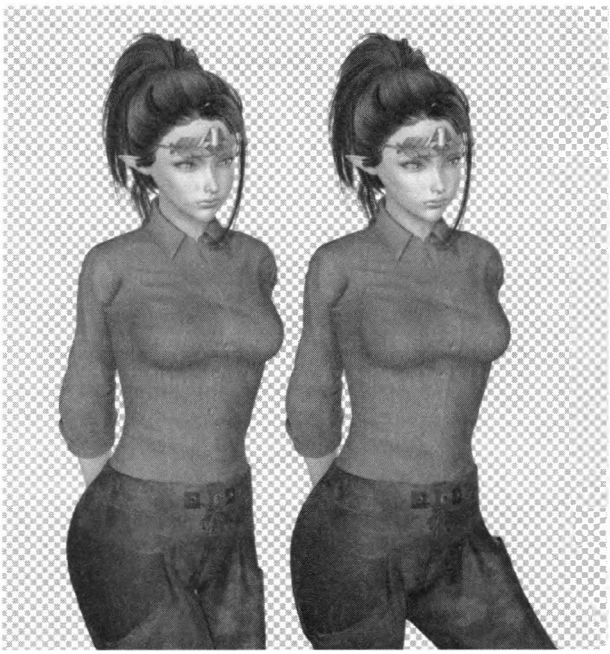


Рис. 7.3

Достаточно двух сменяющихся спрайтов, чтобы создать эффект перестановки ног.

```
image liara_moving: # Изображение шагающей Лиары
    "liara_step1" with dissolve #
    pause 0.8
    "liara_step2" with dissolve
    pause 0.8
    repeat

label part_12:
    scene deck with fade
    show liara_moving at left with dissolve
    pause 2
    show liara_moving at center with moveinright
    # Смещение в центр с эффектом движения вправо
```

7.2. Отображение видео

Благодаря возможности вставки видео, визуальную новеллу можно превратить в полноценный анимационный фильм, состоящий только из видео роликов. Преимуществом перед обычными мультфильмами будет то, что мы можем добавлять интерактив. В результате игрок будет не просто зрителем фильма, но и активным участником, который, делая выборы, может влиять на ход повествования.

Для добавления видеофрагмента в игру используется следующая конструкция:

```
label start:
    $ renpy.movie_cutscene("images/video.webm")
```

Стоит иметь в виду, что поддерживаются не все форматы видео. Так, например, один из самых популярных форматов mp4 движок не воспроизводит. Оптимальными вариантами будут: webm, mpeg, opus, ogg.

В выше описанном примере видео будет проигрываться, пока не закончится, или до клика игрока. Если есть необходимость показать видео до конца, без возможности пропуска, можно перед началом видео открыть модальный экран с таймером, который закроется по истечении видео.

```
label start:
    show screen blocking # Открыли экран поверх видео
    $ renpy.movie_cutscene("images/video.webm")

screen blocking():
    modal True # Не даёт взаимодействовать с нижними слоями
    timer 6 action Hide("blocking") # Закроет сам себя
```

Вместе с тем, функции воспроизведения видео можно добавить дополнительные аргументы.

```
label start:
    $renpy.movie_cutscene("images/video.webm", delay=5, loops=0,
stop_music=True)
```

- **delay=5** – Устанавливает продолжительность воспроизведения ролика в секундах.
- **loops=0** – Устанавливает количество повторов видео. Если установить -1, ролик будет воспроизводиться бесконечно.
- **stop_music=True** – Выключает фоновую музыку. Если установить на *False*, то запущенная ранее композиция продолжит играть.

Ещё один, более удобный способ вывода видео – сделать ролик изображением, аналогично тому, как мы создавали анимированные картинки в предыдущем разделе.

```
image video_1 = Movie(play="images/video.webm") # Создали изображение
label start:
    scene video_1 with fade # Вывели изображение
    # Или так
    show video_1 with dissolve
```

В таком виде ролик можно сделать фоновым изображением, поверх которого будут идти диалоги персонажей. Кроме того, этот способ позволяет вывести поверх экран навигации, например, для переключения между несколькими видео, создав тем самым видеогалерею или имитацию телевизора с несколькими каналами.

Рис. 7.4

Поверх ролика открывается экран, с помощью которого игрок может переключать видео:

```
image video_1 = Movie(play="images/video1.webm") # Три ролика
image video_2 = Movie(play="images/video2.webm") # в виде
image video_3 = Movie(play="images/video3.webm") # изображений

screen video_gallery(): # Экран с кнопками переключения роликов
    modal True # Блокируем нижние слои
    hbox: # Бокс с кнопками
        align (0.5, 0.05)
        textbutton _("1") action Jump("channel_1") # Кнопка перехода к
1-му видео
        textbutton _("2") action Jump("channel_2") # 2-е видео
        textbutton _("3") action Jump("channel_3") # 3-е видео
        textbutton _("X") action [Hide("video_gallery"),Jump("part_14")]
# Выход

label part_12:
    scene deck with fade
    window hide # Скрываем диалоговое окно
    show screen video_gallery # Открыли экран с кнопками
    label channel_1: # Запускается первое видео
        scene video_1 with dissolve
        pause
    label channel_2: # Второе видео
        scene video_2 with dissolve
        pause
    label channel_3: # Третье видео
        scene video_3 with dissolve
        pause
        jump channel_1 # Возвращаемся к первому видео
```

Благодаря паузе, каждый из видеороликов будет бесконечно зациклен пока пользователь не выберет следующее действие. Если паузам присвоить определённое количество времени (pause 15), например, на длину ролика, то после его истечения произойдёт переход к следующему видео.

В примере выше, при установке времени после третьего ролика скрипт перейдёт в начало, тем самым зациклив все три эпизода. Так же можно сделать прыжок к следующему лейблу, и тогда мы получим просто нарезку из нескольких видеосюжетов.

По аналогии можно реализовать кат-сцены в нужном месте игры, но, не добавляя кнопки навигации, чтобы игрок просмотрел все сцены последовательно и перешёл к следующей части повествования.

```
label start:
    scene video_1 with fade # Первый эпизод кат-сцены
    pause 5
    scene video_2 with dissolve # Второй
    pause 7
    scene video_3 with dissolve # Третий
    pause 6
    jump part_1 # Переходим к следующей части сценария
```

Таким образом вся игра может состоять только лишь из озвученных видеоматериалов, представляя собой полноценный фильм, поверх которого можно накладывать диалоги, кнопки выбора и разные эффекты с помощью экранов.

Вместе с тем, видео можно выводить не только в лейблах, но и в экранах (screen). Для примера, давайте откроем файл *screens.rpy*, найдём там экран главного меню, и заменим статичный фон на видео. Предварительно не забудьте создать новое виртуальное изображение, которое будет ссылаться на наше видео.

```
# Создали виртуальное изображение
image video_for_menu = Movie(play="images/video4.webm")

screen main_menu(): # Экран главного меню
    tag menu
    #add "gui/planeta.png" # Закомментировали старый фон
    add "video_for_menu" # Добавили новый фон
    # Далее идут спрайты, кнопки и прочие элементы меню
```

Исходя из выше описанного материала, мы также можем преобразовать наш стартовый логотип, который отображается перед главным меню. Напомню, что он прописывается в специальной метке **splashscreen**.

```
# Создали виртуальное изображение
image splashs_video = Movie(play="images/video5.webm")

label splashscreen:
    scene black with fade # показали чёрный экран
    pause 1 # сделали паузу на одну секунду
    scene splashs_video with fade # показали видеозаставку
    pause 3 # сделали паузу на длину ролика
    scene black with fade # снова затемнение
    return # перешли в главное меню
```

Если вступительный ролик довольно продолжительный, можно поверх видео открыть экран с кнопкой "Пропустить", чтобы пользователю не пришлось смотреть его каждый раз при запуске игры. Делается это аналогично тому, как мы создавали экран навигации для переключения между несколькими видео.

7.3. Преобразование (Transformation)

Transformation (преобразование) – это мощный инструмент в Ren'Py, который позволяет динамически изменять параметры объектов на экране.

С помощью **Transformation** можно изменять положение объектов, перемещать их по экрану, изменять масштаб, вращать и наклонять, менять цвет, использовать градиенты и цветовые эффекты, изменять прозрачность, использовать эффекты размытия, затенения и свечения, а также многое другое.

Благодаря преобразованию, помимо стандартных команд, таких как: **fade**, **dissolve**, **move**, **vpunch** и других, мы можем двигать объекты не только по предустановленным траекториям, но и по своим. Например, спрайт можно заставить двигаться вдоль экрана по зацикленному маршруту.

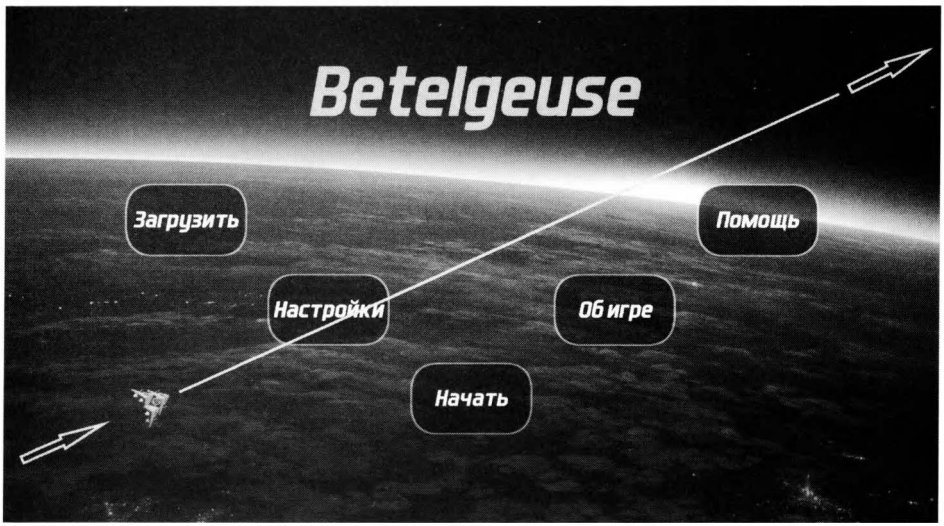


Рис. 7.5

На данном скриншоте спрайт космического корабля будет двигаться по заданному пути. Его траектория зациклена, поэтому в конце он возвращается в исходную позицию и начинает своё движение сначала. Так может продолжаться бесконечно.

Чтобы создать трансформацию, нужно указать соответствующую функцию и присвоить ей название, а затем после двоеточия и отступа прописать все необходимые инструкции. Синтаксис нам знаком по лейблам, экранам, стилям и изображениям.

```
transform firefly: # Создали трансформацию с любым названием
    xalign -0.1 yalign 0.99 rotate -20 # 1-я инструкция
    linear 25 xalign 1.1 yalign 0.05 # 2-я инструкция
    repeat # Повторить

screen main_menu(): # Экран главного меню
    tag menu
    add "video_for_menu" # Добавили новый фон
    # Далее идут спрайты, кнопки и прочее
    add "images/kosmolet.png" at firefly # Летящий спрайт космолёта
```

В примере мы создали трансформацию (transform) с названием *firefly*. Первая инструкция указывает движку установить спрайт на позицию xalign -0.1 (значение меньше нуля находится за пределами экрана, слева) и yalign 0.99.

> Свойство **rotate** указывает поворот спрайта в градусах.

Так как наш спрайт летит снизу вверх по диагонали, нужно немного повернуть его вокруг своей оси, против часовой стрелки.

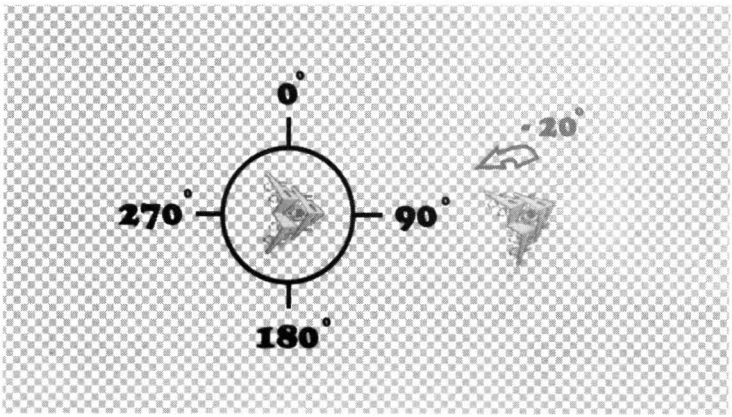


Рис. 7.6

Вторая инструкция сообщает движку, что за указанное время в 25 секунд (linear 25) спрайт переместиться со старой позиции на новую. Координата xalign 1.1 находится за пределами экрана с правой стороны. Свойство **rotate** не указывается, так как больше нет необходимости поворачивать спрайт.

Третья инструкция сообщает, что трансформацию нужно повторить. Спрайт снова появится за пределами экрана слева и начнёт свой путь вправо по диагонали. Подобных инструкций можно написать любое количество, благодаря чему спрайт звездолёта может летать по экрану в разные стороны и поворачивать по необходимости.

Далее, в экране главного меню вызывается наш спрайт (add "images/kosmolet.png") с использованием трансформации (at firefly).

> Функция **linear** перемещает объект линейно на одной скорости на протяжении всей анимации.

Однако, есть ряд других функций, которые работают немного иначе.

- **ease** – Начинает движение с небольшим замедлением, затем скорость немного увеличивается и к финишу снова уменьшается.

```
ease 25 xalign 1.1 yalign 0.05
```

- **ease_back** – Спрайт откатывается немного назад, словно разгоняясь, потом постепенно увеличивает скорость, и к концу замедляется.
- **ease_bounce** – Спрайт несколько раз двигается вперёд-назад, словно примеряясь к прыжку, а затем перемещается в указанном направлении. В конце снова несколько скачков.
- **ease_circ** – Работает похожим образом, как простая функция **ease**, только разгоняется ещё медленней, в середине пути движется быстро, а затем так же медленно останавливается.
- **ease_elastic** – Работает аналогично **ease_bounce**, только движения немного плавнее, словно растягивается резинка, а затем происходит резкий выстрел, как из рогатки.

- **easein_expo** – Происходит выстрел спрайта по направлению движения, с плавным замедлением в конце.
- **easein_quad** – Плавное движение 2/3 пути, с замедлением в конце.
- **easein_quart** – Быстрое движение 2/3 пути, с замедлением в конце.

Вместе с тем, помимо позиций **xyalign** и свойства **rotate**, в трансформациях можно использовать другие свойства позиционирования и движения.

- **xpos, ypos** – Работают аналогично **xyalign**, но с указанием координат в пикселях.

```
transform example:
    pos (50, 720)
    linear pos (150, 1720)
    # Или
    xpos 50 ypos 150
    linear 5 xpos 250 ypos 1340
```

- **xanchor, yanchor** – Задаёт центр координат объекта.
- **rotate** – Поворачивает объект на указанное количество градусов. Положительное число поворачивает по часовой стрелке, отрицательное – против часовой.
- **rotate_pad** – Может иметь значение *True* или *False*. Отвечает за отступы объекта от границ контейнера или экрана, в котором находится. Если значение *False*, то проходя мимо границ, может соприкасаться с ними. Если *True* – будут взяты небольшие отступы, чтобы объект не соприкасался с границами.
- **transform_anchor** – Может иметь значения *True* или *False*. Если *True* – придерживается координат, установленных в якоре. Например, во время вращения.
- **zoom** – Изменяет размер объекта. **xzoom** и **yzoom** изменяют размер только по ширине или высоте.

```
transform firefly:
    zoom 0.5 # Спрайт будет уменьшен на половину своего размера
    linear 10 zoom 2 # Увеличится в два раза от своего размера
```

- **alpha** – Отвечает за прозрачность спрайта. Принимает значения в диапазоне от 0.0 (полностью прозрачный) до 1.0 (непрозрачный).

```
transform firefly:
    alpha 0.0 # Спрайт невидим на экране
    linear 5 zoom 1.0 # Спрайт проявляется в течение пяти секунд
    linear 5 zoom 0.0 # Спрайт снова растворяется
    repeat # Повторяем анимацию, создавая эффект мигания объекта
```

- **blur** – Отвечает за размытие объекта, принимает дробные числа.
- **around** – Устанавливает центр координат относительно экрана или контейнера, в котором находится. Например, around (0.5,0.5).
- **xsize, ysize** – Отвечает за размер объекта, работает аналогично **zoom**.
- **delay** – Если трансформация используется в качестве перехода (как **fade** или **dissolve**), отвечает за длительность этого эффекта. Например, delay 1.5 (длительность эффекта полторы секунды).
- **xpan, ypan** – Отвечают за плавный поворот объекта по "x" и "y". Позволяют создать эффект панорамы (движущегося фона), имитирующий движение за окном автомобиля или космического корабля.

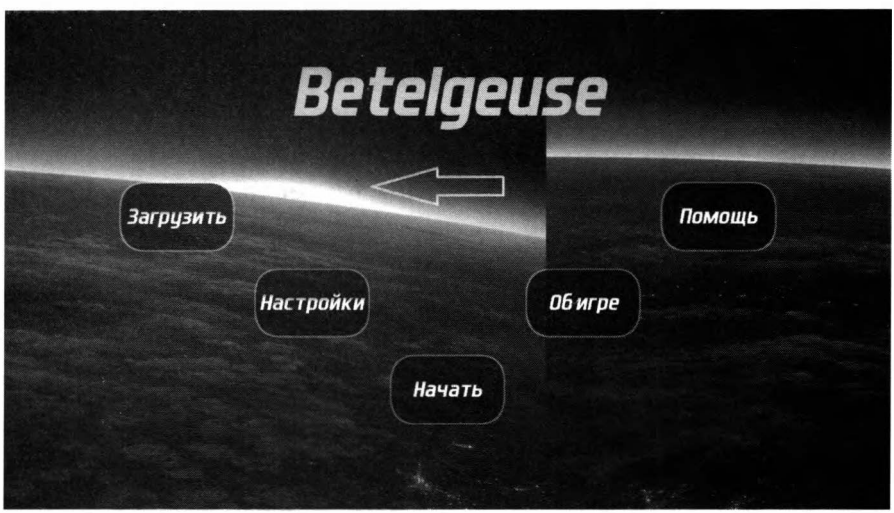


Рис. 7.7

Изображение двигается по кругу по координате "x". Если сделать бесшовные края, например, звёздного неба, это создаст эффект полёта слева на право.

```
transform example:
    xpan 0
    linear 100 xpan 1920
```

- **xtile, ytile** – Отвечает за дублирование объектов на экране по "x" и "y"

```
transform firefly:
    xtile 2
```

Рис. 7.8

- **block** – Позволяет добавлять в трансформацию дополнительные блоки, которые можно повторять отдельно от остальной трансформации.

```
transform firefly:
    pos (100,100) # Стартовая позиция спрайта
    block: # Блок 1
        linear 2 pos (200,200)
        linear 2 pos (300,300)
        linear 2 pos (200,200)
        repeat # Повторяем только блок 1
    time 20 # Через 20 секунд запускается
    block: # инструкция следующего блока
        linear 2 pos (600,600)
        linear 2 pos (700,700)
        linear 2 pos (600,600)
        repeat
```

Таким образом можно записать целую серию блоков, создав сложную анимацию, с множеством перемещений и других эффектов.

- **parallel** – Позволяет запускать несколько разных блоков параллельно. Это может быть полезно, когда две анимации имеют разное время срабатывания. Например, перемещение происходит каждые две секунды (linear 2), а мигание каждые три (linear 3).

```
transform firefly:
    pos (100,100) alpha 1.0 # Стартовая позиция и прозрачность
    parallel:
        linear 2 pos (200,200)
        linear 2 pos (300,300)
        linear 2 pos (200,200)
        repeat
    parallel:
        linear 3 alpha 0.0
        linear 3 alpha 1.0
        repeat
```

- **choice** – Этот оператор нам уже встречался в покадровой анимации, однако его можно использовать и для рандомизации трансформации.

```
transform firefly:
    pos (100,100) alpha 1.0 # Стартовая позиция и прозрачность
    choice: # Ren'Py выберет один из блоков
        linear 2 pos (200,200)
        linear 2 pos (300,300)
        linear 2 pos (200,200)
    choice:
        linear 3 alpha 0.0
        linear 3 alpha 1.0
```

- **crop** – Используется для обрезания части изображения или спрайта. Оператор может включать в себя четыре параметра (**x**, **y**, **width**, **height**). Координаты по "x", "y", ширина и высота.
  На скриншоте обрезали спрайт Лиары в полный рост до одной головы.

```
transform firefly:
    crop (50, -60, 960, 540)
```

Рис. 7.9

- **on** – Этот оператор используется для анимации кнопок при наведении на них курсора.

```
transform firefly:
    anchor (0.5,0.5) # Отсчёт координат на центр кнопки
    on hover: # Событие при наведении курсора
        linear 1 zoom 1.5 # Увеличили в полтора раза
    on idle: # Событие при снятии курсора
        linear 1 zoom 1.0 # Вернули стандартный размер

screen main_menu():
    button at firefly: # Кнопка "Загрузить"
        action ShowMenu("load")
```

В примере кода кнопка увеличивается при наведении курсора, а при снятии – возвращается к первоначальному размеру.

7.4. Анимация кнопок

Все вышеописанные трансформации работают не только со спрайтами, но и с текстом, и с кнопками. В последнем примере мы увеличивали кнопку при наведении на неё курсора. Вместе с тем, её также можно перемещать по экрану.

Такая анимация может быть полезна при создании мини-игр или дополнительного интерактива. Например, спрайт космолёта в главном меню можно сделать летающей кнопкой, при клике на которую игрока будет отправлять в бонусную комнату с пасхалкой (отсылкой на что-либо) или альтернативное начало игры.

В первую очередь нужно создать подходящую анимацию:

```
transform random_flight: # Анимация случайной траектории
    # Случайная координата по "y" за пределами экрана слева
    xpos -100 ypos renpy.random.randint (0,1080)
    # Случайная координата по "y" за пределами экрана справа
    linear 25 xpos 2100 ypos renpy.random.randint (0,1080)
    repeat # Повторить
```

Данная анимация установит нашу кнопку в случайной позиции слева за экраном и отправит в полёт вправо. Теперь, вместо спрайта космолёта сделаем кнопку с анимацией полёта.

```
screen main_menu(): # Экран главного меню
    tag menu
    add "gui/planeta.png" # Фон
    imagebutton at random_flight: # Кнопка с анимацией полёта
        idle "kosmolet.png" # Изображение космолёта
        hover "kosmolet.png" # Изображение при наведении курсора
        action Start("alt_start")# Отправляет в указанную метку
```

Далее нам нужно создать **метку** "alt_start", которая может быть альтернативным стартом или бонусной комнатой с пасхалкой (отсылкой на что-либо).

```
label alt_start: # Альтернативное начало
    scene medcenter with fade # Сцена
    show xeona normal with dissolve # Спрайт
    xeo "О, ты уже пришёл в себя?" # Реплика
    # и т.д.
```

Кроме того, наша кнопка может открывать скрытый экран главного меню, в котором также можно поместить бонусные материалы. Например, галерею, музыкальный плеер или что-то ещё. Для этого нужно создать новый экран, и отправлять пользователя туда при клике.

```
screen bonus_screen():
    tag menu # Экран включён в группу главного меню
    # Использовать экран игрового меню
    use game_menu(_("Об игре"), scroll="viewport"):
        hbox: # Горизонтальный бокс
            align (0.5,0.5) # Бокс по центру экрана
            spacing 100 # Расстояние между изображениями
            box_wrap True # Перенос на новую строку
            add "bonus_image_1" # Бонусные изображения
            add "bonus_image_2"
            add "bonus_image_3"
            add "bonus_image_3"
            add "bonus_image_1"
            add "bonus_image_2"
```

Осталось привязать кнопке новое действие:

```
screen main_menu(): # Экран главного меню
    tag menu
    add "gui/planeta.png" # Фон
    imagebutton at random_flight: # Кнопка с анимацией полёта
        idle "kosmolet.png"
        hover "kosmolet.png"
        action ShowMenu("bonus_screen")# Открывает новый экран
```

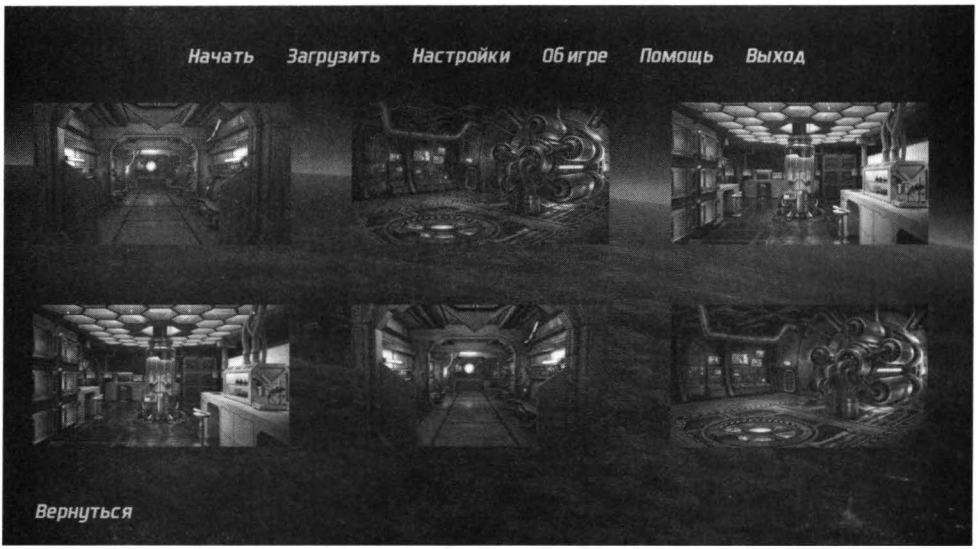


Рис. 7.10

Таким же образом анимируются кнопки в мини-играх, благодаря чему можно сделать простенький шутер (игра-стрелялка) от первого лица.

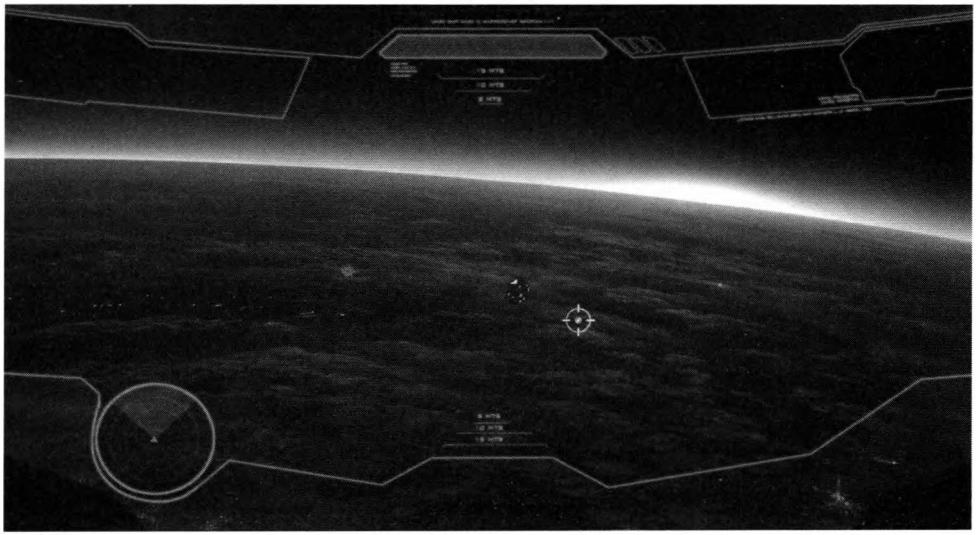


Рис. 7.11

Для начала создадим лейбл, при попадании в который будут установлены настройки и запуск мини-игры:

```
label shooter: # Метка с настройками
    $ quick_menu = False # Выключили быстрое меню внизу экрана
    $ config.rollback_enabled = False # Выключили откат назад
    $ aster1 = True # Включили переменную для кнопки-астероида
    $ default_mouse = "cursor3" # Включили курсор в виде прицела
    $ quantity = 0 # Переменная для подсчёта сбитых астероидов
    show screen asteroids with fade # Показали экран с мини-игрой
    pause
```

Теперь создадим анимацию, которая будет вращать астероид и запускать его по экрану:

```
transform aster_fly: # Анимация для летящих астероидов
    anchor (0.5,0.5) # Якорь координат по центру спрайта
    parallel: # Две параллельные анимации
        rotate 0 # Вращение
        # Вращение со случайной скоростью в диапазоне 5-15
```

```
        linear renpy.random.randint (5,15) rotate 360
        repeat
    parallel: # Полёт от края до края экрана
        # Случайная координата по "y" в диапазоне 0-1080
        xpos -100 ypos renpy.random.randint (0,1080)
        # Случайная скорость полёта и координата по "y"
        linear renpy.random.randint (5,15) xpos 2100 ypos renpy.
random.randint (0,1080)
        repeat
```

Далее создаём экран мини-игры:

```
screen asteroids(): # Экран мини-игры
    modal True # Игнорирует клики по нижним слоям
    # Экран с таймером, по истечении которого игра завершается
    timer 60.0 action [Hide("asteroids"), Jump("final_game")]
    add "planeta" # Фоновое изображение космоса
    if aster1: # Кнопка отображается, если переменная True
        imagebutton at aster_fly: # Кнопка в виде летящего астероида
            idle "aster1" # Изображение астероида
            hover "aster1"
            focus_mask True # Игнорируются прозрачные края кнопки
            # При наведении меняем прицел на красный
            hovered SetVariable("default_mouse", "cursor4")
            # При снятии возвращаем синий прицел
            unhovered SetVariable("default_mouse", "cursor3")
            action [SetVariable("aster1", False),
                    SetVariable("default_mouse", "cursor3"),
                    SetVariable("quantity", quantity+1)]
    # Этот таймер перезапускает астероид каждые 15 секунд
    timer 15.0 repeat True action SetVariable("aster1", True)
```

При клике по кнопке срабатывает **action**, который выключает переменную **aster1**, и астероид пропадает с экрана. Следующая переменная возвращает курсору синий цвет, а переменная **quantity** увеличивается на единицу, подсчитывая тем самым, количество попаданий.

В данном примере, для сокращения количества кода, приведена только одна кнопка-астероид. По желанию их можно добавить больше, каждая из которых будет лететь со своей случайной скоростью и вращением. Более того, можно создать дополнительные блоки в анимации, чтобы метеориты летали в разные стороны.

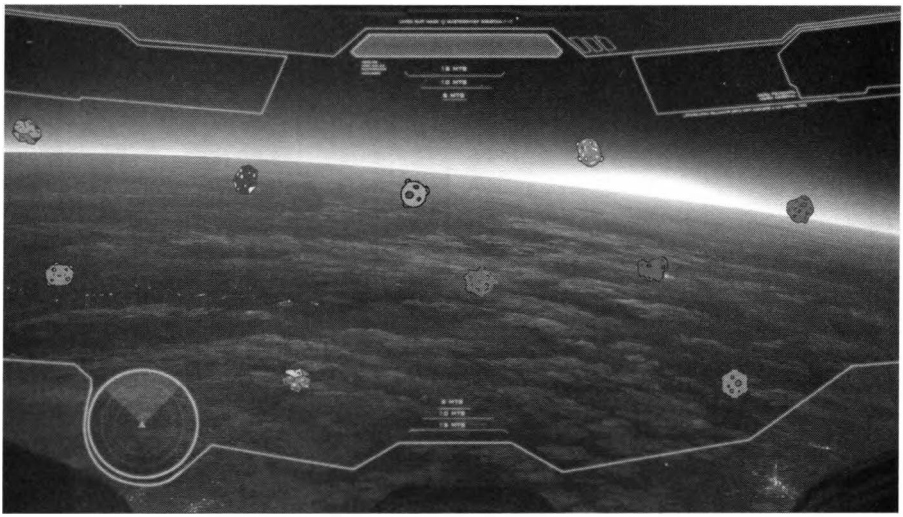


Рис. 7.12

Интерфейс космического корабля является всего лишь декоративной частью фона, но его можно сделать информативным. Например, выводить на экран количество сбитых астероидов, количество промахов, количество снарядов и т.д. Как оформить интерфейс, описывалось в пятой главе "Система экранов".

По истечении 60-ти секунд экран с игрой закрывается, и происходит переход в финальную метку мини-игры, в которой отображается количество сбитых астероидов. В зависимости от этого количества, Лиара хвалит игрока или расстраивается из-за плохого результата. А также меняются различные характеристики и открываются награды.

```
label final_game: # Метка после мини-игры
    $ quick_menu = True # Включили быстрое меню
    $ config.rollback_enabled = True # Включили откат назад
    $ default_mouse = "default" # Вернули обычный курсор
    scene planeta
    "Астероидов сбито: [quantity]" # Количество сбитых астероидов
    if quantity >9: # Если сбили больше 9-ти
        lia "Ты молодец, сбил большинство астероидов!"
        $ persistent.achievement2 = True # Открыли достижение
        $ liara_relat +=5 # Улучшили отношение с персонажем
        $ shooting_skill += 5 # Улучшили навык стрельбы
    elif quantity >4: # Если сбили больше 4-х, но меньше 10
        lia "Неплохо, продолжай тренироваться"
        $ liara_relat +=2 # Улучшили отношение с персонажем
```

```
        $ shooting_skill += 1 # Улучшили навык стрельбы
    else: # Если 4 и меньше
        lia "А если бы это не были учебные астероиды?!"
        lia "Ты бы подверг опасности миллионы жителей планеты!"
    jump part_11 # Переход в следующий эпизод новеллы
```

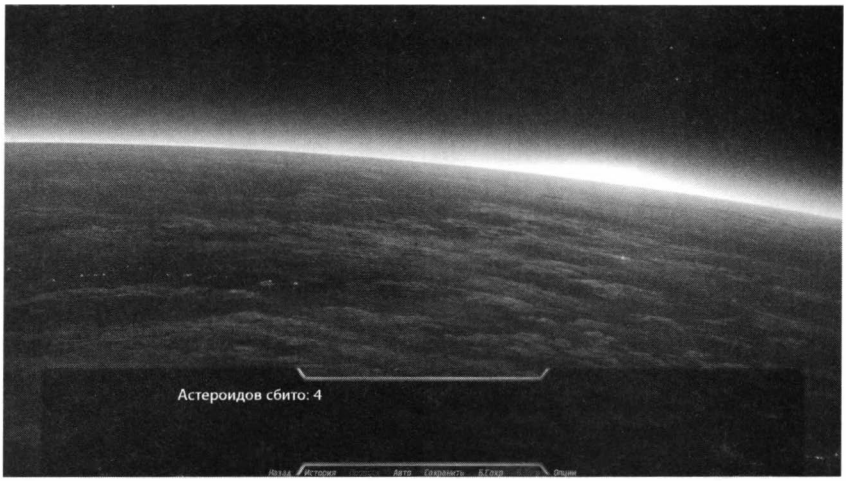


Рис. 7.13

Помимо полётов и вращений, к кнопкам, тексту и изображениям можно применять другие анимации. Например, мигание, что позволит нам реализовать свечение звёзд на заднем плане, светящиеся лампочки внутри помещений или мигающие вывески заведений.

Мерцания, мигания и вспышки можно сделать двумя способами. С помощью покадровой анимации или трансформацией.

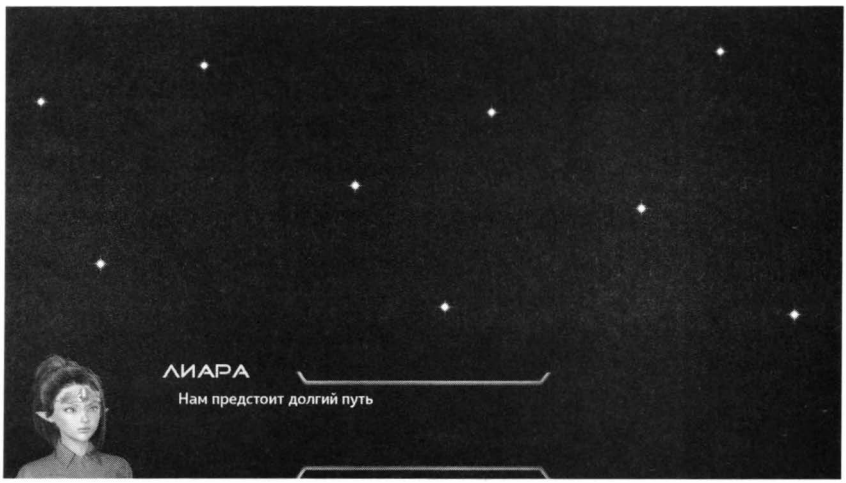


Рис. 7.14

Сначала рассмотрим пример мерцания звёзд с помощью трансформации. Для этого создадим следующую анимацию:

```
transform anim_stars: # Мерцание звёзд
    anchor (0.5,0.5) # Якорь координат по центру спрайтов
    parallel: # Случайное расположение звёзд на небе
        xpos renpy.random.randint (0,1920) ypos renpy.random.randint (0,1080)
    parallel: # Мерцание через разный промежуток времени
        alpha 0.7
        linear renpy.random.randint (5,9) alpha 1.0
        repeat
```

В приведённом коде, в первом блоке параллели каждая звезда на небе получит случайную координату. Во втором блоке каждая звезда будет мерцать в диапазоне от 0.7 до 1.0 за разный промежуток времени.

Теперь создадим экран со звёздами:

```
screen boundless_space(): # Экран с анимированными звёздами
    # Добавляем одно и то же изображение несколько раз
    add "star" at anim_stars
    add "star" at anim_stars
    add "star" at anim_stars
    add "star" at anim_stars
    add "star" at anim_stars
    add "star" at anim_stars
    add "star" at anim_stars
    add "star" at anim_stars
    add "star" at anim_stars
```

После изучения циклов, мы сможем выводить большое количество однотипных изображений более коротким способом, чтобы не прописывать каждую отдельную звезду, так как их может быть гораздо больше.

Завершающим штрихом нам остаётся показать экран с анимацией в нужном месте игры:

```
label part_19:
    scene black # Чёрный экран, который будет ночным небом
    show screen boundless_space with dissolve # Показали экран
    lia "Нам предстоит долгий путь"
    hide screen boundless_space with dissolve # Скрыли экран
```

Второй способ анимирования изображений и кнопок заключается в покадровой анимации. Это делается аналогично тому, как мы заставляли Лиару моргать. Для этого нужно так же создать несколько кадров одного изображения с разной цветовой подсветкой или другим эффектом.

Рис. 7.15

А затем собрать из них виртуальное изображение:

```
image planshet: # Зацикленная анимация
    "tablet1"
    pause 0.3
    "tablet2"
    pause 0.3
    "tablet3"
    pause 0.3
    "tablet4"
    pause 0.3
    "tablet5"
    pause 0.3
    repeat
```

Теперь это анимированное изображение можно привязать к кнопке. В результате она будет мигать, призывая игрока нажать на неё.

```
screen incoming_call():
    imagebutton:
        pos (753,532) # Позиция кнопки на экране
        idle "planshet" # Анимированное изображение
        hover "planshet" # Изображение при наведении курсора
        action [Hide("incoming_call"), Jump ("part_28")]
        # Закрыли этот экран, прыгнули в следующий эпизод
```

Далее открываем экран в нужном месте сценария:

```
label part_21:
    scene medcenter with dissolve
    show screen incoming_call
    pause
```

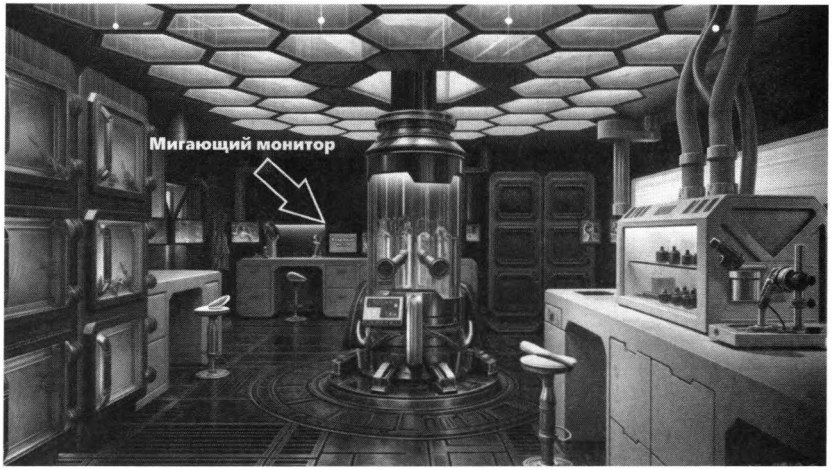


Рис. 7.16

Если в **hovered** изображение вставить более быструю анимацию, то при наведении курсора мигание ускорится. Также можно поставить статичное изображение, тогда при наведении экран будет замирать.

Аналогичным образом можно сделать вывески и лампочки, которые будут мигать, но при клике по ним ничего не будет происходить. Они будут просто анимированной частью фона. В таком случае их можно вывести в лейбле без использования экрана (screen).

```
label part_22:
    scene medcenter with dissolve
    show planshet: # Показали анимированное изображение
        pos (753,532) # На этой позиции
    kay "Хм, здесь никого нет"
```

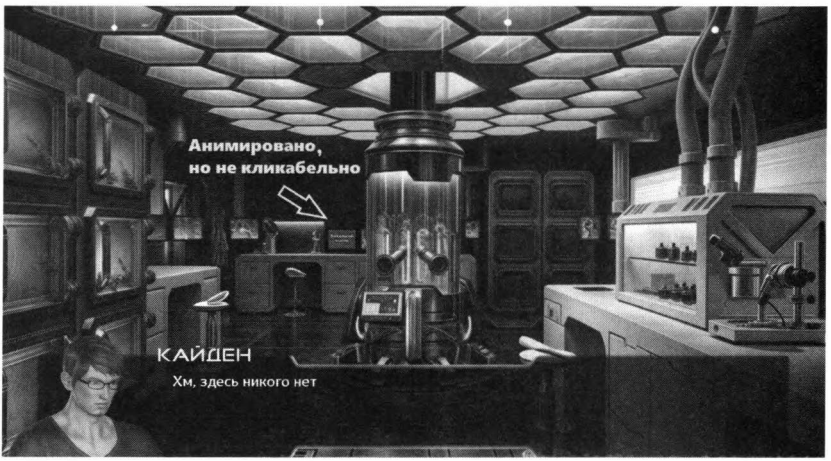


Рис. 7.17

7.5. Создание собственных анимаций

Ранее мы научились двигать изображения и кнопки по экрану. А что если двигать текст? В таком случае можно реализовать титры для финала нашей истории.

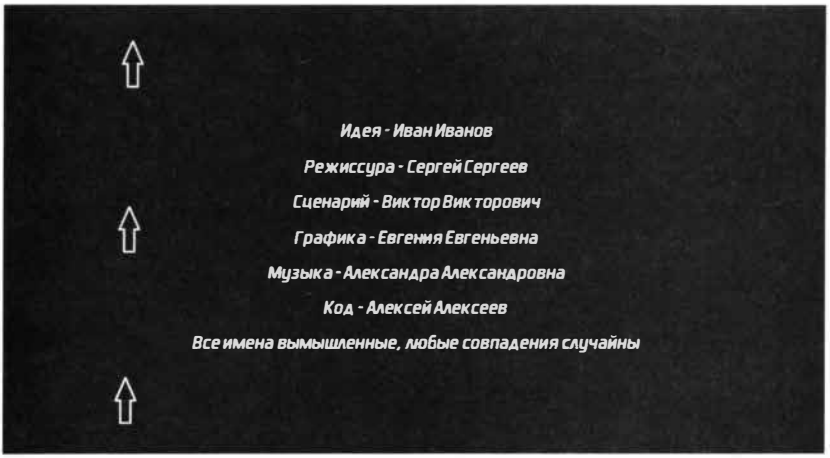


Рис. 7.18

Чтобы создать плавно поднимающиеся титры, создадим соответствующую трансформацию:

```
transform fin_titr: # Анимация финальных титров
    xalign 0.5 yalign 1.9 # Стартовая позиция внизу за экраном
    linear 50.0 xalign 0.5 yalign -0.9 # Конечная позиция вверху
```

Стартовая и конечная позиции по координате "y" (1.9 и -0.9) могут изменяться в зависимости от высоты будущего контейнера с текстом. Их необходимо настраивать после полной реализации всех элементов.

Теперь создадим стиль для текста:

```
style titl_text: # Стиль для текста титров
    color "#fae7b5" # Цвет текста
    xalign 0.5 # Расположение текста по центру
    size 45 # Размер текста
    font "fonts/AeroMaticsBoldItalic.ttf" # Шрифт
```

Наш стиль имеет название **titl**, который также является префиксом для стилизации текста (_text), это сообщает движку, что данные инструкции нужно применить только к тексту. То есть, если в экране будут картинки, кнопки и что-то ещё, этот стиль не будет к ним применён.

Далее, давайте напишем экран с текстом, в котором укажем всю команду разработчиков, принявших участие в создании игры:

```
screen titles(): # Экран с титрами
    # modal True # Если раскомментировать, титры нельзя будет пропустить
    vbox at fin_titr: # Вертикальный бокс с анимацией
        spacing 30 # Расстояние между строками
        style_prefix "titl" # Стиль для текста с префиксом
        text _("Идея - Иван Иванов") # Подчёркивание и скобки
        text _("Режиссура - Сергей Сергеев") # нужны для перевода
        text _("Сценарий - Виктор Викторович") # на другой язык
        text _("Графика - Евгения Евгеньевна")
        text _("Музыка - Александра Александровна")
        text _("Код - Алексей Алексеев")
        text _("Все имена вымышленные, любые совпадения случайны")
```

В настройках экрана мы имеем закомментированную строку **modal True**. То есть этот параметр не учитывается движком. Во время воспроизведения финальных титров игроку не обязательно их досматривать, он может кликнуть в любое место экрана и сразу перейти в главное меню. Если удалить символ решётки, клики по экрану будут игнорироваться, и пользователю придётся досматривать титры до конца или выходить из игры принудительно.

Остаётся только создать метку, в которую будет направлен игрок по завершении игры:

```
label titri: # Метка с отображением титров
    $ quick_menu = False # Скрыли быстрое меню внизу экрана
    scene black with dissolve # Показали чёрный экран
    show screen titles with dissolve # Показали экран с титрами
    pause 50 # Пауза на время анимации
    hide screen titles with dissolve # Закрыли экран
    pause 1
    return # Вернулись в главное меню
```

После команды **show screen titles** движок становится на паузу, пока воспроизводится анимация. Через 50 секунд закроется экран с титрами, которые к этому моменту уйдут за верхний край экрана, и через ещё одну секунду

произойдёт выход в главное меню. Если во время анимации игрок кликнет по экрану, пауза прервётся, и выполнятся следующие инструкции. В случае, когда команда **modal True** активна, клики по экрану не сработают.

Вместе с тем, с помощью трансформаций можно создать и другие анимации. Давайте рассмотрим несколько из них для понимания возможностей движка. Например, стандартный эффект **vpunch**, отвечающий за быструю встряску фона или спрайта, можно сделать самому, со своими настройками длительности и эффектом анимации.

```
transform new_vpunch: # Новая продолжительная встряска
    ypos -0.1
    linear 0.1 ypos 0.05
    linear 0.1 ypos -0.05
    linear 0.1 ypos 0.05
    linear 0.1 ypos -0.05
    linear 0.1 ypos 0.05
    linear 0.1 ypos -0.05
    linear 0.1 ypos 0.0 # Остановили на стандартной позиции

label start:
    scene bridge at new_vpunch # Показали фон с эффектом встряски
```

Чтобы сделать мигание или долгий **эффект растворения**, можно поэкспериментировать с прозрачностью.

```
transform blink: # Мигание объекта
    alpha 1.0
    linear 0.1 alpha 0.5
    linear 0.1 alpha 1.0
    linear 0.1 alpha 0.5
    linear 0.1 alpha 1.0
    linear 0.1 alpha 0.5
    linear 0.1 alpha 1.0
    linear 0.1 alpha 0.5
    linear 0.1 alpha 1.0
label start:
    scene bridge
    show liara normal at blink
```

В примере выше спрайт Лиары мерцает, создавая **эффект плохо работающей голограммы**. А для медленного растворения объекта нужно прописать долгий переход от непрозрачного состояния к полностью прозрачному.

```
transform long_dissolve: # Долгое растворение
    alpha 1.0
    linear 7 alpha 0.0 # Растворение за семь секунд
label start:
    scene bridge
    show liara normal at long_dissolve
```

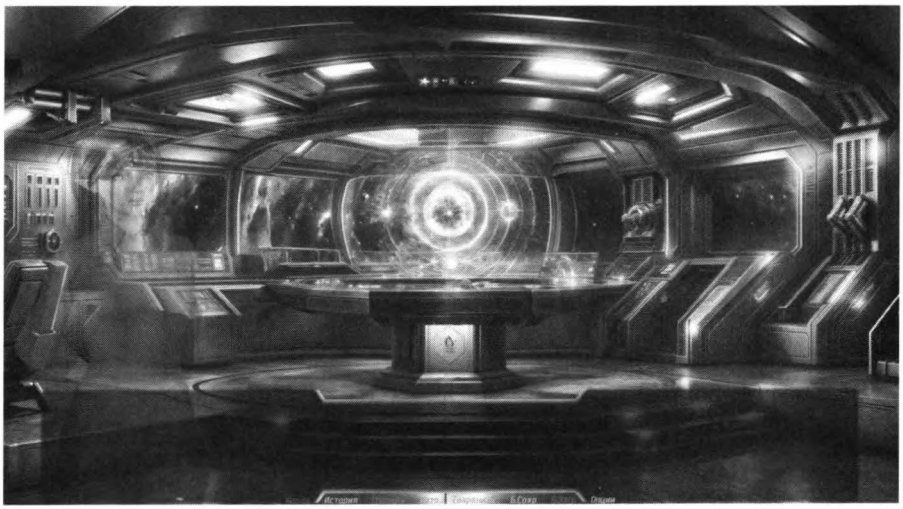


Рис. 7.19

Ранее, с помощью покадровой анимации мы создавали **эффект походки персонажа**. Теперь к нему можно добавить медленное покачивание вверх-вниз и перемещение по экрану, создав эффект прогулки героя по локации.

```
image liara moving: # Изображение шагающей Лиары
    "liara_step1" with dissolve #
    pause 0.8
    "liara_step2" with dissolve
    pause 0.8
    repeat

transform walk: # Слева направо + вверх-вниз
    parallel:
        xalign 0.1
        linear 5 xalign 0.9 # Слева направо за пять секунд
    parallel:
        ypos 10
        linear 0.5 ypos 0
        linear 0.5 ypos 10 # Плавно вверх-вниз
        repeat
```

```
label start:
    scene bridge
    show liara moving at walk # Лиара прогуливается
    pause 5 # Пауза на время прогулки Лиары
    # Заменяем анимированный спрайт статичным
    show liara normal2 with dissolve # Остановилась и повернулась
```

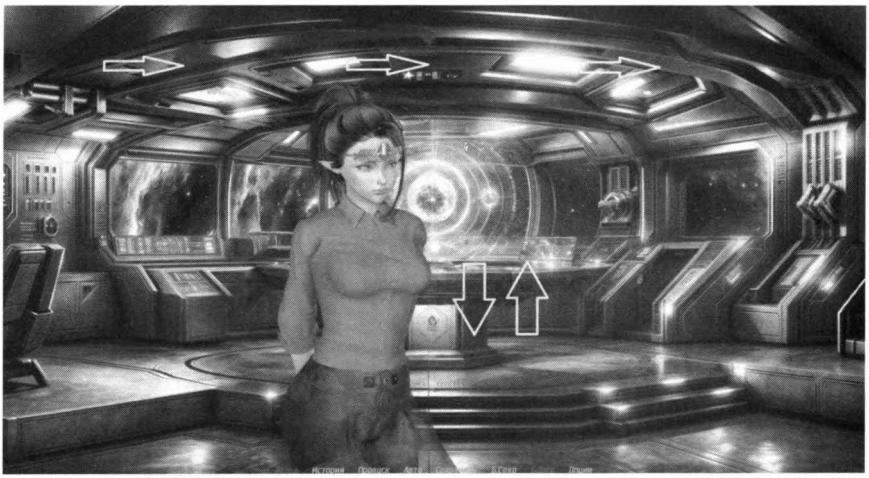


Рис. 7.20

Ещё один интересный эффект – **фокусировка на определённой детали изображения**. Сначала картинка отображается целиком, а затем происходит смещение и увеличение к определённому месту на ней, создавая эффект плавного "наезда" камеры.

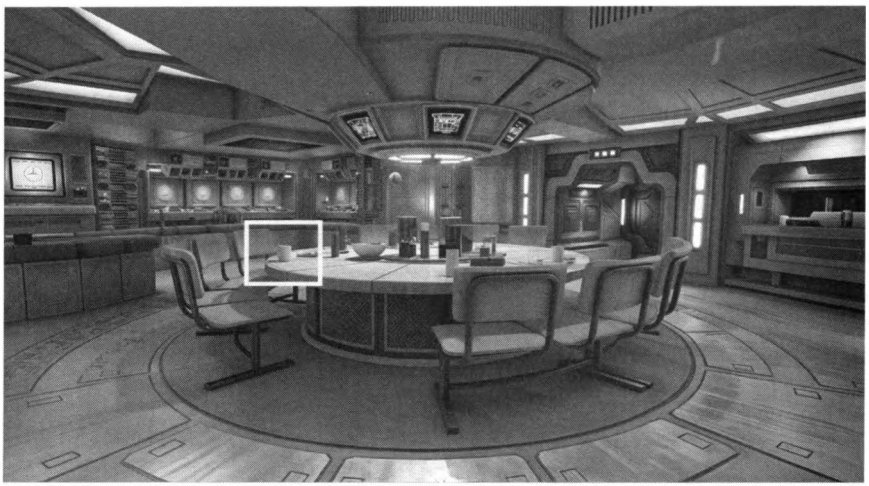


Рис. 7.21

Многие анимации тяжело отобразить в статичных скриншотах, поэтому рекомендуется запускать все примеры в коде, чтобы увидеть, как это работает.

```
transform focus_1: # Фокусировка на объекте
    zoom 0.5 xalign 0.5 yalign 0.5
    linear 5.0 zoom 1.0 xalign 0.2

label part_25:
    scene dining_2 at focus_1
```

Для подобного эффекта фокусировки необходимо брать изображения с большим разрешением, чем стандартные фоны, чтобы при увеличении не ухудшалось их качество.

Рис. 7.22

Аналогичным образом можно сразу показать изображение сфокусированным на определённой детали. Затем отдалить изображение, и приблизить уже к другому элементу.

```
transform focus_2: # Переход от одного объекта к другому
    zoom 1.0 xalign 0.2 yalign 0.5 # Сфокусированы на первом объекте
    linear 5.0 zoom 0.5 xalign 0.5 # Откатили камеру
    # Приблизили к другому элементу
    linear 5.0 zoom 1.0 xalign 0.73 yalign 0.46

label part_25:
    scene dining_2 at focus_2
```

С помощью всё того же перемещения по экрану можно создать эффект падения снежинок, капель дождя или листьев с деревьев. Реализуется это так же, как анимация титров, только в противоположную сторону, сверху вниз.

Развивая тему перемещения объектов на заднем фоне, можно создать многослойный экран, где в иллюминаторе будут плавно смещаться звёзды, создавая эффект полёта. А на переднем плане персонажи будут продолжать вести свои диалоги.

Рис. 7.23

Для этого нужно подготовить два изображения: звёздного неба и каюты с прозрачным иллюминатором.

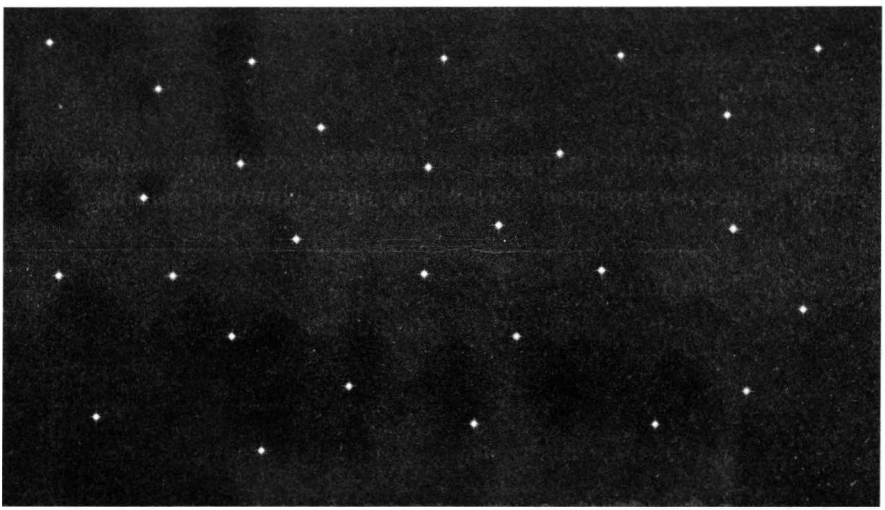


Рис. 7.24

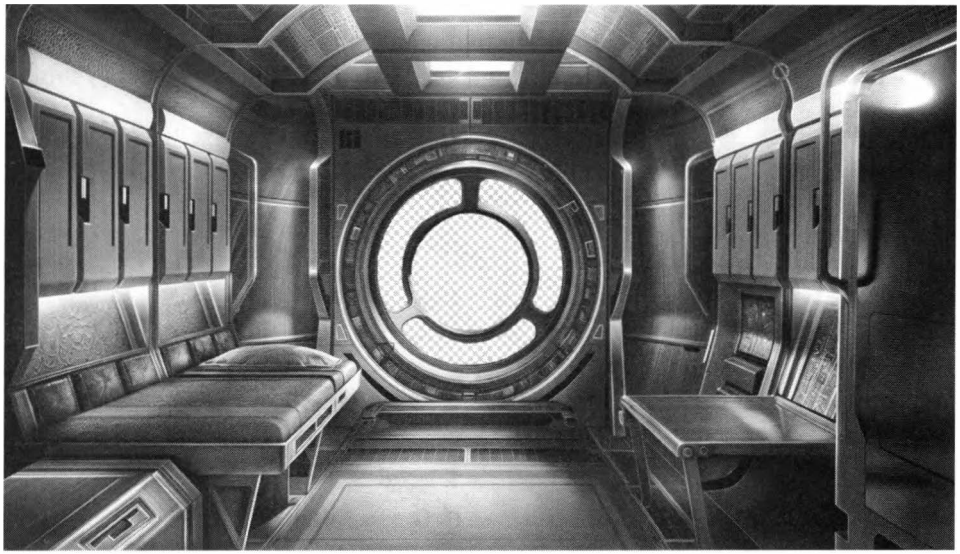


Рис. 7.25

А также простенькую анимацию, которая будет двигать фон справа налево или наоборот.

```
transform starry_sky: # Анимация для бесшовного фона
    xpan 0
    linear 300 xpan 1920
```

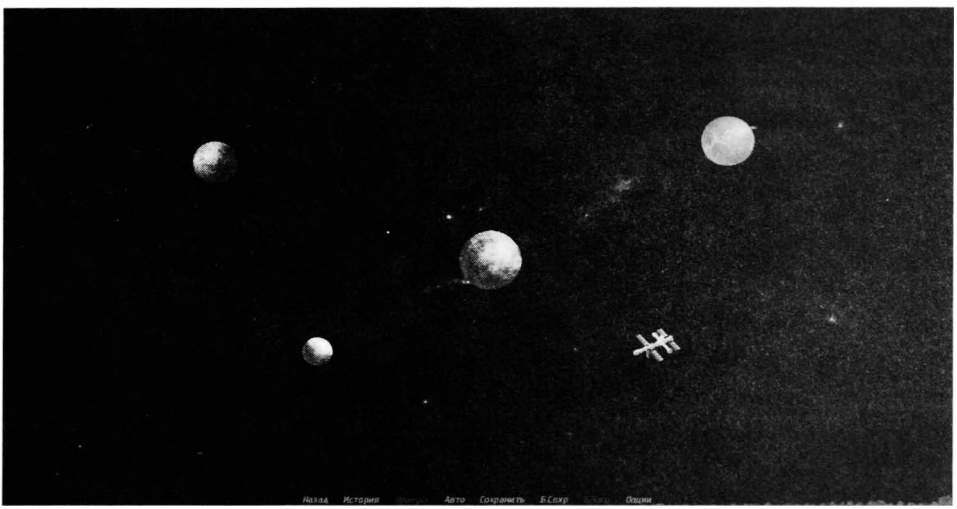
И теперь остаётся только вывести всё это на экран в правильном порядке:

```
label part_25:
    scene space at starry_sky # Зацикленный фон
    show cabin2 # Спрайт на весь экран с прозрачным иллюминатором
    show liara serious at lcenter with dissolve
    show kayden normal at rcenter with dissolve
    lia "Земля в иллюминаторе, Земля в иллюминаторе…"
    kay "Земля в иллюминаторе видна…"
```

Подобным образом можно создать эффект движения на автомобиле или автобусе, где на заднем фоне перемещаются деревья, дома и другие объекты. Поверх отображается спрайт на весь экран, представляющий собой салон автомобиля с прозрачными окнами. Затем вызываются спрайты персонажей и их реплики.

Глава 8.

СПИСКИ



8.1. Создание списка

В этом разделе мы познакомимся со списками и выясним, какие действия можно производить с ними.

Списки позволяют сохранять в одном месте большое количество разных данных, которые могут быть как связаны между собой, так и являться совершенно случайными объектами.

Ранее мы уже сталкивались со списками. Например, когда прописывали несколько действий для кнопок в **action**.

```
imagebutton at aster_fly: # Кнопка в виде летящего астероида
    idle "aster1"
    action [SetVariable("aster1", False),
            SetVariable("default_mouse", "cursor3"),
            SetVariable("quantity", quantity+1)]
```

Любой **список** создается внутри *квадратных скобок*.

В примере мы добавили на событие клика сразу три действия. Все они меняют значения переменных. Данную конструкцию можно записать в одну строку через запятую или разбить на несколько строк, чтобы код не уходил далеко за пределы экрана и был легкочитаемым.

Чтобы создать **список**, создается переменная с любым названием, которая будет хранить несколько объектов.

```
default crew = ["Лиара", "Кайден", "Ксеона", "Капитан"]
# Или
init python:
    crew = ["Лиара", "Кайден", "Ксеона", "Капитан"]
# Или
label start:
    $ crew = ["Лиара", "Кайден", "Ксеона", "Капитан"]
```

Списки, так же как и переменные, можно создать тремя способами. А затем отобразить на экране в диалоговом окне или с помощью экрана **screen**.

```
screen example_list(): # Отображаем на экране
    text _("Экипаж корабля [crew]") align(0.5,0.7)

label start: # Отображаем в метке
    scene deck
    show screen example_list # Показали на экране
         "Экипаж корабля [crew]" # Показали в диалоговом окне
```

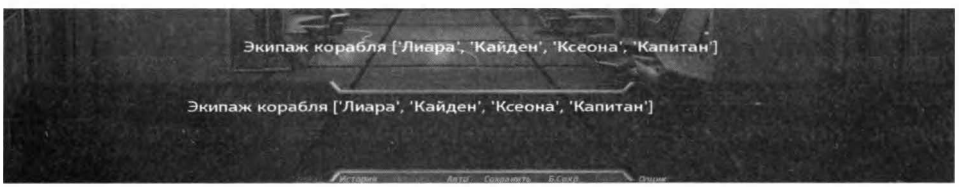


Рис. 8.1.

Помимо имен, **список** может включать в себя разные объекты, такие как: целые числа, дробные числа, строки, булевы значения, другие списки и пр.

```
init python:
    example_list = [23, # Целое число
                    4.9, # Дробное число
                    "Текст", # Какой-то текст
                    "Длинный текст в несколько предложений",
                    True, # Булево значение
                    [crew] # Список внутри списка
                    ] # Закрыли список

label start: # Отображаем список
    scene deck
"Пример списка [example_list]"
```

Рис. 8.2.

То есть большой объем данных, который хранится в списке, можно отобразить в любом месте игры с помощью короткой команды **[example_list]** — квадратных скобок с названием списка внутри.

Также в списке можно хранить множество переменных. Но предварительно необходимо эти переменные создать, иначе Ren'Py сообщит, что не знает о них.

```
init python:
    liara_relat = 0 # Создали переменные
    xeona_relat = 10
    kayden_relat = 7
    capitan_relat = 29
    # Внесли переменные в список
    attitude_char = [liara_relat, xeona_relat,
                     kayden_relat, capitan_relat]

label start: # Отображаем в метке
    scene deck
    "[attitude_char]"
```

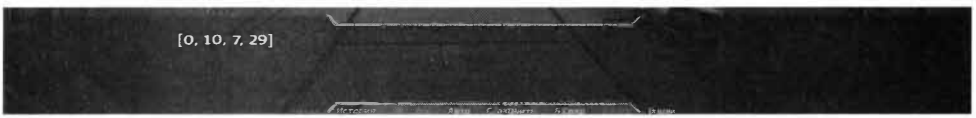


Рис. 8.3.

8.2. Работа со списками

Аналогично переменным, списки можно изменять, добавлять и удалять из них элементы, а также использовать определенную часть списка.

Первая операция над списком — это возможность посчитать количество объектов внутри него. Делается это с помощью функции **len()**.

```
label start:
    scene deck
    if len(attitude_char)>5:
        "[attitude_char]"
    scene bridge
```

> В строке *if len(attitude_char)>5* мы считаем функцией **len** количество объектов в списке **(attitude_char)**, и если объектов больше пяти, список должен отобразиться на экране, однако этого не происходит.

Подобная проверка на количество объектов в списке может быть полезна, например, для отображения статистики только в том случае, если персонаж знаком со всеми членами экипажа. Или отображать инвентарь нужно только в том случае, если в нем находится больше одного предмета (if len(inventory)>0).

Следующее действие над списком — возможность *объединения нескольких в один общий список*. Это называется **сцеплением списков**.

```
label start:
    scene deck
    $ new_list = crew + attitude_char # Объединяем два списка в один
    "[new_list]" # Показали новый список
```

Рис. 8.4.

Вместе с тем каждый список можно пополнять новыми элементами. Делается это аналогично тому, как мы прибавляли значения к переменным. Например, давайте добавим игрока в список экипажа корабля.

```
label start:
    scene deck
    $ crew = crew + [name_mc]
    "[crew]"
```

Также пополнять список можно с помощью соответствующих методов.

Методы — это специальные функции на языке Python.

В эту тему мы не будем углубляться, просто нужно запомнить эти команды и использовать их по необходимости.

Метод **append** используется для добавления элемента в конец списка.

```
label start:
    scene deck
    $ crew.append("Старпом")
    "[crew]"
```

В примере мы с помощью языка Python **($)** к списку **crew** добавили методом **append** новый объект **("")**.

Также нужно иметь в виду, что у каждого элемента в списке есть свой **порядковый номер** — индекс (**index**), благодаря чему можно манипулировать одним объектом в большом списке. Для этого нужно указать его индекс и сообщить необходимое действие.

Обратите внимание, что индекс в Python отсчитывается с **0**, а не с **1**.

То есть в нашем списке **crew** имя "Лиара" имеет индекс 0, "Кайден" имеет индекс 1 и т.д.

Рис. 8.5.

Теперь с помощью индекса и метода **insert** мы можем добавить нового члена экипажа в нужное место в списке.

```
label start:
    scene deck
    $ crew.insert(2,"Старпом") # Указали индекс, куда добавить
    "[crew]"
```

Рис. 8.6.

Вместе с тем отсчет индекса можно вести с обратной стороны списка. Для этого необходимо поставить **знак минуса** перед индексом.

```
label start:
    scene deck
    $ crew.insert(-2,"Старпом") # Указали индекс, куда добавить
    "[crew]"
```

Также, указав индекс, можно отобразить только один элемент списка:

```
label start:
    scene deck
    "[crew[-1]]" # Показали элемент списка с индексом -1
```

Рис. 8.7.

Для изменения определенного объекта в списке нужно указать название списка и индекс элемента, а затем присвоить новое значение:

```
label start:
    scene deck
    $ crew[1] = name_mc # Заменили Кайдена на имя игрока
    "[crew]"
```

Рис. 8.8.

Следующий метод **remove** нам понадобится для удаления определенного объекта из списка. Происходит это аналогично добавлению, с помощью метода **append**.

```
label start:
    scene deck
    $ crew.remove("Старпом") # Удаляем старшего помощника
    "[crew]"
```

Также удалить объект из списка можно по его индексу, с помощью оператора **del**.

```
label start:
    scene deck
    $ del crew[3]
    "[crew]"
```

Недостаток этого способа в том, что не всегда может быть известен индекс определенного объекта, так как на протяжении игры список может много раз измениться. Однако такой вариант пригодится, если нужно удалить любой элемент, находящийся под этим индексом. Например, предмет в инвентаре, который лежит в определенной ячейке.

Метод **reverse** позволяет отобразить список в обратном порядке.

```
label start:
    scene deck
    $ crew.reverse()
    "[crew]"
```

Рис. 8.9.

Метод **sort** позволяет сортировать элементы списка по алфавиту, если список состоит из строк, и по возрастанию чисел, если список состоит из них.

При указании аргумента *reverse=True* сортировка происходит в обратном порядке.

```
label start:
    scene deck
    $ crew.sort() # Сортируем экипаж
    $ attitude_char.sort() # Сортируем по значению отношений
    "[crew] [attitude_char]" # Показали результаты
```

Рис. 8.10.

То же самое с *reverse=True*:

```
label start:
    scene deck
    $ crew.sort(reverse=True)
    $ attitude_char.sort(reverse=True)
    "[crew] [attitude_char]"
```

> Для перемешивания (случайной перестановки) элементов списка можно использовать функцию **shuffle()** из модуля **random**.

Для этого необходимо импортировать модуль **random**, а затем применить функцию **shuffle()** к списку, который нужно перемешать.

```
init python:
    import random # Импортировали модуль random

label start:
    scene deck
    $ random.shuffle(crew) # Перемешали список
    "[crew]" # Показали результат
```

Рис. 8.11.

Для того чтобы использовать часть списка, применяются *срезы*. Благодаря ним можно взять кусок списка от начала до середины, или с середины до конца, или только середину.

При выводе среза необходимо указать индексы объектов, с какого по какой элемент нам нужно достать:

```
label start:
    scene deck
    $ srez = crew[0:4] # Добавили срез в отдельную переменную
    "[srez]" # Показали срез
```

Рис. 8.12.

Обратите внимание, что элементы берутся с 0 до 4, но без четвертого. Срез всегда берется до указанного числа, а не включительно.

Чтобы взять срез конца списка, нужно также указать начало среза и конец:

```
label start:
    scene deck
    $ srez = crew[3:99] # Добавили срез в отдельную переменную
    "[srez]" # Показали срез
```

Несмотря на то, что наш список гораздо короче указанного конечного числа, срез отображает все элементы, которые доступны в этом диапазоне. Такое часто бывает, когда невозможно предугадать, какой длины список будет у пользователя в определенный момент игры. Поэтому можно просто указать начало среза или его конец.

```
label start:
    scene deck
    $ srez = crew[:5] # Взяли срез с начала до пятого индекса
    $ srez2 = crew[4:] # Взяли срез с 4-го до конца списка
    $ srez3 = crew[:] # Взяли срез всего списка
    "[srez]\n[srez2]\n[srez3]" # Показали срезы
    # \n - Следующая часть кода с новой строки
```

Рис. 8.13.

Вместе с тем в срезы можно добавлять элементы с определенным шагом. Например, через одного, через два и т.д. Для этого необходимо указать третий параметр после двоеточия.

```
label start:
    scene deck
    $ srez = crew[1:5:2] # Взяли каждый второй объект в диапазоне 1:5
    "[crew]\n\n[srez]" # Показали срез
```

Рис. 8.14.

Для примера давайте возьмем каждый третий элемент из всего списка, независимо от того, какой он длины.

```
label start:

  scene deck
    $ srez = crew[::3] # Взяли каждый третий элемент
    "[crew]\n\n[srez]" # Показали срез
```

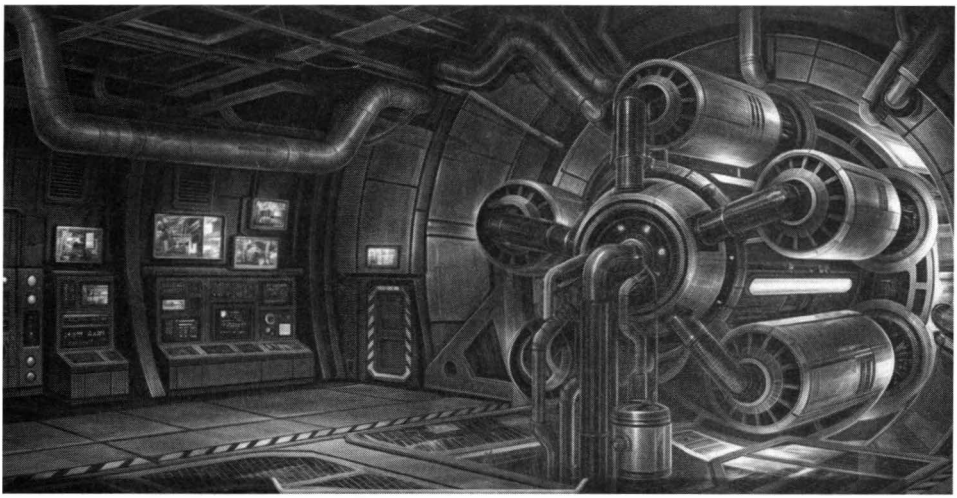
Рис. 8.15.

Также не забываем, что при установке отрицательных значений срез будет отсчитываться с конца списка:

```
label start:
    scene deck
    $ srez = crew[2:-2]
    "[srez]"
```

Глава 9.

ЦИКЛЫ



9.1. Цикл For

Циклы используются в программировании для повторения большого количества однообразных действий. Благодаря этому код программы и работа программиста значительно сокращаются.

> Цикл **for**, в частности, используется для однотипных действий с большим количеством объектов или списков. Он проходит по каждому элементу и производит одно и то же действие над ними, после чего завершается.

Например, мы можем вывести с помощью цикла **for** список всего экипажа, без необходимости выводить каждое имя, прописывая его по отдельности.

```
screen stat_crew(): # Экран с экипажем
    fixed:
        add "gui/tablet.png" align(.5,.5) # Фон в виде планшета
        vbox: # Вертикальный бокс
            align(.5,.5) # Расположение по центру
            for i in crew: # Цикл для отображения списка
                text "[i]"

label start:
    scene deck
    show screen stat_crew # Показали экран со списком
    pause
```

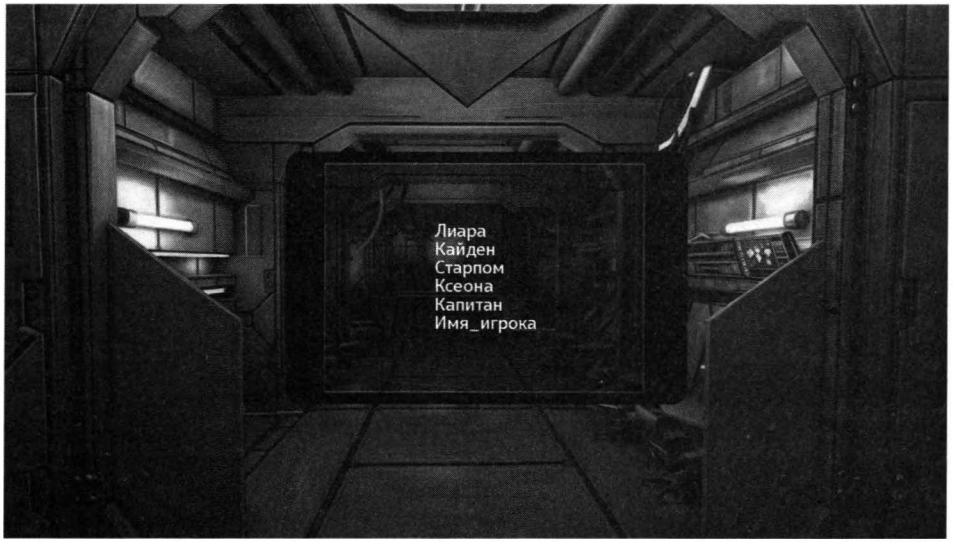


Рис. 9.1.

Как видим, вместо шести строчек типа text "Лиара", text "Кайден" и т.д., благодаря циклу, нам понадобилось только две строки. Сокращение в написании кода более заметно, когда необходимо вывести список с десятком или сотней элементов.

В данном примере переменная **i** поочередно принимает значения из списка **crew**, а в следующей строке команда **text** отображает текущее значение переменной. Цикл **for** сообщает движку, что подобное действие нужно сделать для каждого объекта.

То есть цикл начинается с первого элемента списка — "Лиара", сохраняет этот объект в переменную, которая ниже выводит текст. Затем цикл повторяется сначала и сохраняет в переменную следующий элемент списка — "Кайден", и так продолжается до тех пор, пока не будет пройден весь список.

Аналогичным образом мы можем выводить часть списка с помощью срезов:

```
screen stat_crew():
    fixed:
        add "gui/tablet.png" align(.5,.5)
        vbox:
            align(.5,.5)
            for i in crew[2:5]: # Срез со второго по четвертый
                text "[i]"
```

Рис. 9.2.

А также изображения или другие объекты:

```
screen stat_crew():
    fixed:
        add "gui/tablet.png" align(.5,.5)
        hbox: # Горизонтальный бокс с двумя вертикальными внутри
            align(.5,.5)
            spacing 50
            vbox:
                spacing 10
                # Создали список с иконками персонажей
                $ crew_icon = ["ic_liara","ic_kayden","ic_xeona"]
                for i in crew_icon[:3]: # Вывели иконки персонажей
                    add "[i]" # В переменной названия иконок
            vbox:
                spacing 35
                for i in crew[:3]: # Срез первых трех персонажей
                    text "[i]" # В переменной список персонажей
```

Рис. 9.3.

Вместе с циклом **for** часто используется функция **range**. Она отвечает за ранжирование объектов. С ее помощью можно генерировать список, и выводить его на экран.

```
for i in range(5): # Генерируем список чисел 0-4
    text "[i]" # Выводим список
```

Это можно применить, например, для нумерации:

```
screen stat_crew():
    fixed:
        add "gui/tablet.png" align(.5,.5)
        hbox: # Горизонтальный бокс
            align(.5,.5)
            spacing 50
            vbox:
                spacing 35
                for i in range(1,4): #Ранжируем числа в диапазоне 1-3
                    text "[i]" # В переменную подставляются числа
            vbox:
                spacing 35
                for i in crew[:3]: # Срез первых трех персонажей
                    text "[i]" # В переменной список персонажей
```

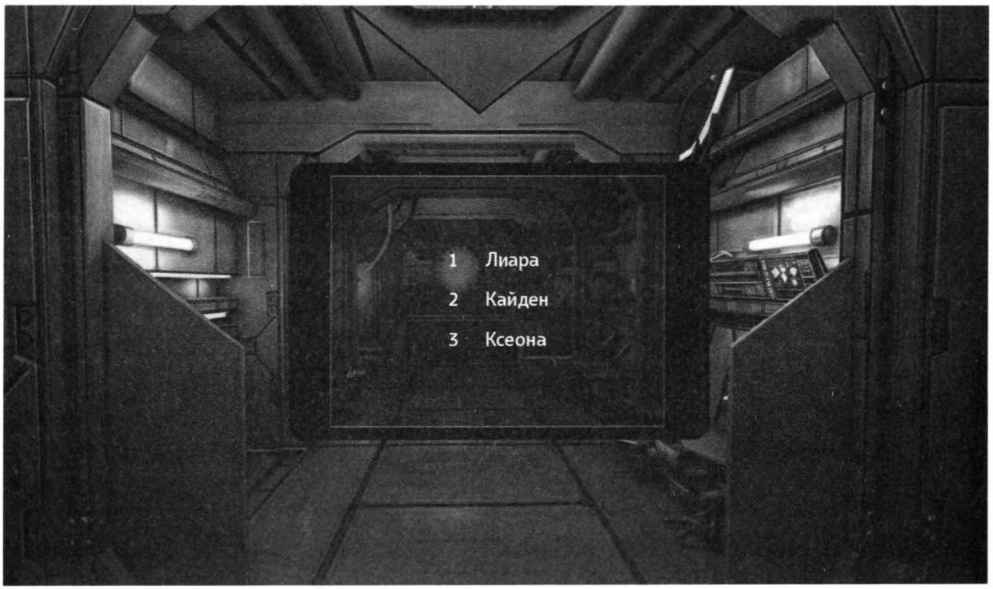


Рис. 9.4.

По аналогии с этим, вместо текста можно выводить много раз изображение звезды со случайной координатой, чтобы создать звездное небо в несколько строк. А также снегопад, дождь или что-то еще.

Не забудьте также добавить трансформацию, чтобы привести объекты в движение или заставить мерцать.

```
screen sky(): # Экран со случайным распределением звезд
    for i in range(99): # 99 звезд на случайных координатах
        add "star" xpos renpy.random.randint (0, 1920) ypos renpy.
random.randint (0, 1080)
        # add "star" at transform для анимации объектов

label start:
    scene black
    show screen sky # Показали экран
    pause
```

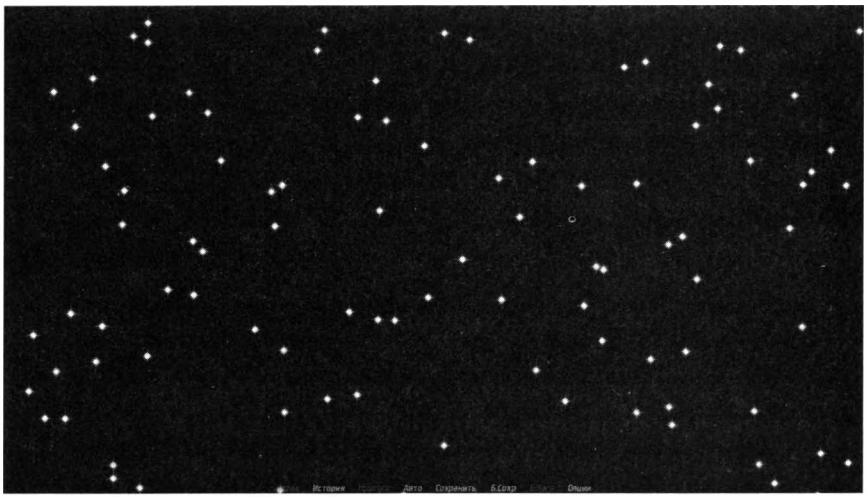


Рис. 9.5.

Помимо этого, функция **range** может создавать последовательности с определенным шагом. Для этого шаг указывается третьим числом в скобках.

```
screen stat_crew():
    fixed:
        add "gui/tablet.png" align(.5,.5)
        vbox:
            align(.5,.5)
            spacing 10
            for i in range(1,12,3): # Список чисел 1-12 с шагом 3
                text "[i]" # Выводим список
```

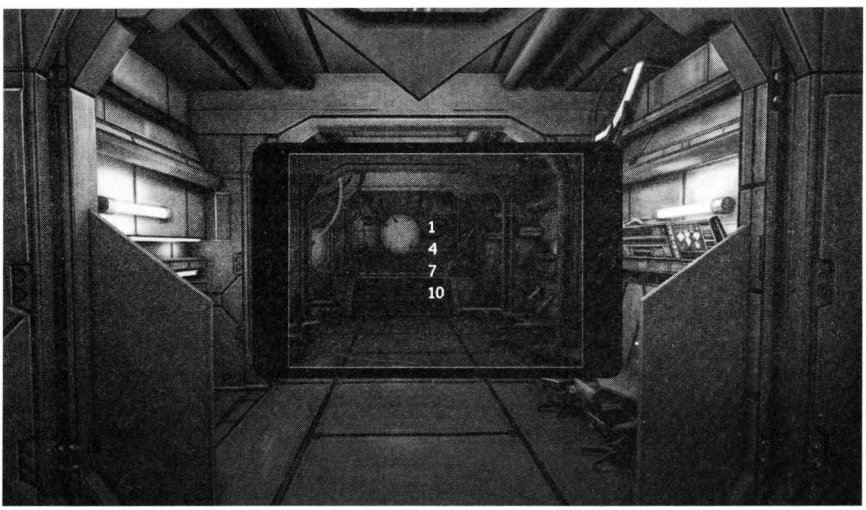


Рис. 9.6.

Таким образом, можно присвоить большому количеству переменных определенные значения.

```
label start:
    scene deck
    # Присваиваем переменным значения в диапазоне 58-99 с шагом 15
    $ li, ka, xe = range(58,99,15) # Присвоили
    "[li]\n[ka]\n[xe]" # Показали результат
```

Рис. 9.7.

9.2. Цикл *While*

В отличие от цикла **for**, цикл **while** работает до тех пор, пока выполняется определенное условие.

Напомню, что цикл **for** проходит по всем элементам один раз, после чего завершается. Цикл **while** может проходить все элементы снова и снова, до бесконечности.

Однако стоит иметь в виду, что в движке Ren'Py установлено ограничение, не позволяющее запускать бесконечные циклы. Это своего рода защита от новичка. Поэтому любой цикл **while** в Ren'Py обязательно должен содержать условие, при котором он должен завершиться.

Ранее мы косвенно встречались с примерами этого цикла. Например, когда создавали покадровую анимацию моргания персонажа. Во время анимации движок последовательно отображал несколько изображений и начинал анимацию сначала.

```
image liara_blink: # Изображение моргающей Лиары
    "liara_blink1" with dissolve # Закрывает глаза
    pause 0.1
    "liara_blink2" with dissolve
    pause 0.1
    "liara_blink3" with dissolve
    pause 0.1
    "liara_blink4" with dissolve # Закрыла
    pause 0.1
    "liara_blink3" with dissolve # Открывает глаза
    pause 0.1
    "liara_blink2" with dissolve
    pause 0.1
    "liara_blink1" with dissolve
    pause 0.1
    repeat # Повторяет анимацию
```

Казалось бы, это бесконечный цикл, который будет отображать изображения бесконечно. Но условие **while** заключается в том, что анимация будет проигрываться до тех пор, пока виртуальное изображение *image liara_blink* показано в ***метке*** (label).

```
label start:
    scene deck
    show liara liara_blink # Анимация моргания
    lia "Я много моргаю, мне что-то попало в глаз"
    show liara normal # Смена спрайта Лиары
    lia "Вот, кажется, прошло"
```

То есть при вызове следующего обычного спрайта или другого анимированного спрайта Лиары предыдущий цикл завершается, так как больше не выполняется условие анимации "повторять, пока отображается на экране". Это скрытое условие, вшитое в движок и облегчающее нашу работу.

Но при создании собственных циклов необходимо указывать условие, до каких пор он должен продолжаться. Иначе Ren'Py сообщит о создании бесконечного цикла и не позволит запустить проект.

```
label part_35:
    scene engine with dissolve
    $ stamina_kayden = 10 # Уровень выносливости Кайдена
    while stamina_kayden >0: # Цикл выполняется, пока выносливость больше 0
        kay "Нужно починить двигатель"
        $ stamina_kayden -= 2 # Кайден теряет два очка выносливости
        "Выносливости осталось [stamina_kayden]"
        # После клика игрока цикл начинается сначала

    # Когда выносливость снизится до 0, условие для while
    # перестанет выполняться, и сценарий пойдет дальше
    kay "Фух, устал. Пойду посплю."
```

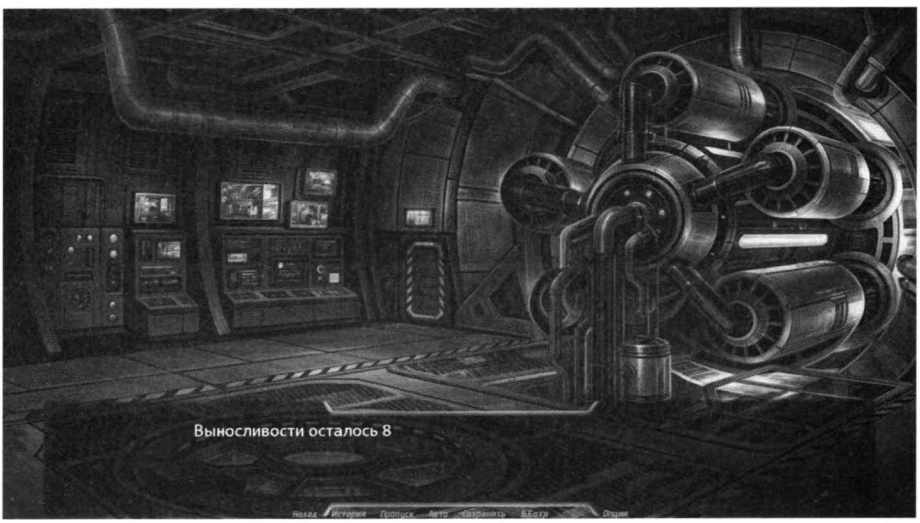


Рис. 9.8.

Также с помощью цикла **while** мы можем работать со списками. Например, сортировать в них элементы. Предположим, Кайден собирает по кораблю различные инструменты и детали, чтобы отремонтировать двигатель. Каждый новый предмет добавляется в список инвентаря методом *$ inventory.append("Деталь_1")*.

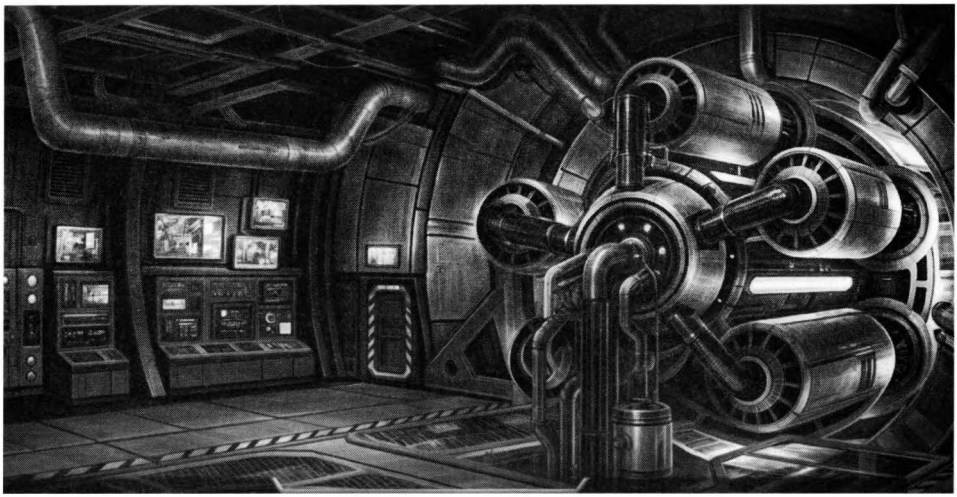
В какой-то момент у Кайдена может скопиться определенных деталей больше, чем предусмотрено нами по сценарию. В таком случае с помощью цикла мы можем удалить все предметы определенного типа.

```
label part_36:
    scene engine with dissolve
    # Список с предметами
    $ inventory = ["Деталь_1", "Деталь_3", "Ненужная_деталь",
                   "Деталь_9", "Ненужная_деталь", "Деталь_14",
                   "Ненужная_деталь"]
    "[inventory]" # Посмотрели содержимое инвентаря
    # Цикл для удаления ненужных деталей
    while "Ненужная_деталь" in inventory:
    # Цикл выполняется, пока ненужные детали есть в инвентаре
        $ inventory.remove("Ненужная_деталь") # Удаляем элемент
    kay "Ого, сколько хлама!"
    "[inventory]" # Посмотрели оставшееся содержимое инвентаря
```

Рис. 9.9.

Глава 10.

ФУНКЦИИ



10.1. Что такое функции

Функции — это еще один из базовых инструментов как движка Ren'Py, так и многих языков программирования.

> Благодаря **функциям** мы можем сокращать код, создавая блоки с инструкциями, к которым потом можно обращаться из любого места в сценарии. Кроме того, они позволяют реализовать различные механики, не предусмотренные движком, работая непосредственно с языком программирования Python.

Система работы функций похожа на то, как работают **метки** (label), **экраны** (screen), **трансформации** (transform) и **анимации** (image). То есть мы указываем ключевое слово **def**, сообщая движку, что это функция, и придумываем ей подходящее название, по которому эта функция будет вызываться. Однако, в отличие от вышеописанных систем, функции создаются непосредственно на языке Python. То есть они должны быть помещены в блок **init python**.

Давайте представим такую ситуацию, когда игрок может выполнить одно и то же действие в разных местах игры. Во всех этих местах будет происходить проверка, хватает ли навыка на выполнение определенного действия. И в зависимости от результата проверки будут происходить некоторые события. Стоит иметь в виду, что во всех таких эпизодах нам потребуется прописывать один и тот же код проверки. В таком случае, создание функции позволит нам сократить время на написание скрипта проверки и уменьшить общее количество кода.

```
init python: # Блок кода на языке Python
    def simple_func(): # Название функции
        if kayden_engin >9: # Проверяем навык инженера
            renpy.notify("Двигатель отремонтирован") # Всплывающее сообщение
            renpy.say(kay,"Так-то лучше") # Реплика персонажа
            renpy.jump("part_38") # Переход в 38-й эпизод
        elif kayden_engin >4:
            renpy.notify("Двигатель частично отремонтирован")
            renpy.say(kay,"Хм, остались странные вибрации")
            renpy.jump("part_39")
        else:
            renpy.notify("Не хватает навыков")
            renpy.say(kay,"Понятия не имею, что нужно делать")
            renpy.jump("part_40")

label part_37:
    scene engine with dissolve
    $ simple_func() # Вызвали функцию проверки навыка
```

Рис. 10.1.

Польза от функции заключается еще и в том, что если в определенный момент разработки мы захотим изменить условия проверки, то нам не придется проходить все места в коде, где происходит эта проверка, и редактировать их. Достаточно будет изменить саму функцию. Например, мы можем захотеть дополнить условие, при котором Кайдену, помимо навыка, необходимо иметь соответствующий инструмент.

```
init python:
    def simple_func():
        if kayden_engin >9 and tool_1 ==True:
```

Как видим, мы просто изменили функцию. Если нам потребуется, чтобы у Кайдена также были подходящие детали, мы снова пойдем редактировать функцию, и она будет работать во всех эпизодах, в которых вызывается.

```
init python:
    def simple_func():
        if kayden_engin >9 and tool_1 ==True and detail_9 >0:
```

Помимо **меток** (label), мы также можем вызывать функцию в **экранах** (screen). Однако система экранов работает немного по другому принципу, и некоторые функции могут противоречить функциям экрана. Нужно иметь это в виду.

```
screen demo_func():
    on "show" action Function(simple_func)

label part_37:
    scene engine with dissolve
    show screen demo_func
```

При открытии экрана (on "show") произойдет вызов функции. Таким же образом можно запустить функцию при закрытии экрана (on "hide"). Кроме того, мы можем привязать действие к нажатию кнопки клавиатуры или к экранной кнопке.

```
screen demo_func():
    textbutton _("Кнопка вызова функции"):
        action Function(simple_func)
```

Внутри функции мы можем изменять переменные. Например, после каждой попытки ремонта двигателя персонаж может увеличивать свой навык на единицу. И здесь нам нужно познакомиться с таким понятием, как "область видимости".

Все переменные делятся на глобальные и локальные. Первые движок видит и использует в любой части кода, а вторые применяются только внутри функций. В других частях кода значения локальных переменных движок не использует.

Глобальные переменные — это те, которые мы создавали ранее. Обращаясь к ним в любом месте игры, движок находит их и читает значения, которые они хранят.

Локальные переменные используются только внутри функций, и они никак не влияют на глобальные переменные.

То есть, изменяя, например, навык *kayden_engin*, мы меняем его только здесь и используем для работы внутри функции. При этом в самой игре навык не повышается.

```
init python:
    def simple_func():
        kayden_engin +=1 # Локальная переменная
```

Это позволяет создавать много разных функций с одинаковыми названиями переменных, где каждая может иметь абсолютно разные значения. Локальные переменные используются только для расчетов внутри функции.

```
init python:
    # Три разные функции с одинаковым названием переменной
    def simple_func():
        kayden_engin = 1 # Локальная переменная
    def skill_training():
        kayden_engin = True # Локальная переменная
    def skill():
        kayden_engin = "Не определен" # Локальная переменная
```

В зависимости от того, где была создана переменная, она становится либо локальной, либо глобальной.

Локальные переменные **можно сделать** *глобальными*, **указав соответствующее ключевое слово внутри функции — global, после которого через запятую указываются имена переменных.**

Они будут использовать значения глобальных переменных. Это позволяет изменять их внутри функций.

Итак, помимо проверки навыка, Кайден будет увеличивать его на единицу при каждой попытке починить двигатель:

```
init python:
    def simple_func():
        global kayden_engin # Эта переменная является глобальной
        kayden_engin += 1 # Изменили глобальную переменную
```

Или так:

```
init python:
    def example():
        # Пример добавления большого количества глобальных переменных
        global kayden_engin, liara_pilot, xeona_medic, blah_blah
        global capitan_relat, kayden_relat, liara_relat
```

Следующий момент, который нужно знать, — это **параметры** и **аргументы функции**.

> Каждая **функция** может получать различные данные, обрабатывать их и возвращать результат.

При создании функции в скобках мы можем указать, какие параметры она способна принимать:

```
init python:
    def example(kayden_engin): # Параметр функции
```

При вызове функции в нужном месте кода мы можем передать аргумент в этот параметр.

```
label part_37:
    scene engine with dissolve
    $ simple_func(15) # Передали аргумент
```

То есть теперь переменная внутри функции имеет значение 15. Функция может иметь любое количество таких параметров, их необходимо указывать через запятую. Также они могут иметь значения по умолчанию.

```
init python:
    # Значения по умолчанию после знака =
    def example(kayden=0, capitan=False, liara="Не
определено"):
```

Аргументы, передаваемые при вызове функции, бывают двух типов: **позиционные** и **именованные**.

Позиционные аргументы определяются их порядком в списке параметров (то есть на первом месте в скобках, на втором и т.д.).

Этот порядок важен для правильной передачи значений.

```
label part_37:
    scene engine with dissolve
    $ simple_func(15, True, "Определено")
```

Здесь мы передали число 15 в переменную **kayden**, затем булево значение в переменную **capitan**, и далее строка "Определено" отправилась в переменную **liara**.

Именованные аргументы, напротив, идентифицируются по своим именам, что позволяет передавать их в любом порядке.

По умолчанию все параметры являются позиционно-именованными.

```
label part_37:
    scene engine with dissolve
    $ simple_func(capitan=True, liara="Определено", kayden_engin=15)
```

Здесь значения переменных передаются в другом порядке, но так как мы непосредственно указываем имя каждой переменной, которая получает новое значение, движок понимает, что к чему относится.

> Параметры могут быть **обязательными** и **необязательными**. Если после знака = указано значение по умолчанию, такой параметр является необязательным.

```
init python:
    # capitan - необязательный параметр
    def example(kayden, capitan=False):
```

Это нужно иметь в виду, так как при передаче аргументов функции их количество должно совпадать с количеством параметров функции, иначе движок выдаст сообщение об ошибке. Но это не распространяется на необязательные параметры, так как они имеют значения по умолчанию и используются в случае, если не были указаны новые значения при вызове функции.

> Также в функциях могут встречаться параметры под одной или двумя звездочками: *args, **kwargs. Первые являются **переменно-позиционными**, а вторые **переменно-именованными**. Эти параметры означают, что их значения необязательны. Они также могут быть переданы или нет.

После передачи значений в функцию все они записываются в локальные переменные. Если какие-то из них имели значения по умолчанию, они меняются на новые. Далее результат обработки аргументов внутри функции может быть возвращен оператором **return** в нужное место кода.

```
init python:
    def example(kayden_engin=0): # Значение по умолчанию
        kayden_engin +=1 # Функция делает подсчет
        return kayden_engin # Возвращает значение

label part_37:
    scene engine with dissolve
    # Передали значение функции и сохранили результат в переменной
    $ kayden_engin = example(10)
"[kayden_engin]" # Посмотрели результат
```

Рис. 10.2.

Параметр *kayden_engin* в функции имеет по умолчанию значение 0. Если в метке вызвать функцию без передачи ей нового значения (10), то ответ будет 1, так как к значению по умолчанию прибавляется единица. Однако после передачи ей нового аргумента значение переменной в функции будет изменено, и результат будет другим.

Это может пригодиться, чтобы использовать одну и ту же функцию в разных случаях. Например, при создании боевой системы мы можем настраивать силу противника в зависимости от прокачки персонажа или развития сюжета. То есть в начале игры противник может наносить урон в размере 1, а далее по сюжету или при достижении персонажем определенного уровня противник будет наносить урон в размере 11.

10.2. Создание "функции" инструментами движка

По сути метки, экраны, трансформации и другие блоки в Ren'Py являются теми же функциями, которые принимают определенные аргументы и выполняют с ними определенные действия. В лейблах (метках) мы вызываем виртуальные изображения (image), трансформации (transform) и экраны (screen), но и сами лейблы могут быть блоками-функциями, вызываемыми в других метках.

Например, персонаж может ежедневно получать задание от старпома. И, чтобы не прописывать сценарий и проверки на каждый день недели, можно создать одну проверку в отдельном лейбле и вызывать ее по необходимости. Принцип работы аналогичен функции из предыдущего раздела.

```
init python:
    day_week = "понедельник" # Создали переменную дня недели

label part_42: # Любой эпизод игры
    scene bridge with fade
    show liara serious at lcenter
    show kayden normal at rcenter
    kay "Лиара, чем я сегодня должен заниматься?"
    lia "Так, сегодня [day_week]"
    call part_157 # Вызываем лейбл с проверкой
    lia "Потом найди меня и предоставь отчет" # Возвращаемся сюда
    # Здесь продолжение сценария
    jump part_43

label part_157: # Лейбл ежедневной проверки
    if day_week == "понедельник": # Если понедельник
        lia "Ты дежуришь в машинном отделении"
    elif day_week == "вторник": # Если вторник
        lia "Ты дежуришь в медцентре"
    elif day_week == "среда": # Если среда
        lia "Ты дежуришь на мостике"
    else: # И т.д.
        lia "Ты сегодня выходной"
    return # Возвращаемся туда, откуда вызывали метку
```

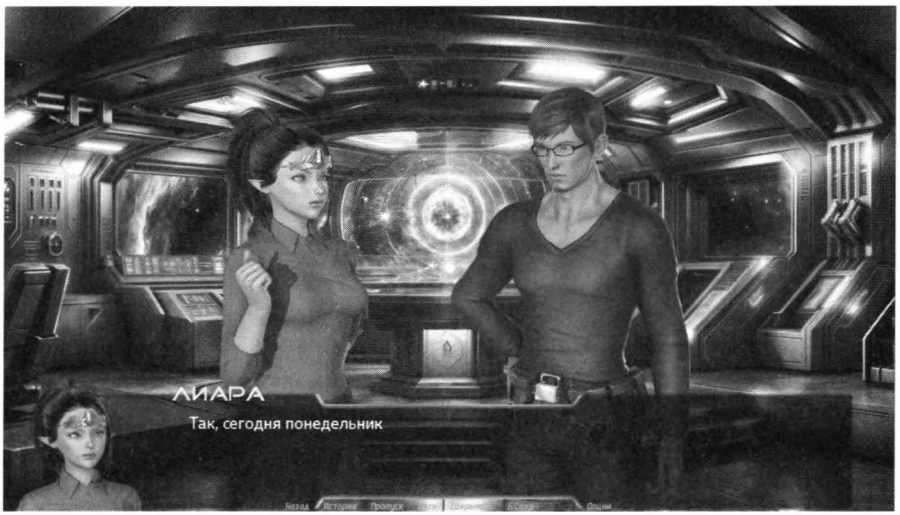


Рис. 10.3.

В примере на строке *call part_157* вызывается метка без перехода в нее (**call** вместо **jump**). В новом лейбле происходит проверка, и команда **return** возвращает нас в то место, откуда был вызван 157-й эпизод.

Таким образом, вместо семи проверок для каждого отдельного дня используется одна. Эта механика может быть применена для создания большой серии зацикленных блоков, попадания в которые будут предусмотрены определенными условиями.

Например, на этом принципе можно построить боевую систему с поочередными атаками противников, где скрипт будет отправлять игрока в несколько лейблов по кругу, пока не завершится бой.

Рис. 10.4.

Для начала потребуется ввести все необходимые переменные:

```
init python: # Переменные с показателями здоровья и силы
    kayden_hp = 80 # Уровень здоровья Кайдена
    kayden_hpmax = 80 # Максимальный уровень
    kayden_strength = 15 # Сила атаки Кайдена
    enemy_hp = 120 # Уровень здоровья Врага
    enemy_hpmax = 120 # Максимальный уровень
    enemy_strength = 20 # Сила атаки Врага
    queue = 1 # Очередь хода, 0 - игрок, 1 - противник
```

Далее давайте создадим экран, в котором будут отражены показатели противников:

```
screen fight_scr(): # Экран боевой системы
    add "kayden serious2" xpos 100 # Спрайт Кайдена на экране
    add "enemy" xpos 1100 ypos 100 # Спрайт противника
    fixed: # Контейнер с баром и текстом Кайдена
        xpos 150 ypos 30 # Позиция контейнера
        xysize (624,140) # Размер контейнера
        text _("Кайден") align(0.3,0.1) size 40
        # Цифровое отображение здоровья поверх полоски бара
        text _("[kayden_hp]/[kayden_hpmax]") align(0.7,0.1) size 40
        bar: # Бар с уровнем здоровья Кайдена
            align(0.5,0.5)
            xsize 620 ysize 60
            value AnimatedValue(value=kayden_hp, range=kayden_
hpmax, delay=1.0)
            left_bar Frame("gui/bar/bbar1.png",10,10)
            right_bar Frame("gui/bar/bbar2.png",10,10)
    fixed: # Контейнер с баром и текстом противника
        xpos 1150 ypos 30 # Позиция контейнера
        xysize (624,140) # Размер контейнера
        text _("Противник") align(0.3,0.1) size 40
        # Цифровое отображение здоровья поверх полоски бара
        text _("[enemy_hp]/[enemy_hpmax]") align(0.7,0.1) size 40
        bar:
            align(0.5,0.5)
            xsize 620 ysize 60
            value AnimatedValue(value=enemy_hp, range=enemy_hpmax,
delay=1.0)
            left_bar Frame("gui/bar/bbar1.png",10,10)
            right_bar Frame("gui/bar/bbar2.png",10,10)
```

Следующий шаг — создание метки, в которой будут установлены настройки боя:

```
label part_45:
    $ quick_menu = False # Скрыли быстрое меню
    $ config.rollback_enabled = False # Выключили откат назад
    $ kayden_hp = 80 # Установили здоровье участников
    $ enemy_hp = 120 #
    $ queue =1 # Установили очередь хода
    scene dining with dissolve
    show screen fight_scr with dissolve # Открыли экран боя
    jump fight_1 # Переходим в метку боя
```

Несмотря на то, что в блоке **init python** мы установили уровень здоровья противников, здесь мы снова устанавливаем эти значения. Это нужно для повторных сражений, так как после первого боя здоровье участников будет снижено, и в данном случае мы возвращаем его к дефолту.

Далее игрок отправляется в первый лейбл боя, который представляет собой небольшую функцию, проверяющую уровень здоровья участников и очередность хода:

```
label fight_1:
    # Если у кого-то здоровье опустилось ниже нуля
    if kayden_hp <= 0 or enemy_hp <= 0:
        hide screen fight_scr # Закрываем экран
        scene black with fade # Затемнение
        jump fight_final # Переход в финальную метку боя
    # Если у всех есть здоровье, проверяем очередь атаки
    if queue ==0: # Если ход равен 0
        jump fight_3 # Переходим к метке с действием игрока
    elif queue ==1: # Если ход равен 1
        jump fight_2 # Переходим к метке с действием противника
```

Теперь создаем еще две метки, которые также будут представлять собой небольшие функции, вычисляющие силу атаки участников.

```
label fight_2: # Метка с действием противника
    "Пират собирается атаковать!" # Реплика - предупреждение
    # Уменьшаем количество здоровья Кайдена
    # в зависимости от сгенерированной силы атаки
    $ kayden_hp -= enemy_strength + renpy.random.randint(-10, 10)
    # То есть из здоровья Кайдена будет вычтена сумма атаки врага
    # и сгенерированного числа, в диапазоне от -10 до 10.
    # Это позволяет сделать каждую атаку разной силы
    $ queue = 0 # Устанавливаем ход игрока
    jump fight_1 # Отправляем в первую метку,
    # где проверяем уровень здоровья и очередь хода
```

Еще один лейбл с действиями игрока:

```
label fight_3: # Метка с действием игрока
    menu: # Варианты действий
        "Атаковать": # Расчет атаки персонажа
```

```
            $ enemy_hp -= kayden_strength+renpy.random.randint(-5, 5)
        "Пополнить здоровье":
            $ kayden_hp += 30 # Пополняем здоровье
    $ queue = 1 # Устанавливаем ход противника
    jump fight_1 # Отправляем в первую метку,
    # где проверяем уровень здоровья и очередь хода
```

Рис. 10.5.

В данном примере три функции в виде лейблов будут отрабатывать по очереди, пока у кого-то из участников битвы не закончится здоровье. Теперь нам осталось создать только финальную метку, в которую будет перенаправлен пользователь из лейбла **fight_1**.

```
label fight_final: # Метка после боя
    scene dining with dissolve
    show kayden normal2 at lcenter with dissolve
    show enemy at rcenter with dissolve
    # Проверяем победителя
    if kayden_hp <= 0: # Если у Кайдена нет здоровья
        kay "Что ж, твоя взяла"
    else: # В противном случае здоровья нет у противника
        kay "Тебе не победить, жалкий пират!"
    $ quick_menu = True # Включили быстрое меню
    $ config.rollback_enabled = True # Включили откат назад
    jump part_46 # Перешли к следующему эпизоду сценария
```

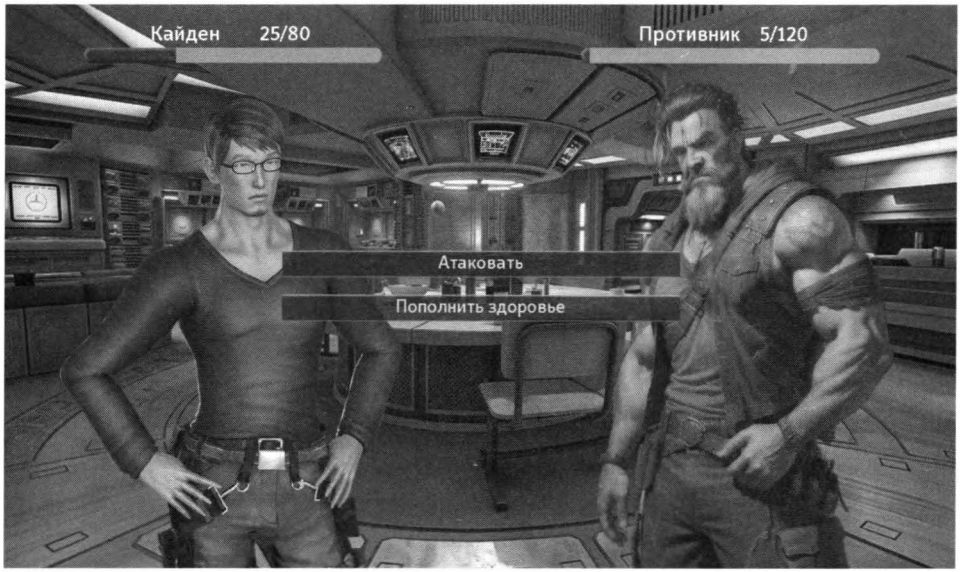


Рис. 10.6.

Пример боевой системы приведен в простейшем варианте, чтобы продемонстрировать взаимодействие нескольких лейблов в виде функций, которые повторяются циклично, друг за другом, пока у персонажей есть достаточно здоровья.

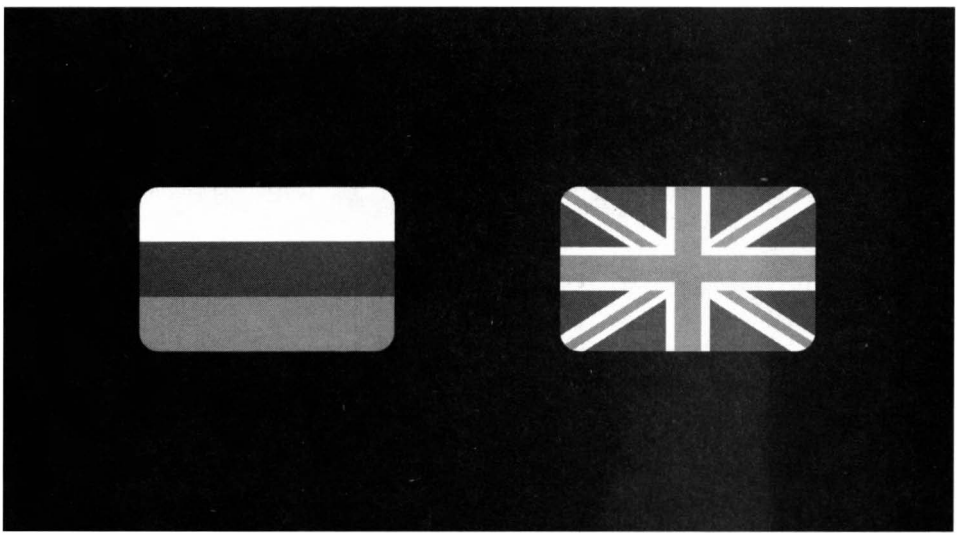
Таким же образом можно добавить несколько врагов, каждый из которых будет атаковать в свою очередь, если ему позволяет уровень его здоровья. Дополнительно можно реализовать различные параметры, такие как уровень брони, атаки холодным и огнестрельным оружием, спецудары, уклоны, увороты и пр. Но в таком случае формулы расчета атаки нужно будет усовершенствовать, добавив все необходимые переменные.

Вместо стандартных кнопок **menu** можно создать свои красивые кнопки *imagebutton* или *button* и открывать их в новом экране, когда наступает очередь игрока. В них можно иконками отобразить атаку, пополнение здоровья, какие-нибудь бафы, дебафы и пр.

С помощью эффектов анимации и трансформаций можно добавить разного рода вспышки, встряхивания экрана при получении урона и т.д. В общем, используйте весь арсенал механик, описанных в предыдущих главах, и свою фантазию.

Глава 11.

Локализация



После завершения работы над игрой есть вероятность, что вам захочется перевести ее на другие языки, чтобы расширить аудиторию пользователей и увеличить продажи, если проект коммерческий. Процесс генерации файлов локализации в движке автоматизирован. Для этого выберите в лаунчере проект для перевода и нажмите "Создать переводы".

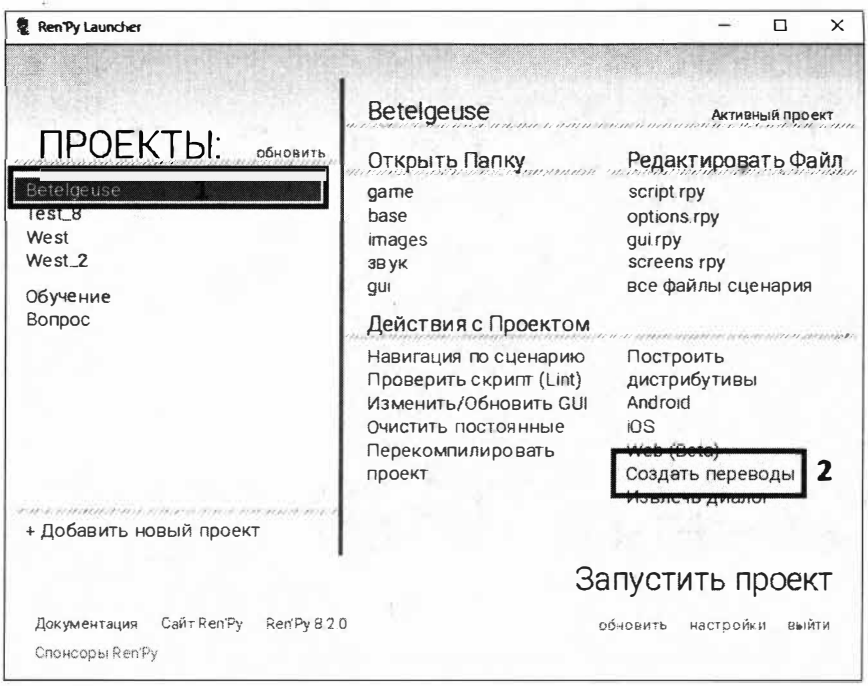


Рис. 11.1.

В следующем окне напишите название языка перевода, затем установите флажок "Генерировать пустые строки для перевода" и нажмите "Объединить строки перевода". После этого запустится автоматическая генерация файлов локализации.

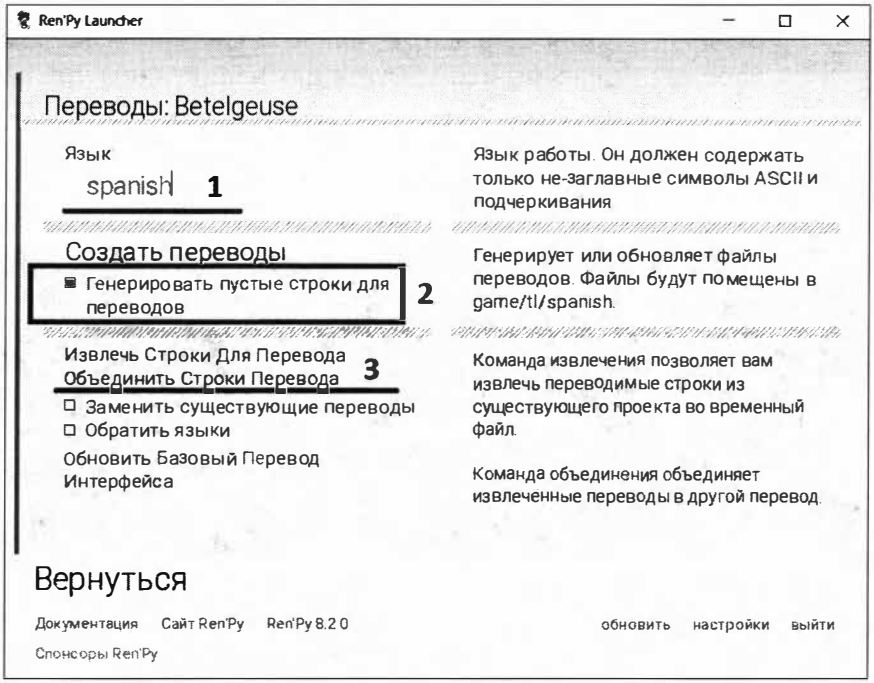


Рис. 11.2.

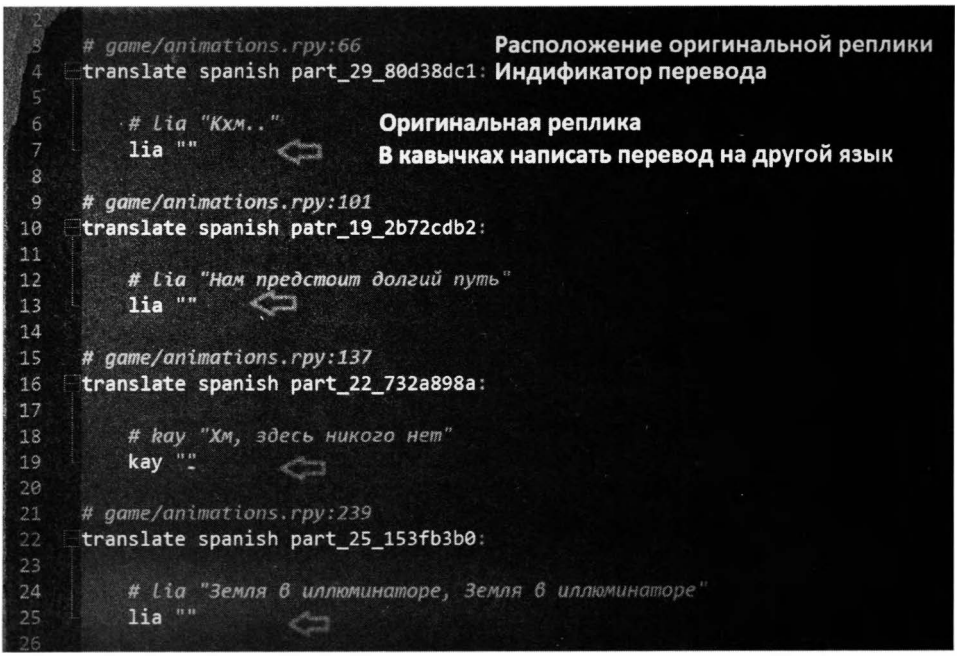
После того как этот процесс завершится, откройте папку с проектом по пути: **game/tl/...** Здесь будет создана директория с переводом, внутри которой находятся файлы, имеющие такие же названия, как и скрипты вашего проекта.

Имя	Дата изменения	Тип	Размер
animations.rpy	11.04.2024 11:17	Файл "RPY"	2 КБ
animations.rpyc	11.04.2024 11:17	Файл "RPYC"	3 КБ
common.rpy	11.04.2024 11:18	Файл "RPY"	46 КБ
common.rpyc	11.04.2024 11:17	Файл "RPYC"	26 КБ
figth.rpy	11.04.2024 11:17	Файл "RPY"	1 КБ
figth.rpyc	11.04.2024 11:17	Файл "RPYC"	3 КБ
options.rpy	11.04.2024 11:17	Файл "RPY"	1 КБ
options.rpyc	11.04.2024 11:17	Файл "RPYC"	1 КБ
screens.rpy	11.04.2024 11:17	Файл "RPY"	9 КБ
screens.rpyc	11.04.2024 11:17	Файл "RPYC"	9 КБ
script.rpy	11.04.2024 11:17	Файл "RPY"	6 КБ
script.rpyc	11.04.2024 11:17	Файл "RPYC"	7 КБ
shooter.rpy	11.04.2024 11:17	Файл "RPY"	1 КБ
shooter.rpyc	11.04.2024 11:17	Файл "RPYC"	2 КБ

Рис. 11.3.

> Файлы с расширением ***.rpy*** можно открыть тем же редактором, который вы используете для написания кода.

Внутри каждого файла находятся все строки, требующие перевода.




```
# game/animations.rpy:66
translate spanish part_29_80d38dc1:

    # lia "Кхм.."
    lia ""

# game/animations.rpy:101
translate spanish patr_19_2b72cdb2:

    # lia "Нам предстоит долгий путь"
    lia ""

# game/animations.rpy:137
translate spanish part_22_732a898a:

    # kay "Хм, здесь никого нет"
    kay ""

# game/animations.rpy:239
translate spanish part_25_153fb3b0:

    # lia "Земля в иллюминаторе, Земля в иллюминаторе"
    lia ""
```

Рис. 11.4.

Здесь вы можете увидеть блоки, каждый из которых состоит из четырех строк, но нам потребуется только вставить текст на другом языке в пустые кавычки. Все остальное оставляйте без изменений, иначе будет нарушена привязка оригинальной реплики и реплики перевода.

Если требуется локализация на несколько разных языков, просто повторите те же операции, только изменив название языка. Если ваш проект выпускается частями, то при выходе нового обновления перевод требуется дополнить новыми диалогами. Для этого снова необходимо нажать "объединение строк", и весь новый текст будет добавлен в аналогичные файлы локализации в самый низ.

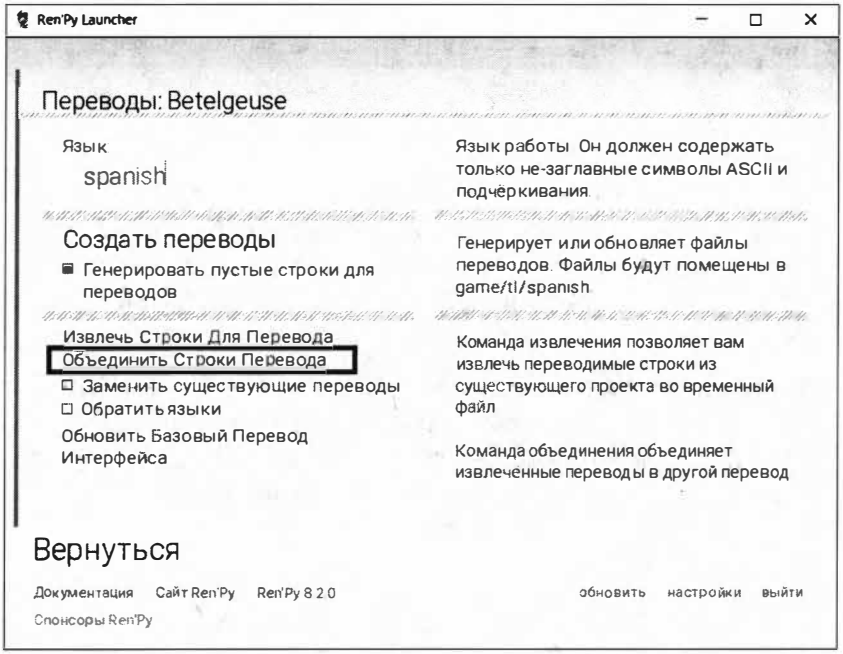


Рис. 11.5.

Стоит иметь в виду, что по умолчанию движок переводит все, что написано в блоках **label** в кавычках. Чтобы создать перевод интерфейса и всего того, что написано в экранах **screen**, нужно предварительно поместить весь текст, предназначенный для перевода, в скобки с нижним подчеркиванием **_()**. Об этом уже упоминалось ранее, и, как правило, эти теги проставляются сразу во время разработки, чтобы не пришлось при создании локализации снова проходить весь код вручную и прописывать скобки, где это необходимо.

```
screen quest_log():
    text _("Журнал заданий") # Этот текст будет переведен _()
    button: # Кнопка
        action Show("interface"), Hide("quest_log")
        idle_background "images/but_s.png"
         text _("Закрыть") # Текст кнопки будет переведен _()
        textbutton _("Закрыть"): # Текстовая кнопка будет переведена
            action Hide("quest_log")
```

Следующим шагом нужно создать соответствующие кнопки для переключения языков в настройках игры. В разделе 5.11 "Оформление экрана настроек" приводился пример таких кнопок. Полный шаблон экрана можно посмотреть там, а здесь будут приведены кнопки для переключения языков.

```
textbutton _("Русский") action Language (None) # Оригинальный язык
textbutton _("Українська") action Language ("ukrainian")
textbutton _("English") action Language ("english")
textbutton _("Espanol") action Language ("spanish")
```

Помимо текстовых кнопок, можно использовать **кнопки-картинки** (imagebutton), если это стилистически подходит вашему проекту. Однако следует иметь в виду, что надписи на изображениях не переводятся. В таком случае художнику придется подготовить однотипные картинки для разных языков.

Завершающим штрихом можно сделать выбор языка при первом запуске игры, чтобы не принуждать игрока искать в настройках этот переключатель. Это будет полезно в случаях, когда игра запускается, например, на русском языке, а иностранный игрок его не понимает, и ему предстоит практически на ощупь искать нужную кнопку на непонятном языке.

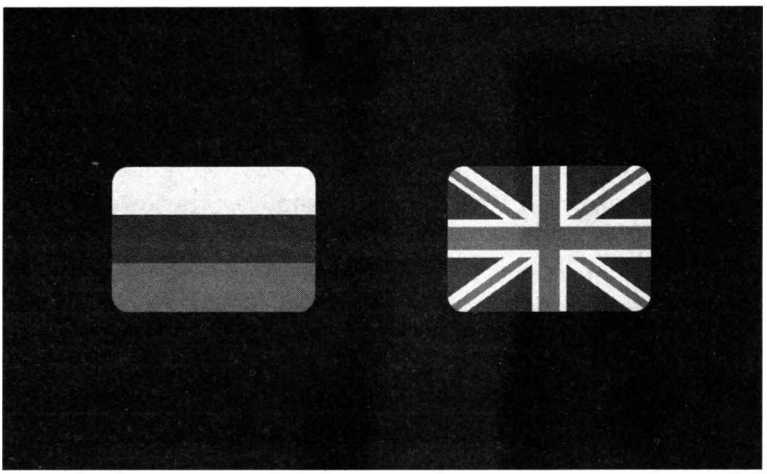


Рис. 11.6.

В разделе 2.2 "Вступительный логотип" мы создавали специальную метку **label splashscreen**, которая всегда запускается перед главным меню. В нее, помимо логотипа или ролика, можно добавить экран с кнопками, в котором будет устанавливаться язык игры.

Для начала создадим переменную, которая будет отслеживать, запускается ли игра в первый раз.

```
default persistent.first = True
```

Далее нам нужен экран с кнопками в виде флажков:

```
screen lang_selection(): # Экран выбора языка
    modal True # Блокируем клики мимо кнопок
    imagebutton:
        align (.2,.5) # Расположение
        idle "russian" # Изображение флага
        # Оставили оригинальный язык, закрыли этот экран
        action [Language(None), Hide("lang_selection"), Jump("logo")]
    imagebutton:
        align (.8,.5) # Расположение
        idle "english" # Изображение флага
        # Переключили язык на испанский, закрыли этот экран
        action [Language("spanish"), Hide("lang_selection"), Jump("logo")]
```

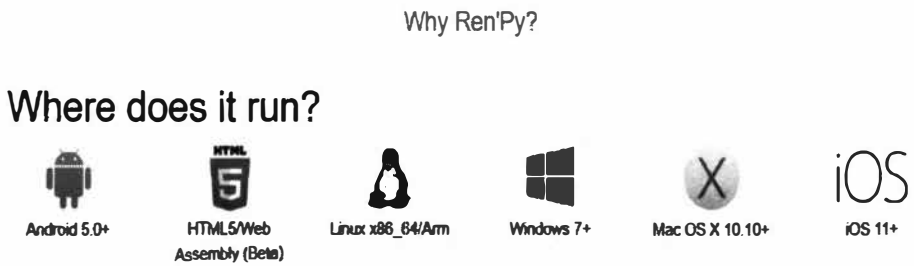
Теперь отрываем экран в метке **splashscreen**:

```
label splashscreen:
    # Проверяем, если игрок запускает игру в первый раз,
    # то выполнится ли этот блок кода
    if persistent.first == True:
        $ persistent.first = False # Выключаем переменную, чтобы выбор языка
больше не предлагался
        scene black with fade # Показали черный фон
        # Открываем экран с выбором языка
        show screen lang_selection with dissolve
        pause # Останавливаем скрипт, пока не сделан выбор
    label logo: # Логотип или видеозаставка
        pause 1 # Сделали паузу на одну секунду
        scene splashs_video with fade # Показали видеозаставку или картинку
        pause 3 # Сделали паузу на длину ролика
        scene black with fade # Затемнение
        return # Перешли в главное меню
```

Таким образом, выбор языка будет предложен только в первый раз. Во всех последующих запусках скрипт будет переходить сразу к метке **label logo** и проигрывать заставку.

Глава 12.

ПРОДВИЖЕНИЕ И МАРКЕТИНГ



После того как игра будет создана, необходимо показать ее миру. Чем больше людей узнают о ней, тем большее их количество смогут сыграть и оценить ее. Если ваш проект коммерческий, от этого будет зависеть прибыль, которую вы сможете получить. Для наибольшего охвата аудитории могут помочь нижеследующие методы продвижения.

Метод 1

В первую очередь необходимо разместить свою игру на максимально возможном количестве площадок, таких как:

- Steam (необходим взнос порядка $100);
- GOG;
- VK Play;
- Яндекс-игры;
- Itch.io — один из крупнейших ресурсов, посвященный инди-играм;
- Lemma Soft, администратором которого является разработчик движка Ren'Py. Renpy.org — официальный сайт. В разделе Ren'Py Games List любой автор визуальной новеллы может разместить свой проект.

Кроме того, можно выложить игру в тематических группах и сообществах различных социальных сетей, таких как Discord, Reddit, VK, Одноклассники и пр.

Метод 2

Заведите социальные сети, посвященные вашему проекту или команде разработчиков. Это отличный способ привлечь внимание аудитории и рас-

сказать о вашей игре. Создайте каналы на популярных платформах, таких как YouTube, TikTok, Instagram, Яндекс Дзен и Rutube. Регулярно публикуйте новости разработки, нововведения, трейлеры, геймплейные видео и другие материалы, которые помогут заинтересовать потенциальных игроков.

Эти ресурсы имеют огромную аудиторию, часть из которой обязательно обратит внимание на ваш проект. Чем больше людей узнает о нем, тем больше шансов на успех. Не забывайте общаться с подписчиками, отвечать на их вопросы и поддерживать интерес к вашему творчеству.

Метод 3

Для успешного продвижения вашего проекта на платформах YouTube и Twitch следует обратить внимание на блогеров и стримеров, специализирующихся на играх. Многие из них готовы рассказать о вашей игре за определенное вознаграждение, однако есть и те, кто готов поделиться своим мнением бесплатно, если им понравится ваше творчество. Таким образом, они получают качественный контент для своих каналов, а вы получаете бесплатную рекламу.

При выборе блогеров обратите внимание на их популярность, активность и лояльность вашему проекту. Сотрудничайте только с теми, кто разделяет ваши интересы и чьи подписчики будут заинтересованы в вашей игре. Помимо оплаты за рекламу, предложите им эксклюзивные материалы или бонусы за продвижение проекта.

Интегрируйте вашу игру в контент блогеров, предложив им готовые сценарии для видео. Это облегчит их задачу и увеличит вероятность их сотрудничества с вами. Отслеживайте результаты этого партнерства, анализируйте количество просмотров, подписчиков и активность аудитории, при необходимости корректируя стратегию продвижения.

Вознаграждайте блогеров после достижения определенных результатов, предоставляя им дополнительные бонусы или подарки. Расширяйте количество блогеров, привлекая новых с похожей аудиторией, поддерживая постоянное общение и предоставляя актуальную информацию о проекте.

Регулярно оценивайте результаты продвижения, сравнивая показатели до начала сотрудничества и после. Определяйте эффективность выбранной стратегии и при необходимости вносите корректировки.

Метод 4

Пройдитесь по сайтам, посвященным играм. Их существует довольно много. Только в ру-сегменте их больше сотни. На каждом из них есть контакты для связи с администрацией.

Составьте письмо с небольшим приветствием и пояснением, что вы являетесь разработчиком игры. Напишите небольшое описание вашего проекта на несколько строк и предложите свою игру на обзор. Это письмо по шаблону отправьте всем администраторам сайтов, контакты которых вы сможете найти.

Если вы только начинающая команда или инди-разработчик, большинство таких ресурсов могут проигнорировать вас. Однако некоторые из них обязательно заинтересуются и опубликуют хотя бы новость о том, что вышла такая игра. Подобные ресурсы ежедневно посещают десятки тысяч игроков, поэтому даже небольшая публикация на нескольких из них позволит привлечь целевую аудиторию.

Метод 5

Вы можете поискать игровые издательства и также написать им. Преимущество этого метода в том, что подобные организации уже имеют большую аудиторию, а также большой опыт и набор своих методик для продвижения. Естественно, с ними придется делиться частью дохода. Часто это довольно существенный процент от общей прибыли.

Метод 6

Еще один способ найти свою целевую аудиторию — это сайты, на которых обсуждаются самые популярные новеллы. Например, можно забить в поисковую строку браузера такие названия, как Doki Doki или Katawa Shoujo. Результатом поиска станут тематические ресурсы, на которых вы также сможете выложить свой проект или создать тему с его обсуждением. Кроме того, можно общаться в чужих темах с ненавязчивыми намеками и отсылками на свою игру.

Этот метод имеет ряд преимуществ. Во-первых, он позволяет привлечь внимание аудитории, которая интересуется определенными жанрами или темами. Во-вторых, обсуждение вашего проекта на тематических ресурсах может способствовать повышению его видимости и узнаваемости среди потенциальных игроков. В-третьих, такое взаимодействие с сообществом поможет вам получить обратную связь от пользователей, что позволит улучшить игру и сделать ее более привлекательной для целевой аудитории.

Метод 7

Аналогично предыдущему методу, можно общаться на тематических форумах, посвященных играм. Особенность большинства форумов в том, что они позволяют создать подпись со ссылкой или небольшим баннером, которые будут добавляться к каждому вашему сообщению. Таким образом, логотип и название игры всегда будут на виду.

Метод 8

Многие группы в социальных сетях имеют фото- и видеогалереи, в которых все участники сообщества могут добавлять свои скриншоты, нарезки видео или мемы. Вам остается добавить в них скриншоты и видео своего проекта, не забыв нанести на них ватермарк (водяной знак вашей лицензии) с названием вашей игры, чтобы пользователи могли самостоятельно ее найти.

Для продвижения подобным методом необходимо выбрать или создать самые яркие и интригующие изображения, чтобы они привлекали внимание, запоминались и побуждали пользователей познакомиться с игрой лично.

Метод 9

Создание разнообразных конкурсов в социальных сетях может стать эффективным методом продвижения вашей игры. Такие конкурсы могут предлагать участникам различные призы, такие как бесплатные ключи, оригинальные версии игры, эксклюзивные арты или другие уникальные материалы.

Важно, чтобы условия конкурса включали в себя действия, благодаря которым участники вольно или невольно будут рекламировать игру. Например, они могут репостить (пересылать) новость о конкурсе или другие публикации о проекте. В таком случае создается эффект «сарафанного радио», повышающий ажиотаж и обсуждение игры.

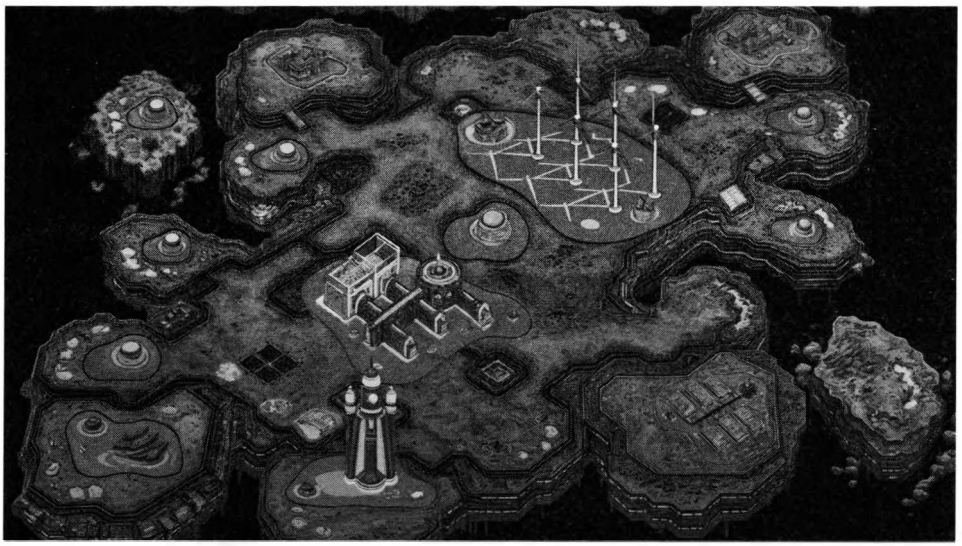
Вот несколько примеров конкурсов, которые можно организовать для продвижения проекта:

- **Предложите участникам создать художественные работы, видеообзоры или музыкальные композиции**, вдохновленные игрой. Лучшие работы могут быть награждены призами, а также опубликованы на официальных страницах вашего проекта.
- **Творческий конкурс на лучший обзор**. Позвольте игрокам создать свои собственные обзоры игры. Это могут быть тексты, видеоролики или стримы (прямые трансляции). Авторы самых интересных и содержательных обзоров могут быть вознаграждены, а их работы использованы для рекламы проекта.
- **Конкурс на создание игрового контента**. Предложите участникам создать модификации, дополнительные сюжетные ответвления или другой игровой контент. Лучшие работы могут быть внедрены в проект как официальное дополнение, а их создатели получат призы и признание в сообществе.
- **Соревнование на лучший фан-арт**. Предложите участникам создать фан-арты, посвященные вашей игре. Это могут быть иллюстрация, комикс или даже косплей (перевоплощение в образ) персонажей. Лучшие работы могут быть опубликованы на официальных ресурсах проекта, а их авторы получат награды и признание в сообществе.

Такие конкурсы не только помогут привлечь внимание к вашей игре, но и будут способствовать ее продвижению, создавая позитивный имидж и укрепляя связи с игроками.

Глава 13.

ТВОРЧЕСКАЯ МАСТЕРСКАЯ



В этом разделе будут представлены полезные механики, которые часто используются в визуальных новеллах и смежных жанрах. Большинство вещей из этого списка востребованы начинающими разработчиками на движке Ren'Py. Некоторые механики довольно сложны в реализации, и о них очень мало информации в сети, поэтому давайте разбираться.

13.1. Создание механик на визуальном примере

Часто авторы вдохновляются чужими работами, и на их основе создают нечто похожее. Но, так как исходники кода не всегда доступны или защищены авторским правом, нужно уметь воспроизводить различный функционал, опираясь только на визуальную демонстрацию.

К примеру, вы зашли на YouTube, увидели ролик с мини-игрой и захотели реализовать что-то похожее в своем проекте. Доступа к коду мини-игры у вас нет. Но вы можете разработать такую же механику, наблюдая, как она работает на видео.

Рис. 13.1.

В нашем примере игра заключается в том, что один из персонажей разбегается и бьет по мячу, затем тот летит в экран и игроку необходимо его отбить.

Рис. 13.2.

Как реализовать такую механику? Первым делом весь функционал необходимо разбить на элементы, чтобы воссоздать каждый по отдельности, а затем соединить их в мини-игру.

Здесь мы видим курсор в виде ладони, который отбивает мяч. Значит, один из элементов — это **замена курсора на новый** (раздел 5.5). Также мы видим, как персонаж разбегается и бьет по мячу. Этот элемент можно создать, анимировав изображение из нескольких спрайтов и двигая их с помощью **трансформации** (разделы 7.1 и 7.3). То есть второй элемент мы тоже способны воссоздать.

Далее, мяч из центра летит в одну из областей экрана — это тоже работа трансформации, которая **перемещает мяч** и увеличивает его, создавая эффект приближения. Мяч летит в случайное место экрана, значит, конечная точка полета должна быть установлена с помощью команды рандома (случайного значения).

Следующий элемент — **пропуск мяча**. В этот момент проигрывается звук столкновения мяча с сеткой ворот, и бьющий персонаж говорит какую-то реплику. Как это сделать, мы тоже знаем.

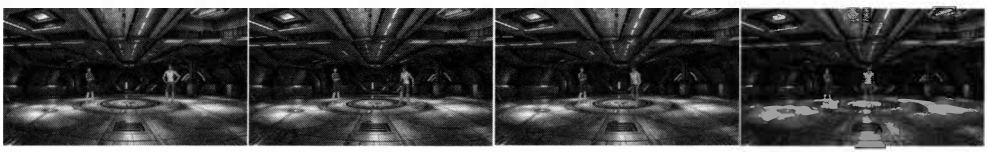
Рис. 13.3.

Последний эпизод — **поимка мяча курсором**. В этот момент проигрывается звук отбивания мяча, и он исчезает с экрана. То есть при касании курсором мяча происходит какое-то взаимодействие, как при **взаимодействии курсора и кнопки** (раздел 7.4). В итоге на визуальном примере мы поняли, как устроена игра, и теперь можем без проблем создать что-то подобное.

Для начала заведем несколько новых переменных для подсчета необходимых событий:

```
default schet_udvor = 0 # Количество ударов по воротам
default schet_otbitiy = 0 # Счетчик перехвата мяча
default schet_liara = 0 # Количество забитых голов Лиарой
default schet_kayden = 0 # Количество забитых голов Кайденом
```

Далее, давайте создадим анимации, которые нам понадобятся:



slide_1 slide_2 slide_3 slide_4

Рис. 13.4.

```
image kayden_udar_pvr: # Анимация разбега Кайдена
    "slide_1" with dissolve # Несколько последовательных изображений,
    pause 0.25 # которые имитируют приближение персонажа к мячу
    "slide_2" with dissolve
    pause 0.25
    "slide_3" with dissolve
    pause 0.25
    "slide_4" with dissolve
    pause 0.25
image liara_udar_pvr: # Аналогичная анимация для Лиары
    "fama1" with dissolve
    pause 0.25
    "fama2" with dissolve
    pause 0.25
    "fama3" with dissolve
    pause 0.25
    "fama4" with dissolve
    pause 0.25
```

Трансформация полета мяча от центра экрана к случайной позиции:

```
transform polet_myacha:
    # Начальный размер изображения и позиция
    zoom 0.15 xpos 945 ypos 740
    # Финальный размер и случайная позиция на экране
    linear 0.7 zoom 1.0 xpos renpy.random.randrange(1920) ypos
renpy.random.randrange(1080)
```

Трансформация длится 0.7 секунды (linear 0.7), то есть за это время игрок должен успеть среагировать и отбить летящий по экрану мяч. Этот параметр важно держать в голове для синхронизации анимаций. Функция **random.randrange(1920)** выбирает случайное целое число в указанном диапазоне (от 0 до 1920).

Теперь давайте последовательно распишем все события игры. Вначале нам необходим стартовый лейбл (метка), в котором будут установлены настройки и в который мы сможем отправлять игрока из любой точки игры, когда это потребуется. Сделать это можно будет командой **jump penalty_game**.

```
label penalty_game: # Стартовая метка для входа в игру
    scene fama1 with fade # Первый слайд для вступительных диалогов
    lia "Сегодня я тебя обыграю"
    kay "Не в этот раз"
    $ config.rollback_enabled = False # Выключили откат назад
    $ default_mouse = "cursor1" # Заменили курсор ладонью
    jump penalty_liara # Переходим к метке анимации Лиары
```

Следующий шаг — создание меток с поочередно повторяющимися событиями, по типу боевой системы, которую мы делали в разделе 10.2.

```
label penalty_liara: # Анимация удара Лиары
    lia "Я бью. Готов?" # Диалоги для предупреждения игрока
    gg "Готов"
    window hide # Закрыли диалоговое окно
    $ schet_udvor +=1 # Добавили +1 к ударам по воротам
    show liara_udar_pvr # Анимация разбега Лиары
    pause 1 # Пауза на время анимации
    play sound kick_ball # Звук удара ноги по мячу
    # Открыли экран полета мяча от Лиары к игроку
    show screen udar_po_vorotam_liara
    pause
label penalty_kayden: # Анимация удара Кайдена
    kay "Моя очередь" # Диалоги для предупреждения игрока
    gg "Бей"
    window hide # Закрыли диалоговое окно
    $ schet_udvor +=1 # Добавили +1 к ударам по воротам
    show kayden_udar_pvr # Анимация разбега Кайдена
    pause 1 # Пауза на время анимации
    play sound kick_ball # Звук удара ноги по мячу
    # Открыли экран полета мяча от Кайдена к игроку
    show screen udar_po_vorotam_kayden
    pause
```

Оба вышеописанных лейбла заканчиваются открытием экрана с анимацией полета мяча. Для каждого персонажа свой экран, в котором переключаются соответствующие переменные и происходят прыжки к меткам с проверками.

```
screen udar_po_vorotam_liara(): # Экран удара Лиары
    modal True # Блокируем клики мимо кнопки
    imagebutton at polet_myacha: # Кнопка с трансформацией мяча
        idle "images/gym/ball.png" # Изображение мяча
```

```
        # Действия при клике на мяч курсором
        action [Hide("udar_po_vorotam_liara"), # Закрыли этот экран
            Play("sound","audio/kick_ball2.mp3"), # Звук отбития мяча
            # Считаем количество отбитий
            SetVariable("schet_otbitiy",schet_otbitiy+1),
            Jump("penalty_game1")] # Переход в метку проверки_1
    # Время полета мяча длится 0.7 секунды,
    # если игрок не успел за это время отбить,
    # срабатывает таймер гола
    timer 0.71 action [Hide("udar_po_vorotam_liara"), # Закрыли экран
        Play("sound","audio/setka.mp3"),# Звук попадания мяча в сетку
        SetVariable("schet_liara",schet_liara+1), # Лиара +1 гол
        Jump("penalty_game1")] # Переход в метку проверки_1
screen udar_po_vorotam_kayden(): # Экран удара Кайдена
    modal True # Блокируем клики мимо кнопки
    imagebutton at polet_myacha: # Кнопка с трансформацией мяча
        idle "images/gym/ball.png" # Изображение мяча
        action [Hide("udar_po_vorotam_kayden"), # Закрыли этот экран
            Play("sound","audio/kick_ball2.mp3"), # Звук отбития мяча
            # Считаем количество отбитий
            SetVariable("schet_otbitiy",schet_otbitiy+1),
            Jump("penalty_game2")] # Переход в метку проверки_2
    # Время полета мяча длится 0.7 секунды,
    # если игрок не успел за это время отбить,
    # срабатывает таймер гола
    timer 0.71 action [Hide("udar_po_vorotam_kayden"), # Закрыли
        Play("sound","audio/setka.mp3"),# Звук попадания мяча в сетку
        SetVariable("schet_kayden",schet_kayden+1), # Кайден +1 гол
        Jump("penalty_game2")] # Переход в метку проверки_2
```

В данном варианте отбитие мяча засчитывается, только если игрок успел кликнуть по мячу. Это довольно сложное действие, учитывая, что времени на это дается 0.7 секунды. Поэтому можно облегчить задачу и сделать отбитие мяча при наведении курсора на кнопку, без необходимости кликать. Для этого нужно добавить опцию **hovered** для каждой кнопки, с такими же командами, как у **action**.

```
hovered [Hide("udar_po_vorotam_liara"), # Закрыли этот экран
    Play("sound","audio/kick_ball2.mp3"), # Звук отбития мяча
    # Считаем количество отбитий
    SetVariable("schet_otbitiy",schet_otbitiy+1),
    Jump("penalty_game1")] # Переход в метку проверки_1
```

Далее, независимо от того, отбит мяч или нет, игрок попадает в метки с проверками. Их можно прописать в одном общем лейбле (метке), но, для процесса обучения и понимания всех событий, сделаем для каждого персонажа отдельно.

```
label penalty_game1: # Проверка после удара Лиары
    hide liara_udar_pvr # Скрыли анимацию разбега Лиары
    # Проверка количества ударов:
    if schet_udvor ==10: # Если ударов по воротам 10,
        jump penalty_final # отправляем в финальный лейбл мини-игры.
    else: # Иначе передаем ход Кайдену
        jump penalty_kayden
label penalty_game2: # Проверка после удара Кайдена
    hide kayden_udar_pvr # Скрыли анимацию разбега Кайдена
    if schet_udvor ==10: # Если ударов по воротам 10,
        jump penalty_final # отправляем в финальный лейбл мини-игры.
    else: # Иначе передаем ход Лиаре
        jump penalty_liara
```

То есть персонажи будут по очереди бить по мячу, пока счетчик ударов по воротам не достигнет десяти. После этого игрок отправляется в финальную метку мини-игры, в которой происходит подсчет результатов. В зависимости от пропущенных/отбитых мячей можно выдать различные награды, достижения и пр.

```
label penalty_final: # Финальная метка мини-игры
    $ default_mouse = "default" # Возвращаем нормальный курсор
    $ config.rollback_enabled = True # Включили откат назад
    scene fama5 with dissolve # Какая-то сцена
    if schet_kayden > schet_liara: # Если счет Кайдена больше Лиары
        kay "Как я и говорил - не сегодня"
        lia "Тебе просто повезло"
    elif schet_liara > schet_kayden: # Если у Лиары больше
        kay "Ладно, твоя взяла"
        lia "Да ну вас, вечно вы поддаетесь"
    else:
        lia "Ничья… Считай, зря потратили время"
        kay "Нужно наслаждаться процессом"
    # Проверяем, сколько мячей отбил игрок
    if schet_otbitiy ==10: # Если 10
        kay "Отлично, Кэп, ты стоял, как этот.. как его.. Джанлуиджи
ван дер Сар"
    elif schet_otbitiy >=6: # Если отбил 6-9 мячей
```

```
    lia "Вы меня удивляете, капитан"
else: # Если меньше
    kay "Что-то ты сегодня не в форме"
scene black with fade # Затемнение
# Сбрасываем все счетчики для следующего раза
$ schet_udvor =0
$ schet_otbitiy = 0
$ schet_liara = 0
$ schet_kayden = 0
jump part_567 # Переходим в следующий эпизод игры
```

Таким образом, не зная, как реализована определенная механика, можно внимательно проанализировать, из каких элементов она состоит, и на основе тех навыков и знаний, что у вас есть, попробовать сделать что-то похожее.

13.2. Создание карты для перемещения по локациям

Карта мира является одной из самых часто используемых механик, если речь не идет о кинетических новеллах. Большинство игр, так или иначе, предлагают игроку свободу выбора, и карта в полной мере может предложить такую свободу. Например, меню выбора (menu:) тоже можно представить как карту с двумя и более выборами.

На самом деле, к текущему моменту вы уже без труда способны создать карту мира самостоятельно. Так как ранее, похожим образом, мы делали **главное меню** (раздел 5.8), в котором использовалась та же механика с кнопками и переходами по разным экранам. В целом, если не прибегать к возможностям языка Python, то большинство функций в движке реализуются с помощью экранов.

В этом разделе в качестве закрепления мы разберем простой пример карты мира, а также несколько мини-игр, которые можно сделать на ее основе.

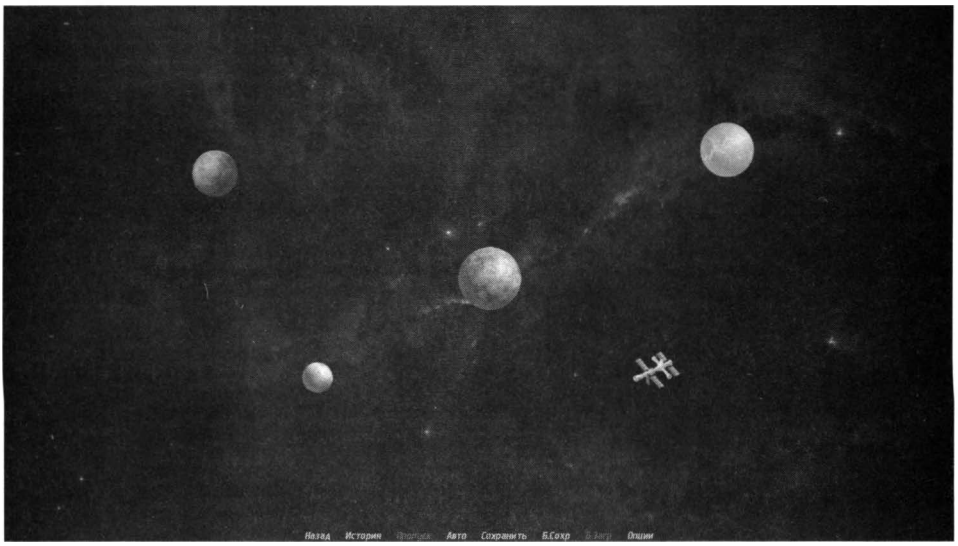


Рис. 13.5.

Карта мира представляет собой *фоновое изображение* с несколькими интерактивными элементами на ней.

Такие элементы реализуются в виде кнопок, при клике на которые происходит переход к той или иной локации.

```
screen map(): # Карта звездной системы
    modal True # Блокируем взаимодействие с другими слоями
    add "star_map" # Добавляем фоновое изображение космоса
    imagebutton: # Звезда Бетельгейзе
        action Hide("map") # Закрываем карту, остаемся на месте
        pos(910,480) # Позиция изображения в пикселях
        idle "images/star_map/pl1.png" # Изображение планеты/звезды
        hover "images/star_map/pl11.png" # Изображение при наведении
        focus_mask True # Игнорируем прозрачные края изображения
        tooltip _("Бетельгейзе") # Всплывающее описание
    imagebutton: # Планета Вапиури
        action [Hide("map"), Jump("vapiuri")] # Переход к локации
        pos(380,285)
        idle "images/star_map/pl2.png"
        hover "images/star_map/pl22.png"
        focus_mask True
```

```
        tooltip _("Планета Вапиури")
    imagebutton: # Планета Дипа
        action [Hide("map"), Jump("dipa")] # Переход к локации
        pos(600,710)
        idle "images/star_map/pl4.png"
        hover "images/star_map/pl44.png"
        focus_mask True
        tooltip _("Планета Дипа")
    imagebutton: # Планета Сусандр
        action [Hide("map"), Jump("susandr")]
        pos(1400,235)
        idle "images/star_map/pl3.png"
        hover "images/star_map/pl33.png"
        focus_mask True
        tooltip _("Планета Сусандр")
    imagebutton: # Космическая станция
        action [Hide("map"), Jump("gagarin_station")]
        pos(1270,695)
        idle "images/star_map/gagarin.png"
        hover "images/star_map/gagarin2.png"
        focus_mask True
        tooltip _("Станция Гагарин")
    $ tooltip = GetTooltip() # Функция Tooltip
    if tooltip:
        frame:
            pos renpy.get_mouse_pos()
            anchor (-0.07, -0.4)
            xsize 350 ysize 60
            background "#00000080"
            text __("[tooltip]") align (.5,.5)
```

В примере каждая из кнопок при нажатии на нее закрывает карту звездной системы и отправляет в определенную локацию. Кнопка звезды в центре экрана работает как кнопка простого закрытия карты без перемещения куда-либо.

Для простоты определения позиций при расположении кнопок на экране можно перед созданием карты в коде создать ее в Фотошопе или в другом графическом редакторе. То есть нужно открыть фоновое изображение в программе и наложить на него спрайты планет в тех местах, где вы хотите, чтобы они располагались. Затем наведите курсор на верхний левый угол каждого спрайта, чтобы узнать их позиции. В графических редакторах они прописываются внизу.

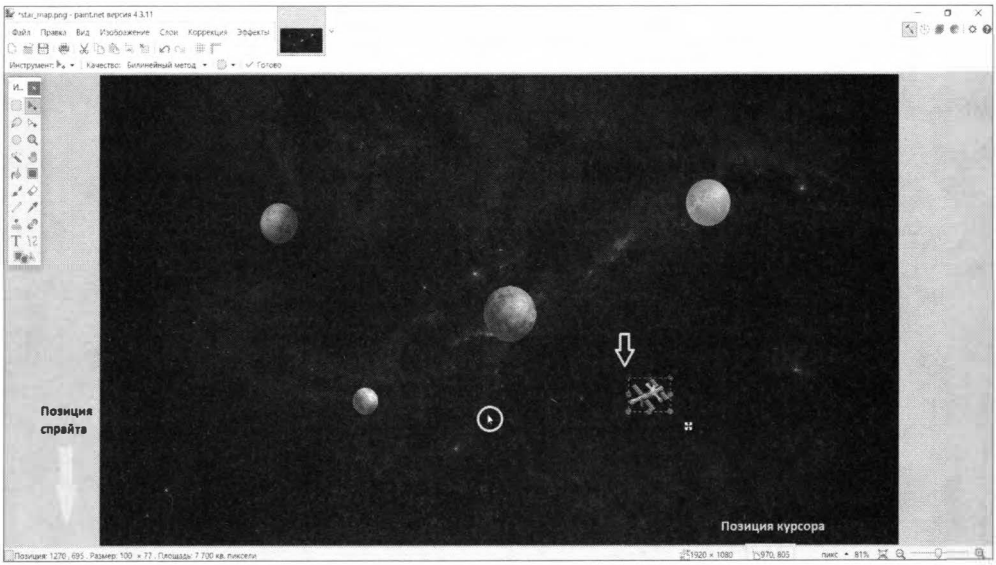


Рис. 13.6.

Далее нам нужно создать лейблы, в которые игрок будет попадать при выборе локации на карте.

```
label vapiuri:
    scene vapiuri with fade # Фоновое изображение локации
    $ renpy.notify(_("Планета Вапиури")) # Всплывающее уведомление
    show kayden normal2 at center # Показали персонажа
    kay "Любопытное местечко"
    # Здесь могут быть любые диалоги, события или карта этой локации
    jump part_589
label dipa:
    scene dipa with fade
    $ renpy.notify(_("Планета Дипа"))
    # Здесь могут быть любые диалоги, события или карта этой локации
    jump part_577
label susandr:
    scene susandr with fade
    $ renpy.notify(_("Планета Сусандр"))
    # Здесь могут быть любые диалоги, события или карта этой локации
    jump part_561
label gagarin_station:
    scene gagarin_st with fade
    $ renpy.notify(_("Станция Гагарин"))
    # Здесь могут быть любые диалоги, события или карта этой локации
    jump part_532
```

Доступ к самой карте можно сделать на кнопке в интерфейсе, как это было описано в разделе 5.3.

```
screen interface():
    imagebutton: # Карта звездной системы
        action Show("map") # Открывает карту
        align(0.16,0.01) # Позиция кнопки на экране
        idle "images/kosmolet.png" # Изображение кнопки
        hover "images/kosmolet.png" # Изображение при наведении
        focus_mask True # Игнорирует прозрачные края изображения
        tooltip _("Карта") # Всплывающее описание
```

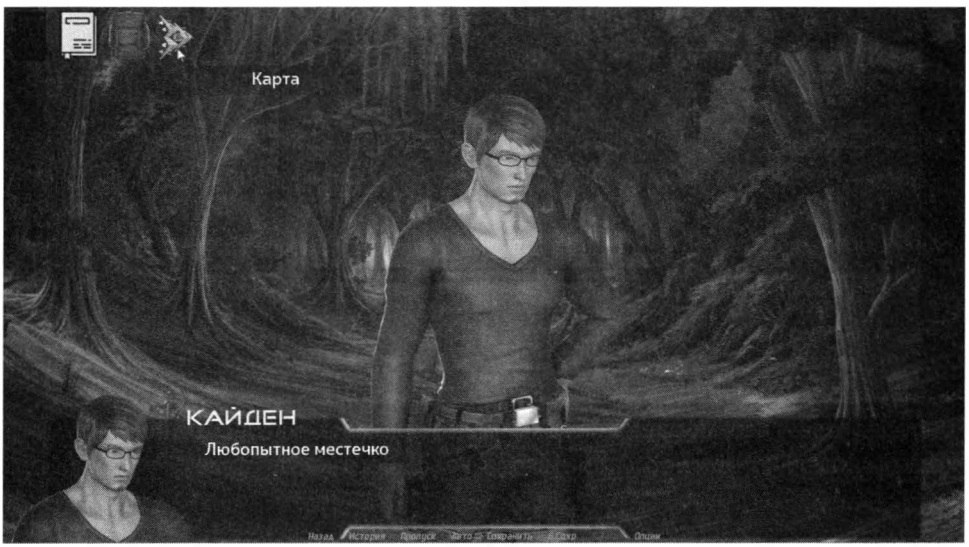


Рис. 13.7.

13.3. Мини-игра "Пианино"

Как было описано выше, **структура карты — это набор из нескольких кнопок, которые выполняют определенные действия**.

Где еще можно использовать большое количество кнопок? Например, на клавиатуре пианино. Из чего следует, что можно создать полноценный цифровой синтезатор, у которого по нажатии каждой клавиши будет воспроизводиться звук определенных нот и полутонов.

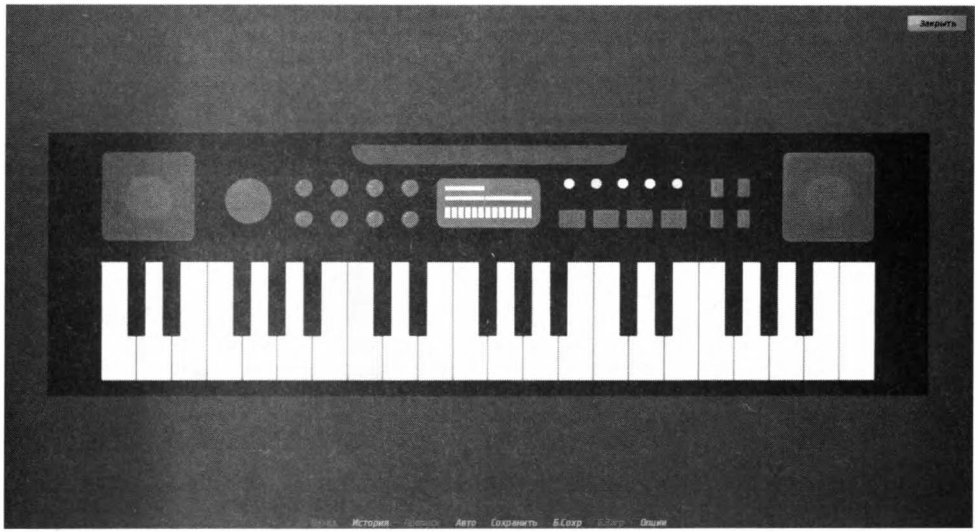


Рис. 13.8.

Это можно использовать для мини-игры. Для краткости кода сделаем наше пианино с двумя функциональными клавишами, остальные вы можете создать, скопировав код предыдущей кнопки и заменив в ней звуковой файл.

Сначала зарегистрируем новые звуковые каналы, чтобы у каждой ноты был свой. Это нужно для того, чтобы новая нота не прерывала предыдущую при проигрывании в том же звуковом канале. То есть ноты будут звучать параллельно, сливаясь в аккорды.

```
init python:
    # В скобках название канала (do), группа звуков (sfx),
    # повтор звука выключен (False)
    renpy.music.register_channel("do", "sfx", False)
    renpy.music.register_channel("re", "sfx", False)
    renpy.music.register_channel("mi", "sfx", False)
    renpy.music.register_channel("fa", "sfx", False)
    renpy.music.register_channel("sol", "sfx", False)
    renpy.music.register_channel("lya", "sfx", False)
    renpy.music.register_channel("si", "sfx", False)
    renpy.music.register_channel("do#", "sfx", False)
    renpy.music.register_channel("re#", "sfx", False)
    renpy.music.register_channel("fa#", "sfx", False)
    renpy.music.register_channel("sol#", "sfx", False)
    renpy.music.register_channel("lya#", "sfx", False)
```

Теперь создадим экран, во многом похожий на экран для звездной карты, с единственным отличием в том, что при нажатии на кнопку будет воспроизводиться звук вместо перехода в новую локацию.

```
screen piano():
    modal True # Игнорируем нажатия мимо кнопок
    add "images/piano/piano.jpg" # Фоновое изображение синтезатора
    imagebutton: # Клавиша "До"
        # В скобках звуковой канал и название файла
        action Play("do", "do.mp3") # Звук при нажатии
        pos(681,530) # Позиция изображения в пикселях
        idle "images/piano/do.png" # Изображение клавиши
        hover "images/piano/do2.png" # Изображение при наведении
        focus_mask True # Игнорируем прозрачные края изображения
    imagebutton: # "Ре"
        action Play("re", "re.mp3")
        pos(751,530)
        idle "images/piano/re.png"
        hover "images/piano/re2.png"
        focus_mask True
    # Вы можете скопировать кнопку "Ре" и изменить
    # звуковой канал, mp3-файл, позицию и изображение
    button: # Кнопка "Закрыть"
        action [Hide("piano"), Jump("part_108")]
        align(0.99,0.02)
        xsize 120 ysize 35 # Размер картинки
        idle_background "images/but_s.png"
        hover_background "images/but_h.png"
        text _("Закрыть")
```

Вместо нажатия курсором на клавишу можно привязать кнопки к клавиатуре (раздел 5.7):

```
screen piano():
    modal True
    add "images/piano/piano.jpg"
    key "q" action Play("do", "do.mp3")
    key "w" action Play("re", "re.mp3")
    key "e" action Play("mi", "mi.mp3")
    key "r" action Play("fa", "fa.mp3")
    key "t" action Play("sol", "sol.mp3")
    key "y" action Play("lya", "lya.mp3")
    key "u" action Play("si", "si.mp3")
```

Ну а чтобы открыть экран с синтезатором, можно сделать кнопку, аналогичную кнопкам в интерфейсе, или в нужном месте сценария использовать команду **show screen**:

```
label part_99:
    stop music # Выключаем текущую композицию, чтобы слышать пианино
    show screen piano with dissolve # Открыли экран
    pause
```

Таким же образом можно создать любой другой музыкальный инструмент.

13.4. Мини-игра "Уборка беспорядка"

Еще один вариант использования механики карты — это удаление предметов с экрана. Например, игроку выдается квест по уборке помещения, поиску улик или сборке определенных материалов.

Реализовать такие мини-игры можно, отобразив на экране предметы в виде кнопок, при нажатии на которые будут переключаться переменные или что-то добавляться в инвентарь.

Рис. 13.9.

Каждая из кнопок отображается на экране только при определенном условии, поэтому сначала создадим для каждой свою переменную:

```
default item1 = True # Переменные для трех предметов
default item2 = True
default item3 = True
```

Для разнообразия, при повторном прохождении квеста по уборке, позиции предметов можно рандомизировать с помощью трансформации:

```
transform for_clean:
    xpos renpy.random.randint (50,1860) ypos renpy.random.randint (730,980)
```

Сам экран будет выглядеть следующим образом:

```
screen cleaning():
    modal True
    add "compart" # Фоновое изображение
    if item1 ==True: # Если переменная равна True, отображается кнопка
        imagebutton at for_clean: # С использованием трансформации
            action SetVariable("item1", False) # Изменили переменную
            # Позиции не пишем, так как установили их в трансформации
            idle "images/cleaning/gr_1.png" # Изображение предмета
            focus_mask True
    if item2 ==True: # Если второй предмет == True
        imagebutton at for_clean:
            action SetVariable("item2", False)
            idle "images/cleaning/gr_2.png"
            focus_mask True
    if item3 ==True: # Если третий предмет == True
        imagebutton at for_clean:
            action SetVariable("item3", False)
            idle "images/cleaning/gr_3.png"
            focus_mask True
    button: # Кнопка "Уйти"
        action [Hide("cleaning"), Jump("part_108")]
        align(0.99,0.02)
        xsize 120 ysize 35
        idle_background "images/but_s.png"
        hover_background "images/but_h.png"
        text _("Уйти")
```

То есть при нажатии на предмет его переменная переключается в значение *False*, и условие для отображения кнопки перестает выполняться. В результате предмет пропадает с экрана, создавая эффект уборки ангара. Этот квест можно повторять, если через какое-то время вернуть игрока в это место и включить переменные для отображения предметов:

```
label part_112: # Отправляем игрока в эту метку командой jump
    $ item1 = True # Включаем переменные
    $ item2 = True
    $ item3 = True
    show screen cleaning # Открываем экран уборки
    pause
```

Благодаря тому, что позиции кнопок выбираются случайно, в следующий раз они будут находиться на новом месте. А если сделать помещение более захламленным, а предметы более сливающимися с фоном, то процесс поиска может сильно усложниться.

13.5. Создание инвентаря/магазина

При удалении предметов с экрана можно помещать их в инвентарь игрока, например, если он искал какую-то улику или собирал детали для ремонта двигателя, которые могут пригодиться в будущем. Инвентарь, как и другие элементы интерфейса, можно отобразить в отдельном экране при нажатии на соответствующую иконку.

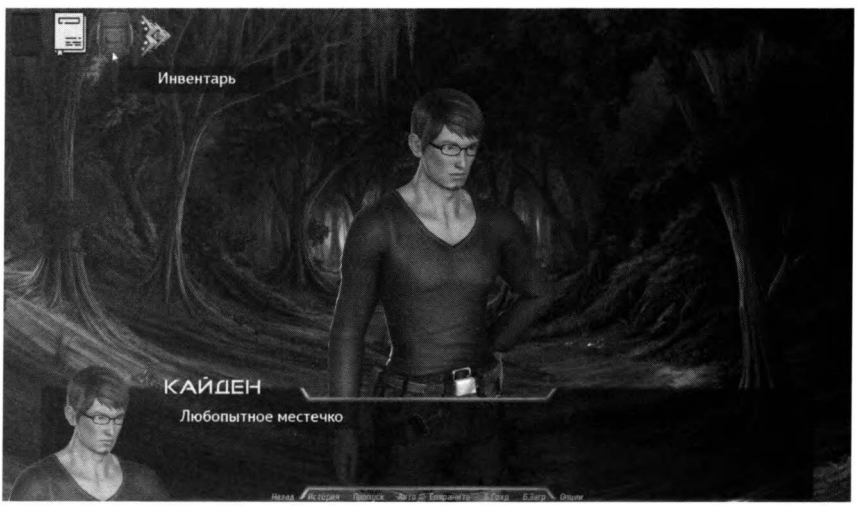


Рис. 13.10

```
screen interface():
    imagebutton: # Инвентарь
        action Show("inventory") # Открывает экран при нажатии
        align(0.11,0.01) # Позиция иконки инвентаря
        idle "images/inventory.png" # Изображение иконки
        hover "images/inventory.png" # Изображение при наведении
        focus_mask True # Игнорируем прозрачные пиксели изображения
        tooltip _("Инвентарь") # Описание при наведении
```

Для учета предметов в инвентаре создадим для каждого свою переменную:

```
default detail_1 = 0 # Количество деталей, по умолчанию 0
default detail_2 = 0
default detail_3 = 0
```

Теперь напишем экран инвентаря:

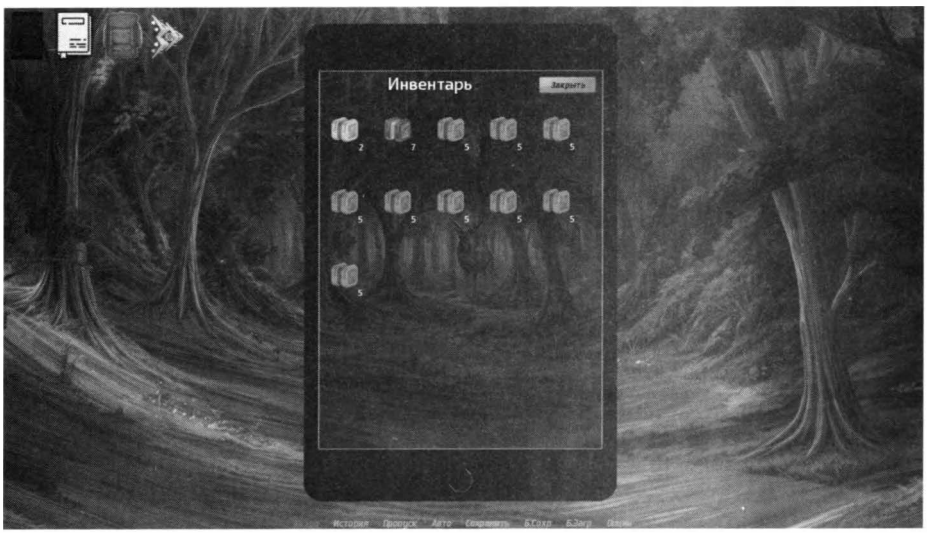


Рис. 13.11.

```
screen inventory(): # Экран инвентаря
    modal True
    add "bg_inv" # Фоновое изображение инвентаря
    text _("Инвентарь") align (0.46,0.14)
    hbox: # Коробка, которая разложит предметы по порядку
        xsize 550 # Ширина коробки
```

```
        pos (670,220) # Позиция коробки на экране
        box_wrap True # Перенос предметов на новую строку
        box_wrap_spacing 50 # Расстояние между предметами
        if detail_1 >0: # Деталь_1 отображается, если больше 0
            fixed: # Контейнер с элементами предмета
                xsize 100 ysize 100 # Размер контейнера
                add "detail_1" # Изображение предмета
                text "[detail_1]" style "inv_text" # Количество
        if detail_2 >0: # Деталь_2 отображается, если больше 0
            fixed:
                xsize 100 ysize 100
                add "detail_2"
                text "[detail_2]" style "inv_text"
        if detail_3 >0: # Деталь_3
            fixed:
                xsize 100 ysize 100
                add "detail_3"
                text "[detail_3]" style "inv_text"
    button: # Кнопка "Закрыть инвентарь"
        action Hide("inventory")
        align(0.62,0.14)
        xsize 120 ysize 35 # Размер картинки
        idle_background "images/but_s.png"
        hover_background "images/but_h.png"
        text _("Закрыть")
```

Текст для каждой детали имеет стиль style "inv_text", в него занесены повторяющиеся элементы, отвечающие за отображение количества данного предмета в инвентаре:

```
style inv_text:
    xalign .8 yalign .8 # Расположение внутри fixed
    size 20 # Размер текста
    color "#ffffff" # Цвет текста
```

Таким образом, каждый предмет в инвентаре отображается, только если у игрока его больше нуля. В противном случае иконка будет скрыта. В примере все детали представляют собой картинки, но их можно выводить в виде кнопок *imagebutton* или *button*. То есть с каждым предметом можно сделать дополнительное взаимодействие.

Теперь при уборке предметов в ангаре из предыдущего раздела их можно не просто удалять с экрана, а и добавлять в инвентарь игроку.

```
screen cleaning(): # Экран уборки ангара из предыдущего раздела
    modal True
    add "compart" # Фоновое изображение
    if item1 ==True: # Если переменная равна True отображается кнопка
        imagebutton at for_clean: # С использованием трансформации
            action [SetVariable("item1", False),
                SetVariable("detail_1", detail_1 +1),
                Notify (_("Деталь добавлена в инвентарь"))]
            idle "images/cleaning/gr_1.png"
            focus_mask True
```

При нажатии на предмет во время уборки в **action** происходят три действия:

1. SetVariable("item1", False) — предмет пропадает с экрана.
2. SetVariable("detail_1", detail_1 +1) — Деталь_1 увеличивает свое значение на 1. То есть она становится больше нуля и отображается в инвентаре. Если деталь уже была в инвентаре, изменится ее количество на +1.
3. Notify (_("Деталь добавлена в инвентарь")) — всплывающее сообщение.

Соответственно, когда персонаж тратит предмет, например, для ремонта двигателя, его нужно удалить из инвентаря.

```
label part_217:
    scene engine with dissolve
    menu:
        "Починить двигатель" if detail_1 >0:
     # Выбор доступен, если в инвентаре есть деталь
            kay "Ну вот, теперь все работает"
            $ detail_1 -= 1 # Отняли одну деталь
            $ renpy.notify(_("Потрачена одна деталь"))
            jump part_218
        "Уйти":
            kay "Пожалуй, в следующий раз"
            jump part_219
```

Данный вариант демонстрирует, как реализовать инвентарь с помощью возможностей движка (системы экранов). Но вы также можете создать его с помощью языка Python.

> Для этого вам понадобится список или словарь, в который с помощью методов **append** и **remove** будут добавляться или удаляться предметы.

Подробнее об этом смотрите в разделах 8.1 и 8.2.

Помимо деталей, в инвентаре могут быть абсолютно разные вещи, которые игрок может не только найти, но и купить. Для этого нам потребуется создать магазин. Делается он аналогично инвентарю, только вместо иконок предметов у нас будут кнопки, при нажатии на которые предмет будет добавляться в рюкзак и со счета будет списываться определенная сумма.

Давайте создадим переменную денег:

```
default money = 100 # Любая сумма по умолчанию
```

Экран магазина будет выглядеть следующим образом:

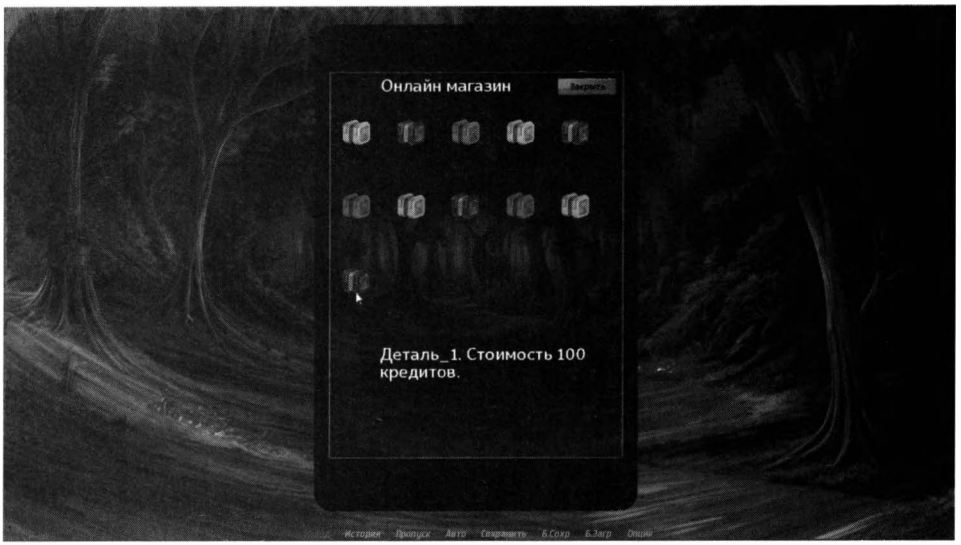


Рис. 13.12.

```
screen online_store(): # Экран магазина
    modal True
    add "bg_inv"
    text _("Онлайн магазин") align(0.46,0.14)
    text _("Кредитов на счете: [money]") align(0.5,0.82)
    hbox: # Коробка, которая разложит предметы по порядку
        xsize 550
        pos (670,220)
        box_wrap True
        box_wrap_spacing 50
        button: # Кнопка для Деталь_1
            xsize 100 ysize 100 # Размер кнопки
            idle_background "images/inventory/detail_1.png"
            hover_background "images/inventory/detail_1.png"
            tooltip __("Деталь_1. Стоимость 100 кредитов.")
            if money >=100: # Если хватает денег на покупку
                action [SetVariable("detail_1", detail_1 +1),
                    SetVariable("money", money-100),
                    Notify(_("Куплена Деталь_1"))]
            else: # Если не хватает
                action Notify(_("Недостаточно средств"))
    $ tooltip = GetTooltip() # Функция Tooltip
    if tooltip:
        frame:
            pos renpy.get_mouse_pos()
            anchor (-0.07, -0.4)
            xsize 500 ysize 120
            background "#00000080"
            text __("[tooltip]") align (.5,.5)
```

В примере при наведении на изображение предмета всплывает подсказка с названием товара и его ценой. А при покупке происходит проверка: если у игрока достаточно денег (if money >=100), то срабатывает **action** с добавлением одной детали в инвентарь, списанием ста кредитов и всплывающим уведомлением о покупке. Если денег недостаточно (else), срабатывает другое действие с соответствующим уведомлением.

13.6. Создание журнала заданий

Журнал заданий облегчает игроку прохождение и позволяет вспомнить список задач после паузы между игровыми сессиями.

В разделах 5.1 и 5.2 приводился пример подобного элемента интерфейса, поэтому основные понятия нам уже знакомы. Здесь мы разберем структуру вывода квестов на экран, которые будут отображаться, если задача активна, и скрываться, когда она выполнена.

Для начала давайте разделим квест-лог на несколько вкладок, чтобы отсортировать задания на основные, второстепенные и пройденные. Для этого нам понадобится одна переменная, которая будет служить переключателем.

```
default vkladka = 1 # Активна первая вкладка по умолчанию
```

Теперь возьмем пример скрипта для квест-лога из раздела 5.2 и немного переработаем его, добавив кнопки для переключения между вкладками.

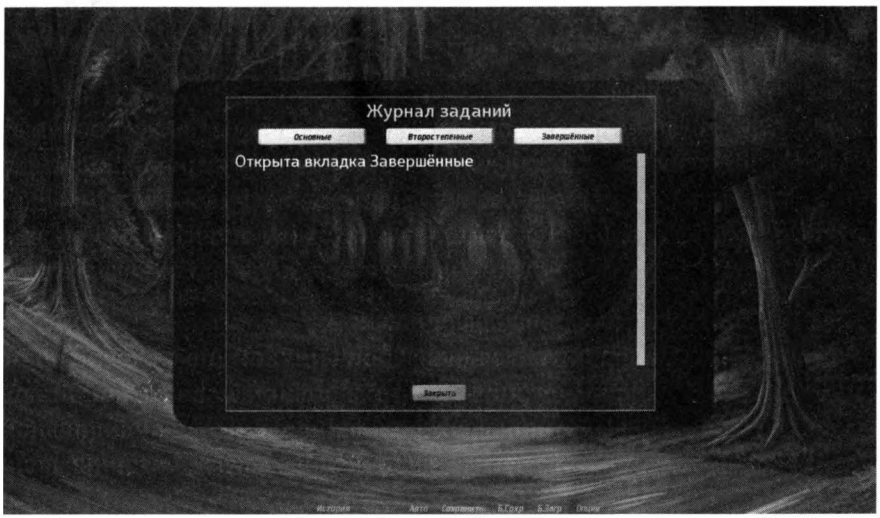


Рис. 13.13.

```
screen quest_log():
    modal True
    add "qwlog" # Фоновое изображение планшета на весь экран
    text _("Журнал заданий") align(0.5,0.2) size 40 color "#88ffff"
    hbox: # Контейнер с кнопками для переключения вкладок
        align(0.5,0.25) # Расположение бокса
        spacing 40 # Расстояние между кнопками
        button: # Основные квесты
            action SetVariable("vkladka", 1) # Установили на 1
            xsize 240 ysize 35
            idle_background "images/bvkl1.png"
```

```
        hover_background "images/bvkl2.png"
        text _("Основные")
    button: # Второстепенные квесты
        action SetVariable("vkladka", 2) # Установили на 2
        xsize 240 ysize 35
        idle_background "images/bvkl1.png"
        hover_background "images/bvkl2.png"
        text _("Второстепенные")
    button: # Завершенные квесты
        action SetVariable("vkladka", 3) # Установили на 3
        xsize 240 ysize 35
        idle_background "images/bvkl1.png"
        hover_background "images/bvkl2.png"
        text _("Завершенные")
viewport: # Область прокрутки
    align(0.5,0.5) # Расположение области
    xysize(900,450) # Размер области
    scrollbars "vertical" # Возможность вертикального скрола
    mousewheel True # Возможность скрола мышью
    draggable True # Возможность скрола перетаскиванием области

    if vkladka==1: # Если нажата первая вкладка
        text "Открыта вкладка Основные" align(0.5,0.2)
    elif vkladka==2: # Если нажата вторая вкладка
        text "Открыта вкладка Второстепенные" align(0.5,0.2)
    elif vkladka==3: # Если нажата третья вкладка
        text "Открыта вкладка Завершенные" align(0.5,0.2)
```

При нажатии на одну из кнопок переменная получает соответствующее значение, и в разделе **viewport** отображается тот текст, который совпадает с условием. То есть, если нажата кнопка "Завершенные", переменная принимает значение **3** (action SetVariable("vkladka", 3)) и в журнале отображается текст, который отвечает условию *elif vkladka==3*. Таким образом, можно сделать сколько угодно вкладок с помощью всего одной переменной и разнести по ним различные квесты, справочники и прочую информацию.

Следующий пункт — отображение только той стадии квеста, которая сейчас активна. Обычно многие квесты в играх состоят из множества последовательных задач, где без выполнения одной невозможно перейти к другой. Например, Лиара может отправить игрока поговорить с Кайденом, тот сообщит, что занят и ему нужна помощь капитана, поэтому игроку нужно найти и уговорить капитана помочь Кайдену. Тот, в свою очередь, может пожаловаться на здоровье, и ему срочно нужна медицинская помощь Ксеоны,

поэтому игрок отправляется уже на ее поиски. А она, в свою очередь, отправит игрока на ближайшую планету за недостающими медикаментами.

В результате наш квест состоит из следующих стадий:

1. Поговорить с Лиарой.
2. Найти и поговорить с Кайденом.
3. Попросить помощи у капитана.
4. Сообщить Ксеоне о ранении капитана.
5. Отправиться на ближайшую планету за лекарствами.
6. Принести Ксеоне медикаменты.
7. Помочь ей с лечением капитана.
8. Отправиться с капитаном к Кайдену.
9. Починить с Кайденом и капитаном двигатель.
10. Сообщить Лиаре о результатах.

Таким образом, мы имеем десять стадий квеста, но в журнале должна отображаться только одна — текущая задача. Поэтому, как и в случае с вкладками в журнале, стадии квеста тоже будут отображаться в зависимости от значения переменной.

```
default liara_quest = 0 # Стадия квеста Лиары по умолчанию
```

В начале игры журнал задач может быть пустым, пока игрок не познакомится с персонажами. После знакомства с Лиарой стадия ее квеста переключается на 1, и в журнале появляется первая запись.

```
label start_2:
    gg "Меня между делом познакомили с Лиарой"
    gg "Нужно как-нибудь найти ее и поболтать"
    $ liara_quest = 1 # Переключили стадию
    $ renpy.notify(_("Журнал заданий обновлен"))
    jump map_kosmolet
```

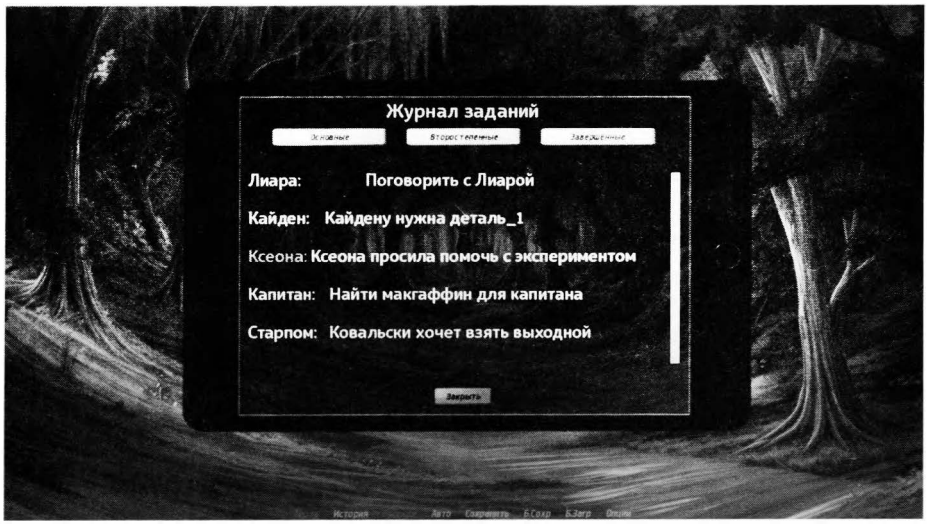


Рис. 13.14.

В коде это будет выглядеть следующим образом:

```
viewport: # Область прокрутки
    align(0.5,0.52)
    xysize(900,400)
    scrollbars "vertical"
    mousewheel True
    draggable True
    #yinitial 20
    if vkladka==1:
        vbox: # Вертикальный бокс со всеми квестами
            xysize(850,400)
            xalign 0.5
            hbox: # Контейнер Лиары, где отображаются ее квесты
                xsize 850
                if liara_quest >0 and liara_quest<11: # Условие отображения
имени в журнале
                    text _("Лиара:") color "#c8ffc8"
                if liara_quest ==1: # Условие отображения 1-го квеста
                    text _("Поговорить с Лиарой") xalign - 0.3
                elif liara_quest ==2:
                    text _("Найти и поговорить с Кайденом")
                elif liara_quest ==3:
                    text _("Попросить помощи у капитана")
                elif liara_quest ==4:
                    text _("Сообщить Ксеоне о ранении капитана")
```

```
            elif liara_quest ==5:
                text _("Отправиться на ближайшую планету за лекарствами")
            elif liara_quest ==6:
                text _("Принести Ксеоне медикаменты")
            elif liara_quest ==7:
                text _("Помочь ей с лечением капитана")
            elif liara_quest ==8:
                text _("Отправиться с капитаном к Кайдену")
            elif liara_quest ==9:
                text _("Починить с Кайденом и капитаном двигатель")
            elif liara_quest ==10:
                text _("Сообщить Лиаре о результатах")

        hbox: # Контейнер с квестами Кайдена

    # Квесты остальных персонажей создаются аналогично
```

После того как стадия квеста Лиары была переключена на 1, в журнале выполнились условия для отображения первой задачи "Поговорить с Лиарой". Теперь игроку доступен соответствующий диалог. Сам диалог может запускаться, если стадия квеста находится на 1.

```
label start_3:
    if liara_quest ==1:
        gg "Привет, я хотел кое-что выяснить"
        lia "Да, конечно, спрашивай"
        $ liara_quest = 2 # Переключили стадию
        $ renpy.notify(_("Журнал заданий обновлен"))
        jump map_kosmolet
    else:
        # Какие-то другие события, в зависимости от стадии квеста
```

После того как игрок поговорит с Лиарой и она отправит его к Кайдену, переменная квеста переключается на следующую стадию ($ liara_quest = 2). В журнале перестает выполняться условие для первой подсказки, но доступна вторая.

Вместе с тем, если *liara_quest* равен двум (if liara_quest ==2:), то у Кайдена становится доступен соответствующий диалог, в конце которого переменная снова переключается на следующую стадию ($ liara_quest = 3). И так до конца квестовой цепочки, которая может состоять хоть из сотни последовательных событий.

В нашем случае после десятой стадии мы переключаем переменную на 11 ($ liara_quest = 11), в результате из списка заданий пропадает ее имя и текущие задачи, так как все события с ней завершены. В разделе завершенных квестов можно добавить соответствующее сообщение.

```
elif vkladka==3:
    vbox: # Вертикальный бокс со всеми квестами
        xysize(850,400)
        xalign 0.5
        hbox:
            if liara_quest >10: # Отображается, если квест >10
                text _("Лиара:") color "#c8ffc8"
                text _("Лиара благодарна мне за помощь")
```

Рис. 13.15.

Если в будущем нам потребуется продлить ее сюжетную линию, то нам остается просто продолжить список задач в журнале (elif liara_quest ==11:) и прописать новый квест, который будет доступен, если переменная равна 11 (if liara_quest == 11:).

Аналогичные линии создаются для каждого героя. Все они доступны параллельно, но по необходимости могут переплетаться. Например, определенные события могут произойти, только если игрок завершил несколько заданий с Лиарой и Кайденом.

```
label part_54:
    if liara_quest >4 and kayden_quest >7:
        "Здесь происходит событие, если оба условия выполнены"
    else:
        "Здесь может ничего не происходить"
```

Преимущество такого варианта журнала над списком или словарем, в который мы добавляем и удаляем квесты методами **append/remove**, в том, что весь лог задач находится в одном месте, его легко редактировать и ориентироваться, когда какая стадия квеста начнется и закончится.

В отличие от списка или словаря, где задачи добавляются и удаляются непосредственно в том месте скрипта, где переключается стадия. То есть все логи и описания квеста в таком случае получаются разбросанными по всему коду. Это неудобно, если по ходу разработки приходится редактировать какое-то ответвление.

13.7. Приложение "Календарь" — смена времени суток и времен года

Календарь представляет собой взаимосвязанное переключение нескольких переменных, которые отвечают за минуты, часы, дни, месяцы и годы.

В зависимости от этих значений можно запускать определенные события и подвязывать квесты к определенному времени суток, дню или числу месяца.

Также, в зависимости от потребностей проекта, календарь можно сделать приближенным к реальному или упрощенным, где у нас может быть всего четыре времени суток и семь дней недели. В данном приложении с этой механикой мы рассмотрим подробный вариант, но, поняв логику, его можно упростить по необходимости или желанию.

Рис. 13.16.

Вначале создадим необходимые переменные:

```
init python:
    minutes = 420 # Время по умолчанию, указываем в минутах (7:00)
    weekday = 1 # День недели по умолчанию - Понедельник
    month = 1 # Месяц по умолчанию - Январь
    date_month = 1 # Число месяца по умолчанию
    year = 2024 # Год по умолчанию
    dayofyear = 1 # Номер дня года от 1 до 365
    yearlim = 366 # Количество дней в году - 365 для обычного и 366 для
високосного
    daylim = 31 # Максимум дней в месяце
    stringweekday = __("Понедельник") # Строчное значение, для замены числа
    stringmonth = __("Сентября") # Так же
    # __() — тег для локализации переменных на другие языки
```

Теперь в том же блоке **init python** создадим функцию, которая будет заниматься перерасчетом времени, в зависимости от изменившихся значений.

```
init python:
    def calendar(): # Функция расчета времени и даты
        if (hasattr(store, 'minutes')): # Рассчитываем минуты, в сутках 1440
минут
            if (store.minutes > 1440): # Если количество минут превышает
1440
                store.minutes = store.minutes - 1440 # Обнуляем минуты
                store.weekday = store.weekday + 1 # Прибавляем день недели
                store.date_month = store.date_month + 1 # Прибавляем число
месяца
                store.dayofyear = dayofyear + 1 # Прибавляем один день года

        if (hasattr(store, 'weekday')): # Рассчитываем дни недели
            if store.weekday > 7: # Если превысили 7, вычитаем 7
                store.weekday = store.weekday - 7 # (8-7)
            if store.weekday == 1: # Если день недели равен 1
                store.stringweekday = __("Понедельник") # Показываем
понедельник
            elif store.weekday == 2:
                store.stringweekday = __("Вторник")
            elif store.weekday == 3:
                store.stringweekday = __("Среда")
            elif store.weekday == 4:
                store.stringweekday = __("Четверг")
            elif store.weekday == 5:
                store.stringweekday = __("Пятница")
            elif store.weekday == 6:
                store.stringweekday = __("Суббота")
            elif store.weekday == 7:
                store.stringweekday = __("Воскресенье")

        if (hasattr(store, 'date_month')):# Рассчитываем месяцы
            if store.date_month > store.daylim: # Если число месяца
превысило лимит
                store.date_month = store.date_month - store.daylim #
Вычитаем
                store.month = store.month + 1 # Прибавляем один месяц

        if month > 12: # Если значение месяцев превысило 12
            store.month = 1 # Устанавливаем месяц на 1 (январь)

        if (hasattr(store, 'month')): # Рассчитываем числа месяцев в
зависимости от лимитов
            if store.month == 1: # Если значение месяца равно 1
                store.stringmonth = __("Января") # Показываем строку
"Январь"
                store.daylim = 31 # Устанавливаем максимум дней в месяце 31
            if store.month == 2:
```

```
                store.stringmonth = __("Февраля")
                store.daylim = 29 # Для високосного
            if store.month == 3:
                store.stringmonth = __("Марта")
                store.daylim = 31
            if store.month == 4:
                store.stringmonth = __("Апреля")
                store.daylim = 30
            if store.month == 5:
                store.stringmonth = __("Мая")
                store.daylim = 31
            if store.month == 6:
                store.stringmonth = __("Июня")
                store.daylim = 30
            if store.month == 7:
                store.stringmonth = __("Июля")
                store.daylim = 31
            if store.month == 8:
                store.stringmonth = __("Августа")
                store.daylim = 31
            if store.month == 9:
                store.stringmonth = __("Сентября")
                store.daylim = 30
            if store.month == 10:
                store.stringmonth = __("Октября")
                store.daylim = 31
            if store.month == 11:
                store.stringmonth = __("Ноября")
                store.daylim = 30
            if store.month == 12:
                store.stringmonth = __("Декабря")
                store.daylim = 31

            if (hasattr(store, 'dayofyear') and hasattr(store, 'yearlim')):
# Годы
                if store.dayofyear > store.yearlim: # Если значение дней года
превысило лимит (365 или 366)
                    store.dayofyear = store.dayofyear - store.yearlim #
Вычитаем
                    store.year = store.year + 1 # Прибавляем один год к
счетчику лет
                if ((((int(store.year) / 4)*4) - store.year) == 0):
                    store.yearlim = 366 # Каждый четвертый год является
високосным
                else:
                    store.yearlim = 365
    # Параметр для перерасчета функции, если значения изменились
    config.python_callbacks.append(calendar)
```

Функция **hasattr()** в Python используется для проверки наличия атрибута у объекта.

В контексте строки **if (hasattr(store, 'minutes'))** она проверяет, существует ли атрибут с именем 'minutes' в объекте **store**.

Слово "store" в переменной store.minutes в Ren'Py означает, что minutes является переменной, хранящей значение, доступное для чтения и записи в рамках хранилища (store) игры.

В Ren'Py **store** представляет собой механизм для хранения и управления данными, которые могут использоваться в игре.

Таким образом, во время прохождения при изменении времени или дат будет происходить перерасчет и появятся соответствующие значения.

Далее мы создадим функцию для отображения нашего календаря на экране. Ее можно расположить все в том же блоке.

```
init python:
    def dat_time(): # Функция для вывода календаря на экран
        ui.text("%d:0%d, %s - %d %s %d" % (int(minutes/60), (minutes -
(int(minutes/60))*60), stringweekday, date_month, stringmonth, year))
```

В данной функции мы используем заполнители **%d** и **%s**, которые вместо себя подставляют значения из переменных, указанных далее в скобках (minutes, stringweekday, date_month, stringmonth, year). Заполнитель **%d** применяется в форматировании целых чисел. Заполнитель %s используется для подстановки строк (Понедельник, Январь).

Обратите внимание, что заполнители и переменные прописываются в одинаковой последовательности, где первый **%d** соответствует первой переменной в скобках и так далее.

Метод **int(minutes/60)** приводит к целому числу значение переменной, которая делится на 60, вычисляя, какой сейчас час. То есть, если сейчас время установлено на 420 минут ($ minutes=420), происходит расчет часов int(420/60), в результате чего отображается время 7 часов, являющееся целым числом.

Второй перерасчет происходит аналогичным образом для минут, чтобы они вместо 420 показывали значение в пределах 60 минут. В итоге на экране мы видим 7:00 и далее остальные значения календаря.

Чтобы отобразить время и дату на экране, нам остается вывести функцию **dat_time**, например, в экране интерфейса. Там же, где у нас кнопки инвентаря, карты и пр.

```
screen interface():
    frame:
        background "#00000070" # Полупрозрачный фон для фрейма
        align (0.5,0.05) # Расположение фрейма на экране
        $ dat_time() # Функция для вывода календаря
        # Далее располагаются кнопки других элементов интерфейса
```

Теперь в любом месте игры по необходимости мы можем изменять время и дату.

Например, после определенных действий игрока или небольшого диалога персонажей может пройти какое-то время.

```
label start:
    scene vapiuri # Задний фон
    show screen interface with dissolve # Экран интерфейса
    show liara serious at lcenter # Показали персонажей
    show kayden normal at rcenter
    lia "Сколько сейчас времени?"
    $ minutes +=60 # Добавили час
    kay "Пока мы с тобой разговаривали, прошел час"
    lia "Какое сегодня число?"
    kay "Сегодня [stringweekday], [date_month] [stringmonth], [year] год"
    $ date_month +1 # Добавили день
    $ month +=1 # Добавили месяц
    $ year +=1 # Добавили год
    jump start
```

Рис. 13.17.

Также, в зависимости от времени, могут происходить разные события или отображаться разные сцены:

```
label part_314:
    if weekday >5: # Если выходные (суббота или воскресенье)
        show liara # Показали Лиару
        lia "Сегодня я дежурю на мостике"
    else: # В остальные дни дежурит капитан
        show captain
        cap "Какая-то реплика"
```

В зависимости от времени суток можно менять освещение локаций, имитируя тем самым утро, день, вечер или ночь. Для этого с помощью виртуальных изображений создадим фильтры, которые будут накладываться поверх сцен.

```
image morning_evening:
    "#ffa50050" # Полупрозрачный оранжевый
image night:
    "#00008b60" # Полупрозрачный синий
```

Далее в экране интерфейса, в самом низу, после всех кнопок и изображений, напишем условия, при которых один из фильтров будет накладываться поверх всего остального, если наступит соответствующее время.

```
screen interface():
    # Если время больше 6 утра и меньше 10 утра
    if minutes >= 360 and minutes <=600:
        add "morning_evening" # Добавили оранжевый фильтр
    # Если время меньше 6 или больше 19:00
    elif minutes < 360 or minutes >= 1140:
        add "night" # Добавили синий фильтр
    # В остальное время будет обычное изображение
```

Рис. 13.18.

Вместе с тем накладывать фильтры можно непосредственно в лейблах, если это необходимо только в определенные моменты игры.

```
label start:
    scene vapiuri # Задний фон
    show morning_evening # Наложили оранжевый фильтр
    show liara serious at lcenter # Показали персонажей
    show kayden normal at rcenter
    lia "Сколько сейчас времени?"
```

В данном примере персонажи выведены после наложения фильтра, поэтому они не подвергнутся цветокоррекции.

13.8. Приложение "Мобильный телефон"

Еще одна востребованная механика на Ren'Py — мобильный телефон. Из примеров с планшетом в предыдущих разделах уже можно понять, как его реализовать. Для создания приложения с данной механикой давайте просто соберем все воедино.

Для начала оставим в интерфейсе только одну иконку, которая будет открывать телефон на весь экран.

```
screen interface(): # Интерфейс только с одной кнопкой
    imagebutton: # Открываем телефон на весь экран
        action [Hide("say"), Show("desktop")]
        align(0.01,0.01)
        idle "images/infoi.png"
        hover "images/infoh.png"
        focus_mask True
        tooltip _("Телефон")
```

Данная кнопка открывает главный экран телефона, на котором расположены все остальные кнопки-иконки. Каждая из них будет открывать соответствующий экран, из тех, что мы делали ранее: инвентарь, онлайн-магазин, журнал заданий, календарь и пр.

Рис. 13.19.

```
screen desktop():
    modal True
    tag phone # Все экраны телефона с этим тегом
    frame: # Дата и время
        background "#00000070"
        align (0.5,0.15)
        $ dat_time()
    hbox: # Контейнер разложит иконки по порядку
        xsize 550
        pos (670,220)
        box_wrap True
        box_wrap_spacing 50
        imagebutton: # 1-я иконка
            action Show("contacts")
            idle "images/icon_1.png"
            hover "images/icon_1h.png"
            tooltip _("Контакты")
        imagebutton: # 2-я иконка
            action Show("gallery")
            idle "images/icon_2.png"
            hover "images/icon_2h.png"
            tooltip _("Галерея")
        # Далее остальные кнопки по такому же принципу
```

Напомню, что объединение всех экранов телефона под общим тегом **tag phone** позволит автоматически закрывать предыдущий экран при открытии нового. Пропишите этот тег каждому из ранее созданных экранов и всем остальным, которые впоследствии станут частью телефона.

```
screen quest_log():
    tag phone

screen inventory():
    tag phone

screen online_store():
    tag phone
```

Таким образом, можно разместить большое количество кнопок и надписей в телефоне или планшете, чтобы все эти элементы интерфейса не захламляли половину экрана.

13.9. Галерея изображений или видеороликов

Во многих визуальных новеллах присутствует галерея изображений с памятными моментами из игры или различными концовками, повторы которых можно пересмотреть в любой момент.

Ранее мы уже рассматривали похожий функционал, когда настраивали **главное меню игры** (раздел 5.12). Кроме того, галерею можно сделать по тому же принципу, что и предыдущие экраны телефона. Приведенный ниже пример можно без труда встроить в главное меню или в любое другое удобное место.

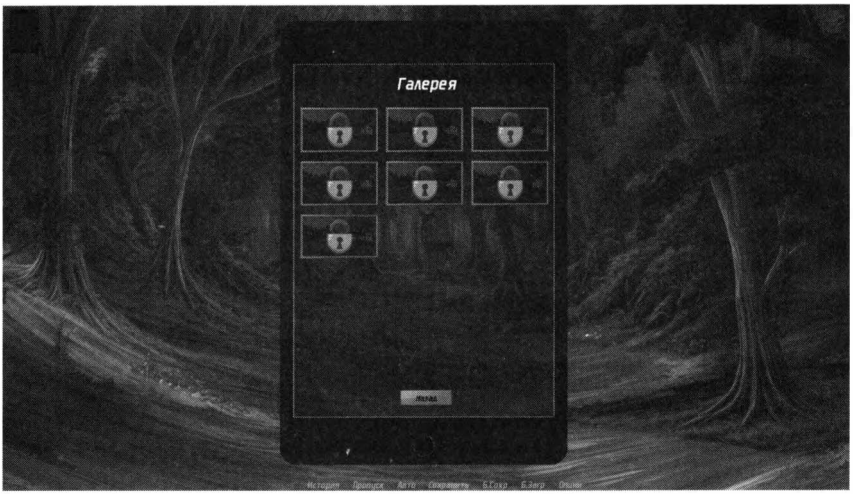


Рис. 13.20.

Для начала нам предстоит определиться с изображениями и их количеством. Каждому из них потребуется уменьшенная копия в виде превью, при нажатии на которое будет показан скриншот на весь экран. Также нам понадобится одна картинка-заглушка размером с превью. Оно будет отображаться, если изображение заблокировано.

Рис. 13.21.

Следующим шагом создадим постоянные переменные, которые будут отвечать за блокировку изображений. Напомню, что постоянные переменные сохраняют свои значения даже с началом новой игры. То есть при повторном прохождении в галерее игрока будут доступны изображения, открытые в предыдущих прохождениях. Если это не нужно, создайте простые переменные, которые будут сбрасывать свое значение при старте новой игры. Галерея, соответственно, будет снова заблокирована.

```
# Постоянные переменные для разблокировки изображений
default persistent.image_g1 = False
default persistent.image_g2 = False
default persistent.image_g3 = False
default persistent.image_g4 = False
default persistent.image_g5 = False
default persistent.image_g6 = False
default persistent.image_g7 = False
# Переменная-переключатель, хранит в себе одно из изображений галереи
default gal_image = "images/gallery/image_g1.jpg" # 1-е по умолчанию
```

Теперь создадим **screen**, который будет открывать изображения на весь экран. Он будет общим для всех изображений и представлять собой полноэкранную кнопку, при нажатии на которую игрока будет возвращать во вкладку галереи с превью-изображениями.

```
screen gallery_image():
    modal True
    tag phone
    imagebutton:
        action Show("gallery")
        align(0.5,0.5) # Расположение по центру на весь экран
        idle gal_image # Переменная хранит изображение
```

Изображение (idle) для вышеописанной кнопки будет подставляться из переменной **gal_image**, в которой в нужный момент будет устанавливаться соответствующее изображение на весь экран.

Далее создадим экран с превью-изображениями:

```
screen gallery():
    modal True
    tag phone
    add "bg_inv" # Фоновая картинка телефона
    text _("Галерея") align(0.5,0.15)
    hbox: # Контейнер, который разложит превьюшки
        xsize 580
        pos (675,230)
        box_wrap True
        box_wrap_spacing 20
        # Кнопки разблокированы, если условие True
        if persistent.image_g1:
            imagebutton:
                idle "images/gallery/prev1.jpg"
                action [SetVariable("gal_image", "images/gallery/image_g1.jpg"),
                        Show("gallery_image"), With(dissolve)]
        else:
            add "images/gallery/prev_bl.jpg"
        if persistent.image_g2:
            imagebutton:
                idle "images/gallery/prev2.jpg"
                action [SetVariable("gal_image", "images/gallery/image_g2.jpg"),
                        Show("gallery_image"), With(dissolve)]
        else:
            add "images/gallery/prev_bl.jpg"
        if persistent.image_g3:
            imagebutton:
                idle "images/gallery/prev3.jpg"
                action [SetVariable("gal_image", "images/gallery/image_g3.jpg"),
                        Show("gallery_image"), With(dissolve)]
        else:
```

```
        add "images/gallery/prev_bl.jpg"
    if persistent.image_g4:
        imagebutton:
            idle "images/gallery/prev4.jpg"
            action [SetVariable("gal_image", "images/gallery/image_g4.jpg"),
                    Show("gallery_image"), With(dissolve)]
    else:
        add "images/gallery/prev_bl.jpg"
    if persistent.image_g5:
        imagebutton:
            idle "images/gallery/prev5.jpg"
            action [SetVariable("gal_image", "images/gallery/image_g5.jpg"),
                    Show("gallery_image"), With(dissolve)]
    else:
        add "images/gallery/prev_bl.jpg"
    if persistent.image_g6:
        imagebutton:
            idle "images/gallery/prev6.jpg"
            action [SetVariable("gal_image", "images/gallery/image_g6.jpg"),
                    Show("gallery_image"), With(dissolve)]
    else:
        add "images/gallery/prev_bl.jpg"
    if persistent.image_g7:
        imagebutton:
            idle "images/gallery/prev7.jpg"
            action [SetVariable("gal_image", "images/gallery/image_g7.jpg"),
                    Show("gallery_image"), With(dissolve)]
    else:
        add "images/gallery/prev_bl.jpg"
# Кнопка "Назад" возвращает на главный экран телефона
button:
    action Show("desktop")
    align(0.5,0.82)
    xsize 120 ysize 35 # Размер картинки
    idle_background "images/but_s.png"
    hover_background "images/but_h.png"
    text _("Назад")
```

В примере выше у каждой кнопки установлено условие. По умолчанию все переменные галереи имеют значения *False*, поэтому вместо кнопок отображаются картинки-заглушки. Во время прохождения, когда будет разблокировано какое-то из изображений, условие для кнопки будет изменено на *True*, и она станет доступна для просмотра.

При нажатии на кнопку происходят три действия:

1. **Show("gallery_image")** — открывает **screen** с кнопкой на весь экран. Так как он входит в группу **tag phone**, предыдущий экран закрывается.
2. **With(dissolve)** — использует эффект растворения для перехода между экранами.
3. **SetVariable("gal_image", "images/gallery/image_g1.jpg")** — изменяет переменную **gal_image**, сохраняя в ней изображение, соответствующее для этого превью. Напомню, что **gal_image** отвечает за изображение кнопки (idle) в этом экране. То есть при клике происходит автоматическая смена картинки, в зависимости от того, какая из превью была нажата.

При желании кнопки можно прописать в одну строку, чтобы сделать код более компактным, или выводить циклом, но для полного понимания пример разложен подробно.

Последний шаг — **разблокировка изображений**. Делается это в том месте кода, где это необходимо, простым переключением переменной с *False* на *True*. То есть после определенного памятного события или после выхода на определенную концовку.

```
label part_241:
    scene vapiuri with dissolve
    show kayden normal at center
    kay "Любопытное местечко"
    $ persistent.image_g2 = True # Разблокировали
    $ renpy.notify(_("Разблокировано изображение в галерее"))
    # И сообщили об этом
```

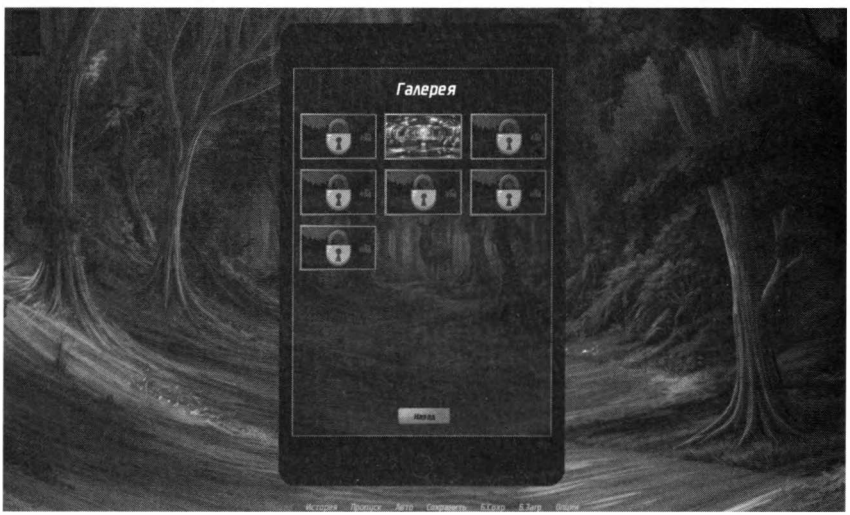


Рис. 13.22.

Таким же образом вместо изображений вы можете открывать повторы видеороликов или целых эпизодов игры, отправляя пользователя в нужные лейблы командой **jump**.

13.10. Многослойные изображения

Аналогично тому, как один экран автоматически заменяет другой, если они оба находятся в одной группе (например, **tag menu** или **tag phone**), можно группировать изображения, которые будут заменять друг друга. Ранее похожую группировку мы делали для спрайтов персонажей, привязывая их к имени и вызывая с дополнительным описанием (liara normal, liara smile и т.д.).

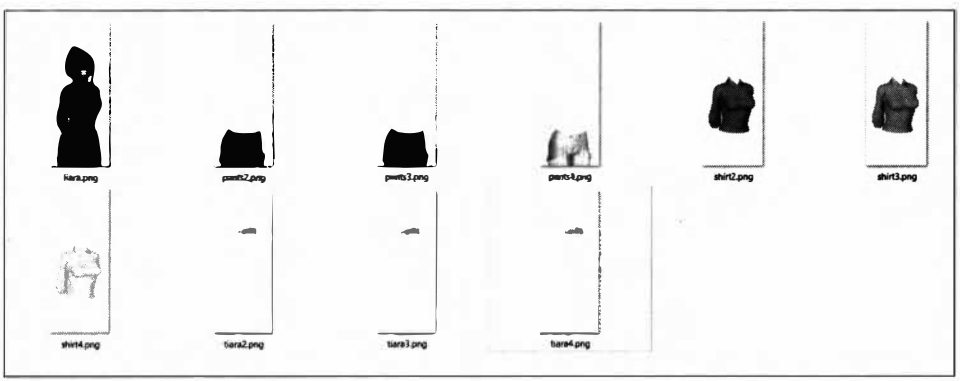


Рис. 13.23.

Вместе с тем каждый такой спрайт можно разбить на отдельные элементы, чтобы дополнительно разнообразить персонажей и уменьшить общее количество итоговых спрайтов в игре.

То есть, имея 10 спрайтов, как на скриншоте, мы можем получить 64 разных изображения, изменив один из элементов одежды. Вместо того чтобы полностью создавать все эти спрайты, на каждом из них будет изменен один элемент.

Связываются подобные многослойные спрайты конструкцией **layeredimage**, которая позволяет собирать их по дополнительным подгруппам.

```
layeredimage liara: # Сборный спрайт с названием liara
    always: # Базовый спрайт, отображается всегда
        "liara" # Название спрайта
    group pants: # Группирует спрайты по элементам (Штаны)
        # Определение изображений по названиям, цветам, фасону
        attribute khaki1 default: # default - элемент по умолчанию
            "pants1" # Изображение спрайта серых штанов
        attribute khaki2: # Зеленые
            "pants2"
        attribute khaki3: # Фиолетовые
            "pants3"
        attribute khaki4: # Белые
            "pants4"
    group shirt: # Рубашки
        attribute shirt default: # Рубашка
            "shirt1"
        attribute blouse: # Блузка
            "shirt2"
        attribute uniform: # Униформа
            "shirt3"
        attribute white: # Белая
            "shirt4"
    group tiara: # Тиары
        attribute tiara default: # Тиара
            "tiara1"
        attribute headband: # Ободок
            "tiara2"
        attribute tiara2: # Тиара2
            "tiara3"
        attribute headband2: # Ободок2
            "tiara4"
```

В данном примере все спрайты относятся к персонажу *liara*. Далее они сортируются по группам: штаны, рубашки, тиары. Помимо этого, можно добавить любые другие группы, например обувь, перчатки, прически, головные уборы, очки и пр. Соответственно, придется подготовить спрайты всех новых элементов.

В каждую группу входят атрибуты, у нас это спрайты одежды, имеющие удобное название, по которому они будут вызываться. Все атрибуты одной группы взаимозаменяемы. То есть при вызове одной из рубашек она автоматически заменит предыдущую. Стоит иметь в виду, что новый спрайт не наложится сверху, а именно заменит старый.

Чтобы один спрайт наложился на другой, он должен быть помещен в отдельную группу или быть отдельным атрибутом, не входящим ни в одну из

групп. Это может быть полезно, например, для причесок и головных уборов, когда панамы будут накладываться поверх волос, а не заменять их.

Вместе с тем следует соблюдать порядок написания групп в **layeredimage**. Если сначала будет идти группа панам, а потом — причесок, то при выводе спрайтов сначала будет отображен головной убор, поверх которого появятся волосы. Поэтому список групп должен начинаться с нижних слоев: сначала волосы, потом панамы; сначала рубашки, потом куртки.

```
layeredimage liara:
    group shirt: # Рубашки
        # Атрибуты
    group jacket: # Верхняя одежда
        # Атрибуты

    group hair: # Волосы
        # Атрибуты
    group hats: # Головные уборы
        # Атрибуты
```

Инструкция **always** определяет, какой спрайт будет выводиться всегда. Обычно это спрайт тела персонажа, на который накладываются предметы одежды.

Инструкция **default** определяет одежду по умолчанию. То есть если при вызове персонажа ничего дополнительно не указано, то поверх тела будут наложены дефолтные вещи.

Чтобы добавить или изменить один из элементов сборного спрайта, при его отображении в игре нужно дополнительно указать название соответствующего атрибута.

```
label part_342:
    scene cabin with dissolve
    show liara khaki2 # Вызвали Лиару в зеленых штанах
    lia "Нет, эти не подходят"
    show liara khaki3 # Изменили на другие
    lia "Хм..."
    show liara blouse headband # Изменили рубашку и тиару
```

Рис. 13.24.

Для скрытия определенного элемента используется знак минус.

```
label part_343:
    scene cabin with dissolve
    show liara1 blouse headband # Изменили рубашку и тиару
    lia "Нет, пожалуй, сниму тиару"
    show liara1 -headband # Показали Лиару минус тиара
```

Вместе с тем спрайтам можно добавлять анимированные изображения. Например, возьмем анимацию моргания глаз и вставим ее вместо статичных. Предварительно не забудьте создать группу для глаз в многослойном спрайте.

```
image blink: # Анимированное изображение моргания глаз
    "liara_blink1" with dissolve # Закрывает глаза
    pause 0.1
    "liara_blink2" with dissolve
    pause 0.1
    "liara_blink3" with dissolve
    pause 0.1
    "liara_blink4" with dissolve # Закрыла
    pause 0.1
```

```
    "liara_blink3" with dissolve # Открывает глаза
    pause 0.1
    "liara_blink2" with dissolve
    pause 0.1
    "liara_blink1" with dissolve
    pause 0.1
    choice: #
        pause 9
    choice:
        pause 12
    choice:
        pause 15
    repeat # Повторяет анимацию
```

Добавляем в сборный спрайт Лиары новую группу для глаз и очков:

```
layeredimage liara:
    always:
        "liara"
    group eyes:
        attribute blink: # Анимированные глаза
            "blink" # Анимация, созданная выше
        attribute glasses: # Очки
            "glasses" # Изображение очков
```

В зависимости от отношения персонажа к главному герою, его выражение лица может изменяться. Например, при плохих значениях Лиара будет выглядеть агрессивно, если отношения на среднем уровне — спокойное лицо, если удастся с ней подружиться — будет улыбаться при встрече. Для этого необходимо добавить соответствующие условия в ее сборный спрайт.

Ранее мы создавали переменные отношений для отображения их на полосках баров. Теперь эти же переменные могут влиять на выражение лиц персонажей.

```
init python:
    liara_relat = 0
    xeona_relat = 10
    kayden_relat = 7
    capitan_relat = 29
```

Добавляем условия в сборный спрайт:

```
layeredimage liara:
    always:
        "liara"
    if liara_relat <20: # Если отношение Лиары меньше 20
        "frowning_face" # Хмурое лицо
    elif liara_relat >70: # Если отношения больше 70
        "happy_face" # Сияющее
    else: # В остальных случаях (между 20 и 70)
        "normal_face" # Нормальное
```

Теперь при вызове спрайта Лиары изображение ее лица будет меняться автоматически, в зависимости от ее отношения к главному герою.

В примерах выше мы изменяем внешний вид персонажа, когда нам это необходимо по сюжету. Но можно добавить функцию кастомизации (настройки различных параметров) героя, чтобы игрок сам настраивал внешний вид. Например, в начале игры или при подходе к шкафчику с одеждой.

Для этого нам потребуется создать экран с кнопками, по нажатии которых будут переключаться переменные, отвечающие за ту или иную часть одежды, прически, выражения лица и пр. А в сборном спрайте необходимо прописать условия, при которых будет отображаться определенная одежда.

```
# Переменные для переключения на кнопки
default liara_pants = 1 # Штаны
default liara_shirt = 1 # Рубашка
default liara_tiara = 1 # Тиара
```

А можно так:

```
# Сборный спрайт для переключения одежды на кнопки-стрелочки
layeredimage liara:
    always:
        "liara1"
    # Штаны
    if liara_pants == 1: # Если переменная равна 1
        "pants1" # Отображается этот элемент одежды
    elif liara_pants == 2:
        "pants2"
    elif liara_pants == 3:
        "pants3"
```

```
elif liara_pants == 4:
    "pants4"
# Рубашки
if liara_shirt == 1:
    "shirt1"
elif liara_shirt == 2:
    "shirt2"
elif liara_shirt == 3:
    "shirt3"
elif liara_shirt == 4:
    "shirt4"
# Тиары
if liara_tiara == 1:
    "tiara1"
elif liara_tiara == 2:
    "tiara2"
elif liara_tiara == 3:
    "tiara3"
elif liara_tiara == 4:
    "tiara4"
```

Экран для переключения будет выглядеть следующим образом:

Рис. 13.25.

Код следующий:

```
screen change_clothes():
    modal True
    textbutton "<<" align(0.37,0.3): # Кнопка переключения тиар влево
        if liara_tiara >1: # Если значение больше единицы
            action SetVariable("liara_tiara", liara_tiara-1)# Минусуем
        else: # Иначе устанавливаем на 4
            action SetVariable("liara_tiara", 4)
    textbutton ">>" align(0.63,0.3): # Тиара вправо
        if liara_tiara <4: # Аналогичная кнопка в другую сторону
            action SetVariable("liara_tiara", liara_tiara+1)
        else:
            action SetVariable("liara_tiara", 1)
    textbutton "<<" align(0.37,0.55): # Рубашка влево
        if liara_shirt >1:
            action SetVariable("liara_shirt", liara_shirt-1)
        else:
            action SetVariable("liara_shirt", 4)
    textbutton ">>" align(0.63,0.55): # Рубашка вправо
        if liara_shirt <4:
            action SetVariable("liara_shirt", liara_shirt+1)
        else:
            action SetVariable("liara_shirt", 1)
    textbutton "<<" align(0.37,0.85): # Штаны влево
        if liara_pants >1:
            action SetVariable("liara_pants", liara_pants-1)
        else:
            action SetVariable("liara_pants", 4)
    textbutton ">>" align(0.63,0.85): # Штаны вправо
        if liara_pants <4:
            action SetVariable("liara_pants", liara_pants+1)
        else:
            action SetVariable("liara_pants", 1)
    button: # Закрыть гардероб
        action Hide("change_clothes")
        align(0.99,0.01)
        xsize 120 ysize 35 # Размер картинки
        idle_background "images/but_s.png"
        hover_background "images/but_h.png"
        text _("Уйти")
```

Подобный экран можно выводить на старте игры или сделать специальную кнопку в нужных локациях, при нажатии на которую будет открываться возможность переключения одежды.

По тому же принципу можно сделать апгрейд космического корабля, оружия или любого другого игрового элемента. После прокачивания (увеличения различных параметров) и улучшения отдельных деталей они будут выглядеть по-другому, а игрок сможет получать дополнительные бонусы.

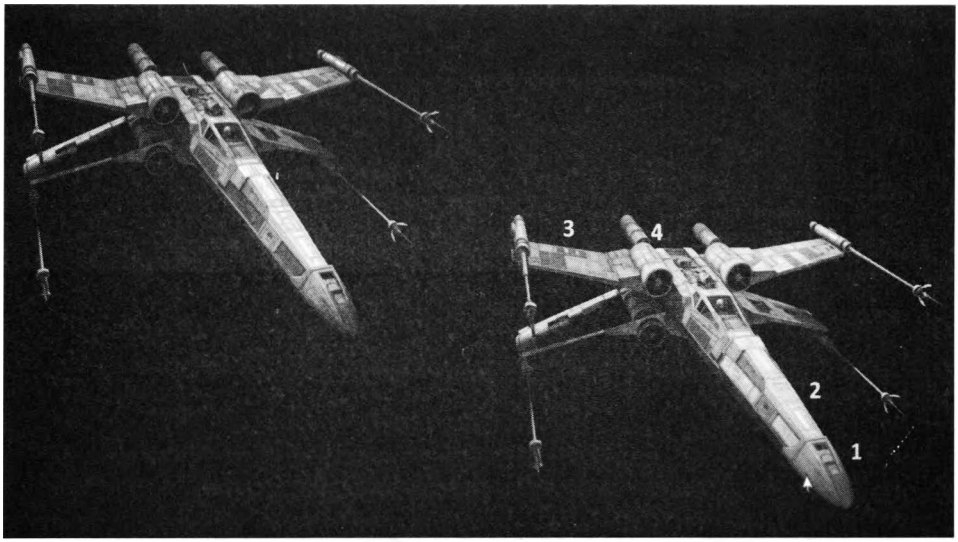


Рис. 13.26.

```
# Переменные для прокачки корабля
default wing_kosmolet = 0
default turbine_kosmolet = 0
default engine_kosmolet = 0
default guns_kosmolet = 0
layeredimage kosmolet: # Пример сборного спрайта корабля
    always:
        "kosmolet1"
    if wing_kosmolet == 1:
        "wing1"
    elif wing_kosmolet == 2:
        "wing2"
    elif wing_kosmolet == 3:
        "wing3"
    elif wing_kosmolet == 4:
        "wing3"
    elif wing_kosmolet == 5:
        "wing3"
    # Далее аналогичный код для других деталей
```

```
    if turbine_kosmolet == 1:
        "turbine1"
    if engine_kosmolet == 1:
        "engine1"
    if turbine_kosmolet == 1:
        "guns1"
```

Таким образом, можно сделать многоуровневую прокачку, где при увеличении левела (уровня) детали могут менять не только цвет, но и форму, превращая космолет в абсолютно другую модель. Каждый прокачанный уровень может добавлять бонус в процентном соотношении или на фиксированное число, в зависимости от необходимого баланса.

Вместе с тем некоторые элементы можно установить в зависимости от навыков игрока и экипажа. Например, если навык ремонта двигателя у Кайдена ниже определенного уровня, то мощный движок установить не получится. Если навык навигации Лиары недостаточен, орудия могут наносить меньше урона и т.д.

```
label part_423:
    if kayden_engin >5: # Если навык Кайдена больше 5
        $ engine_kosmolet = 3 # Апгрейд двигателя
        $ renpy.notify(_("Двигатель улучшен до третьего уровня"))
    else:
        $ renpy.notify(_("Кайдену не хватает навыков"))
```

Скорость корабля может рассчитываться в зависимости от установленных элементов и навыка Лиары, которые могут суммироваться или перемножаться, а затем из этого числа может вычитаться общий вес груза в грузовом отсеке.

```
label part_433: # Расчет скорости в зависимости от деталей, навыков
и груза
    $ speed_ship = (wing_kosmolet + turbine_kosmolet + engine_
kosmolet + liara_navation) - ship_weight
```

Расчет повреждений корабля от атаки противника может также рассчитываться по формуле, где уровень прочности будет зависеть от нескольких факторов. Пример простой боевой системы был приведен в разделе 10.2. Ее можно адаптировать под новые условия, заменив фон локации на изображение космоса, а персонажей — на спрайты звездолетов.

Сила атаки противников может зависеть от установленных деталей, прочности корабля и мощности орудий соперника.

```
default ship_hp = 500 # Прочность корабля
default ships_armor = 100 # Броня корабля
default enemys_weapon = 200 # Сила оружия противника

label enemy_attack:
    $ ship_hp = ship_hp - (enemys_weapon - ships_armor)
```

В приведенном примере "здоровье" корабля рассчитывается по формуле, где в скобках от силы атаки вычитается уровень брони, а затем итоговое значение уменьшает текущую прочность корабля.

После боя, в зависимости от полученного урона, уровень определенных деталей можно уменьшить, имитируя этим повреждения звездолета.

```
label end_figth:
    if ship_hp < 100: # Если уровень прочности корабля меньше 100
        $ wing_kosmolet -= 1 # Уровень деталей -1
        $ turbine_kosmolet -= 1
        $ engine_kosmolet -= 1
        $ guns_kosmolet -=1
```

Как видим, благодаря механике **layeredimage**, простую визуальную новеллу можно превратить в гибрид из нескольких жанров.

Например, игры — симуляторы переодеваний популярны на мобильных платформах. Более того, развивая тему многослойности и прокачки, можно создать стратегию с развитием базы или поселения. С возведением стен, различных построек, добычей ресурсов и фермерством.

Каждый объект на карте является сборным спрайтом, который может быть улучшен. Для апгрейда любого строения могут требоваться один или несколько ресурсов, который предстоит добыть в ближайших рудниках. Рудники, в свою очередь, могут истощаться, а их многослойные спрайты отображаться в обратной последовательности. Где сначала показывается спрайт полного рудника, а с уменьшением ресурсов в нем изображения сменяются на спрайты полупустого и пустого рудников.

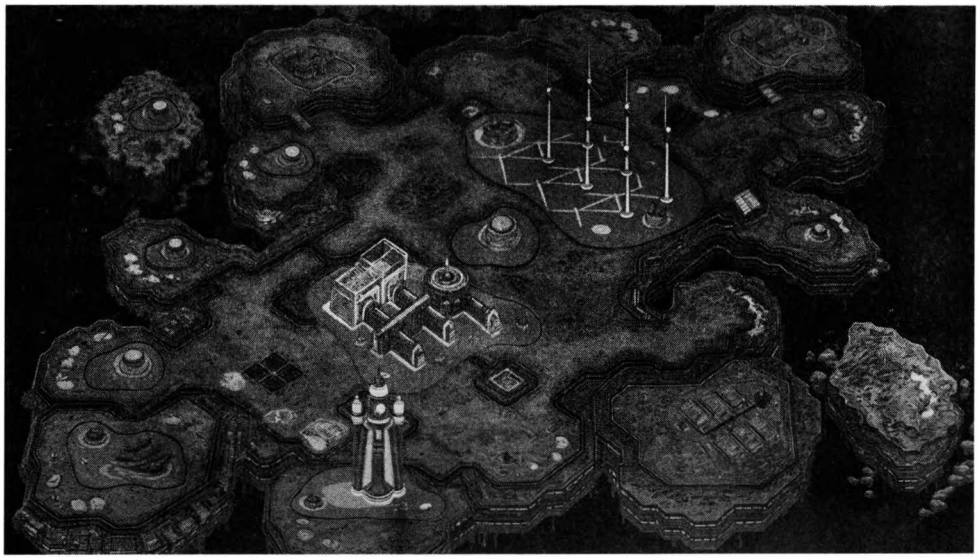


Рис. 13.27.

После улучшения постройка может давать дополнительное жилье для поселенцев, производить дополнительный ресурс, вырабатывать электроэнергию и т.д. Вокруг поселения можно возводить укрепления, чтобы защититься от инопланетной фауны. Которая, в свою очередь, может постепенно разрушать постройки и охотиться на поселенцев. Дальнейшее развитие механик зависит только от нашей фантазии.

13.11. Перетаскивание объектов (Drag and Drop)

Аналогично тому, как мы раскладывали в hbox-контейнере объекты инвентаря, можно раскладывать объекты, которые игрок способен перемещать самостоятельно, с помощью курсора мыши.

Для этого используется система перетаскивания, включающая в себя контейнер **draggroup**, внутри которого происходит перемещение, и объекты **drag**, которые можно перетаскивать.

Рис. 13.28.

```
screen say(who, what): # Перемещаемое диалоговое окно
    draggroup: # Контейнер на весь экран
        drag: # Элемент для перемещения
            drag_name "say" # Имя объекта
            yalign 1.0 # Стартовая позиция
            xalign 0.5
            window id "window": # Диалоговое окно
                xmaximum 1000
                has vbox
                if who:
                    text who id "who"
                text what id "what"
```

Объекты **drag** могут иметь несколько параметров, в зависимости от которых выполняются разные действия:

- **child** — является дочерним элементом, находящимся внутри объекта drag.
- **drag_name** — устанавливает название для объекта.

- **draggable** — если значение *True*, позволяет перетаскивать объект внутри контейнера. По умолчанию включен, поэтому используется в значении *False*, чтобы запретить перемещать элемент.
- **droppable** — если значение *True*, на объект можно помещать другие перетаскиваемые объекты.
- **drag_raise** — если значение *True*, этот элемент при перетаскивании отобразится на верхнем слое. Это полезно, когда несколько перетаскиваемых объектов лежат близко друг к другу или друг на друге.
- **dragged** — во время перетаскивания объекта вызывает функцию с двумя параметрами, включающими в себя список имеющихся перемещаемых элементов в draggroup и объект наложения, при отсутствии которого устанавливается значение *None*.
- **dropped** — запускает функцию при наложении перетаскиваемого объекта на droppable-объект. Она также имеет два параметра, первый из которых хранит значение перенесенного объекта, а второй — список оставшихся.
- **clicked** — запускает функцию, если на объект нажали, но не перетащили.
- **alternate** — действие, которое выполняется при нажатии на перемещаемый объект правой кнопкой мыши (на ПК) или длительном нажатии без перемещения (на мобильных устройствах).
- **hovered** — событие, которое произойдет при наведении курсора на объект.
- **unhovered** — событие при снятии курсора с объекта.
- **tooltip** — функция отображения всплывающей подсказки, как при наведении на кнопки.
- **drag_handle** — определяет область на объекте, за которую его можно перемещать. Должен иметь четыре значения: **x**, **y** — позиции верхнего левого угла перемещаемой области, а также **ширину** и **высоту**. Пример: drag_handle (0,0,50,50). Если параметр не указывать, объект перемещается за любую область.
- **drag_offscreen** — если значение *True*, объект можно переместить за пределы контейнера. Значение "horizontal" позволяет выносить за пределы по горизонтали. Значение "vertical" — за пределы по вертикали. Стоит иметь в виду, что, если элемент окажется за пределами экрана, его будет невозможно достать. В итоге событие с перетаскиванием не будет

завершено, и возможность дальнейшего прохождения может оказаться под вопросом.

- **mouse_drop** — если значение *True*, к курсору при нажатии цепляется ближайший элемент. Если *False* — цепляется элемент с наибольшим перекрытием пикселей.
- **snap (x, y, delay=0, warper=None)** — настраивает анимацию перемещения от плавного к моментальному.

Как все это использовать, разберем на примере из документации, адаптированном под наши нужды. Игроку предлагается переместить рабочего на один из рудников для добычи ресурса.

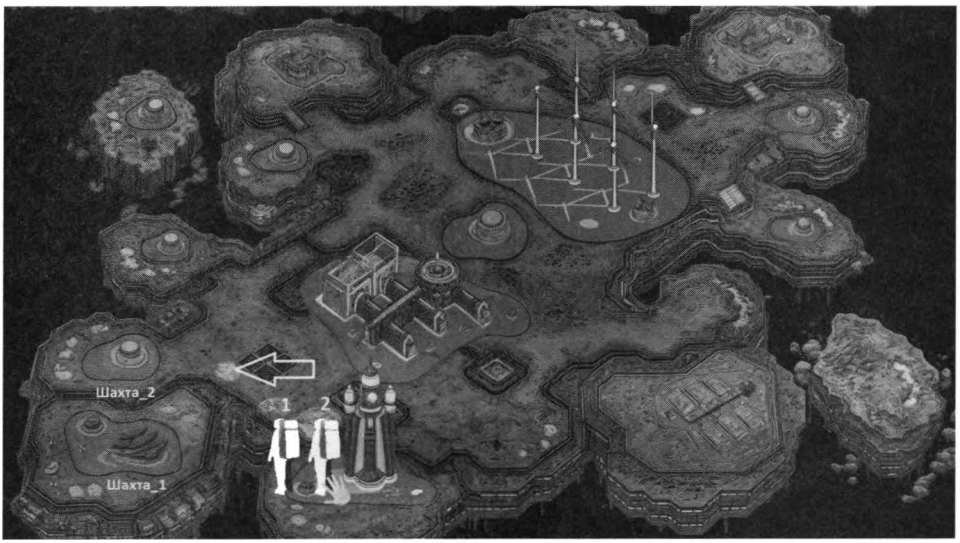


Рис. 13.29.

Для начала создадим несколько переменных и функцию, которая будет запускать события при определенном действии.

```
init python:
    mineral_1 = 1000 # Количество минерала в первой шахте
    mineral_2 = 1000 # Количество минерала во второй шахте
    extracted_mineral = 0 # Количество добытого минерала
    def workers_mines(drags, drop): # Функция при перемещении
        if not drop: # Проверяет, ложно ли условие
            return # Возвращает игрока в место после вызова экрана
```

```
        # Переменная получает имя перетаскиваемого объекта
        store.workers = drags[0].drag_name
        # Переменная сохраняет имя перенесенного к ней объекта
        store.mine = drop.drag_name
        return True # Получаем значение True, действие функции прекращается
```

Далее нам потребуется экран, в котором будут отображены контейнер и перетаскиваемые элементы.

```
screen asteroid_base():
    add "base1327" # Фоновое изображение базы на весь экран
    imagebutton at aster_fly: # Летающий метеорит
        idle "aster7.png"
        action NullAction
    imagebutton at wild_animals: # Блуждающая стая тонаторов
        idle "tonators.png"
        action NullAction
    draggroup: # Контейнер со всеми объектами
        align(.5,.5) # Позиция контейнера
        xysize(1920,1080) # Размер контейнера
        # Объекты, представляющие собой рабочих
        drag:
            drag_name _("Кайден") # Имя объекта
            droppable False # Наложение на объект не вызывает действий
            dragged workers_mines # Запуск функции при перетаскивании
            pos(515,835) # Стартовая позиция объекта
            add "images/base/kosm1.png" # Спрайт персонажа
        drag:
            drag_name _("Лиара")
            droppable False
            dragged workers_mines
            pos(600,835)
            add "images/base/kosm2.png"
        # Объекты, представляющие собой рудники
        drag:
            drag_name _("Шахта_1")
            draggable False # Объект нельзя перетаскивать
            pos(130,785)
            add "images/base/mine1.png"
        drag:
            drag_name _("Шахта_2")
            draggable False
            pos(160,645)
            add "images/base/mine2.png"
```

Обратите внимание, что объекты, представляющие собой шахты, нельзя перетаскивать, так как они являются статичными элементами. С ними происходит взаимодействие, когда персонаж перетаскивается на один из них.

В свою очередь, персонажи имеют параметр *droppable False*, не позволяющий взаимодействие при наложении одного на другой. Подобное взаимодействие возможно только при наложении на шахту, так как у них этот параметр по умолчанию *True*.

При наложении запускается функция **workers_mines**, описанная выше, она завершает мини-игру, если один из персонажей отпускается на объект в виде шахты.

```
label part_555: # Начало мини-игры
    scene base1327
    "Отправьте кого-то из экипажа разрабатывать рудник"
    call screen asteroid_base with dissolve # Открыли экран игры

    # Следующий код срабатывает после завершения игры
    "[workers] разрабатывает [mine]" # Имя персонажа и название шахты
    "Время выработки: 2 часа"
    if mine == "Шахта_1": # Если выбрана первая шахта
        $ mineral_1 -= 100 # Уменьшили количество минерала в шахте_1
    else: # Иначе
        $ mineral_2 -= 100 # Уменьшили количество минерала в шахте_2
    $ extracted_mineral += 100 # Добавили добытый минерал на склад
    $ minutes += 120 # Перемотали время на два часа вперед
    $ renpy.notify(_("Добыто 100 минерала"))
    jump star_map # Отправляем игрока дальше по сюжету или на карту
```

В примере экрана с мини-игрой присутствуют кнопки с использованием трансформаций, которые перемещают их по экрану в случайном направлении. Как это сделать, было описано в разделе 7.3.

При нажатии на них можно запускать функцию или вызывать новый экран с мини-игрой по отстрелу диких животных (раздел 7.4).

Еще одним популярным примером использования Drag and Drop является мини-игра "Пазлы".

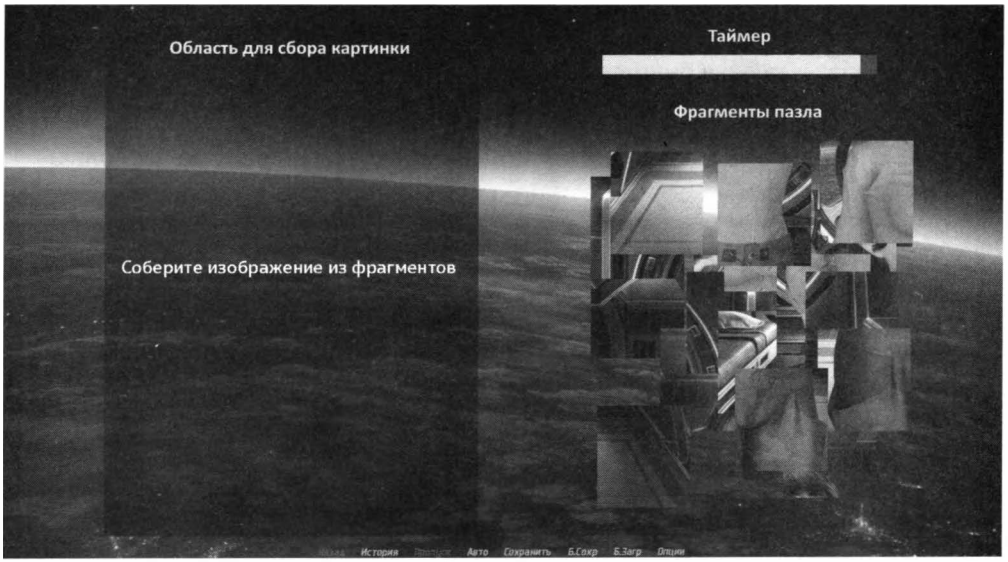


Рис. 13.30.

По традиции, сначала все необходимые переменные:

```
default vremya = 120 # Количество секунд для сбора пазла
default puzzle_complete = False # True, если собрали
default total_fragments = 20 # Общее количество фрагментов
default fragments_complete = 0 # Количество установленных фрагментов
# Список стартовых позиций фрагментов (пустой, будет сгенерирован)
default start_list_positions = []
default final_list_positions = [(204,46), (382,46), (560,46),
                                (738,46), (204,245), (382,245),
                                (560,245), (738,245), (204,444),
                                (382,444), (560,444), (738,444),
                                (204,643), (382,643), (560,643),
                                (738,643), (204,842), (382,842),
                                (560,842), (738,842)]
```

Последняя переменная **final_list_positions** хранит финальный список позиций, на которые необходимо установить каждый из фрагментов.

Все эти позиции нужно смотреть в Фотошопе или другом графическом редакторе, в котором целое изображение делится на фрагменты. Аналогично тому, как мы смотрели позиции для кнопок на звездной карте.

Каждый фрагмент должен иметь одинаковое название и порядковый номер, для удобного использования в цикле.

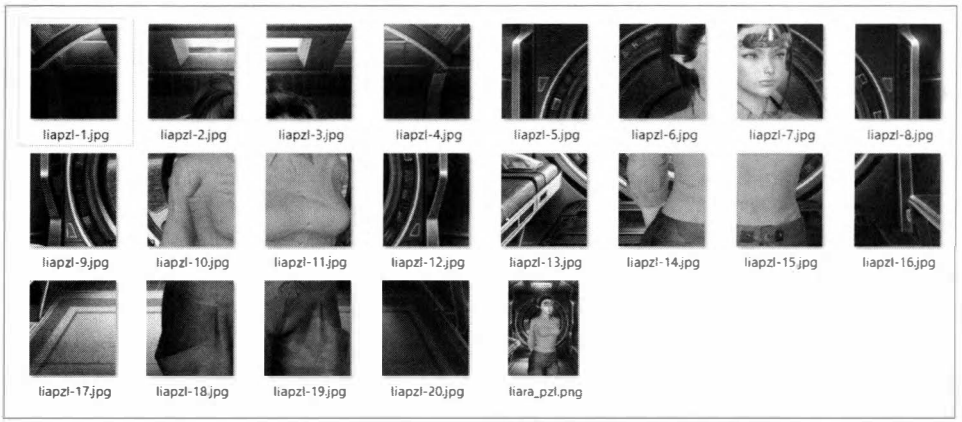


Рис. 13.31.

Напомню, что каждый объект списка имеет свой индекс (порядковый номер), поэтому первая координата в списке **final_list_positions** будет соответствовать фрагменту, имеющему аналогичный индекс, вторая координата соответствовать второму фрагменту и т.д. Чтобы проверить, находится ли фрагмент на своей финальной позиции, нам потребуется соответствующая функция.

```
init python:
    # Функция для расстановки фрагментов по местам
    def check_pos(dropped_f, dragged_frag):
        global fragments_complete, puzzle_complete
        # Перебираем список фрагментов и проверяем,
        # установлен ли он на свою позицию
        if dragged_frag[0].drag_name == dropped_f.drag_name:
            # Анимация фрагментов
            dragged_frag[0].snap(dropped_f.x, dropped_f.y)
            # Запрещаем передвигать фрагмент, если он на своем месте
            dragged_frag[0].draggable = False
            # Если на месте, добавляем +1 к количеству установленных
            fragments_complete +=1
            # Если количество установленных равно общему количеству
            # фрагментов в игре, значит пазл собран
            if fragments_complete == total_fragments:
                puzzle_complete = True # Переключили переменную победы
                renpy.jump("puzzle_end") # Отправили в метку
```

В этом же блоке **init python** напишем еще одну функцию, которая будет генерировать стартовые позиции для фрагментов, после чего они сохранятся в пустом списке **start_list_positions**.

```
init python:
    # Генерация стартовых позиций для фрагментов
    def generat_pos():
        # Цикл перебора фрагментов
        for i in range(total_fragments):
            min_x = 1100 # Диапазон координат,
            min_y = 200 # в котором могут появиться
            max_x = 1600 # фрагменты
            max_y = 800
            # Генерируем случайную позицию
            random_pos = (renpy.random.randint(min_x, max_x), renpy.
random.randint(min_y, max_y))
            # Добавляем в пустой список стартовых позиций
            # только что сгенерированную координату
            #(цикл for делает это для каждого фрагмента)
            start_list_positions.append(random_pos)
```

В диапазоне координат мы указали область, в которой появятся фрагменты. То есть по координате x они появятся на отрезке от 1100 до 1600 пикселей, а по y — от 200 до 800. Затем функция рандома (случайного значения) подставляет эти значения в качестве минимальных и максимальных и выбирает в этом промежутке случайное целое число.

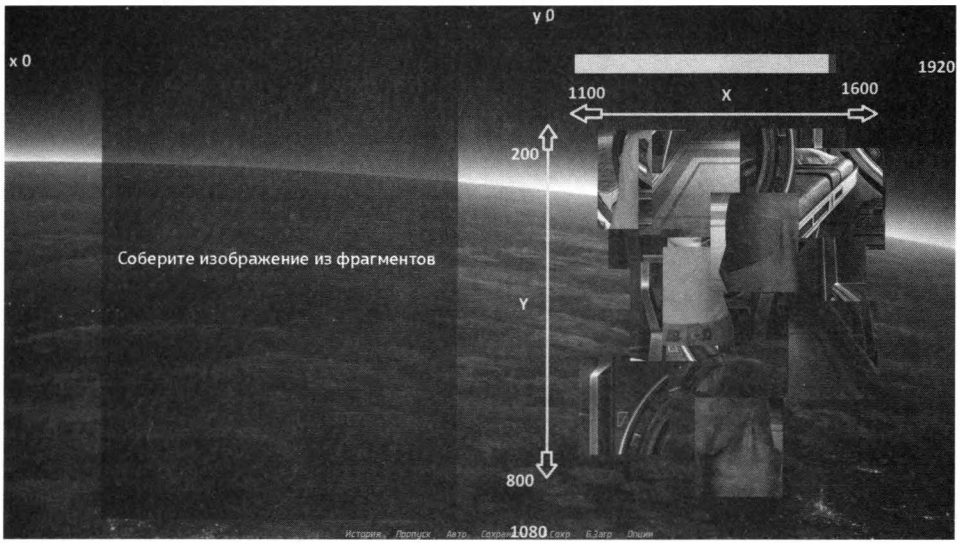


Рис. 13.32.

Далее создадим метку с настройками и стартом игры:

```
label puzzle_start: # Начало мини-игры
    hide screen interface # Скрыли интерфейс
    $ vremya = 120 # Установили таймер игры на нужное время
    $ generat_pos() # Функция для генерации стартовых позиций
    call screen puzzle_1 with dissolve # Вызвали экран с пазлом
```

Экран с пазлом:

```
screen puzzle_1():
    add "images/backgrounds/kosmos1.png" # Фоновое изображение
    # Таймер отображает оставшееся время,
    # по истечении отправляет в финальную метку
    timer 1.00 repeat True action If(vremya > 0, SetVariable("vremya",
vremya - 1), [Hide ("puzzlle_1"), Jump ("puzzle_end")])

    # Бар для визуализации таймера
    bar value AnimatedValue(value=vremya, range=120, delay=1.0) xpos 1150
ypos 100 xmaximum 525
    frame: # Область для сборки пазла
        background "#00000080" # Полупрозрачный фон области
        xysize (715,997) # Размер области
        pos (200,42) # Расположение области
        text _("Соберите изображение из фрагментов") pos (25,450) color
"#ffffff" # Подсказка
    draggroup: # Контейнер для перемещений
        # Чтобы не расписывать 20 "Персонажей" и 20 "Шахт",
        # разложим их с помощью циклов
        for i in range(total_fragments): # Цикл перебора фрагментов
            drag: # Перемещаемые объекты по типу рабочих
                drag_name i # Присваивает имя фрагменту из переменной i
(цикл for по очереди кладет в переменную все фрагменты)
                # При нажатии помещает выбранный фрагмент поверх остальных
                drag_raise True
                # Позиция для фрагмента берется из сгенерированного
                # списка позиций, согласно индексу
                pos start_list_positions[i]
                # Изображение фрагмента,
                # (подставляем число для каждого, как в календаре %s)
                add "images/lia_pzl/liapzl-%s.jpg" % (i+1)
        for i in range(total_fragments):
            drag: # Неперемещаемые объекты по типу шахт
                drag_name i # Аналогичная раздача имен
                droppable True # Объект взаимодействует с другими
                dropped check_pos # Запускает функцию проверки
```

```
            draggable False # Запрещает перемещать этот фрагмент
            # Позиция фрагмента берется из списка,
        # согласно индексу
            pos final_list_positions [i]
            # Для статичных изображений берутся те же фрагменты,
            # но с полной прозрачностью (alpha 0.0)
            add "images/lia_pzl/liapzl-%s.jpg" % (i+1) alpha 0.0
```

После того как пазл будет собран, в функции **check_pos** сработает условие, которое отправит игрока в метку после игры. Если пазл не будет собран, истечет таймер, который закроет экран и отправит в ту же метку.

Финальная метка:

```
label puzzle_end: # Метка после игры
    scene black with fade # Затемнение
    pause 0.5 # Короткая пауза
    # Сбросили переменную для повторной игры
    $ fragments_complete = 0
    scene kosmos2 with fade # Показали новую сцену
    show liara_pzl with dissolve: # Изображение собранной картинки
        align(0.5,0.5) # Позиция картинки
    if puzzle_complete == True: # Если игрок успел собрать
        $ reward = True # Выдаем награду или ачивку
        # Выключаем переменную победы для повторной игры
        $ puzzle_complete = False
        $ renpy.notify (_("Красавчик :)")) # Всплывающее уведомление
        "Пазл успешно собран, возьми пирожок"
    else: # В случае проигрыша
        "В следующий раз обязательно получится"
    jump part_999
```

Стоит отметить, что для создания еще одного похожего пазла достаточно заменить изображения в экране (screen) новыми. То есть можно сделать любое количество таких мини-игр, в которых пазлы с изображениями будут устанавливаться в зависимости от переменной. А саму переменную переключать в стартовой метке мини-игры. Аналогично такому перетаскиванию создаются карточные игры и инвентари с возможностью помещать в них предметы. То есть область, в которой собирался пазл, может выглядеть как сумка с ячейками, а фрагменты пазла заменены изображениями предметов. Вместо генерации случайных позиций их можно разместить по точным координатам. Например, на полке магазина или в сундуке с лутом.

Рис. 13.33.

13.12. Создание Android-версии

Портирование проекта на мобильные платформы — относительно простая задача. При условии, что весь дополнительный софт был установлен корректно.

Для создания версии на Android сначала потребуется установить инструмент **RAPT** (Ren'Py Android Packaging Tool).

При выборе соответствующего раздела в лаунчере Ren'Py сообщит, что RAPT не установлен, и предложит сделать это.

После подтверждения действия начнется скачивание и автоматическая установка RAPT. После этого лаунчер будет перезапущен, и в разделе Android появятся новые функции. В частности, станут доступны эмуляторы для тестирования версий для планшетов, телефонов и телевизоров.

Однако для постройки дистрибутивов необходимо установить Java Development Kit и Android SDK.

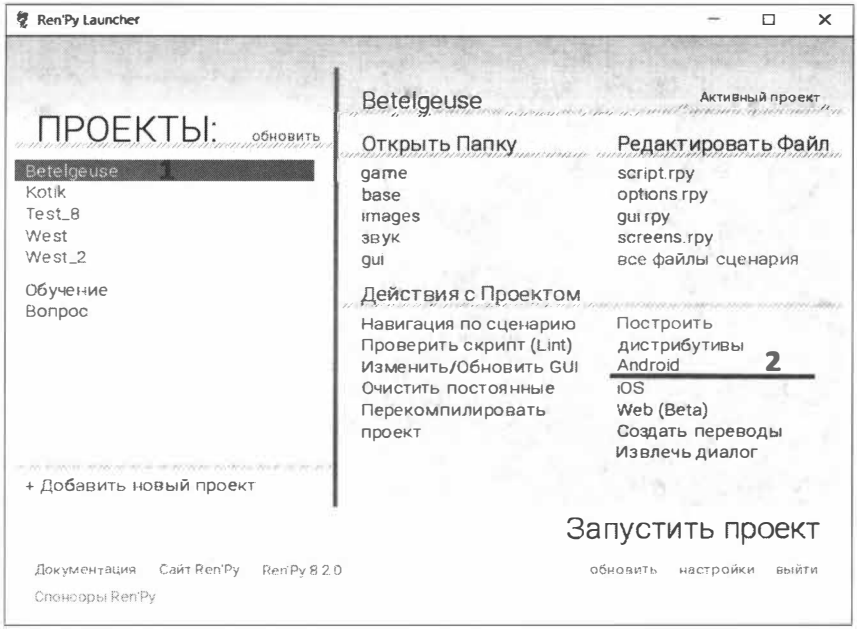


Рис. 13.34.

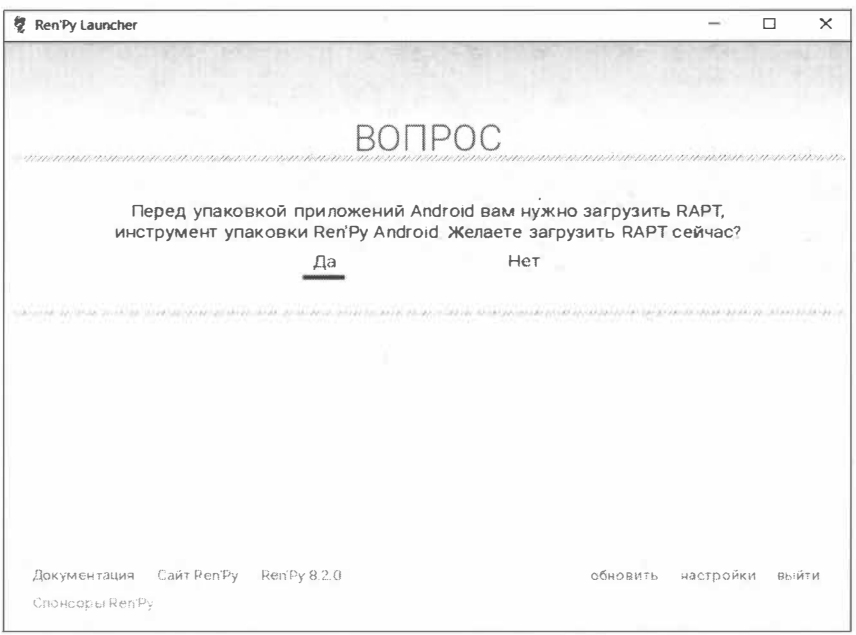


Рис. 13.35.

Если нажать на опцию Android SDK, лаунчер проверит, установлен ли у вас пакет Java и какой он версии. В противном случае будет предложена ссылка на официальный сайт adoptium.net, с которого необходимо скачать и установить Java Development Kit самостоятельно.

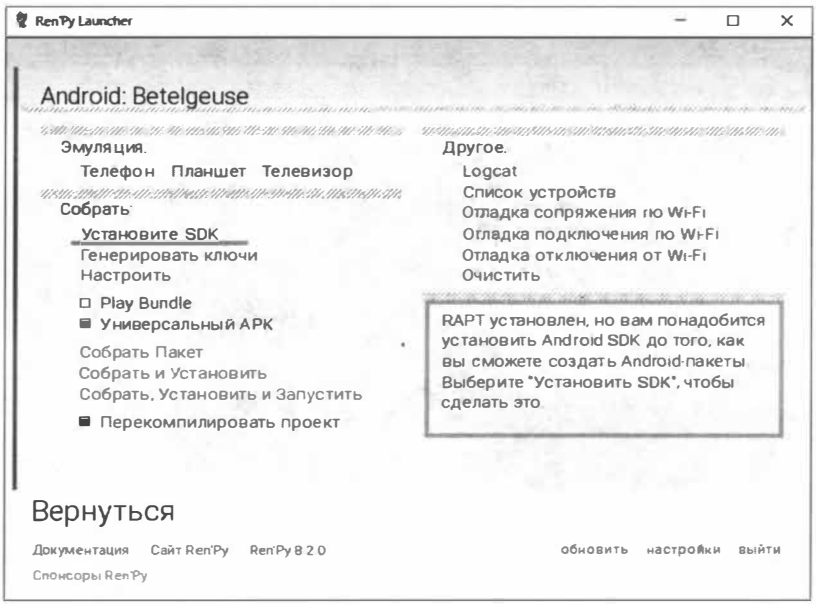


Рис. 13.36.

Рис. 13.37.

Как только инструментарий Java будет установлен, снова нажмите на опцию Android SDK. Откроется следующее окно, и после подтверждения начнется скачивание и установка этого девайса.

Рис. 13.38.

Для установки Android SDK потребуется некоторое время, по истечении которого лаунчер сообщит, что все готово к построению дистрибутивов на Android.

Следующим шагом, если вы планируете выпускать игру в Google Play и подобных, необходимо сгенерировать ключи.

Нажмите соответствующий пункт, затем введите название своей компании и подтвердите, что созданные ключи не будут переданы третьим лицам.

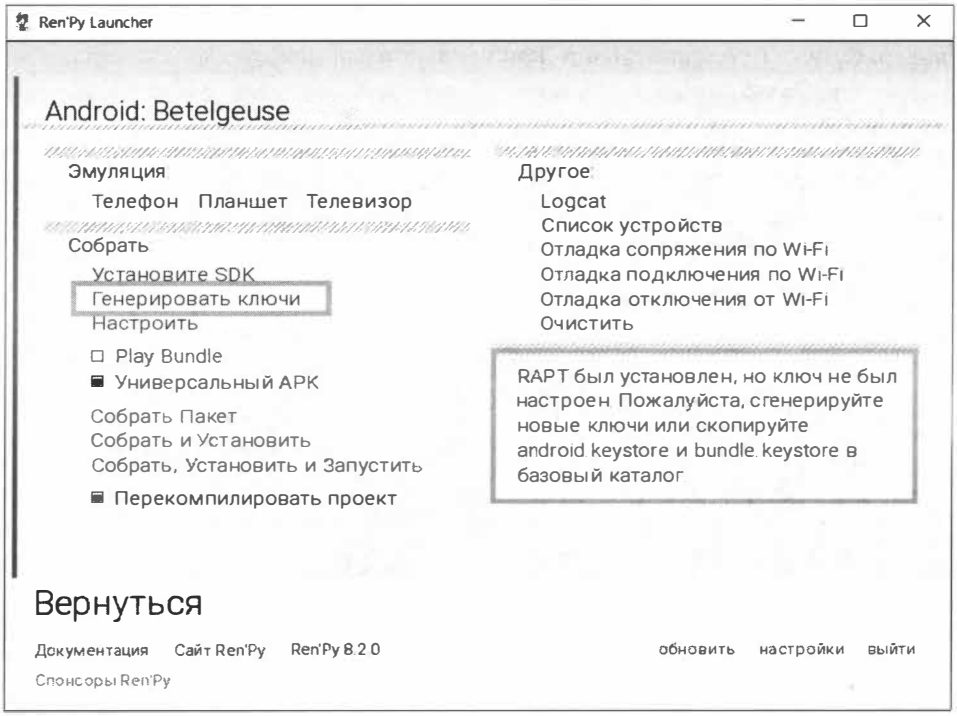


Рис. 13.39.

В итоге готовые файлы с ключами будут созданы в корневой папке вашего проекта. Необходимо сохранить их в надежном месте, недоступном для посторонних.

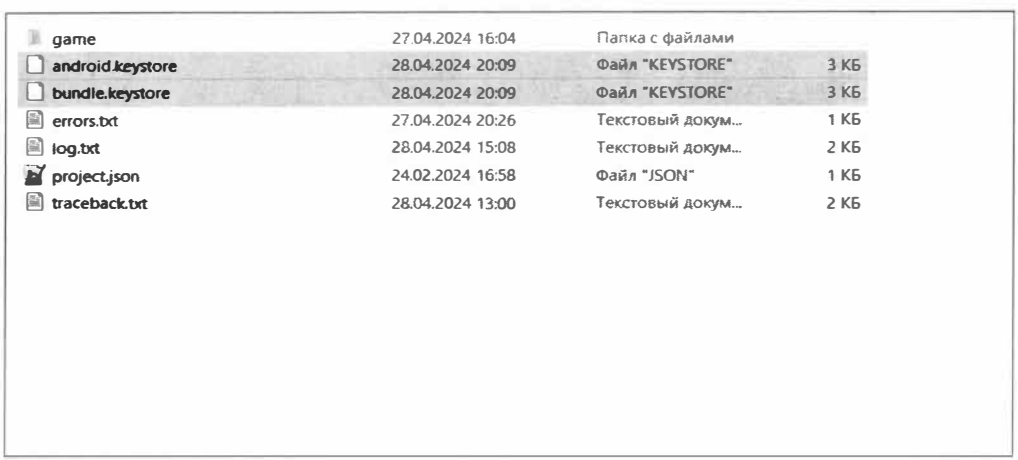


Рис. 13.40.

Завершающим шагом нам остается настроить и собрать Android-дистрибутив. Предварительно Ren'Py запросит некоторую информацию, такую как: название вашего проекта, короткое название для отображения на телефоне и др. Следуйте подсказкам и используйте латинские символы.

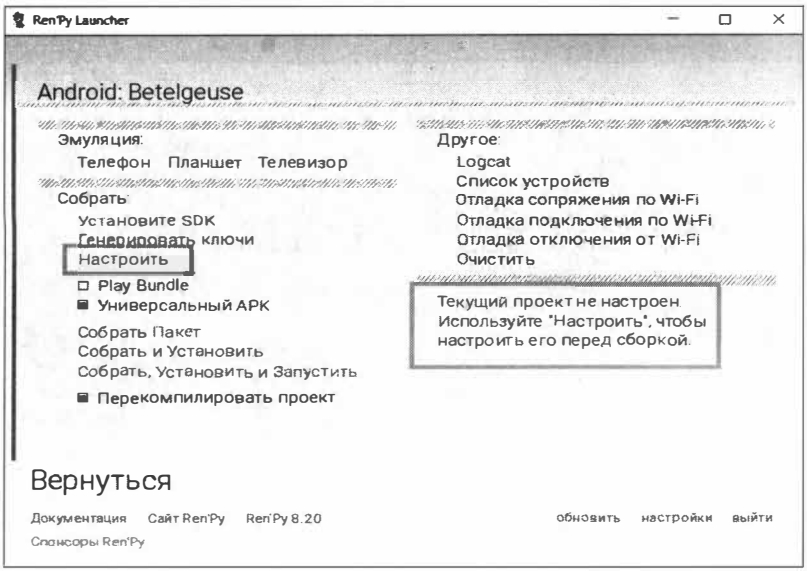


Рис. 13.41.

Далее лаунчер вернется в меню опций для Android, в котором надо нажать "Собрать Пакет" и дождаться результата.

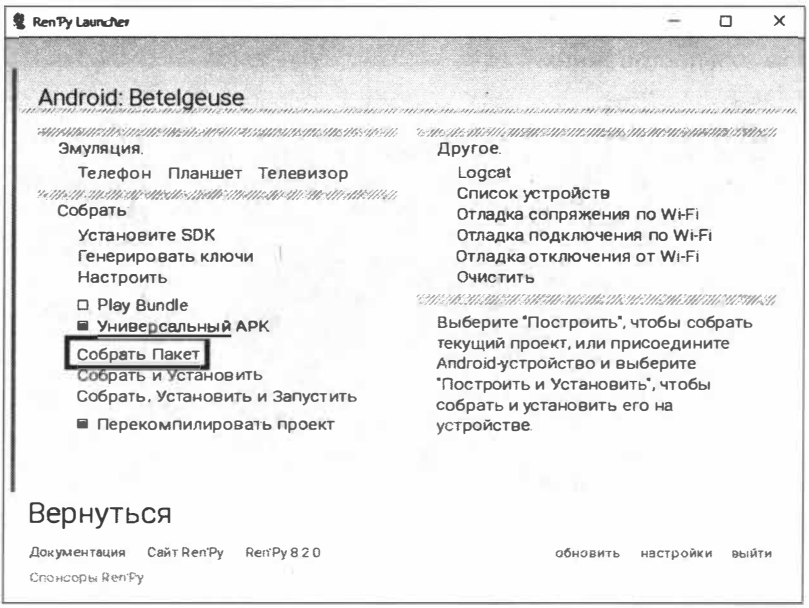


Рис. 13.42.

После завершения компиляции отобразится соответствующее сообщение, и откроется папка с готовым apk-архивом.

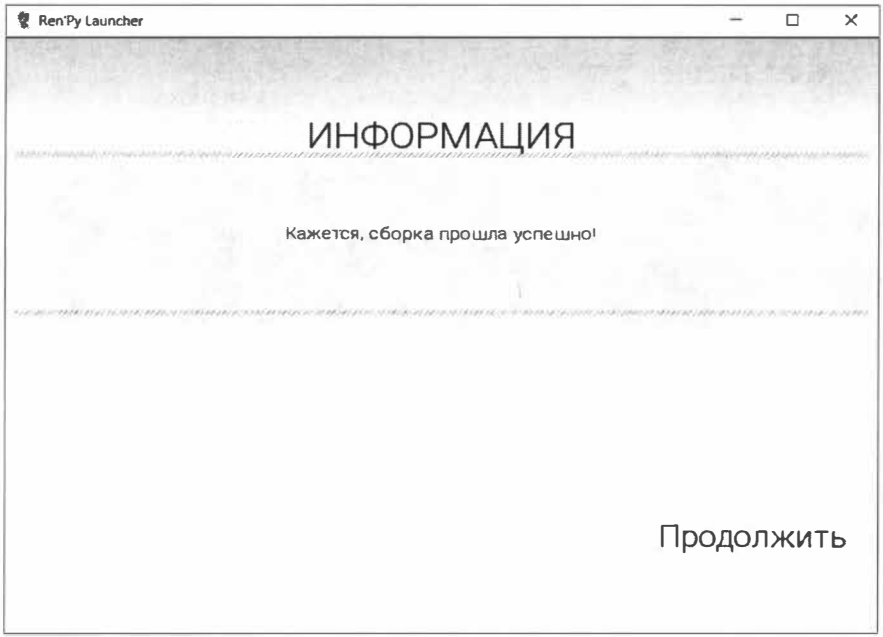


Рис. 13.43.

Согласно документации движка, Ren'Py способен собирать проекты весом не более 2 Гб. На практике, проекты компилируются и с большим размером, но при попытке установить на телефон будет происходить ошибка установки. Поэтому следует придерживаться рекомендуемого объема и при необходимости уменьшать итоговый вес проекта. Например, путем уменьшения разрешения, ухудшения качества изображений или разделения игры на несколько самостоятельных эпизодов.

Прежде чем создавать мобильную версию, стоит иметь в виду, что на небольших экранах некоторые элементы интерфейса могут сползти или быть слишком мелкими. Проверить это можно с помощью встроенного эмулятора. Нажав на один из вариантов, запустите игру с эмуляцией экрана планшета или телефона. Таким образом, вы сможете понять, насколько удобен ваш интерфейс на данных платформах, и адаптировать его по необходимости.

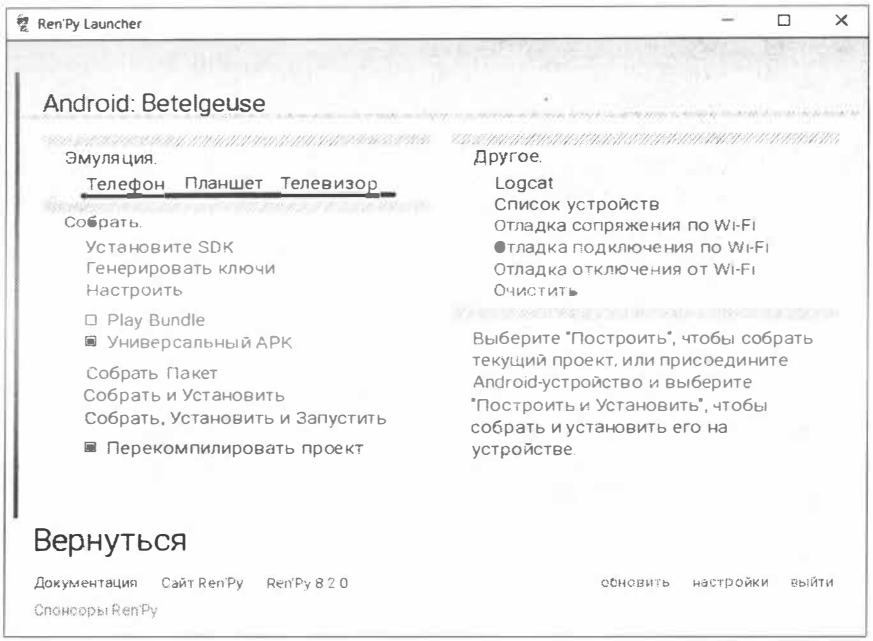


Рис. 13.44.

Кроме того, на сенсорных экранах нет курсора мыши и некоторый функционал может работать некорректно или не работать вовсе. Например, привязка определенных действий к кнопкам клавиатуры приводит к тому, что на мобильной версии цифровая клавиатура может занимать большую часть экрана, скрывая тем самым обзор, экранные кнопки или другие функции. Поэтому мини-игры и механики, привязанные к клавишам, будут неудобны.

При тестировании в эмуляторе телевизора по краям экрана может накладываться черная рамка. Однако наличие ее в готовом дистрибутиве зависит от модели телевизора у пользователя. Поэтому для полного понимания, как портированные версии будут выглядеть на самом деле, рекомендуется устанавливать их на реальные устройства (телефоны, планшеты, телевизоры) для тестирования.

После создания мобильной версии проект будет иметь стандартные иконки и загрузочные экраны движка. Чтобы заменить их, откройте папку, в которую установлен Ren'Py, и пройдите по пути: **rapt/templates**. Там вы найдете соответствующие файлы, которые необходимо поменять на свои с сохранением тех же названий.

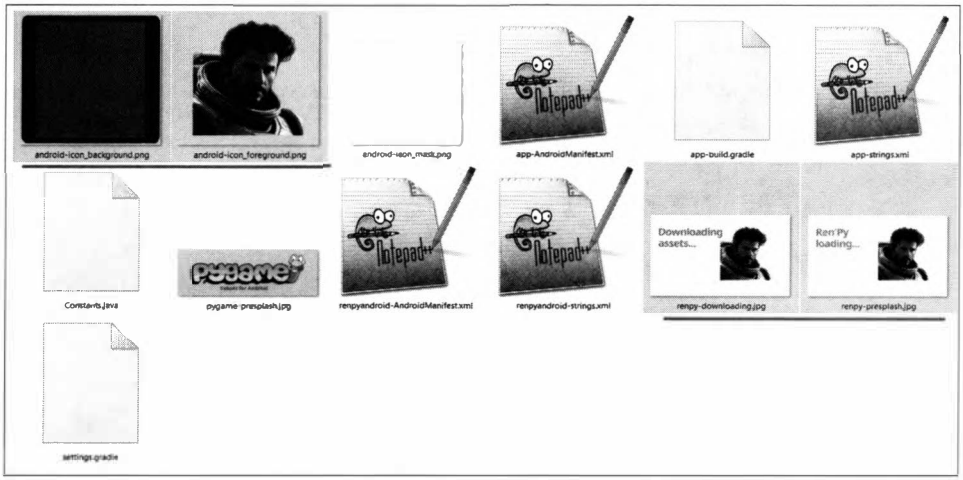


Рис. 13.45.

Также имейте в виду, что в директории **rapt/bin** сохраняются резервные копии созданных проектов. Это может быть полезно в случае потери основной версии или при необходимости освободить место на диске.

13.13. Версия для iOS

Создание iOS-версии для устройств iPhone и iPad требует наличия программ, разработанных Apple (например, Xcode IDE). Которые, в свою очередь, работают только на компьютерах Macintosh. В настоящий момент поддержка движком mac-версий находится на стадии разработки, так как интерфейс Ren'Py по умолчанию не соответствует рекомендациям Apple. С помощью лаунчера можно построить десктопную версию.

Прежде чем пытаться собрать версию для iOS, необходимо произвести некоторые настройки компьютера с системой Mac для работы с приложениями. В частности, необходимо установить Xcode, зарегистрироваться в программе iOS Developer Program и создать профиль инициализации, который позволит вашим приложениям работать на iOS-устройствах.

Разработчики из Apple в руководстве App Distribution Quick Start поясняют, как настроить все вышеперечисленное. Рекомендуется изучить данное руководство и попробовать создать тестовое приложение, прежде чем переходить к играм на Ren'Py.

После подготовки необходимого инструментария в лаунчере Ren'Py нужно выбрать соответствующий пункт, а затем установить плагин Renios, который является аналогом RAPT, но только для iOS. Он также скачивается и устанавливается автоматически.

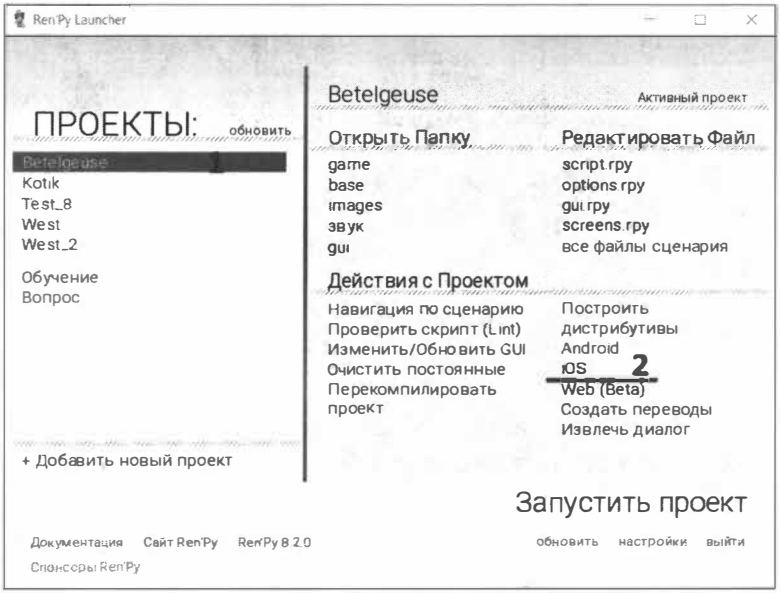


Рис. 13.46.

После этого становятся доступны эмуляторы и возможность создать дистрибутив.

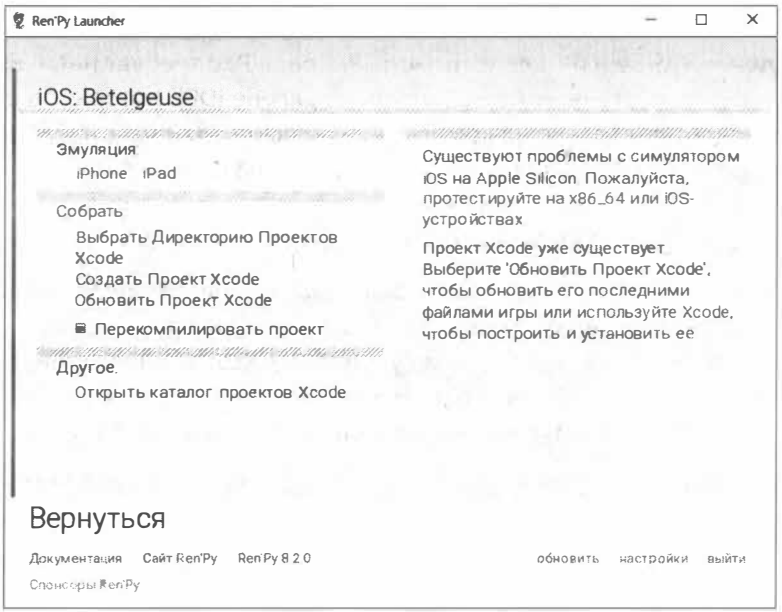


Рис. 13.47.

Для сборки версии на iOS нажимаем соответствующую опцию "Создать проект Xcode". Имя приложению будет присвоено автоматически на основе имени проекта. Однако, при необходимости, его можно переименовать.

13.14. Веб-версия

В настоящий момент Ren'Py поддерживает большинство популярных браузеров, таких как Chrome, Edge, Firefox, Safari и других, созданных на их основе. Например, Яндекс-браузер.

Как и все игры, запускаемые в браузере, игры на Ren'Py сталкиваются с рядом ограничений. Некоторые из них связаны с защитой браузеров, предотвращающей показ рекламы и всплывающих баннеров, другие связаны с ограничением на объем данных, которые можно загрузить перед началом игры.

Так как браузеры используют свои видеоплейеры для воспроизведения видео и не поддерживают некоторые форматы, это накладывает ограничения на форматы, в которые ролики из игры могут быть конвертированы. Кроме того, видео может быть вызвано только инструкцией **renpy.movie_cutscene()**. Тогда как более удобный вариант вызова через **image** (виртуальное изображение) недоступно. Преимущество второго варианта в том, что поверх видео могут быть наложены диалоги.

Вместе с тем некоторые ограничения могут накладывать хостинг-провайдеры. Например, веб-ресурс itch.io ограничивает размер игры и количество файлов, которые могут быть включены в проект. Некоторые браузеры, в свою очередь, устанавливают ограничения на общий объем файлов, которые могут кэшироваться.

Это приводит к тому, что некоторые файлы большого размера будут загружаться заново каждый раз при запуске игры. Все эти моменты следует учитывать во время разработки, чтобы потом не пришлось кардинально переписывать код игры под браузерную версию.

Чтобы создать веб-версию, в лаунчере необходимо выбрать пункт Web (Beta), а затем установить плагин, который схож функционалом с плагинами для мобильных версий.

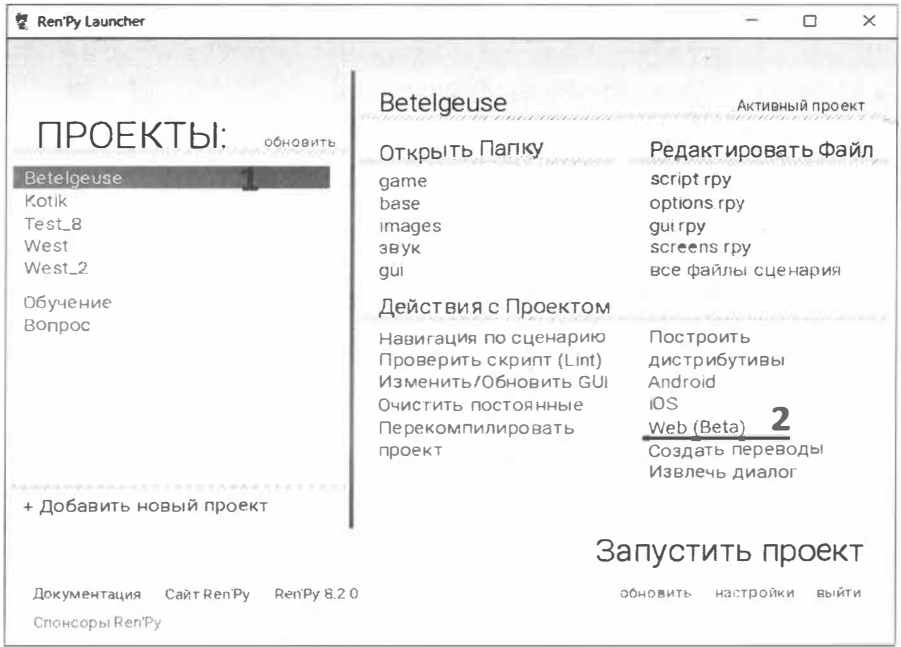


Рис. 13.48.

Здесь нам доступны четыре опции:

1. **Построить приложение для сети** — позволяет создать веб-версию для загрузки на хостинг. Движок создает архив web.zip, а также папку со всеми файлами. Некоторые хостинги не предоставляют прямого доступа к корневой папке, ограничиваясь веб-интерфейсом. В таком случае будет полезен архив.
2. **Построить и открыть в браузере** — этот вариант позволяет собрать веб-версию проекта, которую можно открыть непосредственно в браузере. При этом на вашем компьютере будет запущен веб-сервер, обслуживающий игру. Данная опция полезна для тестирования и контроля того, как будут работать те или иные функции.
3. **Открыть в браузере** — вариант аналогичен предыдущему, только без сборки веб-версии, которую можно загрузить на хостинг.
4. **Открыть папку сборки** — откроется папка, содержащая все файлы, созданные в процессе сборки.

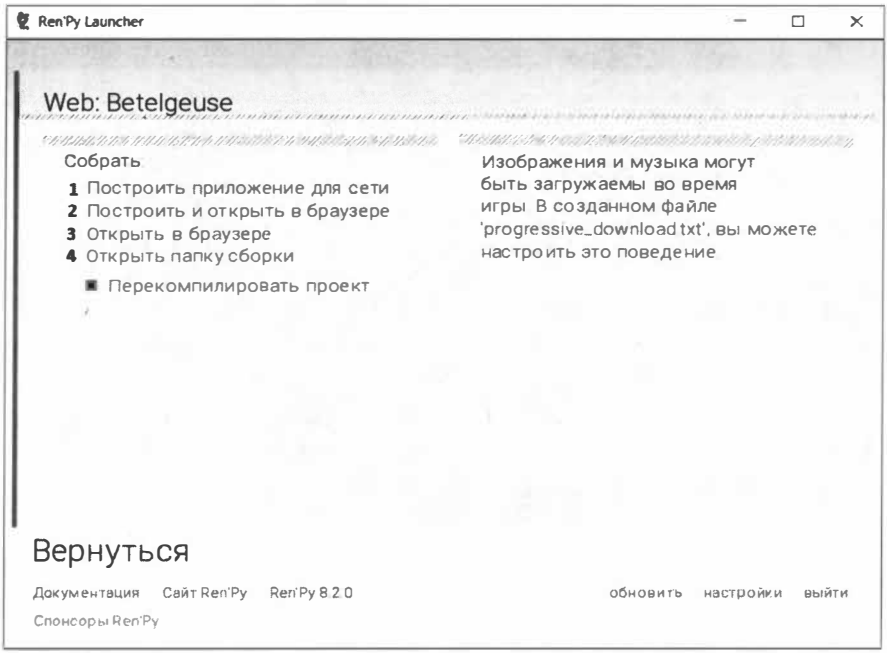


Рис. 13.49.

После того как дистрибутив будет построен, его необходимо загрузить на публичный сервер. Например, itch.io и подобные. В таком случае вам понадобится архив web.zip. Если вы используете полноценный хостинг, то можете загружать файлы по отдельности или папками, настраивать директории и пути.

Во время запуска игры в пресплеше будет использоваться стандартный логотип движка. Чтобы заменить его своим, поместите в корневую папку или в архив web.zip с заменой файл с названием web-presplash.jpg, также подойдут форматы png, webp и видео webm.

Для настройки фавикона (иконки вкладки браузера) создайте изображение с названием web-icon.png и разрешением 512 x 512, а затем также поместите его в архив web.zip или корневую папку вашего проекта. Если изображение не задано, будет использован стандартный логотип.

Вместе с тем Ren'Py поддерживает функцию прогрессивной загрузки, которую можно настроить путем редактирования файла progressive_download.txt в базовом каталоге проекта. Если файл отсутствует, он будет создан автоматически при первой компиляции веб-версии.

По умолчанию в файл записываются следующие инструкции:

```
- image game/gui/**
+ image game/**
+ music game/audio/**
+ voice game/voice/**
```

Данные команды определяют, какие файлы загружаются перед началом игры, а какие по мере необходимости. Строки, начинающиеся с символа "минус" (-), соответствуют файлам, которые должны быть загружены в начале игры. Как правило, это файлы, относящиеся к главному меню и загрузочным экранам. Строки, начинающиеся со знака "плюс" (+), соответствуют файлам, которые следует загружать по мере необходимости.

Стоит иметь в виду, что из-за ограничений браузеров, направленных на блокировку рекламы, видео- и аудиофайлы не будут воспроизводиться до тех пор, пока игрок хотя бы раз не щелкнет мышью внутри игры. Поэтому перед показом анимации или катсцены в самом начале необходимо, чтобы пользователь совершил какое-то действие. Или нужно делать покадровую анимацию с помощью **image** и **transformation**.

Глава 14.

ВСПОМОГАТЕЛЬНЫЕ ПРОГРАММЫ И СОФТ

14.1. Лицензии распространения контента

Часто при создании своих игр разработчик не обладает всеми необходимыми навыками для этого. Например, вы хорошо рисуете, но не умеете играть на музыкальных инструментах, поэтому для музыкального сопровождения придется использовать чужую музыку. Или вы отличный писатель, но совсем не умеете рисовать, тогда фоны и спрайты придется использовать чужие.

Любые материалы, в том числе изображения, спрайты, музыка, скрипты, которые вы найдете в сети, имеют своих авторов. Некоторые из них запрещают использовать свои наработки в чужих проектах, другие разрешают, но с указанием их имени в титрах, третьи разрешают использовать в бесплатных играх, но запрещают в коммерческих проектах, четвертые разрешают использовать свои материалы, как вам будет угодно. То есть дают согласие на любое изменение и распространение своего контента.

Чтобы не запутаться, что вы можете использовать, а что нет, нужно понимать, какие лицензии распространения бывают и какие из них подойдут вам. В любом случае, даже если автор не требует указания своего имени в титрах, хорошим тоном будет сделать это. Таким образом, вы выразите свою благодарность ему.

Creative Commons

Creative Commons (CC) — широко применяется в культуре, образовании и науке. Эта лицензия позволяет использовать литературные, изобразительные и музыкальные произведения искусства, а также научные труды без заключения договора и выплаты вознаграждения правообладателям.

Creative Commons включает в себя совокупность лицензий, разработанных для предоставления авторам более гибкого контроля над правами на

свой контент. Они определяют условия, на которых контент может быть использован другими. На сегодняшний день существует 7 видов разрешений Creative Commons, используемых в интернет-пространстве.

1. **Public Domain (CC0)** — свободное распространение и использование контента с возможностью коммерческой реализации.
2. **(CC BY)** — разрешает свободное использование и изменение контента, но с обязательным указанием автора.
3. **(CC BY-SA)** — условия аналогичны предыдущему разрешению, но финальный продукт должен распространяться с такой же лицензией, как и оригинальный материал (ShareAlike). То есть если какие-то ассеты были предоставлены вам бесплатно, то ваш проект также должен оставаться бесплатным.
4. **(CC BY-NC)** — разрешает использовать материалы исключительно с некоммерческой целью (NonCommercial).
5. **(CC BY-ND)** — разрешает использовать материалы только в неизменном виде с возможностью коммерческого распространения (NoDerivatives).
6. **(CC BY-NC-SA)** — разрешает использовать материалы с возможностью их изменения, но только в некоммерческих целях и с той же лицензией распространения.
7. **(CC BY-NC-ND)** — разрешает использовать материалы только в неизменном виде исключительно с некоммерческой целью.

GNU General Public License

GNU General Public License (GPL) — это соглашение разрешает свободное распространение ПО, а также защищает права пользователей, гарантируя им возможность свободно использовать, изменять, изучать и распространять данное программное решение. Лицензия GPL является одной из наиболее распространенных.

GPL разрешает разработчикам следующее:

- свободно использовать данное ПО в любых целях;
- изменять ПО и адаптировать его под свои потребности;
- модифицировать ПО и распространять в измененном виде.

Однако существует обязательное условие при распространении программ под лицензией GPL — любая производная работа, основанная на исходном коде программы, также должна распространяться под лицензией GPL. Это называется «copyleft», что обеспечивает сохранение свободы ПО и его открытости.

Данная лицензия создает баланс между свободой использования и защитой прав интеллектуальной собственности, обеспечивая открытость и доступность ПО для всех. Кроме того, это стимулирует сотрудничество и обмен опытом в сообществе разработчиков и пользователей.

Massachusetts Institute of Technology License

Massachusetts Institute of Technology License (MIT) — лицензия Массачусетского технологического института. Она также является одной из самых простых и открытых лицензий, используемых для распространения ПО и других материалов.

Основные особенности данной лицензии:

- Позволяет свободно использовать ПО под этой лицензией в любом виде, в том числе в коммерческих целях, без ограничений.
- Предоставляет продукт «как есть», без каких-либо гарантий или условий. То есть разработчики оригинального программного обеспечения не несут ответственности в случае каких-то неисправностей или ошибок в ПО.
- Разрешает свободно модифицировать и распространять данный продукт, а также включать его в свои собственные проекты.
- Обязывает указывать авторов оригинальной работы.

Преимущество данной лицензии заключается в том, что она предоставляет большую свободу в работе с ПО по сравнению с некоторыми другими лицензиями, такими как GPL. Кроме того, разработчики вправе создавать проекты с открытым исходным кодом, не накладывая каких-либо ограничений и позволяя свободно распространять его.

Apache License

Apache — аналогично предыдущей лицензии разрешает использовать программное обеспечение в любом виде, но с соблюдением условий лицензии и указанием авторских прав.

Основные особенности данной лицензии:

- Позволяет свободно использовать ПО и другие материалы в любых целях, включая коммерческое распространение.
- Разработчики оригинального ПО не несут ответственности за работоспособность их программ и предоставляют свой продукт «как есть».
- Авторы оригинального ПО разрешают всем желающим модифицировать их продукт и встраивать в свои программы.
- Вместе с тем данная лицензия требует, чтобы все копии нового продукта содержали уведомление об авторских правах и условиях лицензии.

Основное различие между лицензией Apache и лицензией MIT заключается в дополнительной защите от патентных претензий. Соглашение Apache включает в себя условия, направленные на предотвращение патентных споров в сообществе разработчиков.

Лицензия Apache предлагает баланс между свободой использования и защитой авторских прав, благодаря чему используется во многих проектах с открытым исходным кодом.

Это лишь несколько примеров типов лицензий, которые могут предоставляться для распространения контента. Каждая из них имеет свои особенности и условия использования. Поэтому нужно быть осторожными при заимствовании чужих материалов. Всегда обращайте внимание на тип лицензии и сохраняйте у себя копии лицензионных соглашений, так как со временем они могут изменяться.

14.2. Редакторы для создания спрайтов и анимации

В этом и следующих разделах мы поверхностно познакомимся с редакторами, позволяющими создать все необходимые ассеты (цифровые объекты) без помощи художников, музыкантов, видеомонтажеров и других специалистов.

Однако не стоит забывать, что каждая из программ также имеет свое лицензионное соглашение, исходя из которого некоторые разработчики разрешают использовать созданные материалы только в бесплатных проектах, а другие не накладывают каких-либо ограничений.

Большинство представленных редакторов довольно просты в освоении. К тому же в сети уже созданы сотни видеоуроков, благодаря которым можно быстро обучиться необходимым аспектам.

Adobe Photoshop

Рис. 14.1.

Adobe Photoshop — это многофункциональный редактор, он уже давно стал незаменимым инструментом для фотографов, дизайнеров, художников и специалистов самых разных направлений. Благодаря этому инструменту можно создавать не только спрайты, но и полноценные изображения, а также анимации и видеоролики.

Стоит отметить, что Фотошоп имеет онлайн-версию, благодаря чему им можно воспользоваться для обрезки или редактирования изображений, не скачивая саму программу и не имея на компьютере других фоторедакторов.

GIMP

GIMP (GNU Image Manipulation Program) — это бесплатный и открытый графический редактор, созданный для работы с растровыми изображениями. Своим функционалом он во многом похож на Фотошоп, а многочисленные плагины позволяют расширить его инструментарий еще сильнее.

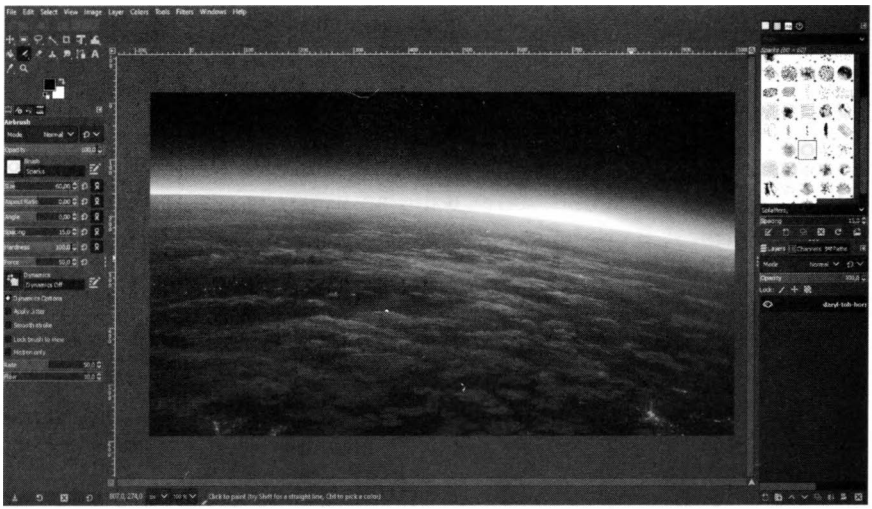


Рис. 14.2.

Если ранее вы уже пользовались Фотошопом или аналогичными редакторами, то вы без труда сможете разобраться в его интерфейсе и возможностях. GIMP является отличной альтернативой Фотошопу.

Paint.net

Рис. 14.3.

Paint.net является встроенным редактором Windows и может стать альтернативой вышеописанным программам. Своим функционалом он немного

уступает им и будет полезен, если нужно сделать несложные манипуляции с изображениями. Например, обрезать, изменить размер, яркость, контрастность и пр.

Однако благодаря установке различных плагинов функционал редактора также можно расширить, что сделает его довольно мощным инструментом, способным решить большинство задач, связанных с графической частью игры.

Krita

Рис. 14.4.

Krita — это еще один свободный и открытый редактор, созданный шведскими разработчиками. Его особенностью является возможность создавать изображения с помощью большого инструментария кистей и карандашей.

Кроме того, как и предыдущие редакторы, Krita обладает схожим функционалом, благодаря чему в нем также можно работать со стандартными инструментами, рисовать покадровую анимацию путем наложения одного слоя на другой и последующего редактирования.

Piskel

Онлайн-редактор **Piskel** позволяет создавать пиксельные спрайты персонажей и любых других объектов. Помимо этого, в него можно загрузить свое изображение и обработать в пиксельном стиле, благодаря чему можно, абсолютно не имея художественных навыков, создавать необходимые ассеты.

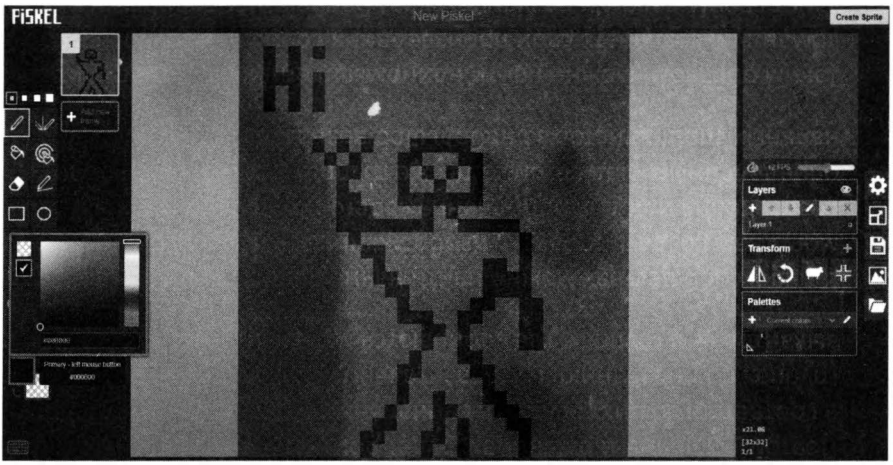


Рис. 14.5.

Интерфейс сервиса прост и удобен. Если ранее вы уже пользовались другими графическими редакторами, то сможете без проблем в нем освоиться. Здесь реализованы те же самые горячие клавиши, поэтому неудачное действие можно быстро отменить с помощью клавиш **Ctrl + Z** или инструмента «Ластика».

Blender

Рис. 14.6.

Blender — это профессиональный 3D-редактор, позволяющий создавать различные трехмерные модели, объекты, эффекты и анимацию. Он широко используется в киноиндустрии и геймдеве. Редактор имеет обширный функционал, но в контексте наших задач мы будем рассматривать его в качестве инструмента для создания спрайтов.

Казалось бы, зачем нам 3D-модели персонажей в двухмерных играх на Ren'Py? Однако, не имея навыков рисования, вы вряд ли в полной мере сможете использовать вышеописанные редакторы для создания героев. Но, создав в «Блендере» изображения 3D-персонажей, с помощью все тех же редакторов, описанных выше, вы сможете сделать из них двухмерные спрайты.

DAZ Studio

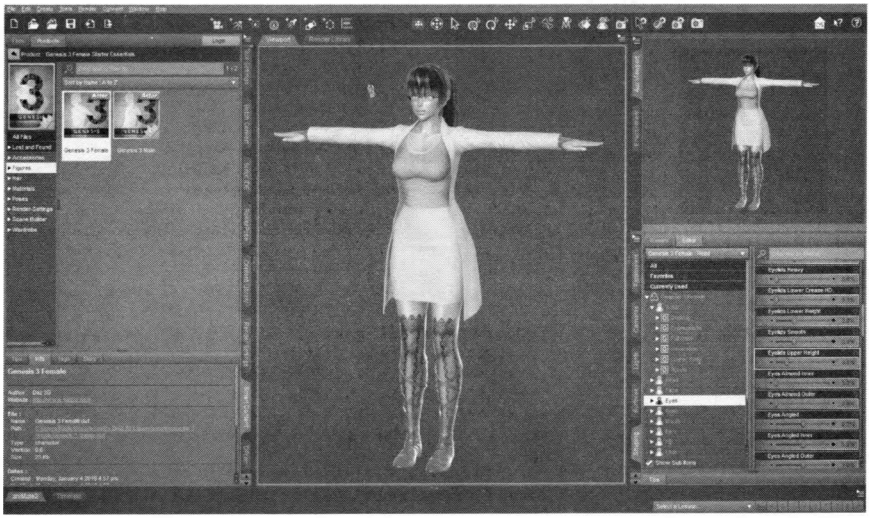


Рис. 14.7.

DAZ Studio — еще один 3D-редактор, но, в отличие от предыдущего, специализируется именно на создании 3D-моделей персонажей. Здесь, как в компьютерной игре, кастомизируя героя, можно настроить внешний вид, подобрать прическу, одежду и прочие аксессуары. Настроить фигуру и черты лица.

На официальном сайте пользователи делятся своими модификациями и ассетами, необходимыми для создания персонажей, а также моделями полноценных локаций, которые будут полезны для создания фоновых изображений.

Koikatsu

Koikatsu представляет собой полноценную игру в стиле аниме. Она обладает мощным редактором персонажей, который позволяет изменить практически любую мелочь. После чего героя можно сохранить с прозрачным фоном и использовать в качестве спрайта в своем проекте.

Рис. 14.8.

Koikatsu имеет большую аудиторию поклонников, которые используют игру непосредственно для создания спрайтов. Для проекта созданы тысячи модификаций и плагинов, благодаря которым можно создать не только уникальных героев, но и полноценные сцены для фоновых изображений. Стоит иметь в виду, что компания-разработчик Illusion Studios разрешает использовать персонажей, созданных в ее редакторе, только в проектах с бесплатным распространением.

Honey Select 1,2

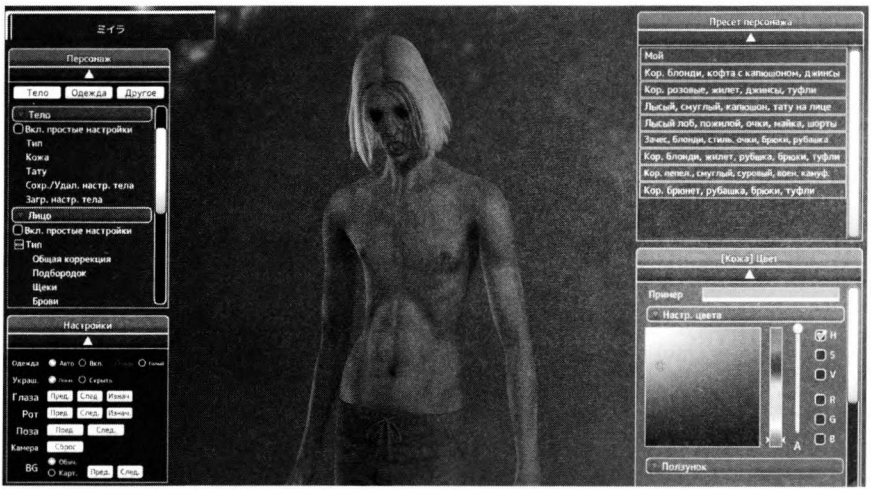


Рис. 14.9.

Honey Select — это еще два проекта от японской студии Illusion. В отличие от Koikatsu, здесь персонажи выполнены в 3D. Своим функционалом обе

игры похожи на предыдущую. Они также имеют продвинутый редактор и огромную базу различных модификаций, благодаря которым можно создать оригинальных героев и построить уникальные локации для фоновых изображений.

Однако стоит отметить, что пул предметов для создания сцен довольно ограничен и для этих целей подойдут другие редакторы, о которых речь пойдет в следующем разделе. Аналогично Koikatsu все модели могут использоваться только в бесплатных проектах.

Virt-A-Mate

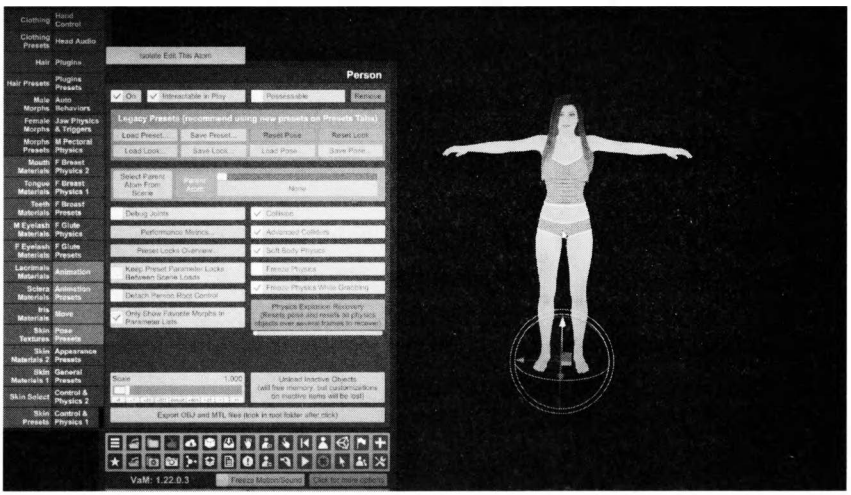


Рис. 14.10.

Virt-A-Mate — это многофункциональный редактор 3D-моделей, который позволяет создавать реалистичные и детализированные объекты, персонажей и сцены. Здесь также есть инструменты для работы с полигонами, текстурами, материалами и освещением. Кроме того, доступны инструменты для анимации и симуляции физики.

Разработчики предлагают несколько видов лицензий — от свободных вариантов для проектов с бесплатным распространением до лицензионных соглашений с правами использования в коммерческих целях.

14.3. Редакторы для создания фоновых изображений

Как в случае спрайтов, не имея навыков рисования и знакомых художников, фоновые изображения придется заимствовать из платных и бесплат-

ных источников в сети. Альтернативным решением может стать создание сцен в соответствующих редакторах, предназначенных для дизайна интерьеров, экстерьеров и ландшафтов.

С готовых 3D-моделей делаются рендеры или скриншоты, которые затем по необходимости обрабатываются в графических редакторах. На выходе получаются качественные фоновые изображения, подходящие для использования в наших проектах.

Sweet Home 3D

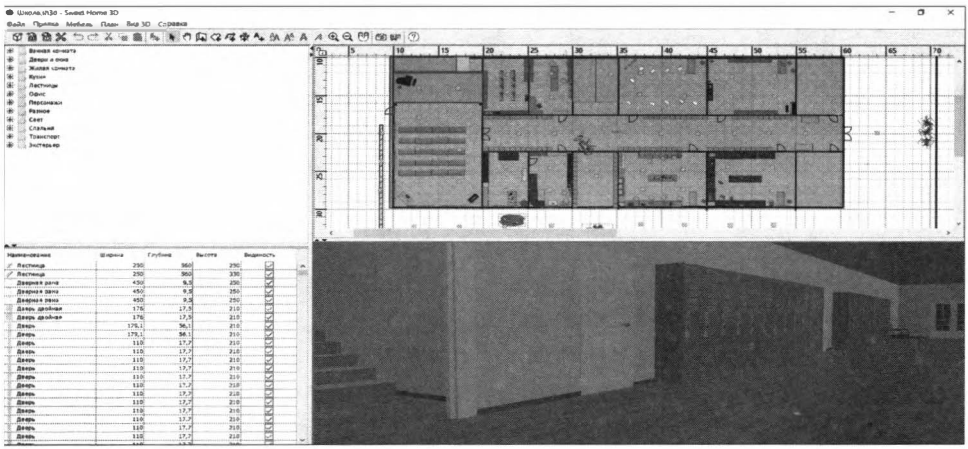


Рис. 14.11.

Sweet Home 3D — это бесплатный редактор с открытым исходным кодом для моделирования интерьеров и экстерьеров. Программа позволяет проектировать настоящие дома и делать фото- и видеовизуализацию.

На рабочем пространстве отображается упрощенный вариант финального результата, а после рендера можно получить фотореалистичное изображение, которое по необходимости обрабатывается различными фильтрами в Фотошопе, чтобы добиться стиля, необходимого для вашего проекта.

Lumion

Lumion является одним из самых популярных редакторов для 3D-визуализации интерьеров и ландшафтного дизайна. Он разрабатывался с расчетом на простых пользователей, не имеющих опыта в проектировании. Поэтому разобраться в интерфейсе сможет даже новичок.

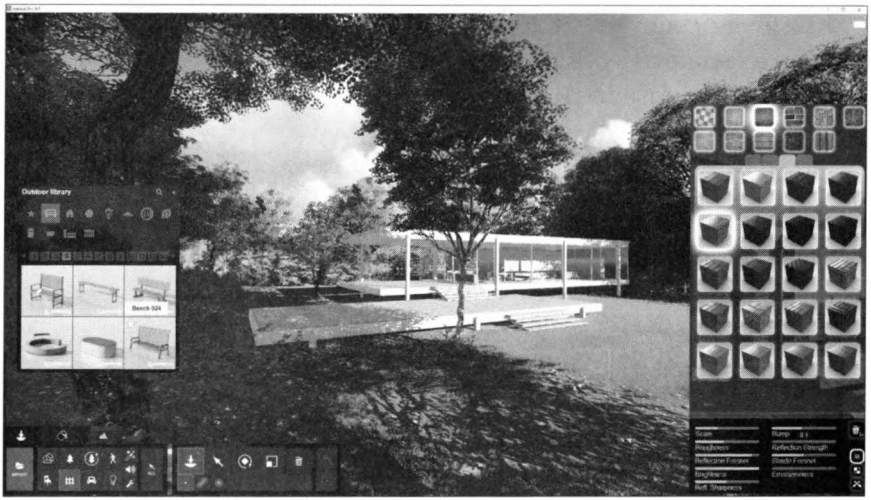


Рис. 14.12.

С 2023 года программа поддерживает трассировку лучей, благодаря чему созданные объекты стали выглядеть еще реалистичней. Хотя и раньше этот редактор мог похвастаться отличной картинкой.

3Ds Max

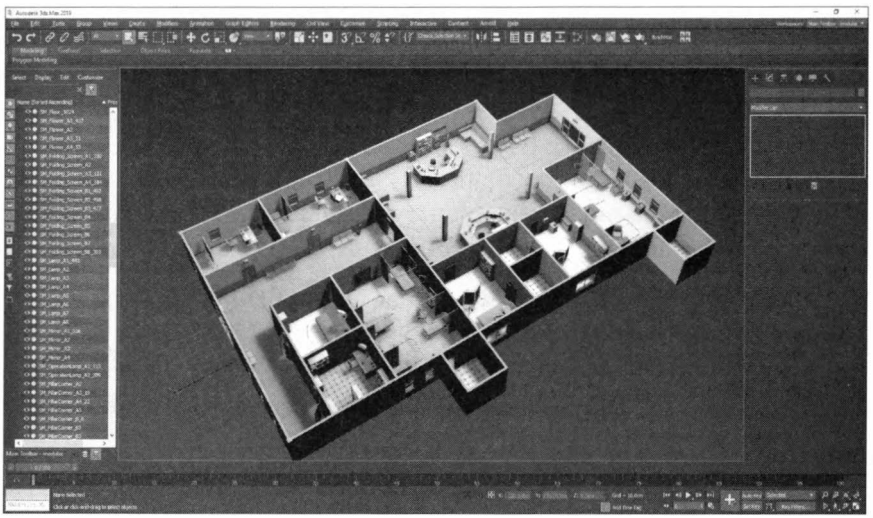


Рис. 14.13.

3Ds Max — это универсальный редактор трехмерной графики, позволяющий создавать не только интерьеры, но и любые другие объекты. Он широко используется архитекторами, веб-дизайнерами и разработчиками компьютерных игр.

Благодаря своей универсальности подходит для решения многих задач. А большое количество разнообразных плагинов позволяет расширить его функционал еще больше. Однако у 3Ds Max есть и недостатки: он совместим только с операционной системой Windows и имеет высокие системные требования.

SketchUp

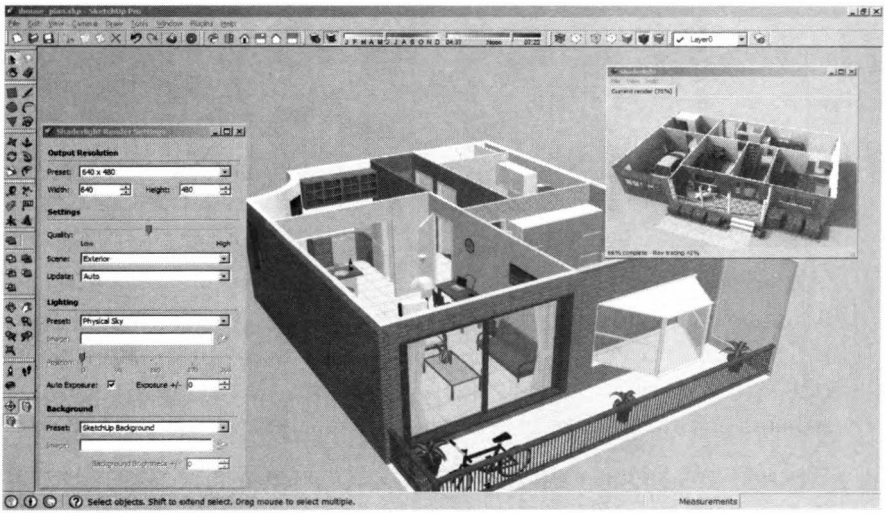


Рис. 14.14.

SketchUp — это еще один мощный инструмент для проектирования строений. Он также имеет простой в освоении интерфейс, позволяющий быстро приступить к работе. С его помощью можно обрабатывать модели, создавать окружение и ландшафты. В программе доступно множество готовых объектов, которые можно использовать для ускорения процесса.

Среди преимуществ SketchUp можно выделить возможность расширения функционала с помощью большого числа плагинов, а также свободу группировки компонентов для применения изменений ко всему проекту.

ArchiCAD

ArchiCAD — это профессиональный инструмент для архитекторов и дизайнеров, позволяющий создавать 2D- и 3D-визуализации, модели зданий и сооружений. Программа оснащена готовыми элементами, такими как мебель, лестницы, окна и двери, что облегчает процесс проектирования.

Рис. 14.15.

В редакторе также имеется встроенный графический движок для рендеринга, который позволяет создавать фотореалистичные изображения готовых объектов. Для удобства существует мобильная версия, благодаря которой можно работать, находясь в дороге или не имея под рукой компьютера.

Помимо графических редакторов, с развитием технологий искусственного интеллекта любой желающий может сгенерировать качественные изображения с помощью нейросетей. В настоящий момент уже существует огромное множество различных ресурсов, предлагающих подобные услуги (платные и бесплатные). Я упомяну лишь пару самых известных — Midjourney и Leonardo, но вы без труда найдете и другие, если этих будет недостаточно.

В подобных сервисах можно сгенерировать практически любые изображения по текстовому описанию. Не всегда нужный результат получается с первого раза, но в любом случае это хорошее подспорье, если вы не имеете навыков рисования и у вас нет знакомых художников, способных помочь.

14.4. Редакторы музыки

Помимо нейросетей, генерирующих изображения, существуют аналоги для создания музыкальных композиций, которые также по текстовому промту и выбранному жанру способны сгенерировать оригинальные треки для вашего проекта.

Еще одним вариантом поиска и подбора музыки могут стать музыкальные стоки, на которых начинающие музыканты и композиторы выкладывают свои работы. Многие из них разрешают использовать свои аудиофайлы в чужих проектах и видео на Youtube. Как правило, они лишь требуют указания их авторства в титрах или в описании к видео.

Вместе с тем после создания подходящих композиций неплохо было бы выровнять все треки по громкости, чтобы во время игрового процесса они не резали ухо игроку и не шокировали его большими перепадами. Для этого можно воспользоваться одним из редакторов, представленных ниже, в большинстве из которых также можно создать свои несложные треки из богатого набора сэмплов.

Ableton Live

Ableton Live — это универсальная программа, предназначенная для создания музыки и записи живых выступлений. Она используется многими известными музыкантами и диджеями. Редактор предлагает широкие возможности для работы со звуком и является решением «все в одном».

FL Studio

FL Studio (ранее известная как Fruity Loops) — это цифровой комбайн и секвенсор для создания композиций. Программа позволяет записывать и сводить аудио- и MIDI-материалы, используя различные плагины, эффекты и сэмплеры. В ней также можно выровнять громкость каждого трека.

Pro Tools

Pro Tools — еще один мощный редактор для работы со звуком. Эта программа предназначена для обработки аудио- и видеоматериалов. Ее основные функции включают в себя запись, редактирование, микширование, сведение и мастеринг аудиофайлов. Программа позволяет работать с различными форматами, а также поддерживает подключение внешних устройств и плагинов.

Cubase

Cubase — один из самых популярных редакторов среди музыкантов. Он предназначен для создания, записи и микширования композиций. Так же как и вышеописанные программы, имеет большой ассортимент сэмплов,

позволяющий собрать свои собственные треки, соединив несколько разных дорожек и музыкальных инструментов.

14.5. Редакторы видео

Для создания катсцен, анимаций, эффектов, различных трейлеров и видеоматериалов по игре потребуются программы для редактирования видео и захвата экрана, чтобы, например, сделать запись игрового процесса. В этом могут помочь следующие видеоредакторы.

Adobe Premiere Pro

Adobe Premiere Pro — это популярный инструмент для редактирования видео среди профессионалов и любителей. Он предлагает широкие возможности детального редактирования, композитинг, кеинг и поддержку множества форматов видео и аудио. Редактор используется для монтажа роликов разного уровня сложности, от видеорепортажей до кассовых фильмов.

DaVinci Resolve

DaVinci Resolve — это профессиональный видеоредактор, разработанный для цветокоррекции, монтажа и обработки видео. Он включает в себя богатый инструментарий для редактирования, работы со звуком и визуальными эффектами, а также предлагает функции автоматической цветокоррекции и нодовую структуру для создания сложных визуальных эффектов.

Sony Vegas Pro

Sony Vegas Pro — это мощный видео и аудиоредактор, позволяющий создавать неограниченные по продолжительности материалы. Он включает в себя продвинутые инструменты для обработки звука, поддержку многоканального ввода-вывода и различные форматы файлов для экспорта готового ролика.

HitFilm Pro

HitFilm Pro — это профессиональный видеоредактор и композер, разработанный для режиссеров и создателей контента. Он объединя-

ет инструменты для реализации визуальных эффектов и традиционные функции нелинейного сбора видео из множества эпизодов, благодаря чему можно добиться качества голливудских фильмов.

Shotcut

Shotcut — это бесплатный инструмент, способный работать с видео высокого разрешения до 4K и поддерживающий все популярные форматы. С его помощью можно производить обрезку и склейку клипов, накладывать спецэффекты и эффекты переходов между эпизодами, обрабатывать аудио и проводить цветокоррекцию. Он также позволяет конвертировать видео в другие форматы и изменять разрешение, частоту кадров, кодек и качество изображения.

Camtasia

Camtasia — это ПО для записи видео с экрана, позволяющее определить область захвата, записать звук с микрофона или динамиков, добавить изображение с веб-камеры и редактировать финальный материал. Camtasia поддерживает большинство популярных форматов видеофайлов: AVI, SWF, FLV, MOV, WMV, RM, GIF и CAMV.

OBS Studio

OBS Studio (Open Broadcaster Software) — это популярный и бесплатный редактор для потокового видео, его записи и проведения онлайн-эфиров. Он подходит для операционных систем Windows, Linux и MacOS и использует открытый исходный код. OBS позволяет захватывать экран, веб-камеру и внешние устройства, настраивать аудио и видео, а также интегрироваться с популярными стриминговыми платформами.

Snagit

Snagit — это утилита для записи изображений, скриншотов и видео с экрана. Она доступна как на Windows, так и на MacOS. Кроме того, позволяет использовать фильтры для обработки захваченного материала. Редактор может отлично подойти для компиляции скринкастов и пакетного конвертирования изображений в различные форматы.

ScreenFlow

ScreenFlow — это профессиональный инструментарий для создания обучающих видео, презентаций и промороликов, предлагающий возможности не только захвата экрана, но также редактирования роликов и наложения на них эффектов. ScreenFlow совместим с MacOS и поддерживает все основные форматы файлов, в том числе MOV, MP4, GIF.

Вместе с тем, если вы используете операционную систему Windows 10 и выше, в ней есть встроенная программа для записи изображения с экрана монитора. Нажмите сочетание клавиш **Win+G**, после чего отобразится ее интерфейс. Он довольно прост и состоит всего лишь из нескольких функций для записи и сохранения. Выключается программа той же комбинацией.

14.6. Привлечение специалистов

В какой-то момент вам может показаться, что создавать самостоятельно графику, музыку, писать сценарий, делать видео и программировать слишком трудоемко для одного человека.

Поэтому отличным решением будет поискать компетентных специалистов в соответствующих группах в социальных сетях. Например, художников в группах по рисованию, изобразительному искусству, арту и подобных. Музыкантов — в группах, где собираются музыканты и т.д.

В таких сообществах можно найти начинающих специалистов, которые помогут вам создать спрайты, изображения, музыку бесплатно. Просто ради развлечения или для пополнения своего портфолио. Если вы располагаете определенным бюджетом, в тех же группах можно найти более профессиональных специалистов за небольшое вознаграждение.

Таким образом, можно найти единомышленников, объединиться в инди-команду и создавать свои произведения сообща. В группе каждый будет занят своим делом и результат будет более качественным и быстрым, чем при работе в одиночку.

Подобные группы можно найти в Discord, на форуме Reddit, на сайте VK и других.

Ну, а на этом все, надеюсь вам понравилось. С вами был Анатолий Адонин.

Мою страничку на сайте VK вы можете найти по следующему адресу: *vk.com/anzheri*

Успехов в творчестве!

"Издательство Наука и Техника" рекомендует:

Кольцов Д. М.

PYTHON. Создаем программы и игры. 3-е издание *— СПб.: "Издательство Наука и Техника" — 432 с., ил.*

Данная книга позволяет уже с первых шагов создавать свои программы на языке **Python**. Акцент сделан на написании компьютерных игр и небольших приложений. Для удобства начинающих пользователей в книге есть краткий вводный курс в основы языка, который поможет лучше ориентироваться на практике. По ходу изложения даются все необходимые пояснения, приводятся примеры, а все листинги (коды программ) сопровождаются подробными комментариями.

Книга будет полезна как начинающим программистам, так и всем, кто хочет быстро и эффективно научиться писать программы на Python.

Издательство «Наука и Техника» выпускает книги более 25 лет!

Уважаемые авторы!

Приглашаем к сотрудничеству по созданию книг
по **IT-технологиям, электронике, электротехнике, медицине, педагогике.**

Наши преимущества:

- являемся одним из ведущих технических издательств страны;
- выпускаем книги большими тиражами, что положительно влияет на гонорар авторов;
- регулярно переиздаем тиражи, автоматически выплачивая гонорар за *каждый* тираж;
- применяем индивидуальный подход в работе с каждым автором;
- работаем профессионально: от корректуры до авторских дизайн-проектов;
- проводим политику доступной цены;
- имеем собственные каналы сбыта: от федеральных сетей, крупнейших книжных магазинов РФ, ведущих маркетплейсов ОЗОН, Wildberries, Яндекс-Маркет и др. до ведущих библиотек вузов, ссузов.

Ждем Ваши предложения:

- **тел. (812) 412-70-26**
- **эл. почта: nitmail@nit.com.ru**

Будем рады сотрудничеству!

Для заказа книг:

- **интернет-магазин: nit.com.ru**
  - более 3000 пунктов выдачи на территории РФ, доставка 3–5 дней
  - более 300 пунктов выдачи в Санкт-Петербурге и Москве, доставка 1–2 дня
  - тел. (812) 412-70-26
  - эл. почта nitmail@nit.com.ru

- **магазин издательства:** г. Санкт-Петербург, пр. Обуховской обороны, д. 107
  - метро Елизаровская, 200 м за ДК им. Крупской
  - ежедневно с 10.00 до 18.00
  - справки и заказ: тел. (812) 412-70-26

- **книжные сети и магазины**
  - «Читай-город» - сеть магазинов тел. +7 (495) 424-84-44
  - «Буквоед» - сеть магазинов тел. +7 (812) 601-0-601
  - Московский дом книги – сеть магазинов тел. +7 (495) 789-35-91
  - ТД «БиблиоГлобус» тел. +7 (495) 781-19-12
  - «Амиталь» — сеть магазинов тел. +7 (473) 223-00-02
  - Дом книги, г. Екатеринбург тел. +7 (343) 289-40-45
  - Дом книги, г. Нижний Новгород тел. +7 (831) 246-22-92
  - Приморский торговый Дом книги тел. +7 (423) 263-10-54

- **маркетплейсы ОЗОН, Wildberries, Яндекс-Маркет, Myshop и др.**

Адонин А.

Ren'Py
Создаем игры и приложения

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *Е.В. Финков*

Редактор: *Н.В. Жерлов*

Корректор: *А.В. Громова*

Изображение на обложке использовано с ресурсов freepik.com и vecteezy.com

12+

ООО "Издательство Наука и Техника"

ОГРН 1217800116247, ИНН 7811763020, КПП 781101001

192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 107, лит. Б, пом. 1-Н

Подписано в печать 17.05.2024. Формат 70x100 1/16.

Бумага офсетная. Печать офсетная. Объем 25 п.л.

Тираж 2000. Заказ 9622.

Отпечатано с готового оригинал-макета

ООО «Принт-М», 142300, М.О., г.Чехов, ул. Полиграфистов, д.1

Ren'Py Создаем игры и приложения

Игровой движок **Ren'Py** (РенПи) – это один из простейших способов познакомиться с основами программирования. Благодаря этой книге, шаг за шагом создавая свою первую игру, вы познакомитесь с основами разработки игр. Функционал движка настолько прост, что первую свою игру вы сможете запустить уже спустя несколько первых глав.

Несмотря на всю простоту, **Ren'Py** позволяет создавать игры во многих жанрах, таких, как: RPG, Симуляторы, Квесты, Стратегии, Новеллы и даже Шутеры. Также на базе движка можно разработать мобильное приложение или программу на ПК.

Ren'Py базируется на языке программирования Python, поэтому многие функции движка, структура, синтаксис и логика схожи с Python, все это будет рассмотрено в книге.

Подробные пошаговые примеры (с разбором кода) в данной книге помогут в создании игр и приложений любому уровню пользователей, от самых начинающих до профессионалов.

издательство НАУКА и ТЕХНИКА рекомендует

ISBN 978-5-907592-50-6

«Издательство Наука и Техника» г. Санкт-Петербург
Для заказа книг: т. (812) 412-70-26
E-mail: nitmail@nit.com.ru
Сайт: nit.com.ru

nit.com.ru