unity

unity 5

Multiplatform Game Development in C# with Unity 5

Unity in Action

Multiplatform Game Development in C#

Joseph Hocking

Unity
в действии

Мультиплатформенная разработка на C#

Джозеф Хокинг

Оглавление

Часть II
Жизнь налаживается

Часть III
Уверенный финиш

Предисловие

Созданием игр я занялся в 1982 году. Это было непросто. Интернета в то время не было. Доступные ресурсы ограничивались небольшим количеством по большей части ужасных книг и журналов с интересными, но запутанными фрагментами кода, а игровых движков попросту не существовало! Написание кода для игр представляло собой гонку с огромным количеством препятствий.

Как я завидую тебе, читатель, держащий в руках эту глубоко информативную книгу! Инструмент под названием Unity дал огромному количеству людей возможность заняться программированием игр. В данном случае был достигнут идеальный баланс между мощностью профессионального игрового движка и его доступностью, так ценимой начинающими.

Но доступность в данном случае достигается только при правильном обучении. Мне довелось поработать в цирковой труппе, которой руководил фокусник. Он был столь любезен, что взял меня на работу и помог стать хорошим артистом. «Стоя на сцене, — говорил он, — ты даешь обещание. Ты обещаешь, что не будешь понапрасну тратить время зрителей».

В этой книге мне больше всего нравится практическая часть. Джо Хокинг не тратит ваше время понапрасну и быстро переходит к написанию кода, причем не каких-то бессмысленных фрагментов, а кода, который вы можете понять и использовать в своих целях. Ведь он знает, что вы не просто хотите прочитать эту книгу и проверить, как работают приведенные им примеры, — вашей целью является *программирование собственных игр*.

И благодаря его указаниям вы научитесь это делать быстрее, чем можно было бы ожидать. Следуйте тропой, которую проложил для вас Джо, но, как только ощутите в себе силы, не колеблясь, уходите с нее и прокладывайте собственную дорогу. Сразу переходите к наиболее интересным вам темам, экспериментируйте, будьте смелым и храбрым! Почувствовав, что заблудились, вы в любой момент сможете вернуться к тексту книги.

Впрочем, довольно вступительных слов — вас с нетерпением ждет будущая карьера разработчика игр! Запомните этот день, ведь именно он изменит вашу жизнь. Именно сегодня вы начали создавать игры.

Джесси Шелл (Jesse Schell),
руководитель фирмы Schell Games,
автор книги Art of game design

Введение

Хотя программированием игр я занимаюсь уже много лет, с Unity я познакомился относительно недавно. Разработку игр я начал осваивать во времена, когда такого инструмента попросту не существовало; его первая версия появилась только в 2005 году. С самого начала это было многообещающее средство разработки, но всеобщее признание оно получило только после выхода нескольких версий. В частности, такие платформы, как iOS и Android (называемые «мобильными»), появились не так давно, и именно они повлияли на рост популярности Unity.

Изначально я смотрел на Unity как на диковинку, интересный инструмент разработки, за развитием которого имеет смысл понаблюдать. В то время я писал игры для настольных компьютеров и сайтов, выполняя проекты для широкого круга клиентов. Я пользовался такими инструментами, как Blitz3D и Flash, великолепно подходящими для программирования, но ограниченными в ряде других аспектов. Но по мере их устаревания я стал искать более совершенные средства разработки игр.

Я начал экспериментировать с Unity версии 3, а после того как компания Synapse Games (в которой я работаю в данный момент) начала разрабатывать игры для мобильных устройств, полностью переключился на этот инструмент. Изначально в фирме Synapse я занимался интернет-играми, но в итоге перешел на разработку игр для мобильных устройств. А затем все вернулось на круги своя, так как инструмент Unity позволял выполнять развертывание одного и того же базового кода как на мобильных устройствах, так и в Интернете!

Я всегда понимал важность передачи знаний, поэтому последние годы занялся обучением разработки игр. Во многом это обусловлено теми примерами, которые подавали мне мои наставники и учителя. Кстати, возможно, об одном из них вы слышали, потому что это потрясающий человек — Рэнди Пауш (Randy Pausch), незадолго до своей кончины выступивший с «Последней общественной лекцией». Я преподавал в нескольких школах и всегда хотел написать книгу, посвященную разработке игр.

Эта книга во многом напоминает другую, о которой я мечтал во времена, когда только начал осваивать Unity. К многочисленным достоинствам Unity можно отнести огромное количество обучающих ресурсов, но, как правило, они представляют собой наборы отдельных фрагментов (таких как ссылки на сценарии или не связанные друг с другом уроки), соответственно, поиск нужного материала затруднен. Мне же в те времена хотелось получить книгу, объединяющую в себе все необходимые знания, представленные в четкой и логически связанной манере, поэтому я написал такую

книгу для вас. Моей целевой аудиторией являются люди, уже имеющие навыки программирования, но не обладающие опытом работы с Unity и, возможно, никогда не занимавшиеся разработкой игр. Выбор проектов отражает мой опыт наработки навыков и уверенности в себе, который я в свое время получил, выполняя различные варианты заказов в достаточно быстром темпе.

Приступая к изучению процесса разработки игр с помощью Unity, вы начинаете захватывающее приключение. Для меня этот процесс подразумевал необходимость не вешать нос перед лицом многочисленных препятствий. У вас же есть преимущество в виде единого логически согласованного ресурса — этой книги!

Благодарности

Я хотел бы поблагодарить издательство Manning Publications, предоставившее мне возможность написать эту книгу. В этом мне помогли редакторы, с которыми я работал, в том числе Робин де Йон (Robin de Jongh) и особенно Дэн Махари (Dan Maharry). Именно взаимодействие с ними сделало книгу намного лучше. Мои искренние благодарности и многим другим людям, сотрудничавшим со мной в процессе написания и издания.

От пристального внимания рецензентов на всем протяжении работы над книгой она только выиграла. Спасибо Алексу Лукасу (Alex Lucas), Крейгу Хоффману (Craig Hoffman), Дэну Кэйсенджару (Dan Kacenjar), Джошуа Фредерику (Joshua Frederick), Люке Кампобассо (Luca Campobasso), Марку Элстону (Mark Elston), Филиппу Таффету (Philip Taffet), Рене ван ден Бергу (René van den Berg), Серджио Арбео Родригесу (Sergio Arbeo Rodríguez), Шайло Моррису (Shiloh Morris) и Виктору М. Пересу (Victor M. Perez). Отдельная благодарность техническому редактору Скотту Шоси (Scott Chaussee) и техническому корректору Кристоферу Хопту (Christopher Haupt). Также хотелось бы поблагодарить Джесси Шелл (Jesse Schell) за предисловие к книге.

Затем я хотел бы выразить свою признательность людям, помощь которых сделала мой опыт работы с Unity крайне плодотворным. Разумеется, этот список начинается с создавшей Unity (игровой движок) компании Unity Technologies. Чувство глубокой благодарности я испытываю и к сообществу gamedev.stackexchange.com. Этот сайт контроля качества я посещал практически ежедневно, учась у других и отвечая на вопросы других. Самый же сильный толчок к работе с Unity я получил от своего руководителя в фирме Synapse Games Алекса Рива (Alex Reeve). Мои коллеги показали мне множество приемов и методов, которые я постоянно применяю при написании кода.

Наконец, я хотел бы поблагодарить мою жену Виргинию за ту поддержку, которая мне оказывалась во время написания книги. До начала работы я понятия не имел, насколько подобные проекты переворачивают твою жизнь, влияя на все ее аспекты. Спасибо ей за ее любовь и помощь.

О книге

Эта книга посвящена программированию игр с помощью Unity. Ее можно считать введением в Unity для опытных программистов. Цель книги крайне проста: научить людей, имеющих опыт программирования, но ни разу не сталкивавшихся с Unity, разрабатывать игры с помощью этого инструмента.

Учить разработке лучше всего на примерах проектов, заставляя обучающихся выполнять практические задания, и именно такой подход используется в данном случае. Темы представлены как этапы построения отдельных игр, и я настоятельно рекомендую вам в процессе знакомства с книгой заняться разработкой этих игр при помощи Unity. Мы рассмотрим ряд проектов, каждому из которых посвящено несколько глав. Бывают книги, целиком посвященные одному крупному проекту, но такой подход исключает возможность чтения с середины, если информация в первых главах покажется вам неинтересной.

В этой книге более строго, чем в большинстве других изданий (особенно предназначенных для начинающих), изложен материал, касающийся программирования. Unity зачастую представляют как набор компонентов, не требующих программирования, что в корне неверно, так как не дает людям знаний, без которых невозможно производство коммерчески успешных продуктов. Если вы пока не имеете навыков программирования, советую сначала их приобрести и только после этого приступать к чтению.

Выбор языка программирования не имеет особого значения; все примеры в книге написаны на C#, но они легко переводятся на другие языки. Первая половина книги в изрядной степени посвящена знакомству с новыми понятиями, и первые шаги по разработке игры с помощью Unity намеренно описаны со всей возможной тщательностью, но затем повествование ускоряется, давая читателям возможность выполнять проекты в различных игровых жанрах. Завершает книгу описание развертывания игр на различных платформах, но в целом мы не будем делать упор на этом аспекте, так как Unity не зависит от платформы.

Что касается прочих аспектов разработки игр, излишне широкий охват различных художественных дисциплин привел бы к сокращению объема представленного в книге конкретного материала по Unity и в значительной степени относился бы к внешним по отношению к Unity программам (например, программам создания анимации). Поэтому обсуждение художественных дисциплин сводится к тем аспектам, которые имеют непосредственное отношение к Unity или должны быть известны всем разработчикам игр. (Впрочем, одно из приложений посвящено моделированию собственных нестандартных объектов.)

Перспективы

Глава 1 знакомит вас с Unity — межплатформенной средой разработки игр. Вы освоите базовую систему компонентов, лежащую в основе Unity, а также научитесь писать и выполнять базовые сценарии.

В главе 2 мы перейдем к написанию программы, демонстрирующей движение в трехмерном пространстве, попутно рассмотрев такие темы, как ввод с помощью мыши и клавиатуры. Детально объясняется определение положения объектов в трехмерном пространстве и операции их поворота.

В главе 3 мы превратим демонстрационную программу в шутер от первого лица, познакомив вас с методом испускания луча и основами искусственного интеллекта. Испускание луча (мы создаем в сцене линию и смотрим, с чем она пересечется) требуется во всех вариантах игр.

Глава 4 посвящена импорту и созданию игровых ресурсов. Это единственная глава в книге, в которой код не играет центральной роли, так как каждому проекту требуются (базовые) модели и текстуры.

Глава 5 научит вас создавать в Unity двухмерные игры. Хотя изначально этот инструмент предназначался исключительно для создания трехмерной графики, сейчас в нем прекрасно поддерживается двухмерная графика.

Глава 6 знакомит с новейшей GUI-функциональностью в Unity. Пользовательский интерфейс требуется всем играм, а последние версии Unity могут похвастаться улучшенной системой создания пользовательского интерфейса.

В главе 7 мы создадим еще одну программу, демонстрирующую движение в трехмерном пространстве, однако на этот раз с точки зрения стороннего наблюдателя. Реализация элементов управления третьим лицом даст вам представление о ключевых математических операциях в трехмерном пространстве, кроме того, вы научитесь работать с анимированными персонажами.

Глава 8 покажет способы реализации интерактивных устройств и элементов в игре. У игрока будет ряд способов применения этих устройств, в том числе прямым касанием, прикосновением к пусковым устройствам внутри игры или нажатием кнопки контроллера.

Глава 9 учит взаимодействию со Всемирной паутиной. Вы узнаете, как отправить и получить данные с помощью стандартных технологий, таких как HTTP-запросы на получение с сервера XML-данных.

В главе 10 вы научитесь добавлять в игры звук. В Unity замечательно поддерживаются как короткие звуковые эффекты, так и долгие музыкальные фонограммы; оба варианта звукового сопровождения критически важны почти для всех видеоигр.

В главе 11 мы соберем воедино фрагменты из различных глав, чтобы получить в итоге одну игру. Кроме того, вы научитесь программировать элементы управления, манипуляция которыми осуществляется с помощью мыши, и сохранять игру.

Глава 12 демонстрирует процесс создания итогового приложения с его последующим развертыванием на различных платформах, таких как настольные компьютеры,

Интернет и мобильные устройства. Unity обладает поразительной независимостью от конкретной платформы, позволяя создавать любые варианты игр!

Затем идут три приложения с дополнительной информацией о навигации по сцене, внешних инструментах и пакете Blender.

Условные обозначения, требования и доступные для скачивания данные

Весь код в этой книге, как в листингах, так и в представленных фрагментах, набран `вот таким шрифтом фиксированной ширины`, позволяющим отличить код от остального текста. Большинство листингов снабжено примечаниями, отмечающими ключевые понятия, кроме того, в тексте периодически встречаются маркированные списки с дополнительными сведениями по поводу кода. Код отформатирован при помощи переносов строки и аккуратных отступов в соответствии с шириной страницы.

Единственной программой, которая вам потребуется, является Unity; в книге используется версия Unity 5.0, которая являлась самой последней на момент написания этого текста. В некоторых главах время от времени обсуждаются другие программы, но это всегда дополнительная информация, не оказывающая решающего влияния на изучение основного материала.

ВНИМАНИЕ В Unity-проектах сохраняется информация о том, в какой версии программы они были созданы, и при попытке открыть их в другой версии выводят предупреждение. Если, открыв относящиеся к этой книге материалы, скачанные из Сети, вы увидите такое предупреждение, просто щелкните на кнопке Continue.

Встречающиеся в книге фрагменты кода в общем случае демонстрируют добавления и изменения, вносимые в существующие файлы; если это не первое появление файла с кодом, не следует заменять файл соответствующим листингом. Можно просто скачать рабочий проект целиком, но обучение пойдет намного быстрее, если вы будете набирать все листинги вручную, используя примеры из проекта только для сравнения. Весь код доступен для скачивания на сайте издателя по адресу www.manning.com/UnityinAction.

Автор в Интернете

На странице www.manning.com/UnityinAction вы найдете информацию о том, как попасть на форум после регистрации, на какую помощь вы можете рассчитывать, а также о правилах поведения на форуме.

Издательство Manning взяло на себя обязательство по предоставлению места, где читатели могут конструктивно пообщаться как друг с другом, так и с автором книги. Но оно не может гарантировать присутствия на форуме автора, участие которого в обсуждениях является добровольным (и неоплачиваемым). Мы надеемся, что вы будете задавать автору по-настоящему трудные вопросы, чтобы его интерес к общению не угас!

Как форум, так и архивы предшествующих обсуждений будут доступны на сайте издательства до тех пор, пока книга находится в продаже.

Об авторе

Джозеф Хокинг живет в Чикаго и занимается разработкой программного обеспечения для интерактивных сред. Он работает в фирме Synapse Games, создавая интернет-игры и игры для мобильных устройств, такие как недавно вышедшая игра-стратегия Tyrant Unleashed. Кроме того, он преподает предмет разработки игр в колледже Колумбия в Чикаго. Его сайт www.newarteest.com.

Часть I
ПЕРВЫЕ ШАГИ

Итак, пришло время познакомиться с Unity. Ничего страшного, если вы не имеете представления о том, что это за инструмент! Я начну с его подробного описания, рассказывая в числе прочего об основах программирования игр с помощью Unity. Затем вы получите инструкцию по разработке простой игры. Первый проект познакомит вас с набором приемов, попутно дав представление об общем ходе процесса.

Приступим к главе 1!

1 Знакомство с Unity

- ✓ Почему следует выбрать Unity
- ✓ Как работает редактор Unity
- ✓ Программирование в Unity
- ✓ Сравнение языков C# и JavaScript

Возможно, вы, как и я, мысленно давным-давно разрабатываете видеоигру. Но переход из лагеря игроков в лагерь создателей игр — это большой прыжок вперед. За последние годы появилось множество инструментов для разработки игр, и мы обсудим один из самых новых и мощных представителей этого семейства. Приложение Unity представляет собой профессиональный игровой движок, который используется в создании видеоигр для различных платформ. Это инструмент, которым ежедневно пользуются опытные разработчики, а также один из наиболее доступных инструментов для новичков. До недавнего времени человек, решивший научиться программированию игр (особенно трехмерных), сразу же сталкивался с множеством серьезных препятствий, в то время как инструмент Unity позволил значительно облегчить жизнь новичкам.

Раз вы читаете эту книгу, значит, интересуетесь компьютерными технологиями и либо разрабатывали игры с помощью других инструментов, либо писали программное обеспечение (ПО) других типов, например приложения для рабочего стола или веб-сайты. Создание видеоигр в своей основе не отличается от написания любого другого ПО; по большей части различия проявляются в количественной плоскости. К примеру, игра намного более интерактивна, чем большинство веб-сайтов, а значит, вам потребуется совсем другой тип кода, но при этом в обоих случаях будут задействованы сходные навыки и процессы. Если вы уже преодолели первое препятствие на пути к карьере разработчика игр, то есть изучили основы написания ПО, следующим вашим шагом станет выбор инструмента и приобретение специализированных

навыков программирования. В этом смысле Unity представляет собой замечательную среду разработки игр.

ПРЕДУПРЕЖДЕНИЕ ПО ПОВОДУ ТЕРМИНОЛОГИИ

Эта книга посвящена программированию в Unity и поэтому будет интересна в основном кодерам. Существует множество ресурсов, где обсуждаются другие аспекты разработки игр вообще и в Unity в частности, но в нашем случае основной упор делается именно на программировании.

Хотелось бы обратить ваше внимание на то, что в контексте разработки игр слово «разработчик» имеет несколько отличный от привычного нам смысл. В таких областях, как веб-разработка, это синоним слова «программист», в то время как в данном случае разработчиком называется любой человек, работающий над созданием игры, а программистом называется разработчик, выполняющий конкретные обязанности. К разработчикам игр относятся также художники и дизайнеры, но в рамках этой книги рассматривается только работа программистов.

Первым делом вам нужно зайти на сайт www.unity3d.com и скачать саму программу. Я пользовался версией Unity 5.0, которая на момент написания этой книги была наиболее новой. В URL-адресе отражен тот факт, что изначально инструмент Unity предназначался для создания трехмерных игр; их поддержка по-прежнему остается главной, но теперь Unity замечательно подходит и для разработки двухмерных игр. В платной версии программы доступны расширенные функциональные возможности, но базовая версия распространяется бесплатно. Все примеры из этой книги прекрасно работают в бесплатной версии, так что Unity Pro вам не потребуется. Разница между этими версиями состоит в уже упомянутых мной расширенных функциональных возможностях (рассмотрение которых выходит за рамки темы данного издания) и коммерческих условиях лицензирования.

1.1. Достоинства Unity

Посмотрим более внимательно на данное в начале этой главы определение: Unity — профессиональный игровой движок, который используется при создании видеоигр для различных платформ. Это достаточно прямой ответ на прямой вопрос «Что такое Unity?». Но что конкретно он означает? И чем примечателен инструмент Unity?

1.1.1. Сильные стороны и преимущества Unity

Любой игровой движок предоставляет множество функциональных возможностей, которые задействуются в различных играх. Реализованная на этом движке игра получает все эти функциональные возможности, кроме того, добавляются ее собственные игровые ресурсы и код игрового сценария. Unity предлагает моделирование физических сред, карты нормалей, преграждение окружающего света в экранном пространстве (Screen Space Ambient Occlusion, SSAO), динамические тени... список можно продолжать долго. Подобным набором функциональных возможностей могут похвастаться многие игровые движки, но у Unity есть два основных преимущества перед другими передовыми инструментами разработки игр: чрезвычайно производительный визуальный рабочий процесс и мощная межплатформенная поддержка.

Визуальный рабочий процесс представляет собой достаточно уникальную вещь, выделяющую данный инструмент из большинства других сред разработки игр. В то время как остальные инструменты разработки игр зачастую представляют собой мешанину разрозненных частей, которые требуется контролировать, или, возможно, библиотеку, для работы с которой нужно настраивать собственную интегрированную среду разработки (Integrated Development Environment, IDE), цепочку сборки и прочее в этом роде, рабочий процесс в Unity привязан к тщательно продуманному визуальному редактору. В этом редакторе вы будете компоновать сцены будущей игры, связывая игровые ресурсы и код в интерактивные объекты. Именно он позволяет быстро и рационально создавать профессиональные игры, обеспечивая невиданную продуктивность труда разработчиков и предоставляя в их распоряжение исчерпывающий перечень самых современных технологий в области видеоигр.

ПРИМЕЧАНИЕ Большинство других инструментов, оснащенных центральным визуальным редактором, страдают от ограниченной и недостаточно гибкой поддержки возможности написания сценариев, но инструмент Unity лишен этого недостатка. Несмотря на то что все создаваемое для Unity в конечном счете проходит через визуальный редактор, основной интерфейс включает в себя множество связанных проектов с нестандартным кодом, запускаемым в игровом движке Unity. Это своего рода аналог связывания классов в параметрах проекта для таких интегрированных сред разработки, как Visual Studio или Eclipse. Поэтому опытные программисты не должны пренебрегать средой Unity, считая, что это чисто визуальный инструмент создания игр с ограниченной возможностью программирования!

Особенно полезен этот редактор при разработке проектов с последовательным улучшением, например в циклах создания прототипов или тестирования. Корректировать объекты в редакторе и двигать элементы в сцене можно даже при запущенной игре. Кроме того, Unity позволяет настраивать и сам редактор при помощи сценариев, добавляющих новые функциональные особенности и элементы меню к интерфейсу.

В дополнение к значительным преимуществам в плане производительности, которые нам дает редактор, у набора инструментов Unity существует еще и сильная межплатформенная поддержка. В данном случае под этим словосочетанием подразумеваются не только места развертывания (вы можете развернуть игру на персональном компьютере, в Интернете, на мобильном устройстве или на консоли), но и инструменты разработки (создание игры может осуществляться на машинах, работающих под управлением как Windows, так и Mac OS). Эта независимость от платформы явилась результатом того, что изначально инструмент Unity предназначался исключительно для компьютеров Mac, а позднее был перенесен на машины с операционными системами семейства Windows. Первая версия появилась в 2005 году, но к настоящему моменту вышли уже пять основных версий (с множеством небольших, но частых обновлений). Изначально инструмент Unity поддерживал разработку и развертывание только для машин Mac, но через несколько месяцев вышло обновление, позволяющее работать и на машинах с Windows. В следующих версиях постепенно добавлялись все новые платформы развертывания, например: межплатформенный веб-плеер в 2006-м, iPhone в 2008-м, Android в 2010-м и даже такие игровые консоли, как Xbox и PlayStation. Совсем недавно появилась возможность развертывания в WebGL — новом фреймворке для трехмерной графики в веб-браузерах. Найдется немного игровых движков,

поддерживающих такое количество целевых платформ развертывания, и ни один из них не делает операцию развертывания на разных платформах настолько простой.

Между тем в дополнение к этим основным достоинствам идет и третье, менее бросающееся в глаза преимущество, обеспечиваемое модульной системой компонентов, которая используется для конструирования игровых объектов. «Компоненты» в такой системе представляют собой комбинируемые пакеты функциональных элементов, поэтому объекты создаются как наборы компонентов, а не как жесткая иерархия классов. Другими словами, компонентная система являет собой альтернативный (и обычно более гибкий) подход к объектно-ориентированному программированию, в котором игровые объекты создаются путем объединения, а не наследования. Сравнение подходов демонстрирует диаграмма на рис. 1.1.

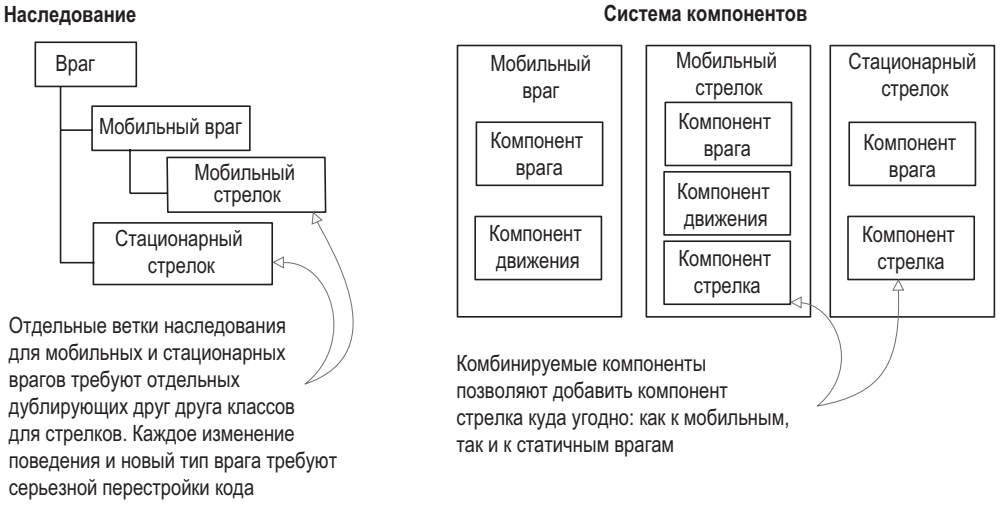


Рис. 1.1. Сравнение наследования и компонентной системы

В компонентной системе объект существует в горизонтальной иерархии, поэтому различные объекты состоят из разных наборов компонентов, а не из структуры наследования, в которой разные объекты оказываются на разных ветвях дерева. Такая компоновка облегчает создание прототипов, потому что взять нужный набор компонентов куда быстрее и проще, чем перестраивать цепочку наследования при изменении каждого объекта.

Разумеется, ничто не мешает вам написать код, реализующий вашу собственную компонентную систему, но в Unity уже существует вполне надежный вариант такой системы, органично встроенный в визуальный редактор. Поэтому у вас есть возможность не только программно управлять компонентами, но и устанавливать и разрывать связи между ними в редакторе. Разумеется, ваши возможности не ограничиваются составлением объектов из готовых деталей; в своем коде вы можете воспользоваться наследованием и всеми наработанными на его базе паттернами проектирования.

1.1.2. Недостатки, о которых нужно знать

Инструмент Unity имеет множество достоинств, которые превращают его в замечательное средство разработки игр, но с моей стороны было бы упущением не упомянуть о его недостатках. В частности, сочетание визуального редактора со сложным кодом при всей его эффективности в рамках компонентной системы Unity является нетипичным и может вызвать затруднения. В сложных сценах можно потерять из виду некоторые из присоединенных компонентов. В Unity существует функция поиска, позволяющая обнаруживать присоединенные сценарии, но она могла бы быть и более надежной — порой возникают ситуации, когда для поиска связанных сценариев приходится вручную просматривать все элементы сцены. Такое случается не часто, тем не менее этой кропотливой и трудоемкой работы хотелось бы совсем избежать.

Другим неожиданным и обескураживающим для опытных программистов недостатком является тот факт, что Unity не поддерживает ссылки на внешние библиотеки кода. Все доступные библиотеки, которые вы планируете задействовать, следует вручную скопировать в проект, вместо того чтобы просто дать ссылку на одну папку общего доступа. Отсутствие единой папки с библиотеками затрудняет коллективное использование функционала разными проектами. Это неудобство можно обойти, рационально применяя системы контроля версий, но готовое решение данной проблемы в Unity отсутствует.

ПРИМЕЧАНИЕ Раньше существенным недостатком была сложность работы с системами контроля версий (такими, как Subversion, Git и Mercurial), но в более поздних версиях Unity все значительно упрощено. Вы можете наткнуться на устаревшие ресурсы, где утверждается, что Unity не работает с системами контроля версий, но на более новых ресурсах есть описание файлов .meta (механизма, который появился в Unity для работы с этими системами) и указания, какие папки проекта следует поместить в репозиторий. Начать лучше всего с чтения страницы документации http://docs.unity3d.com/ru/current/Manual/ExternalVersionControlSystemSupport.html.

Третий недостаток связан с использованием шаблонов экземпляров (prefabs). Эта концепция детально объясняется в главе 3; пока вам достаточно знать, что шаблоны экземпляров предлагают гибкий подход к визуальному созданию интерактивных объектов. Эта крайне мощная концепция существует исключительно в Unity (и, естественно, она связана с компонентной системой Unity), но редактирование таких шаблонов порой оказывается на удивление труднореализуемым. Я считаю шаблоны экземпляров полезным и важным аспектом работы с Unity и надеюсь, что в будущих версиях способ их редактирования будет усовершенствован.

1.1.3. Примеры игр на основе Unity

Итак, вы познакомились с сильными и слабыми сторонами Unity, но, возможно, пока до конца не уверены в том, что с помощью данного инструмента разработки можно получить первоклассные результаты. Зайдите в галерею Unity на странице http://unity3d.com/ru/showcase/gallery и полюбуйтесь постоянно обновляемым списком сотен игр и симуляций, созданных этим инструментом. В данном разделе перечислено небольшое количество игр, демонстрирующих разные жанры и платформы развертывания.

Игры для рабочего стола (WINDOWS, MAC, LINUX)

Так как редактор работает на одной платформе, зачастую наиболее очевидной платформой развертывания становится машина с операционной системой семейства Windows или Mac. Вот пара примеров игр для рабочего стола в различных жанрах:

- Guns of Icarus Online (рис. 1.2) — шутер от первого лица, созданной независимым разработчиком Muse Games.
- Gone Home (рис. 1.3) — квест от первого лица, разработанный независимой студией Fullbright Company.

Рис. 1.2. Guns of Icarus Online

Рис. 1.3. Gone Home

Игры для мобильных устройств (IOS, ANDROID)

Unity позволяет развертывать игры на мобильных платформах, таких как iOS (смартфонах iPhone и планшетах iPad) и Android (смартфонах и планшетах). Вот примеры мобильных игр в различных жанрах:

- Dead Trigger (рис. 1.4) — шутер от первого лица, созданный разработчиком Madfinger Games.
- Bad Piggies (рис. 1.5) — игра-головоломка от финской компании Rovio.

Рис. 1.4. Dead Trigger

Рис. 1.5. Bad Piggies

- Tyrant Unleashed (рис. 1.6) — коллекционная карточная игра, созданная студией Synapse Games.

Рис. 1.6. Tyrant Unleashed

Игры для консолей (PLAYSTATION, XBOX, WII)

Unity поддерживает развертывание на игровых консолях, хотя разработчику потребуется лицензия от Sony, Microsoft или Nintendo. Из-за этого требования и возможности межплатформенного развертывания консольные игры часто доступны и на обычных компьютерах. Вот примеры таких игр в различных жанрах:

- Assault Android Cactus (рис. 1.7) — аркадный шутер от независимого разработчика Witch Beam.
- The Golf Club (рис. 1.8) — спортивный симулятор, созданный фирмой HB Studios.

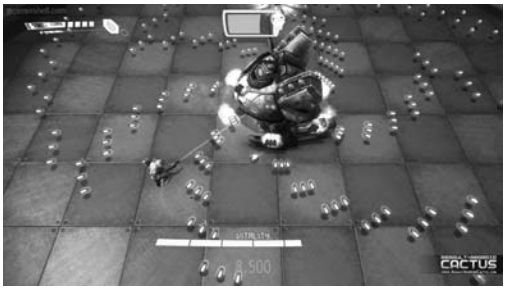


Рис. 1.7. Assault Android Cactus

Рис. 1.8. The Golf Club

Как видно из этих примеров, Unity позволяет создавать игры хорошего товарного качества. Но даже при наличии значительных преимуществ Unity над другими инструментами разработки игр новички зачастую неверно понимают важность программирования. Зачастую Unity описывают как простой набор готовых программных компонентов, для использования которых программирование вообще не требуется. Но это неверная точка зрения, не дающая людям представления о том, как создать коммерчески успешный продукт. Нет, вы, конечно, можете создать из готовых компонентов интересный прототип, пользуясь одной только мышью, но перейти от прототипа к игре, готовой увидеть свет, без программирования не удастся.

1.2. Как работать с Unity

В предыдущем разделе мы много говорили о том, как выгоден встроенный в Unity визуальный редактор с точки зрения производительности, а сейчас пришло время познакомиться с его интерфейсом и узнать, как он работает. Если вы пока этого не сделали, скачайте программу со страницы http://unity3d.com/ru/get-unity и установите на свой компьютер (обязательно установите флажок Example Project, если установщик его сбросит). После этого запустите Unity, чтобы приступить к исследованию интерфейса.

Вам, скорее всего, потребуется пример, поэтому откройте соответствующий проект; при установке новой копии этот проект должен предлагаться автоматически, но вы можете выбрать в меню File команду Open Project и открыть его вручную. Он находится в пользовательской папке общего доступа, адрес которой в операционных системах семейства Windows выглядит примерно так: C:\Users\Public\Documents\Unity Projects\, а в Mac OS — так: Users/Shared/Unity/. Если заодно потребуется открыть пример сцены, дважды щелкните на файле Car (рис. 1.9 демонстрирует, что такие файлы в Unity обозначаются символом куба). Значок этого файла в средстве просмотра файлов, расположенном в нижней части редактора, находится по адресу SampleScenes/Scenes/. Вы должны получить экран, показанный на рис. 1.9.

Рис. 1.9. Части интерфейса Unity

Интерфейс Unity разбит на несколько частей: вкладка Scene, вкладка Game, панель инструментов, вкладка Hierarchy, панель Inspector, вкладки Project и Console. У каждой

части есть собственное предназначение, при этом все они играют важную роль в цикле создания игры:

- Просмотр файлов выполняется на вкладке Project.
- Помещенные в трехмерную сцену объекты просматриваются на вкладке Scene.
- Панель инструментов предоставляет вам элементы управления сценой.
- Менять взаимосвязи между объектами можно методом перетаскивания на вкладке Hierarchy.
- Панель Inspector отображает информацию о выделенных объектах, в том числе и о связанном с ними коде.
- Тестировать полученные результаты можно на вкладке Game, одновременно просматривая сообщения об ошибках на вкладке Console.

Эта компоновка предлагается по умолчанию; все доступные представления помещены на вкладки, которые можно перемещать, можно менять их размер и фиксировать в разных частях экрана. Чуть позже вы поэкспериментируете с выбором компоновки, пока же нам нужно понять назначение каждого элемента интерфейса, поэтому вариант, предлагаемый по умолчанию, является оптимальным.

1.2.1. Вкладка Scene, вкладка Game и панель инструментов

Наиболее заметной частью интерфейса является расположенная в центре вкладка Scene. Именно здесь можно видеть, как выглядит мир игры, и перемещать объекты в сцене. Сеточные объекты в сцене выглядят, как им и положено, в виде сеток. Также можно наблюдать и ряд других объектов, представленных различными значками и цветными линиями. Это камеры, источники света, источники звука, области столкновений и т. п. Разумеется, наблюдаемая тут картинка отличается от того, что вы будете видеть в процессе игры, — можно рассматривать сцену, не ограничиваясь игровым представлением.

ОПРЕДЕЛЕНИЕ Сеточный объект (mesh object) — это визуализация объекта в трехмерном пространстве. Она создается из набора соединенных друг с другом линий и форм, которые формируют сетку.

Игровое представление отображается не на отдельной части экрана, а на вкладке Game, расположенной рядом с вкладкой Scene (переход с вкладки на вкладку осуществляется с помощью кнопок в верхнем левом углу области отображения). В интерфейсе есть и другие элементы, сконструированные подобным образом; для изменения отображаемого ими содержимого достаточно перейти на другую вкладку. После запуска игры начинает отображаться игровое представление, то есть вам не нужно переходить на вкладку Game — переключение выполняется автоматически.

СОВЕТ В режиме воспроизведения игры можно вернуться на вкладку Scene для просмотра объектов. Это крайне полезная возможность, позволяющая понять, как выглядит воспроизводимая игра изнутри и что в ней происходит. Подобный инструмент отладки отсутствует в большинстве игровых движков.

Для запуска игры достаточно щелкнуть на кнопке Play, расположенной над вкладкой Scene. Вся верхняя часть интерфейса занята так называемой панелью инструментов,

и кнопка Play находится как раз в центре этой панели. Рисунок 1.10 получен отбрасыванием остальной части интерфейса редактора и демонстрирует только панель инструментов с расположенными под ней вкладками Scene и Game.

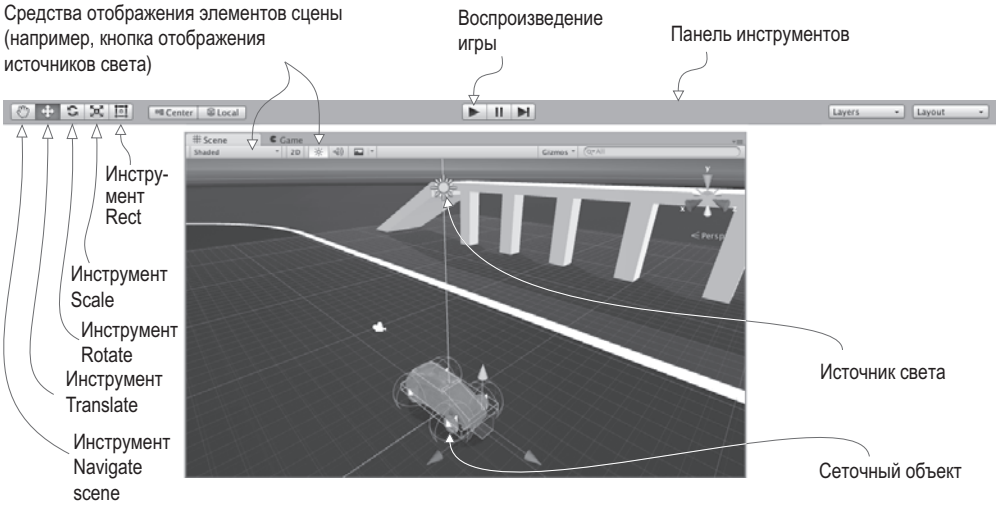


Рис. 1.10. Усеченный вариант редактора, демонстрирующий только панель инструментов и вкладки Scene и Game

На левой стороне панели инструментов расположены кнопки навигации и преобразования объектов. Они позволяют рассматривать сцену с разных сторон и перемещать объекты. Предлагаю вам попрактиковаться в их применении, так как просмотр сцен и перемещение объектов — это два основных действия, выполняемых в визуальном редакторе Unity (они настолько важны, что я посвятил им отдельный раздел). Правую сторону панели инструментов занимают раскрывающиеся меню с перечнем компоновок и слоев. Как я уже упоминал, интерфейс Unity является гибкой, настраиваемой системой, поэтому и существует меню Layouts, позволяющее переходить от одного варианта компоновки к другому. Меню Layers относится к расширенным функциональным возможностям, которые мы пока рассматривать не будем (о слоях речь пойдет в следующих главах).

1.2.2. Работа с мышью и клавиатурой

Навигация в сцене осуществляется в основном с помощью мыши и набора клавиш-модификаторов, влияющих на результат манипуляций мышью. Тремя главными операциями являются перемещение (move), облет (orbit) и масштабирование (zoom). Действия с мышью для совершения каждой из этих операций описаны в приложении А в конце книги, так как они зависят от типа мыши. Но в основном они сводятся к щелчкам и перетаскиванию при нажатых и удерживаемых клавишах Alt (или Option на компьютерах Mac) и Ctrl. Потратьте некоторое время на манипуляции объектами сцены, чтобы понять, как выполняются перемещение, облет и масштабирование.

СОВЕТ Хотя в Unity вполне можно работать с двухкнопочной мышью, рекомендую вам приобрести мышь с тремя кнопками (не сомневайтесь, в Mac OS X она тоже работает).

Преобразование объектов также осуществляется посредством этих трех операций. Более того, каждому типу навигации соответствует собственное преобразование: перенос (translate), поворот (rotate) и изменение размеров (scale). Рисунок 1.11 демонстрирует эти преобразования на примере куба.

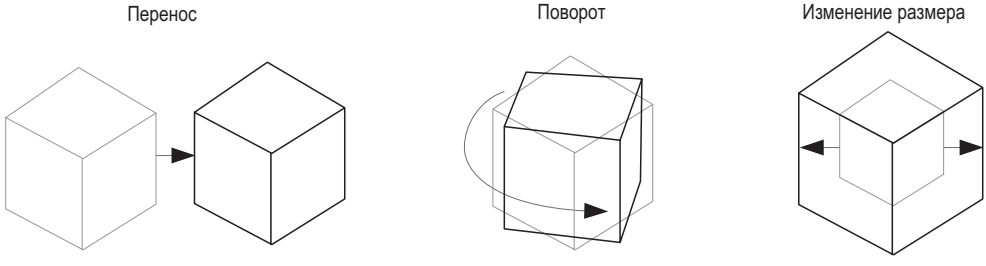


Рис. 1.11. Применение трех вариантов преобразования: переноса, поворота и изменения размеров (более светлые линии обозначают исходное состояние объекта)

После выделения объекта сцены появляется возможность двигать его (или, если брать более точный термин, переносить), вращать и указывать его размер. Если посмотреть с этой точки зрения на процесс навигации в сцене, то перемещение означает перенос камеры, облет соответствует повороту камеры, а масштабирование — изменению размеров камеры. Переход между этими операциями осуществляется не только кнопками панели инструментов, но и нажатием клавиш W, E и R. При входе в режим преобразования у выделенного объекта появляются цветные стрелки или окружности. Это габаритный контейнер преобразования (transform gizmo), перетаскиванием которого вы меняете вид объекта.

Рядом с кнопками преобразований находится еще одна кнопка. Это кнопка инструмента Rect, позволяющего перейти к работе с двухмерной графикой и объединяющего в себе операции переноса, поворота и изменения размеров. В трехмерном пространстве за каждую из этих операций отвечает свой инструмент, но в двухмерном пространстве они объединены, так как у нас становится меньше на одно измерение. В Unity существует также набор клавиатурных комбинаций для ускорения выполнения различных операций. О них вы можете узнать в приложении А. Рассмотрим остальные фрагменты интерфейса.

1.2.3. Вкладка Hierarchy и панель Inspector

На левой стороне экрана примостилась вкладка Hierarchy, в то время как правую заняла панель Inspector (рис. 1.12). Вкладка Hierarchy содержит список всех присутствующих в сцене объектов в виде древовидной структуры, ветви которой вложены друг в друга в соответствии с иерархическими связями между объектами. По сути, эта вкладка предлагает вам возможность выделения объектов по именам, избавляя от необходимости искать нужный объект в сцене, чтобы выделить его щелчком. Иерархические связи объединяют объекты друг с другом. Визуально это оформлено в виде папок, при этом вы можете перемещать одновременно целые группы объектов.

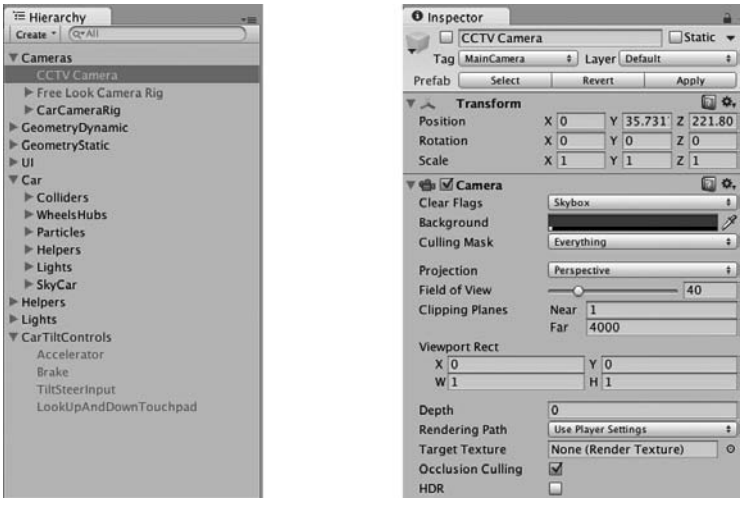


Рис. 1.12. Вкладка Hierarchy и панель Inspector

На панели Inspector отображаются данные выделенного в текущий момент объекта. Выделите любой объект, и вы немедленно увидите, как изменится вид панели Inspector. Отображаемая информация по большей части представляет собой список компонентов, причем вы можете добавлять к объектам новые компоненты или удалять существующие. Все игровые объекты содержат по крайней мере один компонент — Transform, — поэтому на панели Inspector вы всегда будете видеть хотя бы сведения о положении и ориентации выделенного объекта. У многих объектов есть целые списки компонентов, в том числе связанные с этими объектами сценарии.

1.2.4. Вкладки Project и Console

В нижней части экрана находятся показанные на рис. 1.13 вкладки Project и Console. В данном случае мы видим пример такой же организации элементов интерфейса, как и у вкладок Scene и View, что легко позволяет переходить от одного представления к другому. На вкладке Project отображаются все ресурсы (графические фрагменты, код и т. п.) проекта. В левой части этой вкладки вы видите список папок проекта; при выделении папки справа появляются находящиеся в ней файлы. По сути, это такой же список, как и на вкладке Hierarchy, но если эта вкладка показывает вам перечень объектов сцены, то на вкладке Project представлены файлы, не включенные ни в одну конкретную сцену (в том числе и сами файлы сцен — сохраненная сцена появляется на вкладке Project!).

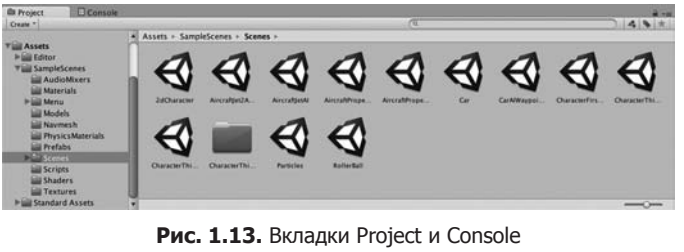


Рис. 1.13. Вкладки Project и Console

СОВЕТ Вкладка Project дублирует содержимое папки Assets на диске, но в общем случае не рекомендуется осуществлять удаление или перемещение файлов непосредственно в этой папке. Выполняйте эти операции на вкладке Project, а о синхронизации с папкой Assets позаботится Unity.

Вкладка Console представляет собой место, где выводятся сообщения, связанные с кодом. Иногда это намеренно вставленные вами в программу сообщения отладчика, иногда это сообщения Unity, появляющиеся при обнаружении ошибок в написанном вами сценарии.

1.3. Готовимся программировать в Unity

Теперь посмотрим, как в Unity выглядит процесс программирования. Хотя компоновка игровых ресурсов происходит в визуальном редакторе, вам потребуется управляющий ими код, обеспечивающий интерактивность игры. Unity поддерживает ряд языков программирования, в частности JavaScript и C#. У каждого из них есть свои достоинства и недостатки, хотя в этой книге все примеры представлены на языке C#.

ПОЧЕМУ C#, А НЕ JAVASCRIPT

Код всех листингов в этой книге написан на языке C#, так как он имеет ряд преимуществ перед языком JavaScript и куда меньше недостатков, особенно с точки зрения профессионального разработчика (именно этим языком я пользуюсь для работы).

Одним из преимуществ является тот факт, что язык C# строго типизирован, чего нельзя сказать о JavaScript. Среди опытных программистов в настоящее время существуют разные точки зрения по поводу того, является ли динамическая проверка типов оптимальным подходом, например к веб-разработке, но при написании программ для определенных игровых платформ (таких, как iOS) она зачастую выгодна, а порой требуется даже статическая типизация. В Unity даже добавлена директива #pragma, принудительно обеспечивающая статическую проверку типов в языке JavaScript. Хотя с технической стороны такое вполне допустимо, при этом нарушается один из основных принципов функционирования JavaScript. Поэтому лучше изначально выбрать язык со строгой типизацией.

Это всего лишь один пример того, как отличается язык JavaScript в Unity. Во многом он напоминает JavaScript в веб-браузерах, но в функционировании языка есть ряд зависящих от контекста отличий. Многие разработчики называют версию для Unity именем UnityScript, которое указывает на сходство, но одновременно и на отличие от JavaScript. Именно это состояние «аналогичный, но отличающийся» становится для программистов проблемой и при попытках применить общие знания языка JavaScript в контексте Unity, и при попытках применять на стороне знания, полученные в процессе работы в Unity.

Рассмотрим пример написания и запуска кода. Откройте Unity и создайте новый проект, выбрав в меню File команду New Project. Откроется окно диалога New Project. Укажите имя проекта и выберите, где бы вы хотели его сохранить. Unity-проект представляет собой обычную папку с файлами различных ресурсов и настроек, поэтому его можно сохранять где угодно. Щелкните на кнопке Create Project, и Unity ненадолго исчезнет, чтобы создать папку проекта.

ВНИМАНИЕ Проекты Unity запоминают, в какой версии программы они создавались, и при попытке открыть их в другой версии появляется предупреждение. Иногда на него можно не обращать

внимания (скажем, если такое предупреждение появится при открытии файлов с примерами к данной книге, просто игнорируйте его), но бывают ситуации, когда перед открытием проекта имеет смысл сделать его резервную копию.

Когда Unity появится снова, вы увидите пустую сцену. А теперь посмотрим, как в Unity запускаются программы.

1.3.1. Запуск кода в Unity: компоненты сценария

Выполнение кода в Unity всегда начинается с файлов, связанных с объектом сцены. В конечном счете все это часть упомянутой ранее компонентной системы; игровые объекты строятся как наборы компонентов, и каждый такой набор может включать в себя исполняемый сценарий.

ПРИМЕЧАНИЕ В терминологии Unity файлы с кодом принято называть сценариями, используя для определения этого слова значение, чаще всего применяемое к ситуации, когда в браузере запускается JavaScript: код выполняется внутри игрового движка Unity, а не фигурирует после компиляции в виде самостоятельного фрагмента. Но зачастую люди определяют данный термин совсем по-другому. К примеру, сценариями иногда называют автономные служебные программы. Сценарии в Unity больше напоминают индивидуальные классы ООП. Присоединенные к объектам сцены сценарии являются экземплярами объектов.

Как вы, наверное, догадались из этого описания, сценарии в Unity представляют собой компоненты. Но прошу заметить, не все сценарии, а только наследующие от класса `MonoBehaviour`, базового класса компонентов-сценариев. Этот класс определяет способ присоединения компонентов к игровым объектам (как показано в листинге 1.1). Наследование от данного класса дает пару автоматически добавляемых методов, которые вы можете переопределить. Это метод `Start()`, вызываемый при активации объекта (а она, как правило, наступает после загрузки содержащего объект уровня), и метод `Update()`, вызываемый в каждом кадре. Соответственно, ваш код запускается, когда вы помещаете его в эти предустановленные методы.

ОПРЕДЕЛЕНИЕ Кадром (frame) называется один прогон зацикленного игрового кода. Практически все видеоигры (не только в Unity, но и вообще) строятся вокруг основного игрового цикла. То есть код циклически выполняется все время, пока запущена игра. Каждый цикл включает в себя прорисовку экрана, откуда, собственно, и возник термин «кадр» (по аналогии с набором статичных кадров в фильме).

Листинг 1.1. Шаблон кода для базового компонента сценария

```
using UnityEngine;          ← Добавляем пространства имен для классов Unity и Mono.
using System.Collections;

public class HelloWorld : MonoBehaviour {  ← Синтаксис для задания наследования.

  void Start() {
    // выполняем некую однократную операцию  ← Помещаем сюда однократно выполняемый код.
  }
```

```
  void Update() {
    // выполняем некую операцию в каждом кадре ← Помещаем сюда код, запускаемый в каждом кадре.
  }
}
```

Именно так выглядит файл, когда вы создаете новый сценарий на C#: минимальный шаблонный код, определяющий корректный компонент в Unity. Существует шаблон сценария, скрытый в недрах Unity. При создании нового сценария приложение копирует этот шаблон и переименовывает класс в соответствии с именем файла (в моем случае — в `HelloWorld.cs`). Есть и пустые оболочки методов `Start()` и `Update()`, так как именно из них чаще всего вызывается создаваемый код (хотя я обычно добавляю к этим функциям немного пробелов, так как количество пробелов в этом шаблоне отличается от того, что я предпочитаю).

Чтобы создать сценарий, выберите команду C# Script в меню Create, доступ к которому можно получить, открыв меню Assets (обратите внимание, что как в меню Assets, так и в меню GameObjects есть варианты команды Create, но это разные команды) или щелкнув правой кнопкой мыши в произвольной точке вкладки Project. Введите имя нового сценария, например «HelloWorld». Как вы увидите чуть позже (рис. 1.15), этот файл сценария можно мышью перетащить на произвольный объект сцены. Дважды щелкните на значке сценария, и он автоматически откроется в программе MonoDevelop, о которой мы поговорим далее.

1.3.2. Программа MonoDevelop — межплатформенная среда разработки

Программирование осуществляется не внутри Unity, код существует в виде отдельных файлов, местоположение которых вы сообщаете Unity. Файлы сценариев могут создаваться в приложении Unity, но в любом случае вам потребуется текстовый редактор или IDE, где будет писаться код для этих изначально пустых файлов. В комплекте с Unity поставляется приложение MonoDevelop как межплатформенная интегрированная среда разработки (IDE) для языка C# с открытым исходным кодом (его внешний вид демонстрирует рис. 1.14). Более подробную информацию об этой программе можно получить на странице www.monodevelop.com, но при этом вы должны пользоваться версией, идущей в комплекте с Unity, а не скачанным с этого сайта приложением, так как в базовую программу был внесен ряд изменений для лучшей интеграции с Unity.

ПРИМЕЧАНИЕ Программа MonoDevelop объединяет файлы в группы, называемые решениями (solution). Инструмент Unity генерирует содержащее все файлы сценариев решение автоматически, поэтому, как правило, вы можете вообще об этом не думать.

Так как язык C# изначально разрабатывался в Microsoft, вероятно, вам было бы интересно узнать, можно ли для программирования в Unity пользоваться Visual Studio. Да, можно. Инструментальные средства поддержки доступны по адресу www.unityvs.com, но лично я предпочитаю программу MonoDevelop, главным образом потому, что Visual Studio работает только в операционных системах семейства Windows, что ограничивает ваши возможности. Я не считаю, что это плохо, и, если вы уже привыкли программировать в Visual Studio, продолжайте пользоваться этой средой разработки. Вы легко

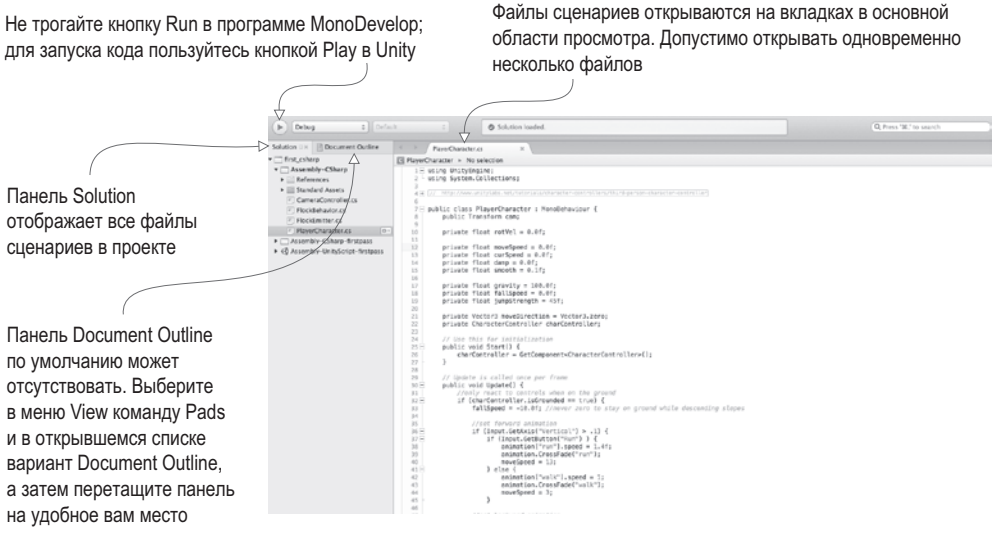


Рис. 1.14. Фрагменты интерфейса программы MonoDevelop

сможете выполнять все практические задания из книги (и после вводной главы больше о IDE мы говорить не будем). Однако привязка рабочего процесса к Windows аннулирует одно из самых важных преимуществ Unity. Трудности возникнут, если вам потребуется сотрудничать с разработчиками, использующими компьютеры Mac, или если вы захотите выполнить развертывание своей игры на платформе iOS. Хотя язык C# является продукцией Microsoft и поэтому изначально работал только в Windows на платформе .NET Framework, в настоящее время он стал открытым стандартом языка, к тому же появился важный межплатформенный фреймворк Mono. Unity использует Mono как основу для программирования, и именно среда MonoDevelop обеспечивает межплатформенные возможности всего процесса разработки.

Все время помните, что в программе MonoDevelop код только пишется, но не запускается. Эта среда разработки всего лишь хорошо оснащенный текстовый редактор, а воспроизводить код следует щелчком на кнопке Play в программе Unity.

1.3.3. Вывод на консоль: «Hello World!»

Итак, в нашем проекте появился пустой сценарий, но пока отсутствует объект, к которому этот сценарий можно было бы присоединить. Вспомните рис. 1.1, демонстрирующий работу системы компонентов; любой сценарий является компонентом, поэтому его нужно превратить в один из компонентов какого-то объекта.

Выберите в меню Select GameObject команду Create Empty, и на вкладке Hierarchy появится пустой объект GameObject. Теперь перетащите значок сценария с вкладки Project на вкладку Hierarchy и поместите на строку GameObject. Как показано на рис. 1.15, приложение Unity выделяет места, куда можно поместить сценарий. Как только вы отпустите кнопку мыши, сценарий будет присоединен к объекту GameObject. Чтобы удостовериться в успешном исходе операции, выделите объект и посмотрите на панель Inspector. Вы должны увидеть там два компонента: Transform — базовый компонент

положения/ориентации/масштаба, присутствующий у всех объектов, который невозможно удалить, а сразу под ним — ваш сценарий.

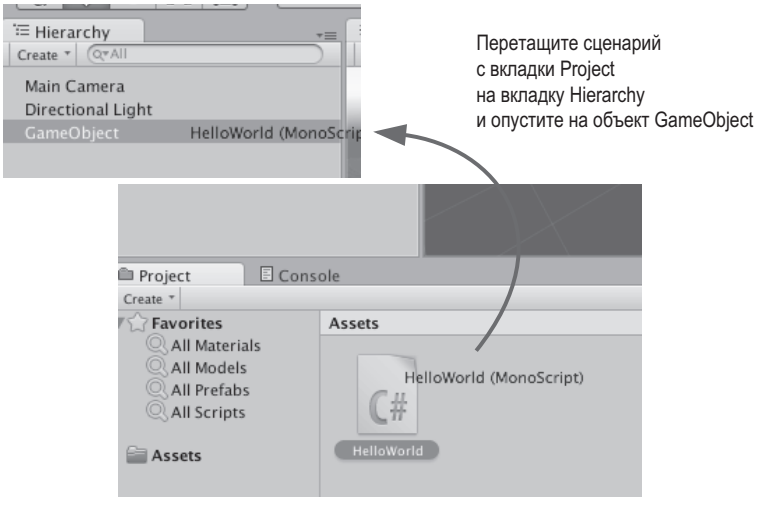


Рис. 1.15. Связывание сценария с объектом GameObject

ПРИМЕЧАНИЕ В конце концов, это действие — перетаскивание объектов и укладывание их на другие объекты — станет вам привычным. Этим способом в Unity создается огромное количество различных связей, а не только привязка сценариев к объектам.

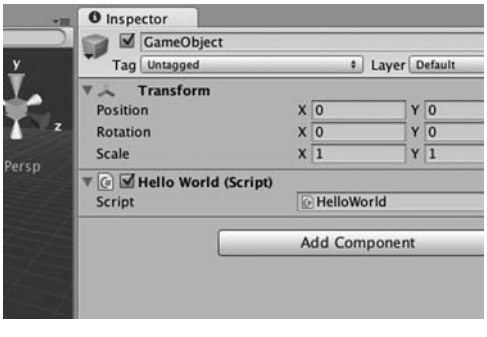


Рис. 1.16. Связанный сценарий отображается на панели Inspector

После связывания сценария с объектом на панели Inspector вы увидите примерно такую картину, которая показана на рис. 1.16. Сценарий будет фигурировать в качестве компонента. Теперь он начнет исполняться при воспроизведении сцены, хотя вы не заметите никаких внешних проявлений, так как код еще не написан. Именно этим мы сейчас займемся!

Откройте сценарий в программе MonoDevelop, чтобы вернуться к листингу 1.1. Классическим примером, с которого всегда начинается знакомство с новой средой программирования, становится вывод текста «Hello World!». Поэтому добавьте эту строку в метод `Start()`, как показано в листинге 1.2.

Листинг 1.2. Добавляем вывод сообщения на консоль

```
void Start() {
  Debug.Log("Hello World!"); ← Добавляем сюда команду регистрации.
}
```

Команда `Debug.Log()` выводит сообщение на вкладку Console в Unity. Строка с этой командой идет в метод `Start()`, так как, как мы уже упоминали, данный метод вызывается однократно после активации объекта. Другими словами, после щелчка на кнопке Play в редакторе метод `Start()` будет вызван всего один раз. Как только вы добавили в сценарий команду регистрации (обязательно сохраните сценарий), щелкните на кнопке Play в программе Unity и перейдите на вкладку Console. Там появится сообщение «Hello World!». Поздравляю, вы написали свой первый сценарий для Unity! В следующих главах мы будем писать куда более сложный код, но сейчас вы сделали важный первый шаг.

ЭТАПЫ НАПИСАНИЯ СЦЕНАРИЯ «HELLO WORLD!»

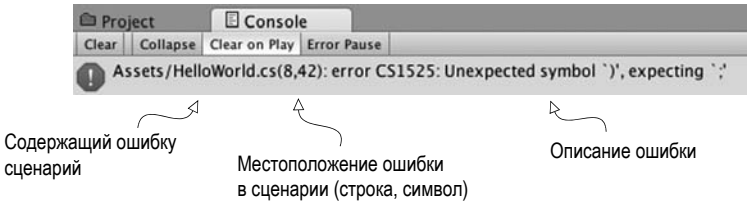
Кратко перечислим, что мы делали при чтении последних страниц:

- Создали новый проект.
- Создали новый сценарий на языке C#.
- Создали пустой объект GameObject.
- Перетащили сценарий на этот объект.
- Добавили к сценарию команду регистрации.
- Щелкнули на кнопке Play!

Теперь сцену можно сохранить; появится файл с расширением .unity и значком Unity. Файл сцены представляет собой снимок всего, что есть в игре в данный момент, поэтому в дальнейшем вы можете легко загрузить сцену в программу. В данном случае сохранять сцену нет особого смысла (ведь она содержит всего один пустой объект GameObject). Но если вы не сохраните сцену и захотите вернуться к ней в будущем, она окажется пустой.

ОШИБКИ В СЦЕНАРИИ

Чтобы посмотреть, каким образом Unity указывает на ошибки, специально сделайте опечатку в сценарии HelloWorld. К примеру, даже лишняя скобка приведет к появлению на вкладке Console сообщения об ошибке:



1.4. Заключение

- Unity — это мультиплатформенный инструмент разработки.
- Визуальный редактор Unity состоит из набора фрагментов, которые работают в связке друг с другом.
- Сценарии присоединяются к объектам как компоненты.
- Код сценариев пишется в программе MonoDevelop.

2 Создание 3D-ролика

- ✓ Знакомство с трехмерным координатным пространством
- ✓ Размещение в сцене игрока
- ✓ Создание сценария перемещения объектов
- ✓ Реализация элементов управления персонажем в игре от первого лица

Глава 1 завершилась традиционным способом знакомства с новыми средствами программирования — написанием программы «Hello World!». Пришло время погрузиться в более сложный Unity-проект, содержащий как средства взаимодействия с пользователем, так и графику. Вы поместите в сцену набор объектов и напишете код, позволяющий игроку перемещаться в этой сцене. По сути, это будет аналог игры Doom без монстров (примерно такой, как на рис. 2.1). Визуальный редактор Unity позволяет новым пользователям сразу приступить к сборке трехмерного прототипа без предварительного написания шаблонного кода (для таких вещей, как инициализация трехмерного представления или установка цикла визуализации).

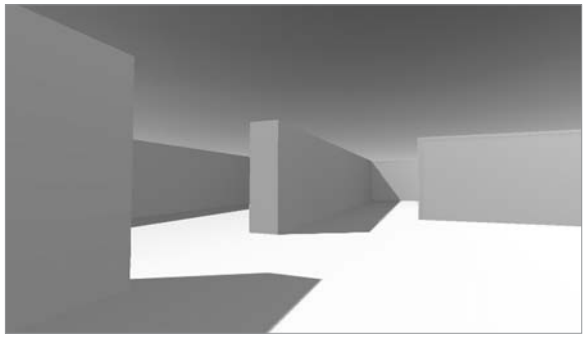


Рис. 2.1. Снимок экрана трехмерной демоверсии (по сути, это Doom без монстров)

Хотя на данном этапе высок соблазн немедленно заняться созданием сцены, особенно с учетом крайней простоты проекта, имеет смысл немного притормозить и продумать порядок своих действий. Сейчас это особенно важно, так как вы пока не знакомы с процессом.

2.1. Подготовка...

Инструмент Unity дает новичкам возможность сразу приступить к работе, но перед тем, как заняться созданием сцены, оговорим пару аспектов. Даже при наличии такого гибкого инструмента, как Unity, следует четко представлять себе, что именно вы хотите получить в результате своих действий. Кроме того, нужно хорошо представлять себе, как функционирует трехмерная система координат, в противном случае вы запутаетесь при первой же попытке вставки в сцену объекта.

2.1.1. Планирование проекта

Перед тем как приступить к программированию, всегда нужно остановиться и ответить на вопрос «что я собираюсь построить?». Проектирование игр — весьма обширная тема, которой посвящено множество толстых книг. К счастью, в данном случае для разработки базового учебного проекта достаточно мысленно представлять структуру будущей сцены. Первые проекты будут несложными, чтобы лишние детали не мешали вам изучать принципы программирования; а задумываться о разработке более высокоуровневого проекта можно (и нужно!) только после того, как вы освоите основы создания игр.

Вашим первым проектом станет создание сцены из шутера от первого лица — мы будем обозначать этот тип игр аббревиатурой FPS (First-Person Shooter). Вы построите комнату, по которой можно перемещаться, при этом игроки будут наблюдать окружающий мир с точки зрения игрового персонажа. Управлять этим персонажем игрок сможет посредством мыши и клавиатуры. Все интересные, но сложные элементы готовой игры мы пока отбросим, чтобы сконцентрироваться на основной задаче — перемещениях в трехмерном пространстве. Рисунок 2.2 иллюстрирует сценарий проекта, по сути, представляющий собой придуманный мной перечень действий:

1. Разработка комнаты: создание пола, внешних и внутренних стен.
2. Размещение источников света и камеры.
3. Создание объекта-игрока (в том числе и присоединение камеры к его верхней части).
4. Написание сценариев перемещения: повороты при помощи мыши и перемещения при помощи клавиатуры.

Пусть столь внушительный сценарий вас не пугает! Кажется, что вам предстоит большая работа, но Unity делает ее очень простой. Разделы, посвященные сценариям перемещения, так велики только потому, что мы подробно рассматриваем каждую деталь, чтобы полностью понять принципы программирования. Мы начинаем с игры от первого лица, чтобы снизить требования к художественному оформлению — в играх

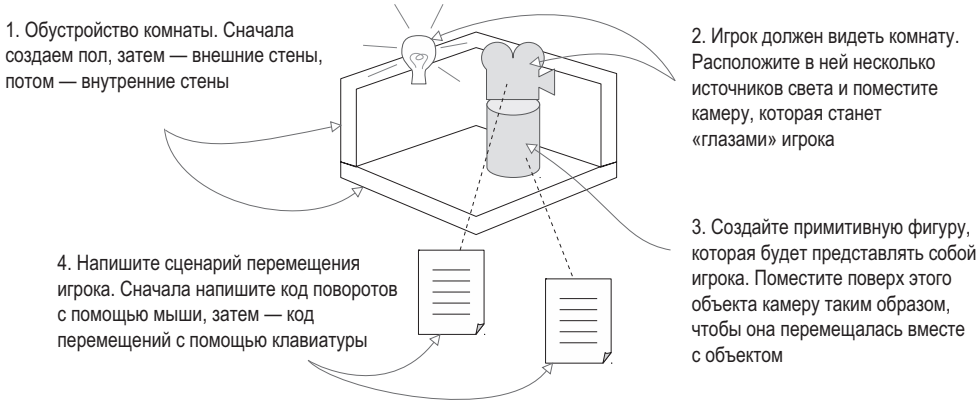


Рис. 2.2. Сценарий трехмерного демонстрационного ролика

от первого лица себя игрок не видит, поэтому вполне допустимо придать персонажу форму цилиндра, на верхней грани которого закреплена камера! Теперь осталось понять, как функционируют трехмерные координаты, и вы сможете легко вставлять объекты в сцены в визуальном редакторе.

2.1.2. Трехмерное координатное пространство

Сформулированный нами для начала простой план включает в себя три аспекта: игровое пространство, средства наблюдения, элементы управления. Для их реализации требуется понимание того, каким образом в трехмерных симуляциях задаются положение и перемещение объектов. Те, кто раньше никогда не сталкивался с трехмерной графикой, вполне могут этого не знать.

Все сводится к числам, указывающим положение точки в пространстве. Сопоставление этих чисел с пространством происходит через оси системы координат. На уроках математики в школе вы, скорее всего, видели показанные на рис. 2.3 оси *X* и *Y* и даже пользовались ими для присваивания координат различным точкам на листе бумаги. Это так называемая прямоугольная, или декартова, система координат.

Две оси дают вам двухмерные координаты. Это случай, когда все точки лежат в одной плоскости. Трехмерное пространство задается уже тремя координатными осями. Так как ось *X* располагается на странице горизонтально, а ось *Y* — вертикально, третья ось должна как бы «протыкать» страницу, располагаясь перпендикулярно осям *X* и *Y*. Рисунок 2.4 демонстрирует оси *X*, *Y* и *Z* для трехмерного координатного пространства. У всех элементов, обладающих определенным положением в сцене (у игрока, у стен и т. п.), будут координаты *XYZ*.

На вкладке Scene в Unity вы видите значок трех осей, а на панели Inspector можно указать три числа, задающих положение объекта. Координаты в трехмерном пространстве вы будете использовать не только при написании кода, определяющего местоположение объектов, но и для указания смещений как значений сдвига вдоль каждой из осей.

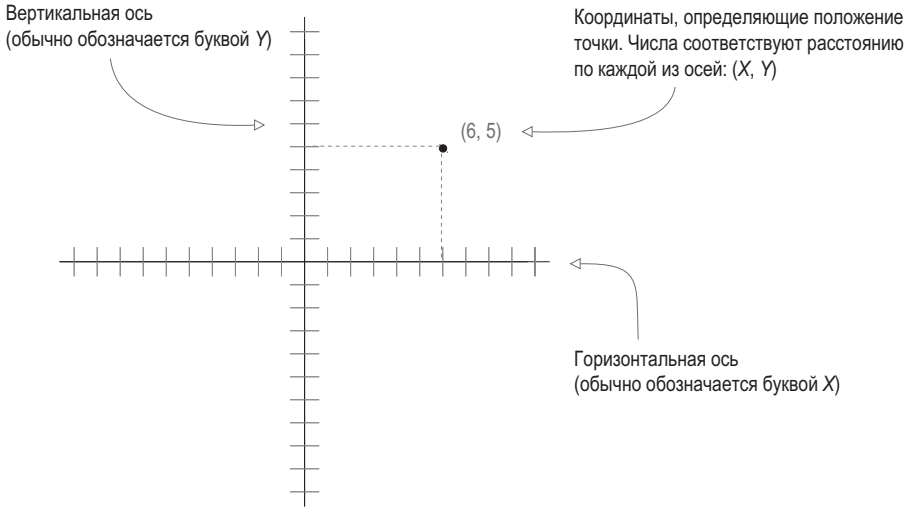


Рис. 2.3. Координаты по осям *X* и *Y* определяют положение точки на плоскости

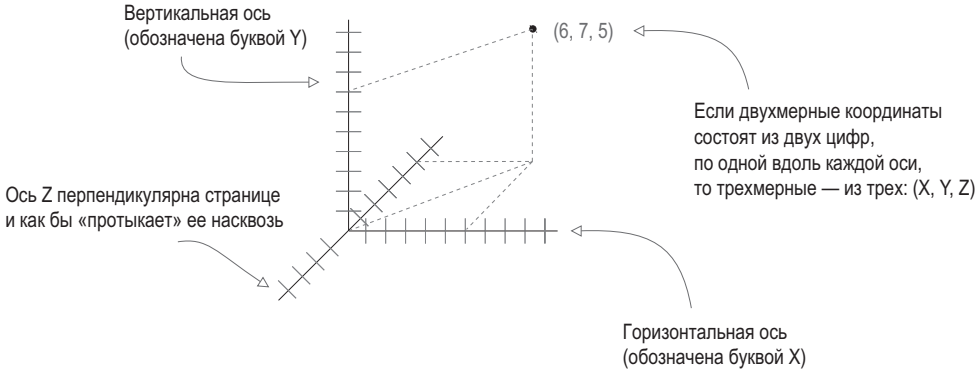
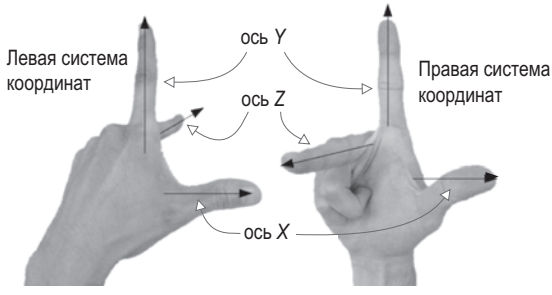


Рис. 2.4. Координаты по осям *X*, *Y* и *Z* определяют точку в трехмерном пространстве

ПРАВАЯ И ЛЕВАЯ СИСТЕМЫ КООРДИНАТ

Положительное и отрицательное направления каждой оси выбираются произвольным образом, в обоих случаях система координат прекрасно работает. Главное, используя инструмент для обработки трехмерной графики (инструмент анимации, инструмент разработки и т. п.), всегда выбирать одну и ту же систему координат.

Впрочем, практически всегда ось *X* указывает вправо, а ось *Y* — вверх; разница между инструментами в основном состоит в том, что где-то ось *Z* выходит из страницы, а где-то входит в нее. Эти два варианта называют «правой» и «левой» системами координат. Как показано на рисунке, если большой палец расположить вдоль оси *X*, а указательный — вдоль оси *Y*, средний палец задаст направление оси *Z*.



У левой и правой руки ось *Z* ориентирована в разных направлениях

В Unity, как и во многих других приложениях для работы с компьютерной графикой, используется левая система координат. Однако существует множество инструментов, в которых применяется правая система координат (например, OpenGL). Помните об этом, чтобы не растеряться, столкнувшись с другими направлениями координатных осей.

Теперь, когда у вас есть не только план действий, но и представление о том, как с помощью координат задать положение объекта в трехмерном пространстве, приступим к работе над сценой.

2.2. Начало проекта: размещение объектов

Итак, начнем с создания объектов и размещения их в сцене. Первыми будут статические объекты — пол и стены. Затем выберем место для источников света и камеры. Последним создадим игрока — это будет объект, к которому вы добавите сценарии, перемещающие его по сцене. Рисунок 2.5 демонстрирует вид редактора после завершения работы.

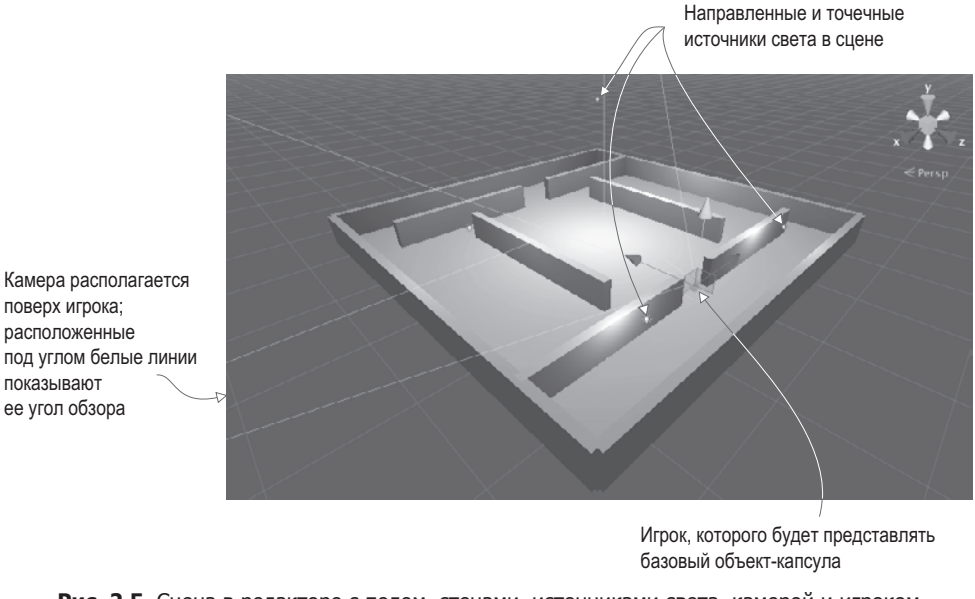


Рис. 2.5. Сцена в редакторе с полом, стенами, источниками света, камерой и игроком

В главе 1 демонстрировался способ создания нового проекта в Unity. Именно это мы сейчас и проделаем. Напоминаю, что нужно выбрать в меню File команду New Project и в появившемся окне указать имя проекта. После этого сразу же сохраните пустую сцену, так как изначально файл сцены у нового проекта отсутствует. Начнем мы с создания наиболее очевидных объектов.

2.2.1. Декорации: пол, внешние и внутренние стены

В расположенном в верхней части экрана меню GameObject наведите указатель мыши на строчку 3D Object, чтобы открыть дополнительное меню. Выберите в нем вариант Cube, так как для нашей сцены требуется куб (позднее вы поработаете и с другими фигурами, такими как Sphere и Capsule). Отредактируйте положение и масштаб появившегося в сцене куба, а также его имя таким образом, чтобы получить пол; значения, которые следует присвоить параметрам этого объекта на панели Inspector, показаны на рис. 2.6 (для превращения куба в пол его нужно растянуть).

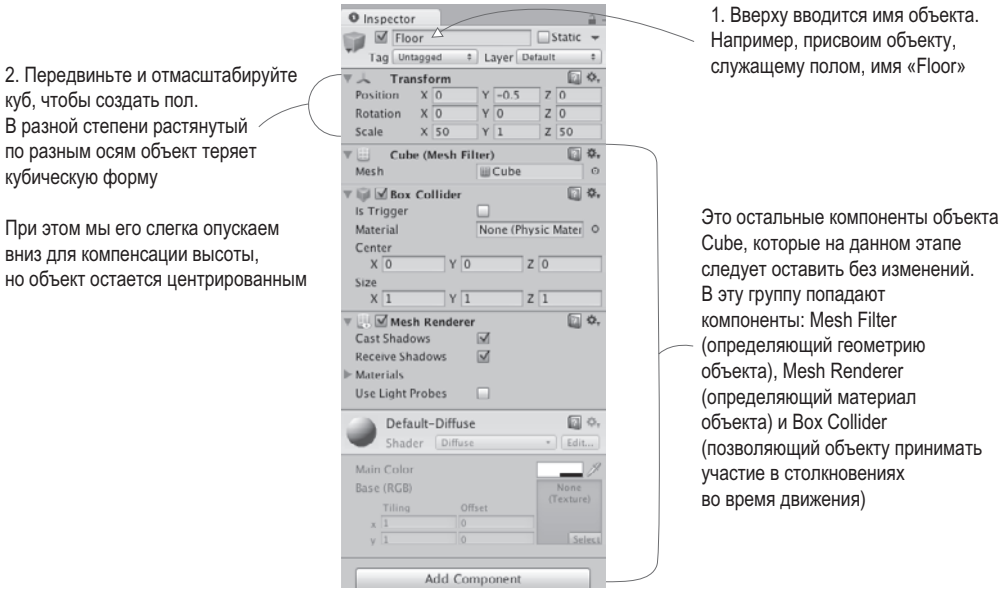


Рис. 2.6. Панель Inspector для пола

ПРИМЕЧАНИЕ Единицы измерения положения могут выбираться произвольным образом, главное, чтобы они использовались для всех элементов сцены. Чаще всего в качестве единицы измерения фигурирует метр. Сам я предпочитаю пользоваться именно метрами, но бывают случаи, когда я выбираю футы. Хотя мне доводилось видеть людей, которые считали, что размеры в сцене измеряются в дюймах!

Повторите описанную последовательность для создания внешних стен комнаты. Можно каждый раз задействовать новый куб, а можно копировать и вставлять существующий объект, указывая стандартные сокращения. Двигайте, поворачивайте

и масштабируйте стены, чтобы получить показанный на рис. 2.5 периметр. Экспериментируйте с различными значениями (например, 1, 4, 50 для полей Scale) или воспользуйтесь инструментами преобразований, с которыми вы познакомились в разделе 1.2.2 (напоминаю, что перемещения и повороты в трехмерном пространстве называют преобразованиями).

СОВЕТ Не забудьте про средства навигации, позволяющие рассматривать сцену под разными углами и менять ее масштаб, например имитируя взгляд с высоты птичьего полета. При этом нажатие клавиши F вернет вас к просмотру выделенного в данный момент объекта.

Точные значения преобразований для стен будут зависеть от того, каким образом вы повернете и отмасштабируете исходные объекты Cube, подогнав их размеры и положение, а также от способа их связывания на вкладке Hierarchy. К примеру, на рис. 2.7 демонстрируется ситуация, когда все стены являются потомками пустого корневого объекта. Содержимое вкладки Hierarchy в этом случае имеет упорядоченный вид. Если вы предпочитаете просто скопировать рабочие значения, скачайте пример проекта и возьмите все данные оттуда.

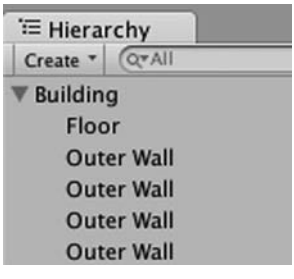


Рис. 2.7. Панель Hierarchy, показывающая, что стены и пол являются потомками пустого объекта

СОВЕТ Связи между объектами устанавливаются простым перетаскиванием объектов друг на друга на вкладке Hierarchy. Объект, к которому присоединены другие объекты, называется предком (parent); объекты, присоединенные к другим объектам, называются потомками (children). Перемещение (поворот или масштабирование) родительского объекта сопровождается аналогичным преобразованием всех его потомков.

СОВЕТ Для систематизации объектов сцены подобным образом применяются пустые игровые объекты. Связывание видимых объектов с корневым позволяет сворачивать списки объектов на вкладке Hierarchy. Но помните, что перед этой операцией следует расположить пустой корневой объект в точке с координатами 0, 0, 0, чтобы в дальнейшем избежать проблем с позиционированием.

ЧТО ТАКОЕ GAMEOBJECT?

Все объекты сцены представляют собой экземпляры класса GameObject аналогично тому, как все компоненты сценариев наследуют от класса MonoBehaviour. Этот факт становится более наглядным, если пустому объекту присвоить имя GameObject. Впрочем, даже если этот объект будет называться Floor, Camera или Player, суть дела не изменится.

На самом деле GameObject представляет собой всего лишь контейнер для набора компонентов. Его основным назначением является обеспечение некоего объекта, к которому можно присоединять класс MonoBehaviour. Как все это будет выглядеть в сцене, зависит от добавленных к объекту GameObject компонентов. К примеру, куб получается добавлением компонента Cube, сфера — добавлением компонента Sphere, и т. п.

Завершив работу над внешними стенами, приступайте к созданию внутренних. Расположите их по своему вкусу. Вам нужны коридоры и препятствия, среди которых будет происходить движение.

В итоге в сцене появится комната, но без источников света игрок ничего в ней не увидит. Поэтому давайте осветим нашу комнату.

2.2.2. Источники света и камеры

Как правило, трехмерные сцены освещаются направленным источником света, к которому добавляется набор точечных осветителей. Начать имеет смысл с направленного источника. Он может по умолчанию присутствовать в сцене, но если его нет, наведите указатель мыши на строку Light в меню GameObject и в открывшемся дополнительном меню выберите вариант Directional Light.

ТИПЫ ОСВЕТИТЕЛЕЙ

Существуют различные типы осветителей, разными способами проецирующие световые лучи. Три основных типа: точечный источник, прожектор и направленный источник.

Все лучи точечного источника (point light) начинаются в одной точке и распространяются во всех направлениях. В реальном мире таким осветителем является лампочка. Яркость света увеличивается по мере приближения к источнику за счет концентрации лучей.

Лучи прожектора (spot light) также исходят из одной точки, но распространяются в пределах ограниченного конуса. В текущем проекте мы с прожекторами работать не будем, но осветители данного типа повсеместно используются для подсвечивания отдельных частей уровня.

Лучи направленного источника света (directional light) распространяются равномерно и параллельно друг другу, одинаково освещая все элементы сцены. Это аналог солнца.

Испускаемый направленным осветителем свет не зависит от местоположения источника, значение имеет только его ориентация, поэтому его можно поместить в произвольную точку сцены. Я рекомендую установить его над комнатой, чтобы он выглядел как солнце и не мешал вам при работе с остальными фрагментами сцены. Поверните источник света и посмотрите, как это повлияет на освещенность комнаты; для получения нужного эффекта я рекомендую слегка повернуть его относительно осей *X* и *Y*. На панели Inspector вы найдете параметр Intensity (рис. 2.8). Как следует из его названия, он управляет яркостью света. Если бы данный направленный осветитель был в сцене единственным, его яркость имело бы смысл увеличить, но, так как мы добавим несколько точечных источников света, его можно оставить тусклым. Например, присвойте параметру Intensity значение 0.6.

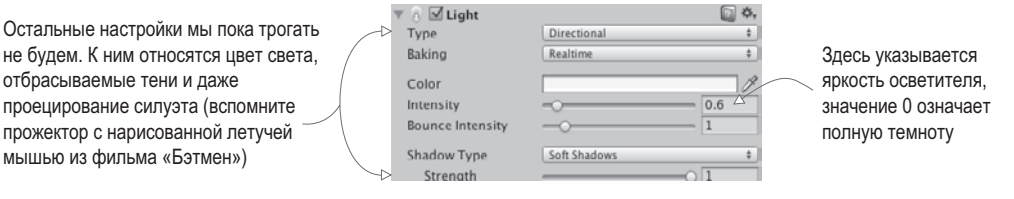


Рис. 2.8. Настройки направленного источника света на панели Inspector

Теперь командой уже знакомого вам меню создайте несколько точечных осветителей, поместив их в темные места комнаты. Вам нужно, чтобы все было как следует освещено. Источников должно быть не слишком много (иначе пострадает

производительность), вполне достаточно одного осветителя в каждом углу (я предлагаю поднять их к верхней кромке стен) и одного, расположенного высоко над сценой (например, со значением 18 координаты *Y*). Это придаст освещению комнаты некое разнообразие. Обратите внимание, что у точечных источников света на панели Inspector появляется дополнительный параметр Range (рис. 2.9). Он управляет дальностью распространения лучей. Если направленный источник света равномерно освещает всю сцену, то яркость точечного источника уменьшается по мере удаления от него. Поэтому для стоящего на полу источника этот параметр может быть равен 18, а для поднятого к верхней кромке стены его необходимо увеличить до 40, иначе он не сможет осветить всю комнату.

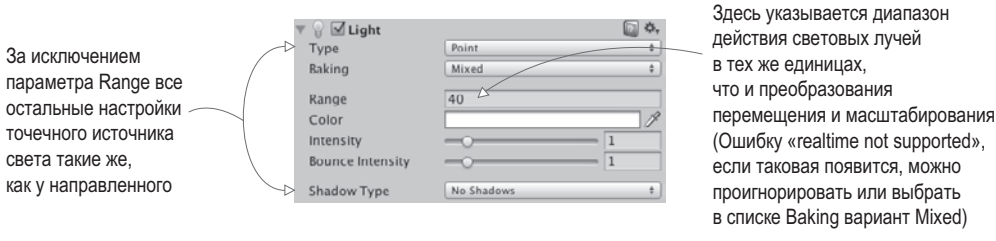


Рис. 2.9. Настройки точечного источника света на панели Inspector

Чтобы игрок мог наблюдать за происходящим, требуется еще один тип объекта — камера. Впрочем, пустая сцена содержит основную камеру, которой мы можем воспользоваться. Если вам когда-нибудь потребуется создать дополнительные камеры (например, для разделения экрана в многопользовательских играх), выберите команду Camera в меню GameObject. Нашу же камеру мы расположим поверх игрока, чтобы наблюдать сцену его глазами.

2.2.3. Коллайдер и точка наблюдения игрока

В этом проекте игрока будет представлять обычный примитив. В меню GameObject (напоминаю, что для открытия этого дополнительного меню нужно навести указатель мыши на строку 3D Object) выберите вариант Capsule. Появится цилиндрическая фигура со скругленными концами — это и есть наш игрок. Сместите объект вверх, сделав его координату *Y* равной 1.1 (половина высоты объекта, плюс еще немного, чтобы избежать перекрывания с полом). Теперь наш игрок может произвольным образом перемещаться вдоль осей *X* и *Z* при условии, что он остается внутри комнаты и не касается стен. Присвойте объекту имя Player.

На панели Inspector вы увидите, что этому объекту назначен капсульный коллайдер. Это очевидный вариант для объекта Capsule, точно так же, как объект Cube по умолчанию обладает коллайдером Box. Но так как наша капсула будет представлять игрока, ее компоненты должны слегка отличаться от компонентов большинства объектов. Капсульный коллайдер мы удалим. Для этого нужно щелкнуть на значке с изображением шестерни справа от имени компонента, как показано на рис. 2.10. Откроется меню, в котором в числе прочих вы найдете и команду Remove Component. Коллайдер

выглядит как окружающая объект зеленая сетка, поэтому после удаления компонента вы обнаружите, что она исчезла.

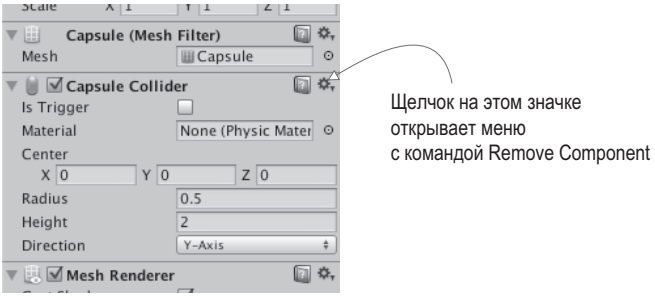


Рис. 2.10. Удаление компонента на панели Inspector

Вместо капсульного коллайдера мы назначим объекту контроллер персонажа. В нижней части панели Inspector вы найдете кнопку Add Component. Щелчок на ней открывает меню с перечнем типов доступных компонентов. В разделе Physics и находится нужная нам строка Character Controller. Как несложно догадаться, именно этот компонент позволит объекту вести себя как персонаж.

Для завершения настройки игрока осталось сделать всего один шаг — присоединить к нему камеру. Как уже упоминалось в разделе, посвященном созданию пола и стен, вы можете перетаскивать объекты и класть их друг на друга на вкладке Hierarchy. Проделайте эту операцию, перетащив камеру на капсулу, чтобы присоединить ее к игроку. Затем расположите ее таким образом, чтобы она соответствовала глазам игрока (я предлагаю указать координаты 0, 0.5, 0). При необходимости верните координатам преобразования поворота значения 0, 0, 0 (если вы вращали капсулу, они будут различаться).

Итак, в сцене присутствуют все необходимые объекты. Осталось написать код перемещения игрока.

2.3. Двигаем объекты: сценарий, активирующий преобразования

Чтобы заставить игрока перемещаться по сцене, нам потребуются сценарии движения, которые будут присоединены к игроку. Напоминаю, что компонентами называются модульные фрагменты функциональности, добавляемые к объектам, поэтому сценарии тоже можно считать своего рода компонентами. Именно они будут реагировать на клавиатурный ввод и манипуляции мышью, но для начала мы заставим игрока поворачиваться на месте. Это продемонстрирует вам, как применять преобразования в коде. Надеюсь, вы помните, что преобразования бывают трех видов — перемещение, поворот и масштабирование; вращение объекта на месте соответствует преобразованию поворота. Но вы должны знать об этой задаче еще кое-что, кроме того, что она «сводится к изменению ориентации объекта».

2.3.1. Схема программирования движения

Анимация объекта (например, его вращение) сводится к смещению его в каждом кадре на небольшое расстояние и к последующему многократному воспроизведению всех кадров. Само по себе преобразование происходит мгновенно, в отличие от движения, растянутого во времени. Но последовательное применение набора преобразований вызывает визуальное перемещение объекта, совсем как набор рисунков в кинеографе. Этот принцип проиллюстрирован на рис. 2.11.

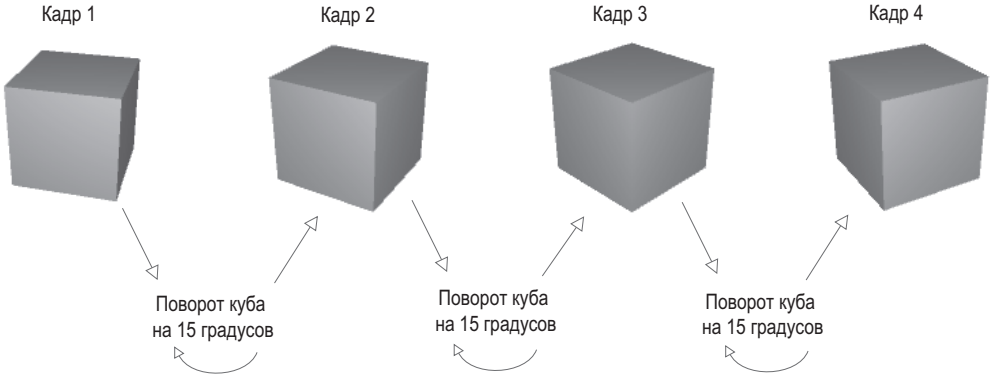


Рис. 2.11. Возникновение движения: циклический процесс преобразования статичных изображений

Напомню вам, что компоненты сценария снабжены запускающимся в каждом кадре методом `Update()`. Поэтому, чтобы заставить куб вращаться вокруг своей оси, следует добавить в метод `Update()` код, поворачивающий его на небольшой угол. Этот код будет запускаться в каждом кадре снова и снова. Не правда ли, все очень просто?

2.3.2. Написание кода

Применим рассмотренную концепцию на практике. Создайте новый сценарий командой C# script (напоминаю, что она находится в дополнительном меню Create, доступ к которому осуществляется через меню Assets), присвойте ему имя Spin и напишите код из следующего листинга (не забудьте после ввода листинга сохранить файл!).

Листинг 2.1. Приводим объект во вращение

```
using UnityEngine;
using System.Collections;

public class Spin : MonoBehaviour {
    public float speed = 3.0f; ← Объявление общедоступной переменной для скорости вращения.

    void Update() {
        transform.Rotate(0, speed, 0); ← Поместите сюда команду Rotate, чтобы она запускалась в каждом кадре.
    }
}
```

Для добавления компонента сценария к игроку перетащите сценарий с вкладки Project на строку Player на вкладке Hierarchy. Щелкните на кнопке Play, и вы увидите, как все

придет во вращение — вы написали код, заставляющий объект двигаться! В основном этот код состоит из шаблона нового сценария, к которому добавлены две строки. Посмотрим, что именно они делают.

Прежде всего, появилась переменная для скорости, добавленная в верхнюю часть определения класса. Превращение скорости вращения в переменную произошло по двум причинам. Первая — это соблюдение стандартного правила программирования: «никаких магических чисел». Вторая же причина связана со способом отображения общедоступных переменных в Unity. Познакомимся с ним.

СОВЕТ Общедоступные переменные появляются на панели Inspector, что позволяет редактировать значения уже добавленных к игровому объекту компонентов. Это называется «сериализацией» значения, так как Unity сохраняет модифицированное состояние переменной.

Рисунок 2.12 показывает вид компонента сценария на панели Inspector. Можно ввести новое значение, и сценарий будет использовать его вместо указанного в коде. Таким способом удобно редактировать параметры компонентов различных объектов непосредственно в визуальном редакторе вместо того, чтобы заниматься правкой кода.

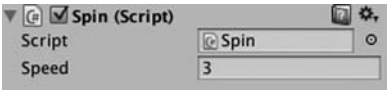


Рис. 2.12. На панели Inspector присутствует объявленная в сценарии общедоступная переменная

Вторая строка, на которую следует обратить внимание в листинге 2.1, касается метода `Rotate()`. Он вставлен в метод `Update()`, а значит, выполняется в каждом кадре. Так как метод `Rotate()` принадлежит классу `Transform`, он вызывается при помощи точечной нотации через компонент преобразования объекта (как и в большинстве объектно-ориентированных языков, если вы введете `transform`, это будет восприниматься как `this.transform`). В рассматриваемом случае преобразование сводится к повороту на `speed` градусов в каждом кадре, что в итоге даст нам плавное вращение. Но почему параметры метода `Rotate()` указаны как (0, speed, 0), а не, допустим, (speed, 0, 0)?

Вспомните, что в трехмерном пространстве есть три оси: *X*, *Y* и *Z*. Интуитивно понятно, каким образом они связаны с местоположением объекта и его перемещениями, но, кроме того, их применяют для описания преобразования поворота. Сходным образом описывают вращение в воздухоплавании, поэтому работающие с трехмерной графикой программисты зачастую пользуются терминами из механики полета: тангаж (pitch), рысканье (yaw) и крен (roll). Значение этих терминов иллюстрирует рис. 2.13; тангаж означает вращение вокруг оси *X*, рысканье — вращение вокруг оси *Y*, крен — вращение вокруг оси *Z*.

Возможность описывать вращение вокруг осей *X*, *Y* и *Z* означает три параметра для метода `Rotate()`. Так как нам требуется только вращение вокруг своей оси без наклонов вперед и назад, меняться должна только координата *Y*, координатам же *X* и *Z* мы просто присвоим значение 0. Надеюсь, вы уже поняли, что получится, если изменить параметры на (speed, 0, 0) и запустить воспроизведение сцены; попробуйте реализовать это на практике!

Существует еще одна тонкость, связанная с вращением и осями координат, реализованная в виде необязательного четвертого параметра метода `Rotate()`.

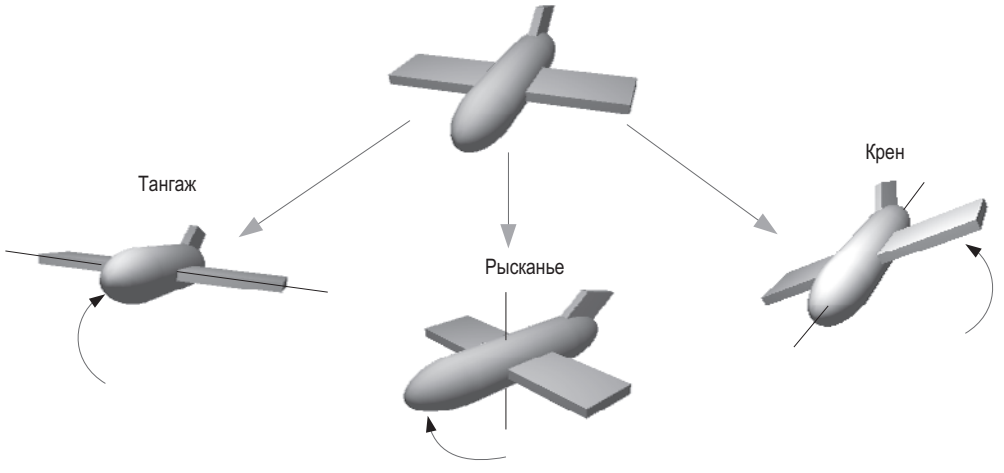


Рис. 2.13. Тангаж, рысканье и крен самолета

2.3.3. Локальные и глобальные координаты

По умолчанию метод `Rotate()` работает с так называемыми локальными координатами, хотя вы также можете использовать глобальную систему координат. Вы сообщаете методу, какой именно системой координат хотите воспользоваться, при помощи четвертого необязательного параметра, который может принимать два значения: `Space.Self` и `Space.World`. Например:

```
Rotate(0, speed, 0, Space.World)
```

Вернитесь к описанию трехмерного координатного пространства и хорошенько подумайте над следующими вопросами: где находится точка (0, 0, 0)? как выбирается положительное направление оси *X*? может ли перемещаться сама координатная система?

Оказывается, у каждого объекта есть собственная точка начала координат и собственные положительные направления осей. И эта координатная система перемещается вместе с объектом. В таких случаях говорят о локальных координатах. При этом трехмерная сцена как целое обладает собственной точкой начала координат и собственными направлениями осей, причем такая координатная система всегда статична. Это так называемые глобальные координаты. Соответственно, указывая вариант системы координат для метода `Rotate()`, вы сообщаете, относительно каких осей *X*, *Y* и *Z* нужно выполнить преобразование поворота (рис. 2.14).

Новичкам в области трехмерной графики эта концепция может показаться сложной для понимания. Различные оси показаны на рис. 2.14 (обратите внимание, что, к примеру, понятие «лево» для самолета отличается от понятия «лево» для сцены в целом), но разницу между локальными и глобальными координатами проще всего понять на примере.

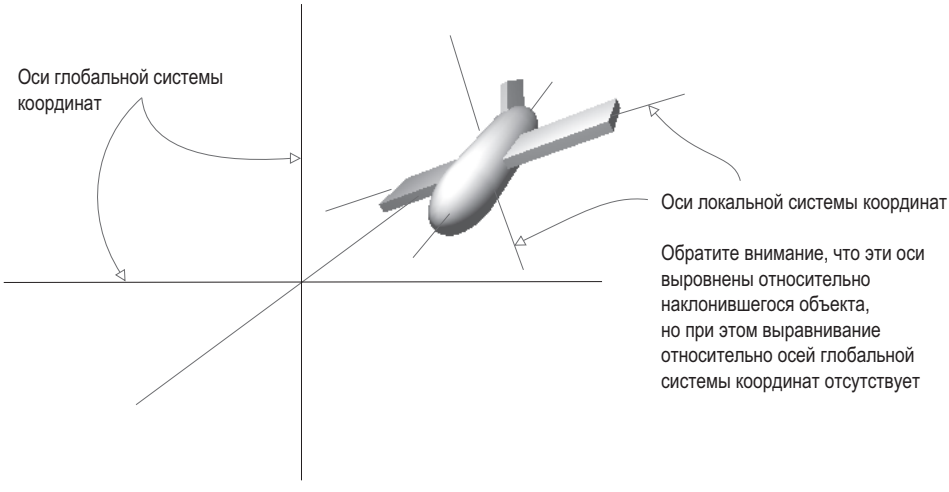


Рис. 2.14. Оси локальной и глобальной систем координат

Для начала выделите игрока и слегка его наклоните (например, повернув относительно оси *X* на 30 градусов). Это приведет к отключению используемых по умолчанию локальных координат, и повороты относительно глобальных и локальных координат начнут выглядеть по-разному. Теперь попробуйте запустить сценарий `Spin` как с добавленным параметром `Space.World`, так и без него; если вам сложно понять, что именно происходит, удалите назначенный игроку компонент вращения и начните вращать расположенный перед игроком наклоненный куб. Вы увидите, что оси вращения зависят от выбранной системы координат.

2.4. Компонент сценария для осмотра сцены: MouseLook

Теперь нужно заставить преобразование поворота реагировать на ввод с помощью мыши. В данном случае подразумевается вращение объекта, к которому присоединен сценарий, то есть игрока. Задача будет решаться в несколько этапов путем постепенного добавления персонажу двигательных возможностей. Сначала мы заставим игрока поворачиваться только из стороны в сторону, а затем — только вверх и вниз. В конечном счете игрок научится смотреть во всех направлениях (поворачиваясь одновременно в горизонтальной и вертикальной плоскостях). Такое поведение называют *слежением за мышью* (mouse-look).

Так как у нас предполагается три типа поведения при повороте (по горизонтали, по вертикали и комбинированный), начнем мы с написания фреймворка, поддерживающего все эти типы. Создайте новый сценарий на C# с именем `MouseLook` и добавьте в него код из следующего листинга.

Листинг 2.2. Фреймворк MouseLook с перечислением для преобразования поворота

```
using UnityEngine;
```

```
using System.Collections;

public class MouseLook : MonoBehaviour {
    public enum RotationAxes {          ← Объявляем структуру данных enum, которая будет
        MouseXAndY = 0,                   сопоставлять имена с параметрами.
        MouseX = 1,
        MouseY = 2
    }
    public RotationAxes axes = RotationAxes.MouseXAndY; ← Объявляем общедоступную переменную,
                                                          которая появится в редакторе Unity.

    void Update() {
        if (axes == RotationAxes.MouseX) {
            // это поворот в горизонтальной плоскости  ← Сюда поместим код для вращения только
        }                                                по горизонтали.
        else if (axes == RotationAxes.MouseY) {
            // это поворот в вертикальной плоскости    ← Сюда поместим код для вращения
        }                                                только по вертикали.
        else {
            // это комбинированный поворот ← Сюда поместим код для комбинированного вращения.
        }
    }
}
```

Обратите внимание, что именно перечисление позволяет сценарию `MouseLook` выбирать, каким образом — в горизонтальной или вертикальной плоскости — будет выполняться поворот. Определив структуру данных `enum`, вы получите возможность задавать значения по имени вместо того, чтобы указывать числа и пытаться запомнить, что означает каждое из них. (Означает ли ноль вращение по горизонтали? Или такому повороту соответствует единица?) Если затем объявить общедоступную переменную, типизированную как перечисление, она появится на панели Inspector в виде раскрывающегося меню, удобного для выбора параметров (рис. 2.15).

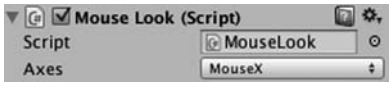


Рис. 2.15. Панель Inspector отображает сопоставленные перечислениям общедоступные переменные в виде раскрывающихся меню

Удалите компонент Spin (тем же способом, которым ранее удалялся капсульный коллайдер) и вместо него присоедините к игроку новый сценарий. В процессе работы над кодом для перехода от одной ветки кода к другой пользуйтесь меню Axes. По очереди выбирая горизонтальное/вертикальное вращение, вы сможете написать код для каждой ветки условной конструкции.

2.4.1. Горизонтальное вращение, следящее за указателем мыши

Первая и наиболее простая ветка соответствует вращению в горизонтальной плоскости. Для начала напишем уже знакомую вам команду из листинга 2.1, заставляющую объект поворачиваться. Не забудьте объявить общедоступную переменную для

скорости вращения; сделайте это после объявления осей, но до метода `Update()`, назвав новую переменную `sensitivityHor`, так как слово «speed» имеет слишком общий смысл и не позволит отличать варианты поворота друг от друга. Увеличьте значение этой переменной до 9, так как совсем скоро она начнет масштабироваться. Исправленный код приведен в следующем листинге.

Листинг 2.3. Поворот по горизонтали, пока не связанный с движениями указателя

```
...
public RotationAxes axes = RotationAxes.MouseXAndY; ← Курсивом выделен код, который уже присутствует в сценарии; здесь он приведен для удобства.

public float sensitivityHor = 9.0f; ← Объявляем переменную для скорости вращения.

void Update() {
    if (axes == RotationAxes.MouseX) {
        transform.Rotate(0, sensitivityHor, 0); ← Сюда мы поместим команду Rotate, чтобы она запускалась в каждом кадре.
    }
...
```

Если запустить сценарий, сцена, как и раньше, начнет вращаться, но намного быстрее, потому что теперь скорость вращения вокруг оси *Y* равна 9, а не 3. Теперь нам нужно сделать так, чтобы преобразование возникало в ответ на движения указателя мыши, поэтому создадим новый метод: `Input.GetAxis()`. Класс `Input` обладает множеством методов для обработки информации, поступающей с устройств ввода (таких, как мышь). В частности, метод `GetAxis()` возвращает числа, связанные с движениями указателя мыши (положительные или отрицательные в зависимости от направления движения). Он принимает в качестве параметра имя нужной оси. А горизонтальная ось у нас называется `Mouse X`.

Если умножить скорость вращения на координату оси, объект начнет поворачиваться вслед за указателем мыши. Скорость масштабируется в соответствии с перемещениями указателя, уменьшаясь до нуля и даже меняя направление. Новый вид команды `Rotate` показан в следующем листинге.

Листинг 2.4. Команда Rotate, реагирующая на движения указателя мыши

```
...
transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0); ← Метод GetAxis() получает данные, вводимые с помощью мыши.
...
```

Щелкните на кнопке Play и подвигайте мышью в разные стороны. Объект начнет поворачиваться вправо и влево вслед за указателем. Видите, как здорово! Теперь нужно заставить игрока вращаться еще и в вертикальной плоскости.

2.4.2. Поворот по вертикали с ограничениями

Вращение в горизонтальной плоскости было реализовано у нас с помощью метода `Rotate()`, но для поворота по вертикали мы воспользуемся другим способом. Дело в том, что указанный метод при всем его удобстве в осуществлении преобразований недостаточно гибок. Он применим только для неограниченного приращения угла поворота, что в нашей ситуации подходит только для вращения в горизонтальной плоскости. В случае же поворота по вертикали требуется задать предел наклона вниз

и вверх. Код этого преобразования для сценария `MouseLook` представлен в следующем листинге.

Листинг 2.5. Поворот в вертикальной плоскости для сценария MouseLook

```
...
public float sensitivityHor = 9.0f;
public float sensitivityVert = 9.0f; ← Объявляем переменные, задающие поворот в вертикальной плоскости.

public float minimumVert = -45.0f;
public float maximumVert = 45.0f;

private float _rotationX = 0; ← Объявляем закрытую переменную для угла поворота по вертикали.

void Update() {
    if (axes == RotationAxes.MouseX) {
        transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
    }
    else if (axes == RotationAxes.MouseY) {
        _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert; ← Увеличиваем угол поворота по вертикали в соответствии с перемещениями указателя мыши.

        _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert); ← Фиксируем угол поворота по вертикали в диапазоне, заданном минимальным и максимальным значениями.

        float rotationY = transform.localEulerAngles.y; ← Сохраняем одинаковый угол поворота вокруг оси Y (т. е. вращение в горизонтальной плоскости отсутствует).

        transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0); ← Создаем новый вектор из сохраненных значений поворота.
    }
...
```

Выбираем в меню Axes компонента MouseLook вращение по вертикали и воспроизводим новый сценарий. Теперь сцена вместо поворотов из стороны в сторону будет вслед за движениями указателя мыши наклоняться вверх и вниз. При этом углы поворота ограничиваются указанными вами пределами.

Этот код знакомит вас с рядом новых понятий, которые следует объяснить более подробно. Во-первых, теперь мы не пользуемся методом `Rotate()`, поэтому нам требуется еще одна переменная (она называется `_rotationX`, так как вертикальное вращение происходит вокруг оси *X*), предназначенная для сохранения угла поворота. Метод `Rotate()` просто увеличивает угол поворота, в то время как на этот раз мы его задаем в явном виде. Другими словами, если в предыдущем листинге мы просили «добавить 5 к текущему значению угла», то теперь мы «присваиваем углу поворота значение 30». Разумеется, нам и в этом случае нужно увеличивать угол поворота, именно поэтому в коде присутствует оператор `-=`. То есть мы вычитаем значение из угла поворота, а не присваиваем это значение углу. Так как методом `Rotate()` мы не пользуемся, манипулировать углом поворота можно разными способами, а не только

увеличивая его. Этот параметр умножается на `Input.GetAxis()`, совсем как в коде для вращения в горизонтальной плоскости, просто на этот раз мы выясняем значение `Mouse Y`, ведь нас интересует вертикальная ось.

Следующая строка также посвящена манипуляциям углом поворота. Метод `Mathf.Clamp()` позволяет ограничить этот параметр максимальным и минимальным значениями. Эти предельные значения задаются объявленными общедоступными переменными, которые гарантируют поворот сцены вверх и вниз всего на 45 градусов. Метод `Clamp()` работает не только с преобразованиями поворота. Он в принципе используется для сохранения числового значения переменной в заданных пределах. В экспериментальных целях превратите строку с методом `Clamp()` в комментарий; вы увидите, как объект начнет свободно вращаться в вертикальной плоскости! Понятно, что нам совсем не обязательно смотреть на сцену, перевернутую с ног на голову; отсюда и появляются ограничения.

Так как угловое свойство преобразования выражается переменной `Vector3`, мы должны создать новую переменную с углом поворота, которая будет передаваться в конструктор. Метод `Rotate()` этот процесс автоматизировал, увеличив угол поворота и затем создав новый вектор.

ОПРЕДЕЛЕНИЕ Вектором (vector) называется набор чисел, сохраняемых как единое целое. К примеру, Vector3 состоит из трех чисел (помеченных как x, y, z).

ВНИМАНИЕ Мы создаем новый вектор Vector3, вместо того чтобы поменять значения у существующего, так как в случае преобразований эти значения предназначены только для чтения. Это крайне распространенная ошибка.

УГЛЫ ЭЙЛЕРА И КВАТЕРНИОНЫ

Возможно, вам интересно, почему свойство называется localEulerAngles, а не localRotation. Для ответа на этот вопрос вам нужно познакомиться с концепцией, называемой кватернионами (quaternions).

Кватернионы представляют собой математическую конструкцию для представления вращения объектов. Они отличаются от углов Эйлера, то есть используемого нами подхода с осями *X*, *Y*, *Z*. Помните обсуждение тангажа, рысканья и крена? В этом случае вращения были представлены с помощью углов Эйлера. Кватернионы… совсем другие. Объяснить их природу сложно, так как для этого нужно углубиться в запутанные дебри высшей математики, включающие в себя движение через четыре измерения. Если вам требуется детальное объяснение, попробуйте прочесть документ http://wat.gamedev.ru/articles/quaternions.

Несколько проще объяснить, почему кватернионы используются для представления вращений: они более равномерно реализуют интерполяцию между углами поворота (то есть переход от одного промежуточного значения к другому).

Свойство localRotation выражено через кватернионы, а не через углы Эйлера. При этом в Unity существует и свойство, выражаемое через углы Эйлера, благодаря которому понять, как именно происходит процесс поворота, становится проще; преобразование одного варианта значений в другой и обратно делается автоматически. Инструмент Unity выполняет все сложные вычисления в фоновом режиме, избавляя вас от необходимости проделывать их вручную.

Осталось написать код для последнего вида вращения, происходящего одновременно в горизонтальной и вертикальной плоскостях.

2.4.3. Одновременные горизонтальное и вертикальное вращения

В последнем фрагменте кода метод `Rotate()` также не применяется, так как нам снова требуется ограничить диапазон углов поворота по вертикали. Это означает, что горизонтальный поворот тоже нужно вычислять вручную. Напоминаю, что метод `Rotate()` автоматизировал процесс приращения угла поворота.

Листинг 2.6. Сценарий MouseLook, поворачивающий объект одновременно по горизонтали и по вертикали

```
...
else {
    _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
    _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

    float delta = Input.GetAxis("Mouse X") * sensitivityHor; ← Приращение угла поворота через значение delta.
    float rotationY = transform.localEulerAngles.y + delta; ← Значение delta – это величина изменения угла поворота.

    transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
}
...
```

Первая пара строк, посвященная переменной `_rotationX`, полностью аналогична таким же строкам из предыдущего сценария. Просто помните, что вращение в вертикальной плоскости происходит вокруг оси *X* объекта. А так как вращение в горизонтальной плоскости больше не реализуется через метод `Rotate()`, появились строки с переменными `delta` и `rotationY`. Дельта — это распространенное математическое обозначение «величины изменения», поэтому наше вычисление переменной `delta` дает величину, на которую следует поменять угол поворота. Затем она добавляется к текущему значению этого угла для придания объекту желаемой ориентации.

В конце оба угла наклона — по вертикали и по горизонтали — используются для создания нового вектора, который назначается угловому свойству компонента `Transform`.

ОТКЛЮЧАЕМ ВЛИЯНИЕ СРЕДЫ НА ИГРОКА

Хотя для данного проекта это не имеет особого значения, в большинстве современных игр от первого лица на все элементы сцены действует модель физической среды. Она заставляет объекты отскакивать и опрокидываться; такое поведение подходит для большинства объектов, но вращение нашего игрока должно контролироваться исключительно мышью, не попадая под влияние смоделированной физической среды.

По этой причине в сценариях, где присутствует ввод с помощью мыши, к компоненту Rigidbody игрока обычно добавляют свойство freezeRotation. Поместите в сценарий MouseLook вот такой метод Start():

```
...
void Start() {
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null) ← Проверяем, существует ли этот компонент.
    body.freezeRotation = true;
}
...
```

Rigidbody — это дополнительный компонент, которым может обладать объект. Моделируемая физическая среда влияет на этот компонент и управляет объектами, к которым он присоединен.

На случай, если вы запутались и не знаете, куда вставить очередной фрагмент кода, привожу полный текст сценария. Кроме того, вы можете скачать пример проекта.

Листинг 2.7. Окончательный вид сценария MouseLook

```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {
  public enum RotationAxes {
    MouseXAndY = 0,
    MouseX = 1,
    MouseY = 2
  }
  public RotationAxes axes = RotationAxes.MouseXAndY;

  public float sensitivityHor = 9.0f;
  public float sensitivityVert = 9.0f;

  public float minimumVert = -45.0f;
  public float maximumVert = 45.0f;

  private float _rotationX = 0;

  void Start() {
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null)
    body.freezeRotation = true;
  }

  void Update() {
    if (axes == RotationAxes.MouseX) {
      transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
    }
    else if (axes == RotationAxes.MouseY) {
      _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
      _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

      float rotationY = transform.localEulerAngles.y;

      transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
    }
    else {
      _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
      _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

      float delta = Input.GetAxis("Mouse X") * sensitivityHor;
      float rotationY = transform.localEulerAngles.y + delta;

      transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
    }
  }
}
```

После запуска этого сценария вы получите возможность смотреть во всех направлениях, просто перемещая указатель мыши. Великолепно! Но вы все еще стоите на месте, как орудийная башня. В следующем разделе игрок начнет перемещаться по сцене.

2.5. Компонент для клавиатурного ввода

Возможность смотреть по сторонам в ответ на перемещения указателя мыши относится к важным фрагментам элементов управления персонажем в играх от первого лица, но это только половина дела. Кроме этого, игрок должен перемещаться по сцене в ответ на клавиатурный ввод. Поэтому в дополнение к компоненту для элемента управления мышью напишем компонент для клавиатурного ввода; создайте новый компонент C# script с именем `FPSInput` и присоедините его к игроку (в дополнение к сценарию `MouseLook`). При этом мы на время ограничим компонент MouseLook только горизонтальным вращением.

СОВЕТ Рассматриваемые нами элементы управления клавиатурным вводом и вводом с помощью мыши помещены в отдельные сценарии. Вы вовсе не обязаны структурировать код подобным образом и вполне можете поместить все в единый сценарий «элементов управления игроком». Но, как правило, модульные системы (такие, как в Unity) стремятся к максимальной гибкости, и, соответственно, эффективнее всего моделировать функциональность как набор отдельных небольших компонентов.

Код, который мы писали, влиял только на ориентацию объекта, теперь же мы будем менять только его местоположение. Как показано в листинге 2.8, мы берем код вращения и вводим его в сценарий `FPSInput`, заменяя метод `Rotate()` методом `Translate()`. После щелчка на кнопке Play сцена начнет скользить вверх, а не вращаться, как раньше. Попытайтесь поменять значения параметров и посмотреть, как изменится движение (например, после того как вы поменяете местами первое и второе числа); после некоторого количества экспериментов можно будет перейти к добавлению клавиатурного ввода.

Листинг 2.8. Код вращения из первого листинга с парой небольших изменений

```
using UnityEngine;
using System.Collections;
public class FPSInput : MonoBehaviour {
  public float speed = 6.0f; ← Необязательный элемент на случай, если вы захотите увеличить скорость.
  void Update() {
    transform.Translate(0, speed, 0); ← Меняем метод Rotate() на метод Translate()
  }
}
```

2.5.1. Реакция на нажатие клавиш

Код перемещения в ответ на нажатия клавиш (показанный в листинге 2.9) аналогичен коду вращения в ответ на движение указателя мыши. Мы снова используем метод `GetAxis()`, причем аналогичным образом.

Листинг 2.9. Изменение положения в ответ на нажатие клавиш

```
...
void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed; ← "Horizontal" и "Vertical" – это дополнительные имена для сопоставления с клавиатурой.
  float deltaZ = Input.GetAxis("Vertical") * speed;
  transform.Translate(deltaX, 0, deltaZ);
}
```

Как и раньше, значения, задаваемые методом `GetAxis()`, умножаются на скорость, давая в итоге величину смещения. Но если раньше в ответ на запрос оси мы получали ответ «Mouse имя оси», то теперь это значения `Horizontal` или `Vertical`. Это абстрактные представления для параметров ввода в Unity; если в меню Edit навести указатель мыши на строку Project Settings и выбрать в открывшемся меню команду Input, откроется список абстрактных имен ввода и соответствующих им элементов управления. Клавиши с указывающими влево/вправо стрелками и с буквами A/D соответствуют имени `Horizontal`, в то время как клавиши со стрелками, указывающими вверх/вниз, и с буквами W/S соответствуют имени `Vertical`.

Обратите внимание, что значения перемещения меняются по координатам *X* и *Z*. Вы могли заметить в процессе экспериментов с методом `Translate()`, что изменение координаты *X* соответствует движению из стороны в сторону, в то время как изменение координаты *Z* означает движение вперед и назад.

Вставив в сценарий этот новый код перемещения, вы получите возможность двигать объект по сцене нажатием клавиш со стрелками или клавиш с буквами WASD. Это стандартная ситуация для большинства игр от первого лица. Сценарий, управляющий перемещениями, практически готов, осталось только внести в него некоторые изменения.

2.5.2. Независимая от скорости работы компьютера скорость перемещений

Пока вы запускаете код только на собственном компьютере, этот аспект незаметен, но на разных машинах игрок будет перемещаться с разной скоростью. Ведь более мощные компьютеры быстрее обрабатывают код и графику. В настоящее время скорость перемещения вашего игрока привязана к скорости работы компьютера. Это называется *зависимостью от кадровой частоты* (frame rate dependent), так как код движения зависит от частоты кадров игры.

Например, представьте, что вы запускаете деморолик на двух компьютерах, один из которых отображает на мониторе 30 кадров в секунду, а второй 60. Это означает, что на втором компьютере метод `Update()` будет вызываться в два раза чаще, двигая объект с одной и той же скоростью 6 при каждом вызове. В итоге при частоте 30 кадров в секунду скорость движения составит 180 единиц в секунду, в то время как частота 60 кадров в секунду обеспечит скорость перемещения в 360 единиц в секунду. Такая вариативность неприемлема для большинства игр.

Решить эту проблему позволяет такое редактирование кода, в результате которого мы получим *независимость от кадровой частоты* (frame rate independent). То есть

скорость перемещения перестанет зависеть от частоты кадров игры. Это обеспечивается сменой скорости в соответствии с частотой кадров. Она должна уменьшаться или увеличиваться в зависимости от того, насколько быстро работает компьютер. Достигнуть такого результата нам поможет умножение скорости на переменную `deltaTime`, как показано в следующем листинге.

Листинг 2.10. Движение с независимостью от кадровой частоты благодаря переменной deltaTime

```
...
void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed;
  float deltaZ = Input.GetAxis("Vertical") * speed;
  transform.Translate(deltaX * Time.deltaTime, 0, deltaZ * Time.deltaTime);
}
...
```

Это простое изменение. Класс `Time` обладает рядом свойств и методов, позволяющих регулировать время. К таким свойствам относится и `deltaTime`. Так как мы знаем, что дельта указывает на величину изменения, переменная `deltaTime` означает величину изменения во времени. А конкретно — это время между кадрами. Его величина зависит от частоты кадров (например, при частоте 30 кадров в секунду `deltaTime` составляет 1/30 секунды). Поэтому умножение скорости на эту переменную приведет к масштабированию скорости на различных компьютерах.

Теперь движение нашего персонажа одинаково на всех машинах. Но сценарий еще не завершен; ведь, перемещаясь по комнате, мы можем проходить сквозь стены.

2.5.3. Компонент CharacterController для распознавания столкновений

Назначенные объекту напрямую преобразования не затрагивают такого аспекта, как распознавание столкновений. В результате персонаж начинает ходить сквозь стены. Поэтому нам требуется компонент CharacterController. Именно он обеспечит естественность перемещений нашего персонажа. Напомню, что в момент настройки игрока мы присоединяли этот компонент, поэтому теперь остается воспользоваться им в коде сценария `FPSInput` (листинг 2.11).

Листинг 2.11. Перемещение компонента CharacterController вместо компонента Transform

```
...
private CharacterController _charController; ← Переменная для ссылки на компонент CharacterController.

void Start() {
  _charController = GetComponent<CharacterController>(); ← Доступ к другим компонентам, присоединенным к этому же объекту.
}

void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed;
  float deltaZ = Input.GetAxis("Vertical") * speed;
  Vector3 movement = new Vector3(deltaX, 0, deltaZ);
```

```
    movement = Vector3.ClampMagnitude(movement, speed);
    movement *= Time.deltaTime;
    movement = transform.TransformDirection(movement);
    _charController.Move(movement);
}
...
```

← Ограничим движение по диагонали той же скоростью, что и движение параллельно осям.

← Преобразуем вектор движения от локальных к глобальным координатам.

← Заставим этот вектор перемещать компонент CharacterController.

В этом фрагменте кода появляется несколько новых концепций. Прежде всего, это переменная для ссылки на компонент CharacterController. Она просто создает локальную ссылку на объект (объект в коде — не путайте его с объектами сцены); ссылаться на этот экземпляр компонента могут разные сценарии. Изначально эта переменная пуста, поэтому, прежде чем ею пользоваться, следует указать объект, на который вы будете ссылаться с ее помощью. Именно здесь появляется метод `GetComponent()`; он возвращает другие компоненты, присоединенные к тому же объекту `GameObject`. Вместо передачи параметра в скобках мы воспользовались синтаксисом C# и определили тип в угловых скобках `<>`.

Теперь, когда у нас появилась ссылка на компонент CharacterController, можно вызвать для этого контроллера метод `Move()`. Мы передаем этому методу вектор тем же способом, которым код вращения в зависимости от указателя мыши использовал вектор для значений поворота. А аналогично тому, как мы ограничивали значения поворота, мы задействуем метод `Vector3.ClampMagnitude()`, чтобы ограничить модуль вектора скоростью движения; мы берем именно этот метод, потому что в противном случае движение по диагонали происходило бы быстрее движения вдоль координатных осей (нарисуйте катеты и гипотенузу прямоугольного треугольника).

Но вектор движения обладает еще одной особенностью, связанной с выбором системы координат. Вы уже сталкивались с ней при обсуждении преобразования поворота. Например, мы создадим вектор, перемещающий объект влево. Но может оказаться, что у игрока это направление не совпадает с направлением координатных осей. То есть мы говорим о локальном, а не о глобальном пространстве. Поэтому методу `Move()` следует передавать вектор движения, определенный в глобальном пространстве, а значит, нам требуется преобразование от локальных к глобальным координатам. Математически это крайне сложная операция, но, к счастью для нас, об этом позаботится Unity. Нам же достаточно вызвать метод `TransformDirection()`, чтобы, как следует из его названия, преобразовать направление.

ОПРЕДЕЛЕНИЕ Слово «transform», используемое в качестве глагола, означает преобразование от одного координатного пространства к другому (если вы не помните, что такое координатное пространство, перечитайте раздел 2.3.3). Не следует путать его с аналогичным существительным, которое означает как компонент Transform, так и изменение положения объектов в сцене. В данном случае значения терминов перекрываются, так как в основе лежит одна и та же концепция.

Запустите код. Если это еще не сделано, установите для компонента MouseLook вращение одновременно по горизонтали и вертикали. Вы можете неограниченно осматривать пространство вокруг себя и летать, нажимая соответствующие клавиши. Это здорово, если вам требуется игрок, умеющий летать. Но как сделать, чтобы он перемещался исключительно по земле?

2.5.4. Ходить, а не летать

Теперь, когда распознавание столкновений работает, в сценарий можно добавить силу тяжести, чтобы игрок стоял на полу. Объявим переменную `gravity` и воспользуемся ее значением для оси *Y*, как показано в следующем листинге.

Листинг 2.12. Добавление силы тяжести в код движения

```
...
public float gravity = -9.8f;
...
void Update() {
  ...
  movement = Vector3.ClampMagnitude(movement, speed);
  movement.y = gravity; ← Используем значение переменной gravity вместо нуля.
  ...
```

Теперь на игрока действует постоянная сила, тянущая его вниз. К сожалению, она не всегда направлена строго вниз, так как изображающий игрока объект может наклоняться вниз и вверх, следуя за движениями указателя мыши. К счастью, у нас есть все, чтобы исправить этот недочет, достаточно слегка поменять настройки компонента по отношению к игроку. Первым делом ограничьте компонент MouseLook только горизонтальным вращением. Затем добавьте этот компонент к камере и ограничьте его только вертикальным вращением. В результате на перемещения указателя мыши у вас будут реагировать два разных объекта!

Так как игрок теперь поворачивается только в горизонтальной плоскости, решилась проблема с наклоном вектора силы тяжести. При этом объект камеры является дочерним по отношению к игроку (помните, как мы установили эту связь на вкладке Hierarchy?), поэтому, обладая способностью поворачиваться в вертикальной плоскости независимо от игрока, она вращается по горизонтали вслед за ним.

СОВЕРШЕНСТВОВАНИЕ ГОТОВОГО СЦЕНАРИЯ

Воспользуйтесь методом RequireComponent(), чтобы проверить, присоединены ли остальные требуемые сценарию компоненты. Иногда такие компоненты являются необязательными (то есть можно сказать, что «если этот компонент присоединен, тогда...»), но бывают ситуации, когда без каких-то компонентов сценарий работать не будет. Поэтому добавьте в верхнюю часть сценария этот метод, чтобы обеспечить выполнение зависимостей, указав требуемый компонент в качестве параметра.

Кроме того, можно вставить в верхнюю часть сценария метод AddComponentMenu(), который добавит сценарий в меню компонентов в редакторе Unity. Достаточно указать имя элемента меню, и вы получите возможность выбирать данный сценарий в списке, открываемом щелчком на кнопке Add Component в нижней части панели Inspector. Очень удобно!

Код добавления обоих методов выглядит примерно так:

```
using UnityEngine;
using System.Collections;
[RequireComponent(typeof(CharacterController))]
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
```

Листинг 2.13 демонстрирует полностью готовый сценарий. В дополнение к слегка скорректированной настройке компонентов игрока мы добавили игроку возможность

ходить по комнате. Несмотря на наличие переменной `gravity`, вы легко можете заставить игрока летать, введя в поле Gravity на панели Inspector значение 0.

Листинг 2.13. Готовый сценарий FPSInput

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
  public float speed = 6.0f;
  public float gravity = -9.8f;

  private CharacterController _charController;

  void Start() {
    _charController = GetComponent<CharacterController>();
  }

  void Update() {
    float deltaX = Input.GetAxis("Horizontal") * speed;
    float deltaZ = Input.GetAxis("Vertical") * speed;
    Vector3 movement = new Vector3(deltaX, 0, deltaZ);
    movement = Vector3.ClampMagnitude(movement, speed);

    movement.y = gravity;

    movement *= Time.deltaTime;
    movement = transform.TransformDirection(movement);
    _charController.Move(movement);
  }
}
```

Поздравляю вас с созданием 3D-проекта! В этой главе вы получили большое количество новой информации и теперь знаете, как создать код перемещения в Unity. Но как бы ни радовал нас первый демонстрационный ролик, до готовой игры ему еще очень далеко. В плане проекта этот пункт описан как создание базовой сцены в шутере от первого лица, но какой же это шутер, если мы лишены возможности стрелять? Поэтому похвалите себя за успешно завершенный фрагмент проекта и готовьтесь к следующему шагу.

2.6. Заключение

- Трехмерное координатное пространство определяется осями *X*, *Y* и *Z*.
- Сцену создают помещенные в комнату объекты и источники света.
- Игрока в сцене от первого лица, по сути, представляет камера.
- Код движения в каждом кадре циклически повторяет небольшие преобразования.
- Элементы управления персонажем в игре от первого лица состоят из указателя мыши, отвечающего за вращение, и клавиатуры, отвечающей за перемещения.

3 Добавляем в игру врагов и снаряды

- ✓ Как игрок и его враги получают возможность целиться и стрелять
- ✓ Попадания и реакция на них
- ✓ Заставляем врагов перемещаться
- ✓ Порождение новых объектов сцены

Ролик, демонстрирующий перемещения объекта, который вы создали в предыдущей главе, выглядит здорово, но до полноценной игры ему далеко. Давайте попробуем превратить его в шутер от первого лица. Для этого нам не хватает возможности стрелять, а также врагов, которых требуется поразить. Начнем мы с написания сценариев, превращающих нашего игрока в стрелка. После этого мы заполним сцену врагами, добавив в числе прочего код, позволяющий бесцельно перемещаться по сцене и реагировать на попадание. В заключение мы дадим врагам возможность отстреливаться, кидая в игрока огненные шары. Написанные в главе 2 сценарии при этом редактироваться не будут; мы просто добавим в проект новые сценарии с дополнительными функциональными возможностями.

Я выбрал для этого проекта шутер от первого лица по двум причинам. Во-первых, подобные игры популярны, ведь людям нравится стрелять. Во-вторых, я учел набор приемов, с которыми вы можете ознакомиться в процессе работы. Создание такой игры дает возможность изучить несколько фундаментальных концепций трехмерного моделирования. Например, вы узнаете, что такое *бросание луча* (raycasting). Подробно особенности этого приема я объясню чуть позже, а пока достаточно информации о том, что он позволяет решать множество различных задач в трехмерном моделировании и применяется во множестве ситуаций, из которых интуитивно наиболее понятной является реализация стрельбы.

Создание блуждающих целей, которые требуется поразить, дает нам замечательный повод исследовать код контролируемых компьютером персонажей, а также техники

отправки сообщений и порождения объектов. Более того, поведение блуждающих целей — это еще один аспект, в моделировании которого нам пригодится прием испускания луча, так что мы познакомимся с еще одним вариантом его применения. Также широко применим демонстрируемый в рамках этого проекта подход к рассылке сообщений. В следующих главах вы встретите другие варианты применения указанных приемов. Впрочем, даже в рамках одной главы они используются в различных ситуациях.

По сути дела, мы добавляем в проект по одной функциональной возможности за раз, причем на каждом этапе в игру можно играть, но всегда остается ощущение, что каких-то элементов не хватает. Вот план нашей будущей работы:

1. Написать код, позволяющий игроку стрелять.
2. Создать статичные цели, реагирующие на попадание.
3. Заставить цели перемещаться по сцене.
4. Вызвать автоматическое появление новых блуждающих целей.
5. Дать возможность целям/врагам кидать в игрока огненные шары.

ПРИМЕЧАНИЕ Проект, которым мы будем заниматься в этой главе, предполагает наличие демонстрационного ролика с перемещающимся по сцене персонажем. Этот ролик создавался в главе 2, но если вы ее пропустили, скачайте соответствующие файлы с сайта книги.

3.1. Стрельба путем бросания лучей

Первой дополнительной функциональной возможностью, которую мы добавим в наш демонстрационный ролик, станет стрельба. Умение оглядываться по сторонам и перемещаться в шутере от первого лица является, без сомнения, решающим, но игра не начнется, пока игроки не смогут влиять на окружающее пространство и показывать свое мастерство. Стрельба в трехмерных играх реализуется несколькими способами, наиболее важным из которых является бросание лучей.

3.1.1. Что такое бросание лучей?

Название приема подразумевает, что вам предстоит бросать лучи. Но что именно в данном случае подразумевается под словом *луч*?

ОПРЕДЕЛЕНИЕ Лучом (ray) в сцене называется воображаемая, невидимая линия, начинающаяся в некоторой точке и распространяющаяся в определенном направлении.

Прием бросания луча иллюстрирует рис. 3.1. Вы формируете луч и затем определяете, с чем он пересекается. Подумайте, что происходит, когда вы стреляете из пистолета: пуля вылетает из точки, в которой находится пистолет, и летит по прямой вперед, пока не столкнется с каким-нибудь препятствием. Луч можно сравнить с путем пули, а бросание луча аналогично выстрелу из пистолета и наблюдению за тем, куда попадет пуля.

Думаю, очевидно, что реализация метода бросания лучей зачастую требует сложных математических расчетов. Трудно не только просчитать пересечение линии

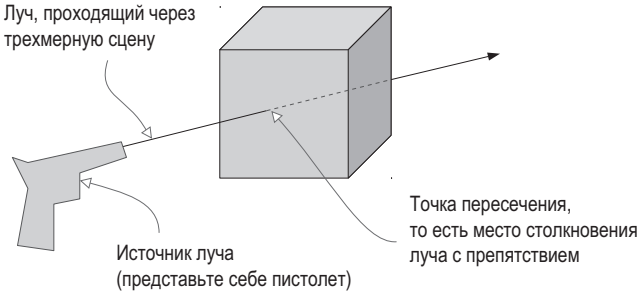


Рис. 3.1. Луч представляет собой воображаемую линию, и при бросании луча выясняется, с чем этот луч пересекается

с трехмерной плоскостью, но и сделать это для всех сеточных объектов в сцене (напоминаю, что сеточным объектом называется трехмерный видимый объект, сконструированный из множества соединенных друг с другом линий и фигур). К счастью, Unity берет на себя все сложные математические расчеты, оставляя вам заботу о задачах более высокого уровня, например о месте и причине появления лучей.

В текущем проекте ответом на второй вопрос (почему возникает луч) является имитация выпускания пули. В шутерах от первого лица луч, как правило, начинается из места, где располагается камера, и распространяется по центру ее поля зрения. Иными словами, вы проверяете наличие объекта непосредственно перед камерой — в Unity есть соответствующие команды. Рассмотрим их более подробно.

3.1.2. Имитация стрельбы командой ScreenPointToRay

Итак, мы реализуем стрельбу проецированием луча, начинающегося в месте нахождения камеры и распространяющегося по центральной линии ее поля зрения. Проецирование луча по центральной линии поля зрения камеры представляет собой частный случай действия, которое называется *выбором с помощью мыши*.

ОПРЕДЕЛЕНИЕ Выбор с помощью мыши (mouse picking) означает действие по выбору в трехмерной сцене точки, непосредственно попадающей под указатель мыши.

Эта операция в Unity осуществляется методом `ScreenPointToRay()`. Происходящее иллюстрирует рис. 3.2. Метод создает луч, начинающийся с камеры, и проецирует его по линии, проходящей через указанные экранные координаты. Обычно при выборе с помощью мыши используются координаты указателя, но в шутерах от первого лица эту роль играет центр экрана. Появившийся луч передается методу `Physics.Raycast()`, который и выполняет его «бросание».

Напишем код, использующий только что рассмотренные нами методы. Создайте в редакторе Unity новый компонент C# script и присоедините его к камере (а не к объекту, представляющему игрока). Добавьте в него код следующего листинга.

В листинге следует обратить внимание на несколько моментов. Во-первых, компонент Camera вызывается в методе `Start()`, совсем как компонент CharacterController в предыдущей главе. Остальная часть кода помещена в метод `Update()`, так как положение

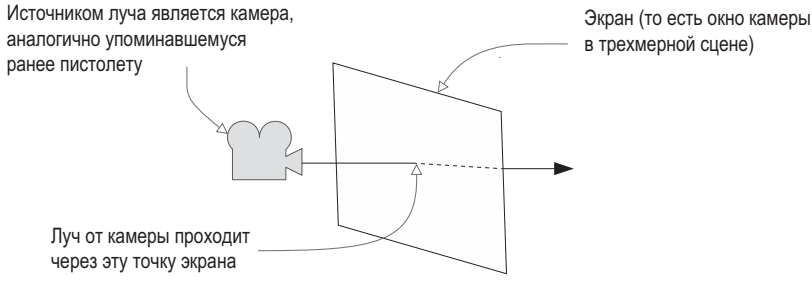


Рис. 3.2. Метод ScreenPointToRay() проецирует луч от камеры через указанные экранные координаты

Листинг 3.1. Сценарий RayShooter, присоединяемый к камере

```
using UnityEngine;
using System.Collections;

public class RayShooter : MonoBehaviour {
  private Camera _camera;

  void Start() {
    _camera = GetComponent<Camera>(); ← Доступ к другим компонентам, присоединенным к этому же объекту.
  }

  void Update() {
    if (Input.GetMouseButtonDown(0)) { ← Реакция на нажатие кнопки мыши.
      Vector3 point = new Vector3( ← Середина экрана – это половина его ширины и высоты.
        _camera.pixelWidth/2, _camera.pixelHeight/2, 0);
      Ray ray = _camera.ScreenPointToRay(point); ← Создание в этой точке луча методом ScreenPointToRay().
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit)) { ← Испущенный луч заполняет информацией переменную, на которую имеется ссылка.
        Debug.Log("Hit " + hit.point); ← Загружаем координаты точки, в которую попал луч.
      }
    }
  }
}
```

мыши нам требуется проверять снова и снова. Метод `Input.GetMouseButtonDown()` в зависимости от того, нажимается ли кнопка мыши, возвращает значения `true` и `false`. Помещение этой команды в условную инструкцию означает, что указанный код выполняется только после щелчка кнопкой мыши. Мы хотим, чтобы выстрел возникал по щелчку мыши, именно поэтому условная инструкция проверяет ее состояние.

Вектор создается, чтобы определить для луча экранные координаты (напоминаю, что вектором называются несколько связанных друг с другом чисел, хранимых как единое целое). Параметры `pixelWidth` и `pixelHeight` дают нам размер экрана. Определить его центр можно, разделив эти значения пополам. Хотя координаты экрана являются двухмерными, то есть у нас есть только размеры по вертикали и горизонтали, а глубина отсутствует, вектор `Vector3` все равно создается, так как метод `ScreenPointToRay()` требует данных этого типа (возможно, потому, что расчет луча

включает в себя операции с трехмерными векторами). Вызванный для этого набора координат метод `ScreenPointToRay()` дает нам объект `Ray` (программный, а не игровой объект; эти два объекта часто путают).

Затем луч передается в метод `Raycast()`, причем это не единственный передаваемый в этот метод объект. Есть также структура данных `RaycastHit`; она представляет собой набор информации о пересечении луча, в том числе о точке, в которой возник луч, и об объекте, с которым он столкнулся. Используемый в данном случае синтаксис языка C# гарантирует, что структура данных, с которой работает команда, является тем же объектом, существующим вне команды, в противоположность ситуациям, когда в разных областях действия функции используются разные копии объекта.

В конце мы вызываем метод `Physics.Raycast()`, который проверяет место пересечения рассматриваемого луча, собирает информацию об этом пересечении и возвращает значение `true` в случае столкновения луча с препятствием. Так как возвращаемое значение принадлежит типу `Boolean`, метод можно поместить в инструкцию проверки условия, совсем как метод `Input.GetMouseButtonDown()` чуть раньше.

Пока что на пересечения код реагирует консольным сообщением с координатами точки, в которой луч столкнулся с препятствием (значения *X*, *Y*, *Z* мы обсуждали в главе 2). Но понять, в каком именно месте это произошло, или показать, где находится центр экрана (то есть место, через которое проходит луч), практически нереально. Поэтому давайте добавим в сцену визуальные индикаторы.

3.1.3. Добавление визуальных индикаторов для прицеливания и попаданий

Теперь нам нужно добавить в сцену два вида визуальных индикаторов: точку прицеливания в середине экрана и метку в месте столкновения луча с препятствием. Второе в случае шутера от первого лица лучше показывать как дырки от пуль, но мы пока будем обозначать это место сферой (и воспользуемся *сопрограммой*, через секунду убирающей сферу). Рисунок 3.3 демонстрирует то, что вы увидите в сцене.

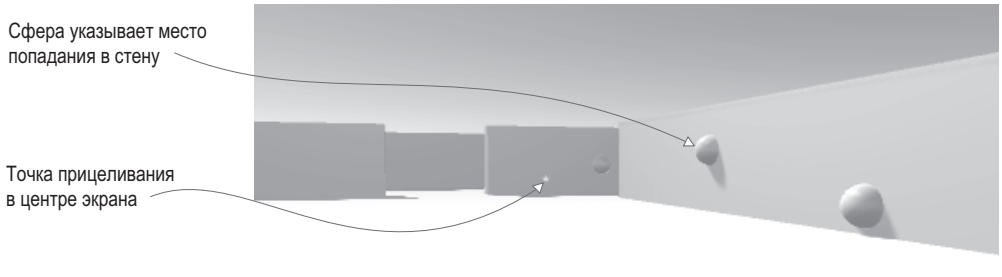


Рис. 3.3. Многократные выстрелы после добавления индикаторов прицеливания и попаданий

ОПРЕДЕЛЕНИЕ Сопрограммы (coroutines) в Unity выполняются параллельно программе в течение некоторого времени; этим они отличаются от большинства функций, заставляющих программу ждать окончания своей работы.

Начнем с добавления индикаторов в точки попадания. Готовый сценарий показан в листинге 3.2. Подвигайтесь по сцене, стреляя; индикаторы в виде сфер выглядят довольно забавно!

Листинг 3.2. Сценарий RayShooter после добавления индикаторных сфер

```
using UnityEngine;
using System.Collections;

public class RayShooter : MonoBehaviour {
  private Camera _camera;

  void Start() {
    _camera = GetComponent<Camera>();
  }

  void Update() { ← Эта функция по большей части содержит знакомый нам код бросания луча из листинга 3.1.
    if (Input.GetMouseButtonDown(0)) {
      Vector3 point = new Vector3(
        _camera.pixelWidth/2, _camera.pixelHeight/2, 0);
      Ray ray = _camera.ScreenPointToRay(point);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit)) {
        StartCoroutine(SphereIndicator(hit.point)); ← Запуск сопрограммы в ответ на попадание.
      }
    }
  }

  private IEnumerator SphereIndicator(Vector3 pos) { ← Сопрограммы пользуются функциями IEnumerator.
    GameObject sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere);
    sphere.transform.position = pos;

    yield return new WaitForSeconds(1); ← Ключевое слово yield указывает сопрограмме, когда следует остановиться.

    Destroy(sphere); ← Удаляем этот GameObject и очищаем память.
  }
}
```

Из нового у нас появился метод `SphereIndicator()` и однострочная модификация существующего метода `Update()`. Новый метод создает сферу в указанной точке сцены и через секунду удаляет ее. Вызов метода `SphereIndicator()` внутри кода испускания луча гарантирует появление визуальных индикаторов точно в местах попадания. Данная функция определена с помощью интерфейса `IEnumerator`, связанного с концепцией сопрограмм.

С технической точки зрения сопрограммы не асинхронны (асинхронные операции не останавливают выполнение остальной части кода; представьте, к примеру, скачивание изображения в сценарии веб-сайта), но продуманное применение перечислений в Unity заставляет сопрограммы вести себя аналогично асинхронным функциям. Секретным компонентом сопрограммы является ключевое слово `yield`, временно прерывающее ее работу, возвращающее управление основной программе и в следующем кадре возобновляющее сопрограмму с прерванной точки. В результате создается впечатление, что сопрограммы работают в фоновом режиме.

Как следует из ее имени, функция `StartCoroutine()` запускает сопрограмму. После этого она начинает работать до завершения выполнения, периодически делая паузы. Обратите внимание на небольшую, но важную деталь. Переданный в `StartCoroutine()` метод имеет набор скобок, следующий за именем. Такой синтаксис означает, что вы не передаете имя функции, а вызываете ее. И эта функция работает, пока не встретится команда `yield`. После этого ее выполнение на время прервется.

Функция `SphereIndicator()` создает сферу в определенной точке, останавливается после инструкции `yield` и после возобновления сопрограммы удаляет сферу. Продолжительность паузы контролируется значением, которое возвращается в момент появления инструкции `yield`. В сопрограммах допустимо несколько типов возвращаемых значений, но проще всего в явном виде вернуть время ожидания. Возвращая `WaitForSeconds(1)`, мы заставляем сопрограмму остановить работу на одну секунду. Создание сферы, секундная остановка и разрушение сферы — такая последовательность дает нам временный визуальный индикатор.

Код для таких индикаторов был дан в листинге 3.2. Но нам требуется также точка прицеливания в центре экрана. Она создается в следующем листинге.

Листинг 3.3. Визуальный индикатор для точки прицеливания

```
...
void Start() {
  _camera = GetComponent<Camera>();

  Cursor.lockState = CursorLockMode.Locked;  | Скрываем указатель мыши
  Cursor.visible = false;                    | в центре экрана.
}

void OnGUI() {
  int size = 12;
  float posX = _camera.pixelWidth/2 - size/4;
  float posY = _camera.pixelHeight/2 - size/2;
  GUI.Label(new Rect(posX, posY, size, size), "*"); ← Команда GUI.Label() отображает на экране символ.
}
...
```

Еще один новый метод, добавленный в класс `RayShooter`, называется `OnGUI()`. В Unity возможны как базовая, так и усовершенствованная системы пользовательского интерфейса (UI). Так как базовая система обладает множеством ограничений, в следующих главах мы построим более гибкий, усовершенствованный пользовательский интерфейс, но пока нам проще отображать точку в центре экрана средствами базового интерфейса. Любой сценарий класса `MonoBehaviour` автоматически реагирует как на методы `Start()` и `Update()`, так и на функцию `OnGUI()`. Эта функция запускается в каждом кадре после визуализации трехмерной сцены, прорисовывая поверх сцены дополнительные элементы (представьте себе бумажные этикетки, наклеенные на нарисованный пейзаж).

ОПРЕДЕЛЕНИЕ Визуализацией (rendering) называется работа компьютера по прорисовыванию пикселов трехмерной сцены. Хотя сцена задается с помощью координат *X*, *Y* и *Z*, монитор отображает двухмерную сетку цветных пикселов. Соответственно, для показа трехмерной сцены компьютер должен рассчитать цвет всех пикселов двухмерной сетки; работа этого алгоритма и называется визуализацией.

Код внутри функции `OnGUI()` определяет двухмерные координаты для отображения (слегка смещенные с учетом размера метки) и затем вызывает метод `GUI.Label()`. Этот метод отображает текстовую метку; так как переданная ему строка состоит из символа звездочки (*), именно он появляется в центре экрана. С ним прицеливаться в нашей будущей игре станет намного проще!

Кроме того, листинг 3.3 добавляет в метод `Start()` настройки указателя мыши, а именно возможность управлять его видимостью и возможность блокировки. В принципе, сценарий прекрасно работает и без этих настроек, но они делают элементы управления более удобными в использовании. Указатель мыши все время будет располагаться в центре экрана, а чтобы не заслонять обзор, мы сделаем его невидимым. Он будет появляться только при нажатии клавиши Esc.

ВНИМАНИЕ Помните, что вы в любой момент можете снять блокировку с указателя мыши, нажав клавишу Esc. При заблокированном указателе щелкнуть на кнопке Play и остановить игру невозможно.

Итак, код, управляющий поведением игрока, готов. Фактически, мы завершили работу над взаимодействиями игрока со сценой, но у нас все еще отсутствуют цели.

3.2. Создаем активные цели

Возможность стрелять — это замечательно, но в данный момент стрелять нашему игроку не во что. Поэтому нам нужно создать целевой объект и присоединить к нему сценарий, программирующий реакцию на попадание. Точнее, мы слегка отредактируем код стрельбы, чтобы получать уведомления о поражении цели, а присоединенный к целевому объекту сценарий будет запускаться в ответ на такое уведомление.

3.2.1. Определяем точку попадания

Первым делом нам нужен объект, который послужит мишенью. Создайте куб (выбрав в меню GameObject команду 3D Object, а затем вариант Cube) и поменяйте его размер по вертикали, введя в поле Y для преобразования Scale значение 2. В полях X и Z оставьте значение 1. Поместите новый объект в точку с координатами 0, 1, 0, чтобы он стоял на полу в центре комнаты, и присвойте ему имя Enemy. Создайте сценарий с именем `ReactiveTarget` и присоедините его к объекту. Скоро мы напишем для этого сценария код, но пока оставьте его как есть; мы создали этот файл, так как без него код из следующего листинга компилироваться не будет. Вернитесь к сценарию `RayShooter.cs` и отредактируйте код испускания луча в соответствии с показанным листингом. Запустите новый вариант кода и попробуйте выстрелить в цель — вместо сферического индикатора на консоли появится отладочное сообщение.

Листинг 3.4. Распознавание попаданий в цель

```
...
if (Physics.Raycast(ray, out hit)) {
  GameObject hitObject = hit.transform.gameObject; ← Получаем объект, в который попал луч.
  ReactiveTarget target = hitObject.GetComponent<ReactiveTarget>();
  if (target != null) { ← Проверяем наличие у этого объекта компонента ReactiveTarget.
    Debug.Log("Target hit");
```

```
  } else {
    StartCoroutine(SphereIndicator(hit.point));
  }
}
...
```

Обратите внимание, что вы получаете объект, с которым пересекся луч, совсем так же, как получали координаты для сферических индикаторов. С технической точки зрения вам возвращается вовсе не игровой объект, а попавший под удар компонент Transform. Далее вы получаете доступ к объекту `gameObject` как к свойству класса `transform`.

Затем к этому объекту применяется метод `GetComponent()`, проверяющий, является ли он активной целью (то есть присоединен ли к нему сценарий `ReactiveTarget`). Как вы уже видели раньше, этот метод возвращает компоненты определенного типа, присоединенные к объекту `GameObject`. Если таковые отсутствуют, не возвращается ничего. Поэтому остается проверить, было ли возвращено значение `null`, и написать два варианта кода, которые будут запускаться в зависимости от результата проверки.

Если пораженным объектом оказывается активная цель, вместо запуска сопрограммы для сферических индикаторов код отображает отладочное сообщение. Теперь нам нужно проинформировать целевой объект о попадании, чтобы он смог отреагировать на это событие.

3.2.2. Уведомляем цель о попадании

В коде нужно поменять всего одну строку, как показано в следующем листинге.

Листинг 3.5. Отправка сообщения целевому объекту

```
...
if (target != null) {
  target.ReactToHit(); ← Вызов метода для мишени вместо генерации отладочного сообщения.
} else {
  StartCoroutine(SphereIndicator(hit.point));
}
...
```

Теперь отвечающий за стрельбу код вызывает связанный с мишенью метод, который нам нужно написать. Введите в сценарий `ReactiveTarget` код следующего листинга. При попадании мишень будет опрокидываться и исчезать, как показано на рис. 3.4.

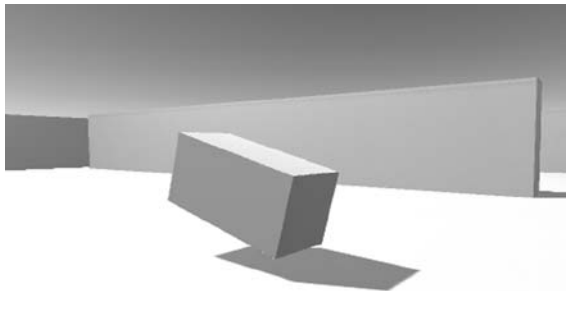


Рис. 3.4. Целевой объект, падающий после попадания

Листинг 3.6. Сценарий ReactiveTarget, реализующий смерть врага при попадании

```
using UnityEngine;
using System.Collections;

public class ReactiveTarget : MonoBehaviour {

  public void ReactToHit() { ← Метод, вызванный сценарием стрельбы.
    StartCoroutine(Die());
  }

  private IEnumerator Die() { ← Опрокидываем врага, ждем 1,5 секунды и уничтожаем его.
    this.transform.Rotate(-75, 0, 0);

    yield return new WaitForSeconds(1.5f);

    Destroy(this.gameObject); ← Объект может уничтожать сам себя точно так же, как любой другой объект.
  }
}
```

Большая часть кода должна быть вам уже знакома по предыдущим сценариям, поэтому рассматривать его мы будем совсем кратко. Первым делом мы определяем метод `ReactToHit()`, так как именно он вызывается в сценарии стрельбы. Этот метод запускает сопрограмму, аналогичную коду для сферических индикаторов, но на этот раз она призвана манипулировать объектом этого же сценария, а не создавать отдельный объект. Такие выражения, как `this.gameObject`, относятся к объекту `GameObject`, к которому присоединен данный сценарий (ключевое слово `this` указывать не обязательно, то есть можно ссылаться просто на `gameObject`).

Первая строка сопрограммы заставляет мишень опрокинуться. Как обсуждалось в главе 2, преобразование вращения можно определить как угол поворота относительно каждой из трех осей, *X*, *Y* и *Z*. Так как объект не должен поворачиваться из стороны в сторону, оставьте координатам *Y* и *Z* значение 0, а угол поворота назначьте координате *X*.

ПРИМЕЧАНИЕ Преобразование происходит мгновенно, но, возможно, вы предпочитаете видеть, как опрокидываются мишени. После более детального знакомства с методами моделирования, возможно, вы захотите воспользоваться анимацией по начальной и конечной точкам (tweening), позволяющей смоделировать плавное движение объектов.

Во второй строке метода фигурирует ключевое слово `yield`, останавливающее выполнение сопрограммы и возвращающее количество секунд, через которое она возобновит свою работу. В последней строке функции игровой объект уничтожается. Функция `Destroy(this.gameObject)` вызывается после времени ожидания точно так же, как раньше мы вызывали код `Destroy(sphere)`.

ВНИМАНИЕ Аргумент функции Destroy() в данном случае должен выглядеть как this.gameObject, а не просто this! Не путайте эти два случая; само по себе ключевое слово this относится к компоненту данного сценария, в то время как this.gameObject означает объект, к которому присоединен данный сценарий.

Теперь наша мишень реагирует на попадание! Но больше она ничего не делает. Давайте добавим ей дополнительные поведения, чтобы превратить ее в настоящего врага.

3.3. Базовый искусственный интеллект для перемещения по сцене

Статичная цель не очень интересна, поэтому давайте напишем код, который заставит врагов перемещаться по сцене. Такой код является одним из простейших примеров искусственного интеллекта (Artificial Intelligence, AI). Этот термин относится к сущностям, поведение которых контролируется компьютером. В данном случае такой сущностью служит враг в нашей игре, но в реальной жизни это может быть, например, робот.

3.3.1. Диаграмма работы базового искусственного интеллекта

Существует множество подходов к реализации искусственного интеллекта (это одна из основных областей исследований ученых, работающих в области теории вычислительных машин и систем), но для наших целей подойдет достаточно простой вариант. По мере накопления опыта и усложнения создаваемых вами игр вы, скорее всего, захотите познакомиться с другими способами реализации AI. Но пока мы ограничимся процессом, показанным на рис. 3.5.

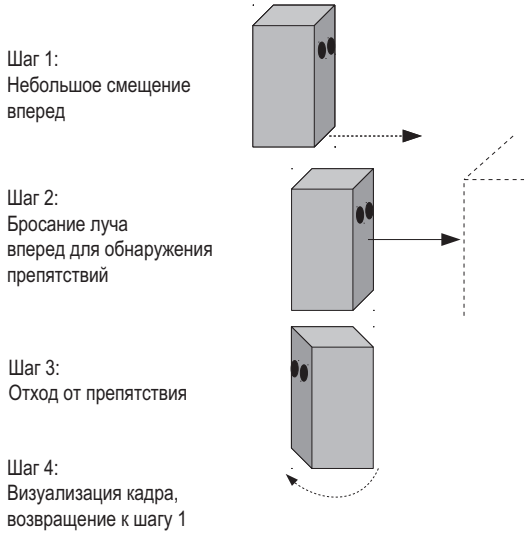


Рис. 3.5. Базовый искусственный интеллект: циклический процесс движения вперед с обходом препятствий

В каждом кадре код искусственного интеллекта сканирует окружающее пространство, чтобы определить свое дальнейшее поведение. Если на пути появляется препятствие, враг меняет направление движения. При этом независимо от поворотов он неуклонно двигается вперед. В итоге враг будет в разных направлениях ходить по комнате, непрерывно перемещаясь вперед и поворачиваясь, чтобы не сталкиваться со стенами.

Сам код будет иметь знакомый вам вид, так как движение врагов вызывают те же команды, что и движение игрока. Кроме того, в коде вы снова увидите метод бросания луча, но уже в другом контексте.

3.3.2. «Поиск» препятствий методом бросания лучей

Как вы узнали во введении к данной главе, бросание луча представляет собой прием, позволяющий решать различные задачи трехмерного моделирования. Первым делом на ум приходит имитация выстрелов, но кроме того, этот прием позволяет сканировать окружающее пространство. А так как в данном случае нам требуется решить именно эту задачу, код испускания лучей попадет в наш код для AI.

Раньше мы создавали луч, который брал начало из камеры, так как именно она служит глазами игрока. На этот раз луч будет начинаться в месте расположения врага. В первом случае луч проходил через центр экрана, теперь же он будет распространяться перед персонажем, как показано на рис. 3.6. Затем аналогично тому, как код стрельбы использовал информацию из структуры `RaycastHit`, чтобы определить, поражена ли какая-нибудь цель и где она находится, код AI задействует эту же информацию, чтобы определить наличие препятствия по ходу движения и расстояние до этого препятствия.

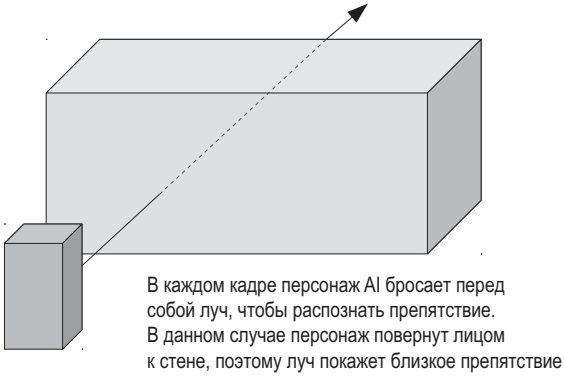


Рис. 3.6. Обнаружение препятствий методом бросания луча

Основным различием между лучом, бросаемым в случае выстрела, и лучом в коде AI является радиус распознаваемого пространства. При стрельбе мы обходились бесконечно тонким лучом, в то время как у луча для AI будет большое сечение. Соответственно, мы воспользуемся методом `SphereCast()` вместо метода `Raycast()`. Все дело в том, что пули имеют маленький размер, в то время как, проверяя наличие препятствий перед персонажем, мы должны учитывать ширину самого персонажа.

Создайте сценарий с именем `WanderingAI` и присоедините его к целевому объекту (вместе со сценарием `ReactiveTarget`). Введите в него код из следующего листинга. Запустите код, и вы увидите, как враг перемещается по комнате; при этом вы можете выстрелить в него, и он среагирует на попадание так же, как и раньше.

Листинг 3.7. Базовый сценарий WanderingAI

```
using UnityEngine;
using System.Collections;

public class WanderingAI : MonoBehaviour {
  public float speed = 3.0f; ← Значения для скорости движения и расстояния, с которого
  public float obstacleRange = 5.0f;          начинается реакция на препятствие.

  void Update() {
    transform.Translate(0, 0, speed * Time.deltaTime); ← Непрерывно движемся вперед в каждом
                                                         кадре, несмотря на повороты.
    Ray ray = new Ray(transform.position, transform.forward); ← Луч находится в том же положе-
    RaycastHit hit;                                             нии и нацеливается в том же
                                                                направлении, что и персонаж.
    if (Physics.SphereCast(ray, 0.75f, out hit)) { ← Бросаем луч с описанной
      if (hit.distance < obstacleRange) {            вокруг него окружностью.
        float angle = Random.Range(-110, 110); ← Поворот с наполовину случайным
        transform.Rotate(0, angle, 0);           выбором нового направления.
      }
    }
  }
}
```

В листинге появилась пара новых переменных. Одна — для скорости движения врага, а вторая — для расстояния, на котором враг начинает реагировать на препятствие. Затем мы добавили внутрь метода `Update()` метод `Translate()`, обеспечив непрерывное движение вперед (в том числе воспользовавшись переменной `deltaTime` для движения, не зависящего от частоты кадров). Кроме того, в метод `Update()` помещен код бросания луча, во многом напоминающий написанный нами ранее сценарий поражения целей. В данном случае прием бросания луча применяется для осмотра сцены, а не для стрельбы. Луч создается на базе положения врага и направления его движения, а не на базе камеры.

Как уже упоминалось, для расчета траектории луча применяется метод `Physics.SphereCast()`, в качестве параметра принимающий радиус окружности, в пределах которой будут распознаваться пересечения. Во всех же прочих отношениях он аналогичен методу `Physics.Raycast()`. Сходство наблюдается в способе получения информации о столкновении луча, способе проверки пересечений и применении свойства `distance`, гарантирующего, что враг среагирует только тогда, когда приблизится к препятствию.

Как только враг окажется перед стеной, код поменяет направление его движения наполовину случайным образом. Я использую словосочетание «наполовину случайным», так как значения ограничены минимумом и максимумом, имеющими смысл в данной ситуации. Мы прибегаем к методу `Random.Range()`, которым Unity позволяет получить случайное значение в указанном диапазоне. В нашем случае ограничения немного превышают величины поворота влево и вправо, что дает персонажу возможность развернуться на достаточный угол, чтобы избежать препятствия.

3.3.3. Слежение за состоянием персонажа

Текущее поведение врага имеет один недостаток. Движение вперед продолжается даже после попадания в него пули. Ведь метод `Translate()` запускается в каждом кадре вне зависимости от обстоятельств. Внесем в код небольшие изменения,

позволяющие следить за тем, жив персонаж или мертв. Говоря техническим языком, мы хотим отслеживать «живое» состояние персонажа. Код, по-разному реагирующий на разные состояния, представляет собой паттерн, распространенный во многих областях программирования, а не только в AI. Более сложные реализации этого паттерна называются *конечными автоматами*.

ОПРЕДЕЛЕНИЕ Конечным автоматом (Finite State Machine, FSM) называется структура кода, в которой отслеживается текущее состояние объекта, существуют четко определенные переходы между состояниями и код ведет себя по-разному в зависимости от состояния.

Разумеется, речь о настоящей реализации конечного автомата не идет, но нет ничего странного в том, что при обсуждении искусственного интеллекта упоминаются основы FSM. Конечный автомат обладает множеством состояний для различных вариантов поведения сложного искусственного интеллекта, в случае же базового искусственного интеллекта достаточно отследить, жив персонаж или уже нет. В следующем листинге в начальную часть сценария добавляется логическая переменная `_alive`, значение которой будет периодически проверяться в коде. Благодаря этим проверкам код движения запускается только для живого персонажа.

Листинг 3.8. Сценарий WanderingAI после добавления «живого» состояния

```
...
private bool _alive; ← Логическая переменная для слежения за состоянием персонажа.

void Start() {
  _alive = true; ← Инициализация этой переменной.
}

void Update() {
  if (_alive) { ← Движение начинается только в случае живого персонажа.
    transform.Translate(0, 0, speed * Time.deltaTime);
    ...
  }
}

public void SetAlive(bool alive) { ← Открытый метод, позволяющий внешнему коду
  _alive = alive;                    воздействовать на «живое» состояние.
}
...
```

Теперь сценарий `ReactiveTarget` может сообщить сценарию `WanderingAI`, в каком состоянии находится враг.

Листинг 3.9. Сценарий ReactiveTarget сообщает сценарию WanderingAI, когда наступает смерть

```
...
public void ReactToHit() {
  WanderingAI behavior = GetComponent<WanderingAI>();
  if (behavior != null) { ← Проверяем, присоединен ли к персонажу сценарий WanderingAI; он может и отсутствовать.
    behavior.SetAlive(false);
  }
  StartCoroutine(Die());
}
...
```

СТРУКТУРА КОДА ДЛЯ AI

Приведенный в этой главе код для AI помещен в один класс, так что изучить и понять его достаточно просто. Такая структура кода совершенно нормальна для простого искусственного интеллекта, поэтому не бойтесь, что вы сделали что-то не так и что на самом деле требуется более сложная структура. Для усложненных реализаций искусственного интеллекта (например, в игре с широким диапазоном интеллектуальных персонажей) облегчить разработку AI поможет более надежная структура кода.

Как упоминалось в главе 1 при сравнении компонентной структуры с наследованием, иногда фрагменты кода для AI имеет смысл помещать в отдельные сценарии. Это позволит сочетать и комбинировать компоненты, генерируя уникальное поведение для каждого персонажа. Подумайте, в чем ваши персонажи одинаковы, а чем различаются. Именно эти различия помогут вам при разработке архитектуры кода. К примеру, если в игре есть враги, сломя голову несущиеся на персонажа, и враги, тихо подкрадывающиеся в тени, имеет смысл создать для них разные компоненты перемещения и написать отдельные сценарии для перемещения бегом и перемещения подкрадываясь.

Точная структура кода для AI зависит от особенностей конкретной игры; единственного «правильного» варианта в данном случае не существует. Средства Unity позволят вам легко спроектировать гибкую архитектуру.

3.4. Увеличение количества врагов

В настоящее время в сцене присутствует всего один враг, и после его смерти сцена становится пустой. Давайте заставим игру порождать врагов, сделав так, чтобы после смерти существующего врага сразу появлялся новый. В Unity это легко делается с помощью механизма *шаблонов экземпляров* (prefabs).

3.4.1. Что такое шаблон экземпляров?

Шаблоны экземпляров предлагают гибкий подход к визуальному определению интерактивных объектов. В двух словах, это полностью сформированный игровой объект (с уже присоединенными и настроенными компонентами), существующий не внутри конкретной сцены, а в виде ресурса, который может быть скопирован в любую сцену. Копирование может осуществляться вручную, чтобы гарантировать идентичность объектов-врагов (или других объектов) в каждой сцене. Но куда важнее то, что эти шаблоны могут порождаться кодом; поместить копии объектов в сцену можно не только вручную в визуальном редакторе, но и с помощью команд сценария.

ОПРЕДЕЛЕНИЕ Ресурсом (asset) называется любой файл, отображаемый на вкладке Project; это могут быть двухмерные изображения, трехмерные модели, файлы с кодом, сцены и т. п. Термин «ресурс» вскользь упоминался в главе 1, но до текущего момента мы не акцентировали на нем внимания.

Термин *экземпляр* (instance) относится также к создаваемым на основе класса объектам кода. Попытайтесь не запутаться в терминологии; словосочетание «шаблон экземпляра» относится к игровому объекту, существующему вне сцены, в то время как экземпляром называется помещенная в сцену копия объекта.

3.4.2. Создание шаблона врага

Конструирование шаблона начинается с создания объекта. Так как мы планируем получить шаблон из объекта-врага, этот шаг уже сделан. Теперь нужно перетащить строку с названием этого объекта с вкладки Hierarchy на вкладку Project, как показано на рис. 3.7. Объект автоматически сохранится в качестве шаблона. На панели Hierarchy его имя будет выделено синим цветом, означающим, что теперь он связан с шаблоном экземпляров. Редактирование этого шаблона (например, добавление компонентов) осуществляется посредством редактирования объекта сцены, после чего в меню GameObject выбирается команда Apply Changes To Prefab. Но сейчас данный объект в сцене уже не нужен (мы собираемся порождать новые шаблоны, а не пользоваться уже имеющимся экземпляром), поэтому его следует удалить.

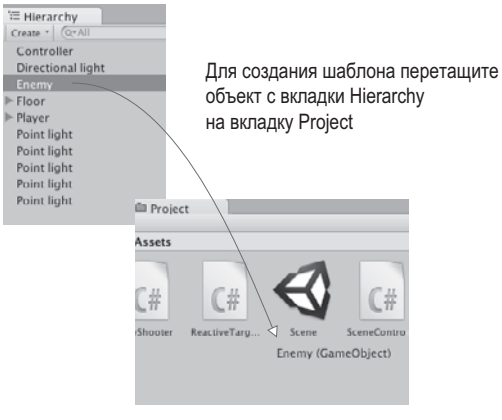


Рис. 3.7. Процесс получения шаблона экземпляров

ВНИМАНИЕ Интерфейс для работы с шаблонами экземпляров не слишком удобен, а соотношения между шаблонами и их экземплярами в сценах порой достаточно нестабильны. К примеру, зачастую требуется перетащить шаблон в сцену для последующего редактирования, а после завершения этого процесса удалить объект. В первой главе я упоминал об этом как о недостатке Unity и надеюсь, что в следующих версиях последовательность действий будет усовершенствована.

Теперь у нас есть шаблон для заполнения сцены врагами, поэтому давайте напишем код, создающий его экземпляры.

3.4.3. Экземпляры невидимого компонента SceneController

Хотя сам по себе шаблон в сцене отсутствует, нам нужен некий объект, к которому будет присоединяться код, порождающий врагов. Поэтому мы создадим пустой игровой объект и добавим к нему сценарий, при этом сам объект останется невидимым.

СОВЕТ Использование пустого объекта GameObject для присоединения к нему компонентов-сценариев является распространенной практикой при разработке в Unity. Этот трюк применяется для решения абстрактных задач и нереализуем при работе с конкретными объектами сцены. Unity-сценарии присоединяются только к видимым объектам, но не для каждой задачи это имеет смысл.

Выберите в меню GameObject команду Create Empty и присвойте новому объекту имя Controller. Убедитесь, что он находится в точке с координатами 0, 0, 0 (с технической точки зрения местоположение этого объекта не имеет значения, так как его все равно не видно, но, поместив его в начало координат, вы облегчите себе жизнь, если в будущем решите использовать его в цепочке наследования). Создайте сценарий `SceneController`, показанный в следующем листинге.

Листинг 3.10. Сценарий SceneController, порождающий экземпляры врагов

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour {
  [SerializeField] private GameObject enemyPrefab; ← Сериализованная переменная для связи с объектом-шаблоном.
  private GameObject _enemy; ← Закрытая переменная для слежения за экземпляром врага в сцене.

  void Update() { ← Порождаем нового врага, только если враги в сцене отсутствуют.
    if (_enemy == null) {
      _enemy = Instantiate(enemyPrefab) as GameObject; ← Метод, копирующий объект-шаблон.
      _enemy.transform.position = new Vector3(0, 1, 0);
      float angle = Random.Range(0, 360);
      _enemy.transform.Rotate(0, angle, 0);
    }
  }
}
```

Присоедините этот сценарий к объекту-контроллеру, и на панели Inspector появится поле для шаблона врага с именем Enemy Prefab. Оно работает аналогично общедоступным переменным, но есть и важное отличие.

ВНИМАНИЕ Для ссылки на объекты в редакторе Unity я рекомендую закрытые переменные с атрибутом SerializeField, так как нам нужно отобразить поле новой переменной на панели Inspector, но при этом не хотелось бы, чтобы ее значение могли менять другие сценарии. Как объяснялось в главе 2, открытые переменные отображаются на панели Inspector по умолчанию (другими словами, их сериализацией занимается Unity), поэтому в большинстве руководств и примеров для всех сериализованных значений фигурируют общедоступные переменные. Но эти переменные могут быть модифицированы другими сценариями (ведь они являются общедоступными); в большинстве же случаев редактирование значений должно разрешаться только через панель Inspector.

Перетащите шаблон врага с вкладки Project на пустое поле переменной; при наведении на него указателя мыши вы должны увидеть, как поле подсвечивается, демонстрируя допустимость присоединения объекта (рис. 3.8). После присоединения к шаблону врага сценария `SceneController` воспроизведите сцену, чтобы посмотреть, как работает код. Враг, как и раньше, будет возникать в центре комнаты, но если вы его застрелите, на его месте появится новый враг. Это намного лучше, чем единственный враг, который умирает навсегда!

СОВЕТ Перетаскивание объектов на поля переменных панели Inspector — весьма удобный прием, используемый в самых разных сценариях. В данном случае мы связали шаблон со сценарием, но можно связывать между собой объекты сцены и даже отдельные компоненты (так как код в этом конкретном компоненте должен вызывать открытые методы). В следующих главах мы еще не раз воспользуемся этим приемом.

Рис. 3.8. Процедура соединения шаблона со сценарием

Центральной частью сценария является метод `Instantiate()`, поэтому обратите внимание на содержащую его строку. Созданные экземпляры шаблона появляются в сцене. По умолчанию метод `Instantiate()` возвращает новый объект обобщенного типа `Object`, который напрямую практически бесполезен, поэтому его следует обрабатывать как объект `GameObject`. В языке C# приведение типов осуществляется при помощи ключевого слова `as` (указывается исходный объект, ключевое слово `as` и желаемый новый тип).

Полученный экземпляр сохраняется в закрытой переменной `_enemy` типа `GameObject` (снова напоминаю о разнице между шаблоном экземпляра и самим экземпляром; переменная `enemyPrefab` хранит в себе шаблон, в то время как переменная `_enemy` — экземпляр этого шаблона). Инструкция `if`, проверяющая сохраненный объект, гарантирует, что метод `Instantiate()` будет вызван только при пустой переменной `_enemy` (или на языке кода — при равенстве этой переменной значению `null`). Изначально эта переменная пуста, соответственно, код создания экземпляра запускается в самом начале сеанса. Возвращенный методом `Instantiate()` объект затем сохраняется в переменной `_enemy`, блокируя повторный запуск кода создания экземпляров.

После попадания объект-враг разрушается, переменная `_enemy` становится пустой, что приводит к вызову метода `Instantiate()`. Благодаря этому враг всегда присутствует в сцене.

РАЗРУШЕНИЕ ИГРОВЫХ ОБЪЕКТОВ И УПРАВЛЕНИЕ ПАМЯТЬЮ

Тот факт, что существующие ссылки при разрушении объекта начинают указывать на значение null, является до некоторой степени неожиданным. В языках программирования с автоматическим управлением памятью, к которым относится C#, мы, как правило, не можем напрямую удалять объекты; можно обнулить все ведущие на них ссылки, после чего удаление объекта произойдет автоматически. Это верно и в Unity, но фоновый способ обработки объектов GameObject выглядит в Unity так, как если бы удаление совершалось напрямую.

Для отображения объектов в Unity ссылки на все эти объекты должны присутствовать в графе сцены. Соответственно, даже после удаления всех ссылок на конкретный игровой объект в коде на него все равно будет ссылаться граф сцены, что сделает автоматическое удаление невозможным.

Поэтому в Unity существует метод Destroy(), заставляющий игровой движок удалять объект из графа сцены. Как часть этой фоновой функциональности в Unity также перегружается оператор == и начинает возвращать значение true при проверке на наличие значения null. Технически объект все еще находится в памяти, но при этом он может больше не существовать, поэтому Unity показывает его равенство значению null. Это можно проверить, вызвав для уничтоженного объекта метод GetInstanceID().

Впрочем, разработчики Unity обдумывают замену этого поведения более стандартным вариантом управления памятью. Если подобное произойдет, придется поменять и код порождения врагов, указав вместо проверки (_enemy==null) новый параметр, например (_enemy.isDestroyed). Следите за новостями в социальной сети Facebook по адресу https://www.facebook.com/unity3d/posts/10152271098591773.

(Если большая часть данного примечания осталась вам непонятной, просто не обращайте на него внимания; это было лирическое отступление для пользователей, заинтересованных в непонятных деталях.)

3.5. Стрельба путем создания экземпляров

Хорошо, добавим врагам еще немного способностей. Пойдем по тому же пути, что и при работе над игроком. Первым делом мы научили врагов двигаться, теперь дадим им возможность стрелять! Как я упоминал, знакомя вас с приемом испускания луча, этот прием является всего лишь одним из подходов к реализации стрельбы. Другой вариант реализации — создание экземпляров из шаблона. Воспользуемся им, чтобы заставить врагов отстреливаться. Результат, который нужно получить, показан на рис. 3.9.

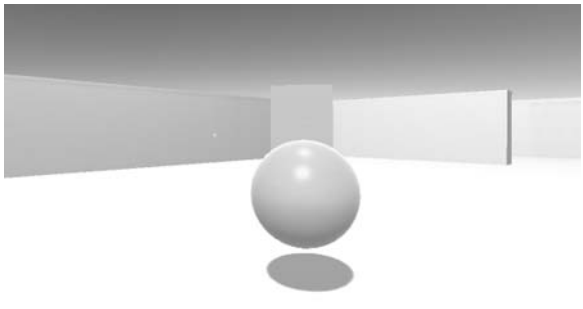


Рис. 3.9. Враг бросает в игрока «огненный шар»

3.5.1. Шаблон снаряда

Если раньше мы стреляли без применения реальных снарядов, то теперь они нам понадобятся. Стрельба приемом испускания лучей, по сути, мгновенна, и попадание регистрируется в момент щелчка кнопкой мыши, враги же будут бросать «огненные шары», летающие по воздуху. Разумеется, хотя они двигаются довольно быстро, у игрока будет возможность уклониться. Фиксировать попадания мы будем не методом испускания лучей, а методом распознавания столкновений (это уже знакомый вам метод, не дающий игроку проходить сквозь стены).

Код будет порождать огненные шары тем же способом, которым порождаются враги, — созданием экземпляров из шаблона. Как вы знаете из предыдущего раздела, первым шагом в подобных случаях является создание объекта, который послужит основой для шаблона. Так как нам требуется огненный шар, выберите в меню GameObject команду 3D Object и в дополнительном меню команду Sphere. Присвойте появившейся сфере имя Fireball. Теперь создайте новый сценарий с таким же именем и присоедините его к объекту. Чуть позже мы напишем для него код, а пока оставьте вариант, предлагаемый по умолчанию, так как первым делом мы завершим работу над снарядом. Чтобы он напоминал огненный шар, ему нужно присвоить ярко-оранжевый цвет. Такие свойства поверхностей, как цвет, контролируются при помощи *материалов*.

ОПРЕДЕЛЕНИЕ Материалом (material) называется пакет информации, определяющий свойства поверхности любого трехмерного объекта, к которому этот пакет присоединен. К свойствам поверхности относятся цвет, блеск и даже небольшая шероховатость.

Выберите в меню Assets команду Create и в дополнительном меню команду Material. Присвойте новому материалу имя Flame. Выберите его на вкладке Project, чтобы увидеть его свойства на панели Inspector. Как показано на рис. 3.10, щелкните на цветовой ячейке с именем Albedo (это технический термин, означающий основной цвет поверхности). В отдельном окне откроется палитра цветов; переместите расположенный справа ползунок и точку в основной области таким образом, чтобы выбрать оранжевый цвет.

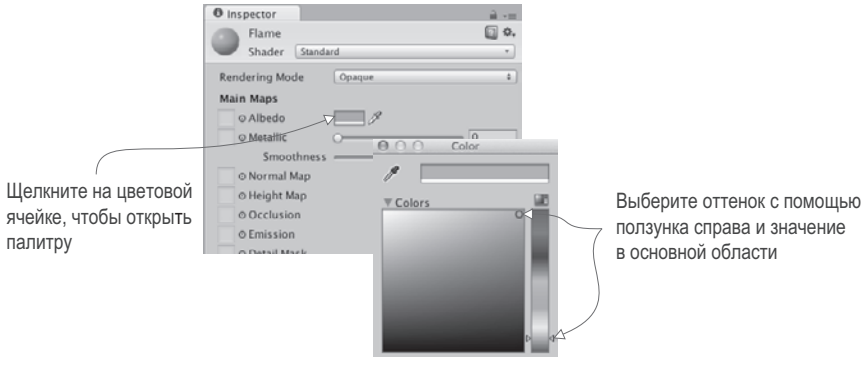


Рис. 3.10. Выбор цвета материала

Еще мы увеличим яркость материала, чтобы придать ему большее сходство с пламенем. Отредактируйте параметр Emission (он тоже находится в списке атрибутов панели Inspector). По умолчанию он равен 0, присвойте ему значение 0.3, чтобы сделать материал более ярким.

Теперь можно превратить наш огненный шар в шаблон, перетащив его со вкладки Hierarchy на вкладку Project, как мы делали, создавая шаблон врага. Теперь у нас есть основа для наших снарядов! Осталось написать код стрельбы.

3.5.2. Стрельба и столкновение с целью

Отредактируем шаблон врага, наделив его способностью кидать огненные шары. Чтобы код распознавал игрока, потребуется новый сценарий (аналогично тому, как сценарий `ReactiveTarget` требовался для распознавания мишеней), поэтому создадим сценарий с именем `PlayerCharacter` и присоединим его к игроку.

А теперь откроем сценарий `WanderingAI` и введем в него код следующего листинга.

Листинг 3.11. Сценарий WanderingAI с возможностью кидать огненные шары

```
...
[SerializeField] private GameObject fireballPrefab; ← Эти два поля добавляются перед любыми методами, как и в сценарии SceneController.
private GameObject _fireball;
...
if (Physics.SphereCast(ray, 0.75f, out hit)) {
  GameObject hitObject = hit.transform.gameObject;
  if (hitObject.GetComponent<PlayerCharacter>()) { ← Игрок распознается тем же способом, что и мишень в сценарии RayShooter.
    if (_fireball == null) { ← Та же самая логика с пустым игровым объектом, что и в сценарии SceneController.
      _fireball = Instantiate(fireballPrefab) as GameObject; ← Метод Instantiate() работает так же, как и в сценарии SceneController.
      _fireball.transform.position = ← Поместим огненный шар перед врагом и нацелим в направлении его движения.
      transform.TransformPoint(Vector3.forward * 1.5f);
      _fireball.transform.rotation = transform.rotation;
    }
  }
  else if (hit.distance < obstacleRange) {
    float angle = Random.Range(-110, 110);
    transform.Rotate(0, angle, 0);
  }
}
...
```

Думаю, вы заметили, что все примечания к этому листингу ссылаются на сходные (или аналогичные) фрагменты предыдущих сценариев. Фактически, в предыдущих листингах вы уже видели весь код, необходимый для бросания огненных шаров; мы просто смешали эти фрагменты друг с другом в соответствии с новым контекстом.

Как и в листинге `SceneController`, в верхнюю часть сценария нужно добавить два поля `GameObject`: сериализованную переменную, с которой будет связываться шаблон, и закрытую переменную для слежения за созданным кодом экземпляром этого шаблона. После испускания луча идет проверка попадания в объект `PlayerCharacter`; это работает аналогично тому, как код стрельбы проверял наличие у пораженного объекта компонента ReactiveTarget. Код, создающий экземпляр огненного шара при отсутствии таких шаров в сцене, работает как код создания экземпляров врага. Отличаются только размещение и ориентация объектов; на этот раз мы помещаем экземпляр, полученный из шаблона, перед врагом и нацеливаем в направлении его движения.

Как только новый код окажется на своем месте, на панели Inspector появится новое поле Fireball Prefab, точно так же, как в свое время для компонента SceneController появилось поле Enemy Prefab. Щелкните на шаблоне врага на вкладке Project, и на панели Inspector появятся компоненты этого объекта, как если бы вы выделили его в сцене.

Хотя упомянутое ранее неудобство интерфейса часто проявляется при редактировании шаблонов, именно интерфейс дает нам возможность легко редактировать компоненты объекта, чем мы сейчас и воспользуемся. Перетащите шаблон огненного шара со вкладки Project на поле Fireball Prefab панели Inspector, как показано на рис. 3.11.

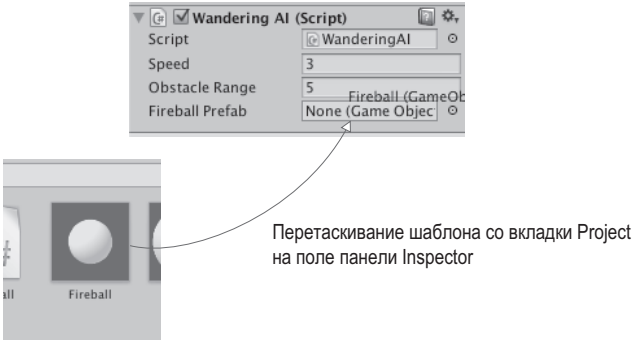


Рис. 3.11. Соединение огненного шара со сценарием

Теперь, как только игрок окажется непосредственно перед врагом, в него полетит огненный шар... Попробуйте выполнить воспроизведение — перед врагом появится огненная сфера, но никуда не полетит, потому что мы пока не написали соответствующий сценарий. Давайте сделаем это сейчас. Код сценария `Fireball` приведен в следующем листинге.

Листинг 3.12. Сценарий Fireball, реагирующий на столкновения

```
using UnityEngine;
using System.Collections;

public class Fireball : MonoBehaviour {
  public float speed = 10.0f;
  public int damage = 1;

  void Update() {
    transform.Translate(0, 0, speed * Time.deltaTime);
  }

  void OnTriggerEnter(Collider other) {   // ← Эта функция вызывается, когда с триггером сталкивается другой объект.
    PlayerCharacter player = other.GetComponent<PlayerCharacter>();
    if (player != null) {   // ← Проверяем, является ли этот другой объект объектом PlayerCharacter.
      Debug.Log("Player hit");
    }
    Destroy(this.gameObject);
  }
}
```

Существенным новшеством в этом коде является метод `OnTriggerEnter()`. Он автоматически вызывается при столкновении объекта, например, со стеной или с игроком. Пока что наш код работает не совсем корректно; после его запуска огненный шар

благодаря строке `Translate()` полетит вперед, но триггер не сработает, а вместо этого будет запрошен новый шар путем разрушения уже существующего. Требуется внести в компоненты огненного шара еще пару дополнений. Прежде всего, нужно превратить коллайдер в триггер. Для этого установите флажок Is Trigger в разделе Sphere Collider.

СОВЕТ После превращения в триггер компонент Collider продолжит реагировать на соприкосновение/перекрытие с другими объектами, но уже не будет препятствовать физическому пересечению с этими объектами.

Огненному шару также требуется компонент Rigidbody, относящийся к системе моделирования законов физики. Именно он позволит этой системе зарегистрировать триггеры столкновений для рассматриваемого объекта. Щелкните на кнопке Add Component на панели Inspector и по очереди выберите команды Physics и Rigidbody. После добавления указанного компонента сбросьте флажок Use Gravity, как показано на рис. 3.12, чтобы на огненный шар перестала действовать сила тяжести.

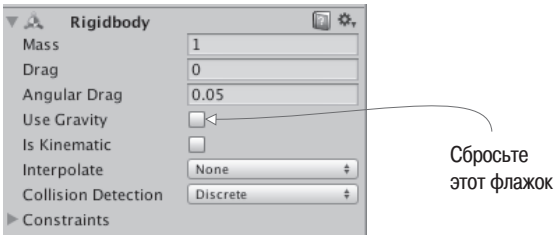


Рис. 3.12. Отключение гравитации для компонента Rigidbody

Воспроизведите сцену, и вы увидите, как огненные шары разрушаются после столкновения с каким-либо объектом. Так как код генерации этих шаров запускается сразу после исчезновения очередного экземпляра, обстрел игрока теперь становится непрерывным. Осталось добавить всего одну деталь: заставить игрока реагировать на попадание.

3.5.3. Повреждение игрока

Ранее мы создали сценарий `PlayerCharacter`, но оставили его пустым. Теперь введите в него из следующего листинга код реакции на попадание.

Листинг 3.13. Игрок может получать повреждения

```
using UnityEngine;
using System.Collections;

public class PlayerCharacter : MonoBehaviour {
  private int _health;

  void Start() {
    _health = 5; ← Инициализация переменной health.
  }
```

```
  public void Hurt(int damage) {
    _health -= damage; ← Уменьшение здоровья игрока.
    Debug.Log("Health: " + _health);
  }
}
```

В листинге определяется поле для здоровья игрока, и по команде этот показатель уменьшается. В следующих главах вы научитесь отображать на экране текстовую информацию, пока же сведения о состоянии игрока можно вывести в виде отладочного сообщения.

Теперь нужно вернуться к сценарию `Fireball`, чтобы вызвать для игрока метод `Hurt()`. Вставьте вместо отладочной строчки в сценарии `Fireball` строку `player.Hurt(damage)`, которая будет сообщать игроку о попадании. Всё. Последний фрагмент кода встал на свое место!

Ну что ж, это была достаточно насыщенная глава с большим количеством новой информации. В предыдущей и этой главах вы реализовали большую часть функциональности, необходимой в шутере от первого лица.

3.6. Заключение

- Луч — это спроецированная в сцену воображаемая линия.
- Как для стрельбы, так и для обнаружения объектов применяется метод бросания лучей.
- Для моделирования хаотичных перемещений персонажа по сцене используется базовый искусственный интеллект.
- Новые объекты генерируются путем создания экземпляров из существующего шаблона.
- Сопрограммы позволяют растянуть функцию во времени.

4 Работа с графикой

- ✓ Основные сведения о графических ресурсах
- ✓ Процесс создания геометрической модели сцены
- ✓ Использование в Unity двухмерных изображений
- ✓ Импорт собственных трехмерных моделей
- ✓ Моделирование эффектов частиц

Пока что мы рассматривали в основном принципы функционирования игры, не обращая особого внимания на то, как она выглядит. И это не случайно, ведь книга посвящена программированию игр в Unity. Тем не менее важно знать, каким образом создается и улучшается визуальная картинка. В следующей главе мы вернемся к основной теме — написанию кода для различных частей игры, — а пока поговорим о такой вещи, как графика, чтобы ваши проекты не превращались в набор одинаковых с виду примитивов, плавно перемещающихся по сцене.

Все, что составляет визуальное содержимое игры, называется *графическими ресурсами* (art assets). Но что именно скрывается за этим термином?

4.1. Основные сведения о графических ресурсах

Графическим ресурсом называется отдельная единица визуальной информации (обычно — файл), используемая игрой. Это всеобъемлющее собирательное название для всего визуального содержимого; к графическим ресурсам относятся файлы изображений, трехмерные модели и т. п. На самом деле, это всего лишь частный случай ресурса, который, как вы знаете, представляет собой любой используемый игрой файл (например, сценарий). Все они в Unity находятся в основной папке Assets. В табл. 4.1 перечислены и описаны пять основных видов графических ресурсов, применяемых при создании игр.

Таблица 4.1. Типы игровых ресурсов

Тип	Определение типа
Двухмерное изображение	Плоские изображения. В реальном мире им соответствуют картины и фотографии
Трехмерная модель	Виртуальные трехмерные объекты (почти синоним для «сеточных объектов»). В реальном мире им соответствуют скульптуры
Материал	Пакет информации, определяющий свойства поверхности любого объекта, к которому присоединен материал. К таким свойствам относят цвет, блеск и даже небольшие шероховатости
Анимация	Пакет информации, определяющий движение связанного с ним объекта. Существуют как заблаговременно созданные детализированные последовательности перемещений, так и код, вычисляющий положения объектов «на лету»
Система частиц	Механизм создания большого количества небольших движущихся объектов и управления ими. Позволяет создавать различные визуальные эффекты, такие как огонь, дым и водяные брызги

Создание графики для новой игры в общем случае начинается с двухмерных изображений или трехмерных моделей, так как эти ресурсы формируют базу для всего остального. Как несложно догадаться, двухмерные изображения служат основой для двухмерной графики, в то время как трехмерные модели — для трехмерной. Точнее говоря, двухмерные изображения представляют собой плоские картинки. Даже если у вас нет представления о графике, используемой в играх, с ними вы, скорее всего, сталкивались, например на сайтах. При этом трехмерные модели могут оказаться совершенно незнакомым новичкам понятием, поэтому я дам им определение.

ОПРЕДЕЛЕНИЕ Модель (model) — это трехмерный виртуальный объект. В главе 1 вы столкнулись с термином «сеточный объект»; трехмерная модель — это практически синоним. Оба термина часто используются в одном и том же значении, но термин «сеточный объект» относится исключительно к геометрии трехмерных объектов (соединенные друг с другом линии и фигуры), в то время как термин «модель» имеет более общий смысл и часто подразумевает и другие атрибуты объекта.

Следующие два типа ресурсов в нашем списке — это материалы и анимация. В отличие от двухмерных изображений и трехмерных моделей они не имеют смысла сами по себе, поэтому новичкам бывает непросто понять, что это такое. Изображения и модели легко представимы благодаря аналогам из реального мира. Первым соответствуют картины, вторым — скульптуры. Материалы и анимация не имеют прямых ассоциаций с реальными объектами. Это абстрактные пакеты информации, накладываемые на трехмерные модели. Например, базовый смысл материалов уже был рассмотрен в главе 3.

ОПРЕДЕЛЕНИЕ Материалом (material) называется пакет информации, определяющий свойства поверхности любого трехмерного объекта, к которому он присоединяется. К таким свойствам относятся, к примеру, цвет, блеск и даже небольшая шероховатость.

Если продолжить аналогию с искусством, материал можно представить как вещество (глина, бронза, мрамор и т. п.), из которого создается скульптура. Подобным

же образом анимация представляет собой присоединяемый к видимому объекту абстрактный слой информации.

ОПРЕДЕЛЕНИЕ Анимацией (animation) называется пакет информации, определяющий движение связанного с ним объекта. Так как эти движения можно задать независимо от объекта, их можно использовать, сочетая и комбинируя с различными объектами.

В качестве конкретного примера рассмотрим перемещающийся по сцене персонаж. Его положение в каждый момент определяется кодом игры (например, сценариями движения, которые вы писали в главе 2). Но подробные движения ног, ступающих по земле, размахивающих рук и поворачивающихся бедер представляют собой воспроизводимые в цикле анимационные последовательности, которые и являются графическим ресурсом.

Чтобы понять, как связаны друг с другом анимация и трехмерные модели, представьте себе кукольный театр: трехмерная модель будет куклой, аниматор — кукольником, заставляющим ее двигаться, а анимация — записью движений куклы. Определенные таким способом движения записываются заранее и обычно происходят в небольшом масштабе, не меняя общего положения объекта. Это отличается от крупномасштабных перемещений, которые мы программировали в предыдущих главах.

Последний графический ресурс, упоминавшийся в табл. 4.1, — это система частиц.

ОПРЕДЕЛЕНИЕ Системой частиц (particle system) называется механизм создания большого количества движущихся объектов и управления ими. Обычно эти объекты имеют маленький размер — именно поэтому они называются частицами, хотя это не является обязательным условием.

Системы частиц служат для создания таких визуальных эффектов, как огонь, дым или водяные брызги. Роль частиц (то есть отдельных объектов, контролируемых системой) может играть любой сеточный объект, но для большинства эффектов достаточно квадрата, воспроизводящего изображение (например, искры пламени или клуба дыма).

В основном создание графики для игры выполняется во внешних программах, никак не связанных с Unity. В Unity можно генерировать только материалы и системы частиц. Список внешних программ вы найдете в приложении Б; для создания трехмерных моделей и анимации применяются самые разные графические редакторы. Полученные во внешней программе трехмерные модели затем сохраняются как графические ресурсы, то есть импортируются в Unity. Я пользуюсь программой Blender, о которой рассказывается в приложении В. Ее можно скачать с сайта www.blender.org. Мой выбор обусловлен тем, что эта программа с открытым исходным кодом, бесплатно доступная всем желающим.

ПРИМЕЧАНИЕ Проект, который можно скачать для этой главы, включает в себя папку с именем scratch. Хотя эта папка помещена в проект Unity, она не является его частью; я поместил в эту папку дополнительные внешние файлы.

Работая над проектом этой главы, вы познакомитесь с примерами большинства типов графических ресурсов (анимация пока относится к слишком сложным темам,

поэтому она рассматривается в следующих главах). Вам же предстоит построить сцену, в которой фигурируют двухмерные изображения, трехмерные модели, материалы и система частиц. В некоторых случаях вы будете пользоваться уже готовыми графическими ресурсами, импортируя их в Unity, но будут и ситуации (особенно с системой частиц), когда графический ресурс придется создавать с нуля.

В этой главе мы только слегка затронем тему создания игровой графики. Все-таки эта книга посвящена в основном программированию в Unity, поэтому подробное рассмотрение вопросов, связанных с графикой, уменьшит количество информации по главной теме. Создание игровой графики — огромная предметная область, для детального описания которой потребуется не одна книга. В большинстве случаев программисты работают в паре со специалистами по графике. Принимая во внимание сказанное, человек, занимающийся программированием игр, должен понимать, как Unity работает с графическими ресурсами, и, возможно, даже уметь создавать их грубые заменители; их еще называют *программистской графикой* (programmer art) и позднее (в готовой игре) заменяют нужными графическими ресурсами.

ПРИМЕЧАНИЕ Проекты из предыдущей главы для выполнения заданий вам не потребуются. Воспользуйтесь сценариями движения из главы 2, чтобы получить возможность перемещаться по сцене; при необходимости можно взять модель игрока и сценарии из скачанного с сайта проекта. К концу этой главы вы создадите движущиеся объекты, напоминающие полученные в предыдущих главах.

4.2. Создание геометрической модели сцены

Разговор о моделировании сцен мы начнем с рассмотрения процесса создания геометрической модели сцены (whiteboxing). Обычно это первый шаг моделирования уровня с помощью компьютера (следующий за разработкой этого уровня на бумаге). Как следует из английского термина, объекты сцены создаются из набора примитивов, то есть белых ящиков (white boxes). В списке различных ресурсов пустые декорации соответствуют базовому виду трехмерной модели и являются основой для отображения двухмерных картинок. Вспомните сцену, которую мы создали из примитивов в главе 2. Это и есть геометрическая модель (просто на тот момент вы еще не знали этого термина). В некоторых разделах вы найдете отсылки к вещам, которые мы делали в начале главы 2, но на этот раз я буду касаться их совсем коротко, попутно сократив обсуждение новой терминологии.

ПРИМЕЧАНИЕ В английском языке также используется термин grayboxing (gray boxes — серые ящики). Он также означает геометрическую модель сцены, но я предпочитаю слово whiteboxing, потому что узнал его раньше. Реальный цвет примитивов может различаться, точно так же, как светокопии, называемые «синьками» (blueprints), далеко не всегда имели синий цвет.

4.2.1. Назначение геометрической модели

Составление сцены из примитивов практикуется по двум причинам. Во-первых, это позволяет быстро получить «набросок», который со временем будет постепенно совершенствоваться. Эта деятельность близко связана с проектированием игровых уровней.

ОПРЕДЕЛЕНИЕ Проектирование уровней (level design) — это дисциплина, касающаяся планирования и создания в игре сцен (или уровней). Проектировщик уровней — это тот, кто занимается проектированием уровней.

По мере увеличения количества разработчиков в группе и сужения специализации каждого отдельного члена группы создание первой версии каждого игрового уровня в виде геометрической модели отдается на откуп проектировщикам уровней. Затем эта заготовка передается специалистам по графике для визуальной доработки. Но даже в маленькой группе, где за проектирование уровней и работу с графикой может отвечать один и тот же человек, такая последовательность действий является оптимальной. В конце концов, нужно с чего-то начинать, а геометрическая модель становится хорошей основой для создания визуальных эффектов.

Второй причиной использования геометрических моделей является возможность быстро привести сцену в подходящее для игры состояние. Она может быть не окончена (более того, уровень, на котором построена только геометрическая модель, очень далек от завершения), но это уже функциональная версия, поддерживающая игровой процесс. Как минимум, игрок в состоянии перемещаться по сцене (вспомните демонстрационный ролик из главы 2). Можно провести тестирование и убедиться, что игровой уровень построен корректно (например, комнаты имеют подходящий для игры размер), и только после этого тратить время и энергию на его проработку. Если окажется, что чего-то не хватает (скажем, вы поняли, что требуется больше места), несложно внести изменения и провести повторное тестирование на стадии геометрической модели.

Более того, возможность поиграть даже на стадии конструирования уровня хорошо поднимает моральный дух. Не сбрасывайте это преимущество со счетов. Создание богатой в визуальном отношении сцены может занять долгое время, и вы почувствуете усталость от того, что никак не можете воспользоваться плодами своего труда. Геометрическая модель сразу дает вам готовый (хотя и примитивный) игровой уровень и возможность играть в постоянно совершенствующуюся игру.

Итак, теперь, когда вы понимаете, почему разработки всех уровней начинаются с геометрических моделей, давайте приступим к созданию игры!

4.2.2. Рисуем план уровня

Созданию игрового уровня на компьютере предшествует его проектирование на бумаге. Мы не будем подробно обсуждать тему проектирования уровней; достаточно сделанного в главе 2 примечания по поводу проектирования игр. Проектирование уровней (представляющее собой разновидность проектирования игр) — это обширная область знаний, достойная отдельной книги. Мы же нарисуем базовый план уровня, чтобы обозначить цель, к которой нужно стремиться.

Рисунок 4.1 демонстрирует вид сверху простого помещения, состоящего из четырех комнат и центрального коридора. Это все, что нам на данный момент нужно: набор отдельных областей и внутренние стены, которые требуется установить на свои места. План реальной игры будет содержать больше деталей, например таких, как враги и фрагменты обстановки. Попрактиковаться в создании геометрической модели сцены можно как на примере этого плана, так и взяв за основу собственные идеи. Конкретное

расположение комнат в этом упражнении не имеет значения. Важно только наличие у вас нарисованного плана, что позволит перейти к следующему этапу.

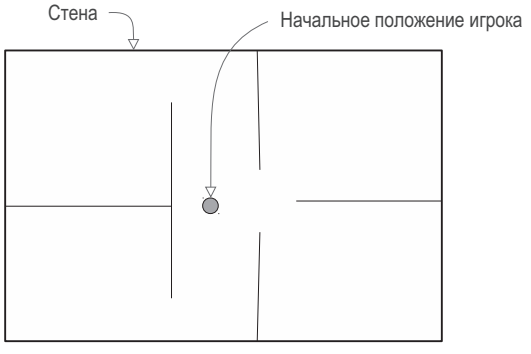


Рис. 4.1. План игрового уровня: четыре комнаты и центральный коридор

4.2.3. Расставляем примитивы в соответствии с планом

Построение геометрической модели сцены в соответствии с имеющимся планом включает в себя позиционирование и масштабирование множества параллелепипедов, которые будут играть роль стен. Как описано в разделе 2.2.1, выберите в меню GameObject команду 3D Object, а затем команду Cube, чтобы получить куб для дальнейших преобразований.

ПРИМЕЧАНИЕ При желании вместо кубов вы можете воспользоваться объектом QuadsBox, скачанным вместе с проектом. Он представляет собой куб, созданный из шести частей, что дает дополнительную гибкость при назначении материала. Выбор в данном случае зависит только от вашего желания; лично я не стал пользоваться объектом QuadsBox, так как вся геометрия со временем все равно заменяется графикой.

Первым объектом сцены станет пол; на панели Inspector поменяйте имя примитива и его положение по координате *Y* на –0.5, как показано на рис. 4.2. Это делается для компенсации высоты объекта. Затем растяните куб по осям *X* и *Z*.

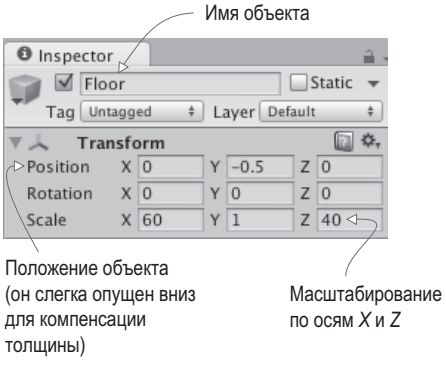


Рис. 4.2. Параметры куба на панели Inspector после перемещения и масштабирования

Повторите эти шаги, чтобы получить стены. Скорее всего, вы захотите привести в порядок вкладку Hierarchy, сделав стены потомками общего базового объекта (напоминаю, что такой объект нужно поместить в точку с координатами 0, 0, 0, а затем на вкладке Hierarchy перетащить на него остальные объекты), но это действие не является обязательным. И не забудьте расположить в сцене несколько простых источников света, чтобы видеть окружающее пространство. В главе 2 вы уже узнали, что для создания источника света нужно выбрать его тип в дополнительном меню, которое появляется после выбора команды Light в меню GameObject. Примерный вид вашего уровня после этих операций показан на рис. 4.3.

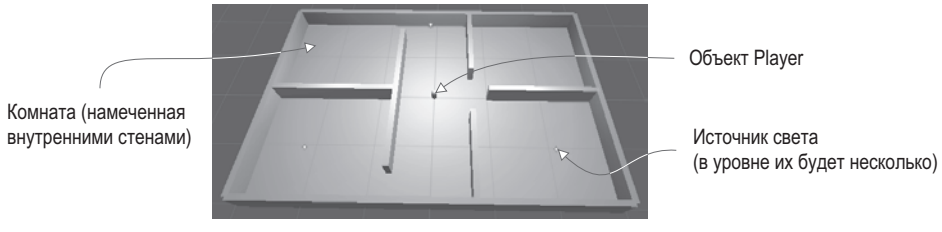


Рис. 4.3. Геометрическая модель игрового уровня, построенного по плану, представленному на рис. 4.1

Заставьте объект, представляющий игрока или камеру, двигаться по сцене (для создания игрока воспользуйтесь контроллером персонажа и сценариями движения — эта тема подробно объяснялась в главе 2). Теперь можно походить по сделанной из примитивов сцене, чтобы протестировать получившийся уровень! Все очень просто, но пока у вас есть только чистая геометрия. Декорируем ее с помощью двухмерных изображений.

ЭКСПОРТ ГЕОМЕТРИЧЕСКИХ МОДЕЛЕЙ В ДРУГИЕ ПРОГРАММЫ

Большая часть работы по визуальному оформлению сцены выполняется во внешних приложениях для работы с трехмерной графикой, например в программе Blender. Удобнее всего это делать, загрузив во внешнюю программу свою геометрическую модель. Хотя по умолчанию возможность экспорта скомпонованных примитивов в Unity отсутствует, существуют сценарии, позволяющие добавить в редактор такую функциональность. Большинство из них дают возможность выделить в сцене всю геометрию и щелкнуть на кнопке Export (я упоминал их в главе 1, в разделе, посвященном настройкам редактора).

Эти сценарии обычно экспортируют геометрию как OBJ-файл (этот тип файлов обсуждается чуть позже). На сайте Unity3D щелкните на кнопке поиска и введите запрос obj exporter. Или можете посмотреть пример такого сценария на странице http://wiki.unity3d.com/index.php?title=ObjExporter.

4.3. Наложение текстур

Пока что наш уровень представляет собой грубый набросок. Он уже доступен для игры, но очевидно, что над внешним видом сцены требуется еще долго работать. Следующим шагом по совершенствованию уровня будет наложение текстур.

ОПРЕДЕЛЕНИЕ Текстурой (texture) называется двухмерное изображение, применяемое для улучшения качества трехмерной графики. Это обобщенное определение термина; различные способы использования текстур тут не учитываются. Впрочем, каким бы способом изображение ни использовалось, оно все равно будет называться текстурой.

В трехмерной графике текстуры имеют ряд применений, но наиболее простым является отображение их на поверхности трехмерных моделей. Позднее вы узнаете, как это происходит в случае сложных моделей, сейчас же мы, по сути, покроем стены обоями, как показано на рис. 4.4.

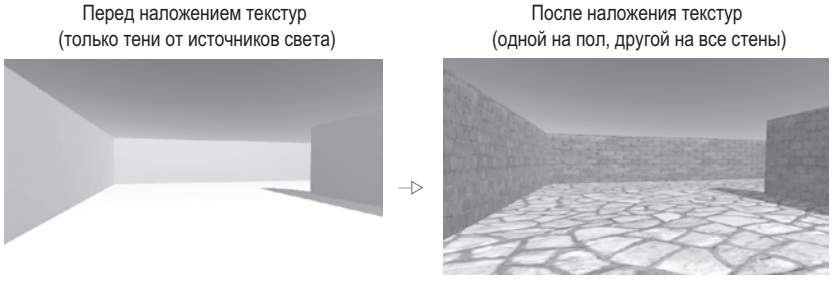


Рис. 4.4. Внешний вид игрового уровня до и после наложения текстур

Как видно из рисунка, текстура превращает откровенно нереалистичную цифровую конструкцию в кирпичную стену. Другие варианты применения текстур включают в себя маски, позволяющие вырезать фигуры, и карты нормалей для имитации рельефа. Рекомендую вам самостоятельно почитать дополнительные материалы о работе с текстурами.

4.3.1. Выбор формата файла

Для сохранения двухмерных изображений существует множество форматов. Какой из них лучше выбрать? Форматы, поддерживаемые Unity, перечислены в табл. 4.2.

Таблица 4.2. Форматы файлов двухмерных изображений, поддерживаемые Unity

Формат	Достоинства и недостатки
PNG	Повсеместно используется в Интернете. Сжатие без потерь; есть альфа-канал
JPG	Повсеместно используется в Интернете. Сжатие с потерями; нет альфа-канала
GIF	Повсеместно используется в Интернете. Сжатие с потерями; нет альфа-канала. (Технически потери возникают не из-за сжатия, а в результате преобразования к 8-битному изображению. Но конечный результат все равно один и тот же)
BMP	Формат, по умолчанию используемый в Windows. Применяется без сжатия; нет альфа-канала
TGA	Повсеместно применяется в трехмерной графике; во всех прочих областях малоизвестен. Используется без сжатия, но возможно и сжатие без потерь; есть альфа-канал

Таблица 4.2 (продолжение)

Формат	Достоинства и недостатки
TIFF	Повсеместно применяется в цифровой фотографии и издательском деле. Используется без сжатия, но возможно и сжатие без потерь; нет альфа-канала
PICT	Формат по умолчанию на старых компьютерах Mac. Сжатие с потерями; нет альфа-канала
PSD	Собственный формат Photoshop. Используется без сжатия; есть альфа-канал. Основным достоинством является возможность непосредственной работы с файлами в Photoshop

ОПРЕДЕЛЕНИЕ Альфа-канал (alpha channel) служит для хранения информации о прозрачности изображения. Видимые цвета поступают по трем «каналам»: красному, зеленому и синему. Альфа — это дополнительный невидимый канал, управляющий прозрачностью изображения.

Хотя Unity допускает импорт и использование в качестве текстур изображений всех перечисленных в табл. 4.2 форматов, форматы значительно различаются по количеству поддерживаемых функций. Для файлов, импортируемых в качестве текстур, особенно важны два фактора: как именно сжимается изображение и есть ли у него альфа-канал. С альфа-каналом все просто: так как он часто используется в трехмерной графике, лучше, чтобы у изображения он был. Объяснить важность разных аспектов сжатия чуть сложнее. Впрочем, объяснение можно свести к фразе «сжатие с потерями — это плохо». Изображения, используемые без сжатия и допускающие сжатие без потерь, сохраняют свое качество, в то время как при сжатии с потерями качество падает по мере уменьшения размера файла.

С учетом этих соображений я рекомендовал бы использовать в качестве текстур в Unity файлы формата PNG или TGA. Формат Targas (TGA) был любимым вариантом для создания текстур, пока в Интернете не получил распространение формат PNG. В наше время PNG с технологической точки зрения является практически эквивалентом формата TGA, но получил большее распространение благодаря применению как в качестве текстур, так и в Интернете. К числу рекомендуемых форматов относится также PSD, потому что удобно работать с одним и тем же файлом как в Photoshop, так и в Unity. Хотя я предпочитаю хранить рабочие файлы отдельно от «готовых» экспортированных в Unity вариантов (аналогичных взглядов я придерживаюсь касательно хранения трехмерных моделей, но об этом речь пойдет позже).

В результате все изображения, представленные в примере проекта, имеют формат PNG, и я рекомендую вам работать именно с ним. А теперь пришло время импортировать в Unity несколько изображений и применить их к объектам сцены.

4.3.2. Импорт файла изображения

Начнем с создания/подготовки наших будущих текстур. Все изображения, используемые в качестве текстур, обычно являются бесшовными, что позволяет многократно повторять их на больших поверхностях.

ОПРЕДЕЛЕНИЕ Бесшовное изображение (tileable image) представляет собой рисунок, края которого совпадают друг с другом. Именно это позволяет повторять его на поверхности без видимых швов в местах соединения. Концепция назначения текстур в трехмерном моделирования сходна с использованием фоновых рисунков на веб-страницах.

Получить бесшовное изображение можно различными способами, например обработать фотографию или нарисовать собственный вариант картинки. Учебные пособия и объяснения можно найти в различных книгах и на сайтах, но сейчас мы не будем тратить на это время. Вместо этого мы воспользуемся изображениями с сайтов, предлагающих наборы графики для трехмерного моделирования. Например, показанные на рис. 4.5 текстуры я скачал с сайта www.textures.com. Именно их я собираюсь назначить полу и стенам; вы можете выбрать собственные картинки, подходящие, по вашему мнению, для этой цели.

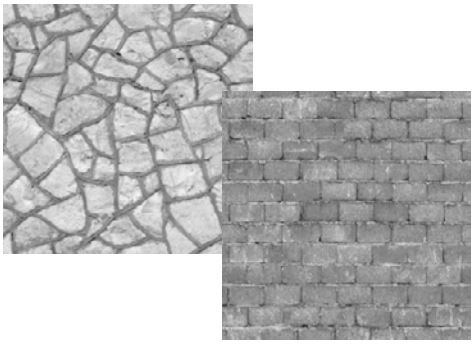


Рис. 4.5. Бесшовные текстуры камня и кирпичей, скачанные с сайта textures.com

Скачайте выбранные вами изображения и подготовьте их к использованию в качестве текстур. С технической точки зрения ничто не мешает задействовать их сразу, но в исходном виде они далеки от идеала. Разумеется, они являются бесшовными (именно поэтому мы их и скачали), но рисунок имеет некорректный размер, а файл — не тот формат, который нам нужен. Размер текстуры должен выражаться в степенях двойки. Графические процессоры показывают максимальную эффективность при обработке изображений, размер которых выражается числом $2N$: 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 (следующее в этом ряду — число 4096, но это слишком большое изображение, чтобы использовать его в качестве текстуры). В графическом редакторе (это может быть Photoshop, GIMP или любой другой вариант из перечисленных в приложении Б) отмасштабируйте скачанные изображения до размера 256×256 и сохраните их в формате PNG.

Теперь перетащите эти файлы из папки на вашем компьютере на вкладку Project в Unity, как показано на рис. 4.6. Это действие позволит скопировать их в Unity-проект, после чего их можно будет использовать в трехмерной сцене. Если перетаскивать файлы вам по каким-то причинам неудобно, щелкните правой кнопкой мыши на вкладке Project и выберите в появившемся меню команду Import New Asset, чтобы открыть окно выбора файлов.

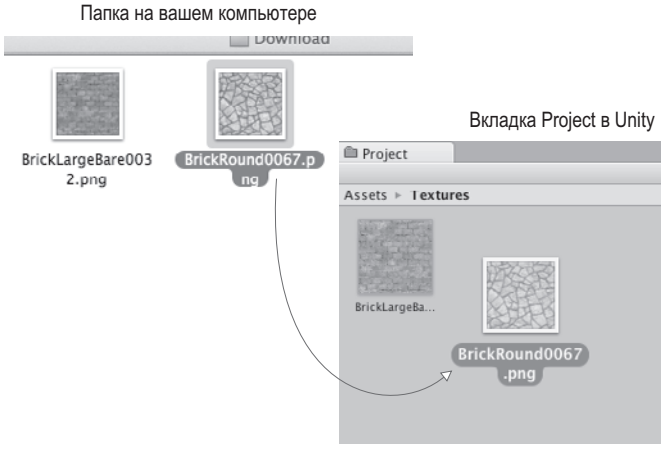


Рис. 4.6. Перетащите изображения на вкладку Project, чтобы импортировать их в Unity

СОВЕТ По мере усложнения проектов имеет смысл распределить ресурсы по отдельным папкам; на вкладке Project создайте папки для сценариев и текстур и перетащите в них соответствующие ресурсы.

ВНИМАНИЕ В Unity есть несколько ключевых слов, совпадающих с именами папок. Они инициируют обработку содержимого этих папок специальным образом. Это ключевые слова Resources, Plugins, Editor и Gizmos. Зачем нужны эти папки, вы узнаете позже, а пока просто избегайте этих слов, выбирая имена для своих папок.

Теперь изображения импортированы в Unity как текстуры и готовы к использованию. Как же назначить их объектам сцены?

4.3.3. Назначение текстуры

С технической точки зрения наложить текстуру непосредственно на геометрию невозможно. Текстуры должны входить в состав материалов, которые, собственно, и назначаются объектам. Как объяснялось во введении, материалом называется пакет информации, описывающий свойства поверхности; эта информация может включать в себя и отображаемую текстуру. Подобный подход имеет смысл, так как позволяет использовать одну и ту же текстуру для разных материалов. Но так как обычно все текстуры фигурируют в составе разных материалов, для удобства в Unity можно просто поместить текстуру на объект, в результате новый материал создается автоматически. Если вы перетащите текстуру с вкладки Project на объект сцены, как показано на рис. 4.7, Unity создаст новый материал и назначит его объекту. Попытайтесь таким образом получить материал для пола.

Кроме этого удобного метода автоматического создания материалов существует еще и «корректный» способ через подменю, которое появляется после выбора в меню Assets команды Create; новый ресурс появляется на вкладке Project. Остается только выделить полученный новый материал, чтобы его свойства отобразились на панели Inspector, и, как показано на рис. 4.8, перетащить текстуру на ячейку с именем Albedo (это технический термин для базового цвета). Перетащите полученный материал со вкладки Project

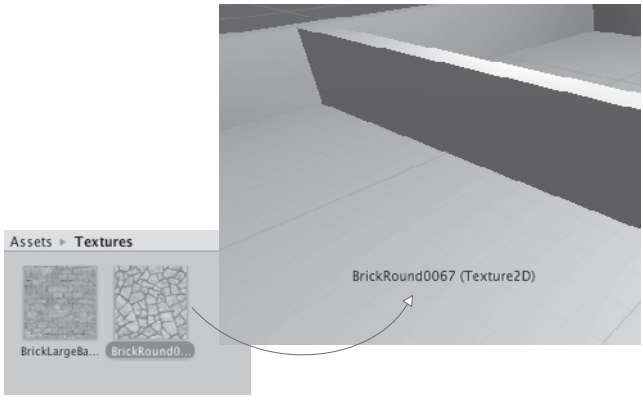


Рис. 4.7. Одним из способов наложения текстур является их перетаскивание со вкладки Project на объекты сцены

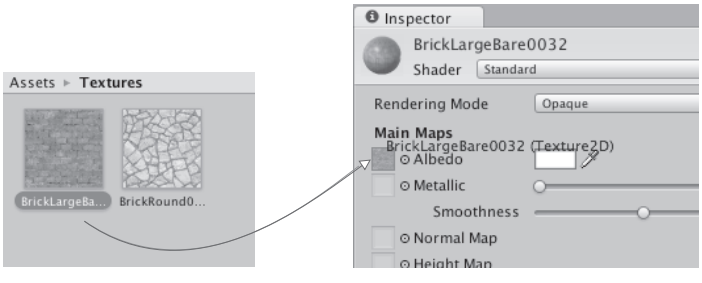


Рис. 4.8. Выделите материал для просмотра его свойств на панели Inspector и перетащите структуру на ячейку одного из свойств

на объект сцены. Попробуйте проделать все описанное с текстурой для стены: создайте новый материал, перетащите в него текстуру и назначьте его стене.

Вы увидите, что на поверхности пола и стен появились изображения камня и кирпичей, но они выглядят растянутыми и размытыми. Как получилось, что единственное изображение оказалось растянутым на весь пол? Мы же хотели, чтобы оно повторялось на поверхности несколько раз. Такой эффект дает свойство Tiling: выделите материал на вкладке Project и измените числа в полях Tiling на панели Inspector (существуют отдельные значения для координат *X* и *Y*, отвечающие за количество повторений в каждом направлении). Проверьте, что вы задаете повторение основной, а не вторичной карты (данный материал поддерживает вторичную карту текстуры для усовершенствованных эффектов). По умолчанию число повторений равно 1 (то есть единственная текстура растягивается на всю поверхность); присвойте этому параметру, к примеру, значение 8 и посмотрите, как изменится вид пола. Подберите и для второго материала кратность, обеспечивающую оптимальный вид.

Итак, пол и стены нашей комнаты обзавелись текстурами! Но вы можете назначить текстуру и небу; давайте посмотрим, как это делается.

4.4. Создание неба с помощью текстур

Текстуры камня и кирпича придали стенам и полу намного более естественный вид. Но небо пока выглядит пустым и ненатуральным, мы же хотим придать сцене реалистичность. Чаще всего эта задача решается при помощи специальных текстур с изображениями неба.

4.4.1. Что такое скайбокс?

По умолчанию камера показывает фоновый темно-синий цвет. Он заполняет все пустое пространство сцены (например, пространство над стенами). Но в качестве фона можно визуализировать и изображение неба. В этом нам поможет *скайбокс*.

ОПРЕДЕЛЕНИЕ Скайбоксом (skybox) называется окружающий камеру куб, на грани которого находится изображение неба. В каком бы направлении ни смотрела камера, она будет отображать небо.

Корректная реализация скайбокса — дело непростое; принцип его работы иллюстрирует рис. 4.9. Существует ряд приемов, позволяющих отобразить грани куба как удаленный фон. К счастью, все детали реализации в Unity уже учтены.

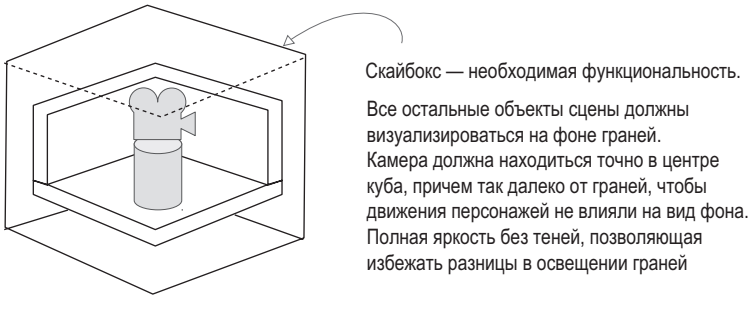


Рис. 4.9. Схема скайбокса

Новые сцены создаются с уже готовым скайбоксом. Именно поэтому вместо равномерного темно-синего фона цвет неба постепенно меняется от светлого к темно-синему. Если открыть окно диалога с параметрами освещенности (выбрав в меню Window команду Lighting), первым вы увидите параметр Skybox со значением Default. Этот параметр находится в свитке Environment Lighting; окно диалога разделено на свитки, связанные с усовершенствованной системой освещения в Unity. Впрочем, пока нас интересует только самый первый параметр.

Текстуры для скайбокса, как и текстуры кирпича и камня, можно найти на различных сайтах. Воспользуйтесь поисковым запросом *текстуры для скайбокса* (skybox textures). Например, я нашел несколько прекрасных вариантов на сайте www.93i.de, в том числе набор TropicalSunnyDay. После добавления к скайбоксу текстуры неба сцена начнет выглядеть так, как показано на рис. 4.10.

Как и прочие текстуры, изображения для скайбокса сначала назначаются материалу и только потом используются в сцене. Давайте попробуем создать для скайбокса новый материал.

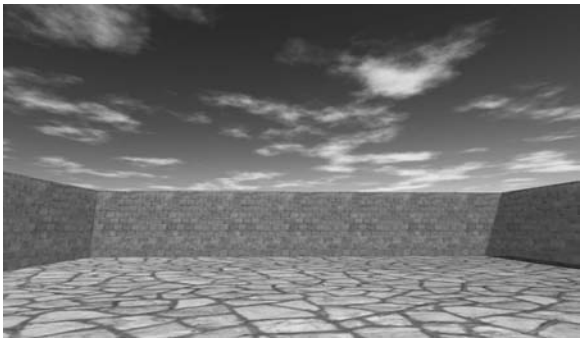


Рис. 4.10. Сцена с фоновым изображением неба

4.4.2. Создание нового материала для скайбокса

Сначала создайте новый материал (как обычно, щелкнув правой кнопкой мыши и выбрав команду Create или выбрав эту же команду в меню Assets). Его параметры отобразятся на панели Inspector. Первым делом нам нужно поменять раскраску материала. В верхней части списка настроек находится меню Shader, показанное на рис. 4.11. В разделе 4.3 мы не обращали на него внимания, так как *раскраска*, предлагаемая по умолчанию, подходит большинству стандартных текстур, но скайбокс требует другого варианта.

ОПРЕДЕЛЕНИЕ Раскраской (shader) называется короткая программа с инструкциями, описывающими способ рисования поверхности. В ней указываются, в частности, используемые текстуры. Компьютер задействует эти инструкции для вычисления пикселов в процессе визуализации изображения. В наиболее распространенной раскраске цвет материала затемняется в соответствии с освещенностью. Раскраски применяются для всех видов визуальных эффектов.

Каждый материал имеет раскраску, которая определяет его вид (можно представить материал как экземпляр раскраски). Новому материалу по умолчанию назначается раскраска Standard. Она отображает цвет материала (включая назначенную текстуру), одновременно применяя к поверхности основные настройки теней и освещенности.

Для скайбоксов используется другая раскраска. Щелкните на этом меню, чтобы открыть выпадающий список с перечнем доступных раскрасок. Выделите строку Skybox и выберите в появившемся дополнительном меню вариант 6 Sided, как показано на рис. 4.11.

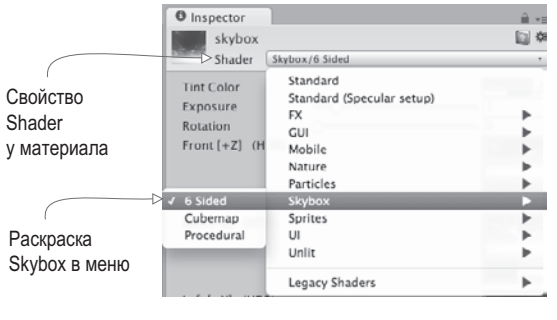


Рис. 4.11. Выпадающее меню доступных раскрасок

Теперь в настройках материала появилось шесть ячеек для текстур (вместо одной маленькой ячейки Albedo, которую мы видели у стандартной раскраски). Эти шесть текстур соответствуют шести сторонам куба. Они должны совпадать друг с другом в местах стыка, чтобы картинка получилась бесшовной. Например, рис. 4.12 демонстрирует изображения для солнечного скайбокса.

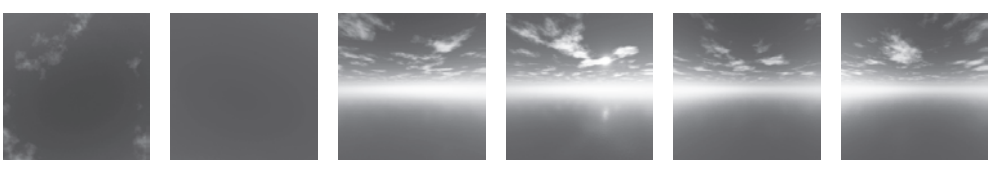


Рис. 4.12. Шесть сторон скайбокса

Импортируйте в Unity изображения для скайбокса тем же способом, которым импортировалась текстура кирпича: перетащите файлы на вкладку Project или щелкните правой кнопкой мыши на вкладке Project и выберите команду Import New Asset. Впрочем, в данном случае есть одно небольшое отличие; щелчком выделите импортированную текстуру, чтобы увидеть ее свойства на панели Inspector, и поменяйте значение параметра Wrap Mode с Repeat на Clamp (рис. 4.13); не забудьте после этого щелкнуть на кнопке Apply. Обычно текстуры укладываются на поверхность как плитки, а чтобы результат такой укладки выглядел бесшовным, противоположные края изображений накладываются друг на друга. Но в случае неба подобная операция может привести к появлению небольших линий, поэтому значение Clamp (аналогичное знакомой вам по главе 2 функции `Clamp()`) очертит границы текстуры и уберет результат их наложения.

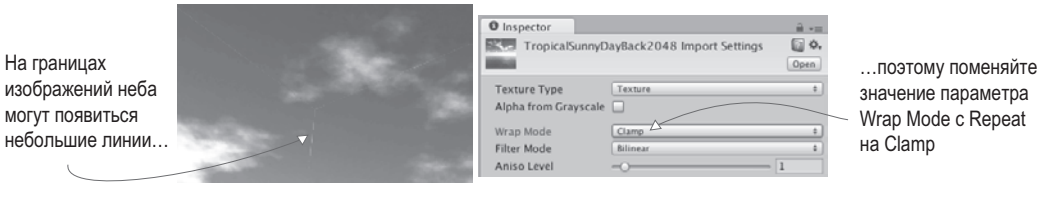


Рис. 4.13. Избавление от линий путем редактирования параметра Wrap Mode

Теперь можно перетащить изображения на ячейки для текстур. Имена изображений должны совпадать с именами ячеек (например, left или front). Как только все текстуры окажутся на своих местах, можно использовать материал для скайбокса. Снова откройте окно с параметрами освещенности и перетащите новый материал на ячейку Skybox или щелкните на маленьком кружке с точкой в центре, расположенном справа от ячейки, чтобы открыть окно выбора материала.

СОВЕТ По умолчанию Unity отображает скайбокс (или, по крайней мере, его основной цвет) на вкладке Scene редактора. Если это мешает редактированию объектов, видимость скайбокса можно отключить. В верхней части вкладки Scene располагаются кнопки, управляющие видимостью различных элементов; щелчок на крайней правой кнопке, которая называется Effects, открывает меню, через которое можно отключить видимость скайбокса.

Итак, вы узнали, как смоделировать настоящее небо! Скайбокс предлагает вам элегантный способ создать иллюзию бескрайнего пространства вокруг игрока. Следующим шагом по совершенствованию внешнего вида сцены станет получение более сложных трехмерных моделей.

4.5. Собственные трехмерные модели

В предыдущем разделе мы накладывали текстуры на большие плоские стены и пол. А что делать с более детализированными объектами? Предположим, мы хотим обставить комнаты мебелью. Для решения этой задачи нам потребуется внешнее приложение для работы с трехмерной графикой. Вспомните определение, которое было дано в начале этой главы: трехмерные модели — это помещенные в игру сеточные объекты (то есть трехмерные фигуры). В этом разделе мы импортируем в игру сетку, имеющую форму скамейки.

Для моделирования трехмерных объектов широко применяются такие приложения, как Maya от Autodesk и 3ds Max. Но это дорогие коммерческие инструменты, поэтому мы воспользуемся приложением с открытым исходным кодом, которое называется Blender. Скачанный с сайта проект включает в себя файл с расширением .blend, которым вы можете воспользоваться; рис. 4.14 демонстрирует модель скамейки в программе Blender. На случай, если вы захотите научиться моделировать такие объекты собственными руками, в приложение В включено упражнение по созданию скамейки.

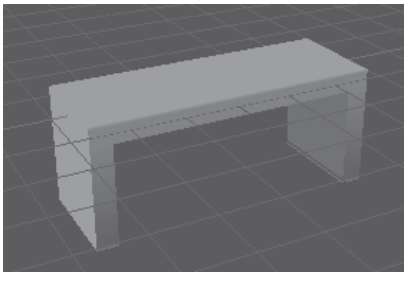


Рис. 4.14. Модель скамейки в программе Blender

Кроме моделей, созданных лично вами или сотрудничающим с вами художником, можно скачивать подходящие варианты со специальных сайтов. К примеру, существует такой замечательный ресурс, как Asset Store: https://www.assetstore.unity3d.com.

4.5.1. Выбор формата файла

Созданную в приложении Blender модель требуется экспортировать. Как и в случае с двухмерными изображениями, существует множество разных форматов экспорта, каждый из которых обладает своими достоинствами и недостатками. Поддерживаемые в Unity форматы трехмерных файлов перечислены в табл. 4.3.

Таблица 4.3. Форматы файлов трехмерных моделей, поддерживаемые в Unity

Формат	Достоинства и недостатки
FBX	Сетки и анимация; рекомендуемый формат
Collada (DAE)	Сетки и анимация; еще один хороший вариант, если формат FBX недоступен
OBJ	Только сетки; это текстовый формат, который иногда используется для трансляции в Интернет
3DS	Только сетки; достаточно старый и примитивный формат
DXF	Только сетки; достаточно старый и примитивный формат
Maya	Работает через FBX; требует установки этого приложения
3ds Max	Работает через FBX; требует установки этого приложения
Blender	Работает через FBX; требует установки этого приложения

Выбор варианта сводится к поддержке анимации. Так как единственными удовлетворяющими этому условию вариантами являются Collada и FBX, выбирать приходится между ними. Когда есть такая возможность (не все инструменты для работы с трехмерной графикой экспортируют данные в этом формате), лучше всего пользоваться форматом FBX, в противном случае подойдет и формат Collada. К счастью, приложение Blender допускает экспорт файлов в формате FBX.

Обратите внимание, что в нижней части табл. 4.3 перечислено несколько приложений для работы с 3D-графикой. Инструмент Unity позволяет прямо переносить их файлы в ваши проекты, что сначала кажется удобным, но эта функциональность имеет несколько подводных камней. Во-первых, Unity не загружает непосредственно сами файлы. Модель загружается в фоновом режиме, затем загружается экспортированный файл. Но так как модель в любом случае экспортируется в формате FBX или Collada, лучше делать это в явном виде. Во-вторых, для подобной операции у вас должно быть установлено соответствующее приложение. Если вы планируете организовать доступ к файлам с разных компьютеров (например, для группы разработчиков), это обстоятельство становится большой проблемой. Я не рекомендую загружать файлы из приложения Blender (или Maya, или еще откуда-то) напрямую в Unity.

4.5.2. Экспорт и импорт модели

Итак, пришло время экспортировать модель из Blender и импортировать ее в Unity. Первым делом откройте файл со скамейкой в приложении Blender и выберите в меню File команду Export ▸ FBX. Сохраненный файл импортируйте в Unity тем же способом, которым осуществлялся импорт изображений. Перетащите FBX-файл на вкладку Project или щелкните на этой вкладке правой кнопкой мыши и выберите команду Import New Asset. Трехмерная модель будет скопирована в Unity-проект, готовая к вставке в сцену.

ПРИМЕЧАНИЕ В доступный для скачивания пример проекта включен файл с расширением .blend, чтобы вы могли попрактиковаться в экспорте FBX-файлов из приложения Blender; даже если вы не хотите ничего моделировать самостоятельно, зачастую требуется конвертировать скачанные файлы в доступный для Unity формат. Если вы предпочитаете пропустить все шаги, связанные с приложением Blender, задействуйте имеющийся FBX-файл.

При импорте моделей желательно сразу же поменять несколько параметров. Unity масштабирует импортируемые модели до очень маленьких размеров (на рис. 4.15 показано, что вы увидите на панели Inspector, выделив такую модель), поэтому введите в поле Scale Factor значение 100, чтобы частично скомпенсировать параметр File Scale, равный 0.01. Можно также установить флажок Generate Colliders (генерировать коллайдеры), но это не обязательно; просто без коллайдера персонажи смогли бы проходить сквозь скамейку. Затем перейдите на вкладку Animation в параметрах импорта и сбросьте флажок Import Animation (импорт анимации) — ведь эту модель мы анимировать не будем.

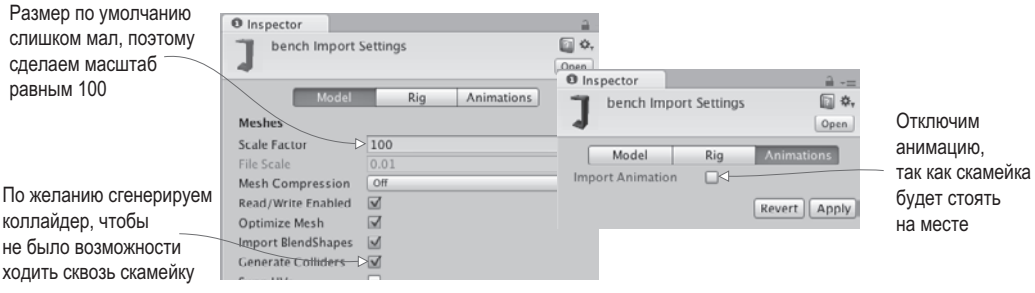


Рис. 4.15. Редактирование параметров импорта трехмерной модели

Это обеспечит нам корректный импорт сетки. Теперь что касается текстуры; при импорте FBX-файла инструмент Unity заодно создал материал для скамейки. По умолчанию он пустой (как любой новый материал), поэтому назначьте ему текстуру (показанную на рис. 4.16) тем же способом, каким вы назначали текстуру кирпича стене: перетащите изображение на вкладку Project, чтобы импортировать его в Unity, а затем — на ячейку текстуры в настройках материала скамейки. Изначально изображение будет выглядеть странновато, его разные части окажутся на разных частях скамейки, поэтому текстурные координаты были отредактированы для приведения изображения в соответствие с сеткой.

Рис. 4.16. Двухмерное изображение, служащее текстурой для скамейки

ОПРЕДЕЛЕНИЕ Текстурными координатами (texture coordinates) называется дополнительный набор значений для каждой вершины, проецирующий полигоны на области текстуры. Представьте, что трехмерная модель — это ящик, а текстура — оберточная бумага. Текстурные координаты будут указывать, на какую из сторон ящика должен накладываться каждый фрагмент бумаги.

ПРИМЕЧАНИЕ Даже если вы не собираетесь вникать в тонкости моделирования скамейки, имеет смысл ознакомиться с информацией о текстурных координатах (см. приложение В). Понимание этой концепции (как и связанных с ней понятий UV-координат и проецирования) пригодится при программировании игр.

Новые материалы часто имеют слишком сильный блеск, поэтому имеет смысл уменьшить параметр Smoothness до нуля (чем более гладкой является поверхность, тем сильнее она блестит). После этого скамейку можно поместить в сцену. Перетащите модель с вкладки Project в одну из комнат. Как только вы поставите объект на место, вы увидите нечто, напоминающее рис. 4.17. Поздравляю, вы создали для игрового уровня текстурированную модель!

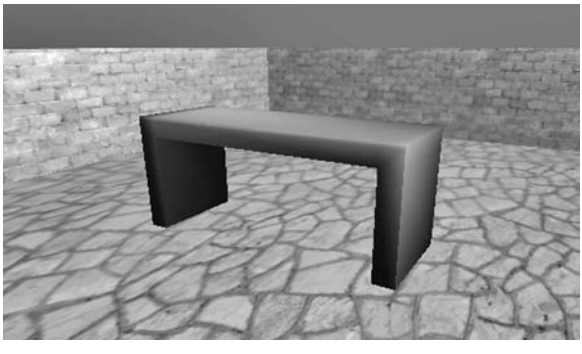


Рис. 4.17. Скамейка, импортированная в игровой уровень

ПРИМЕЧАНИЕ В этой главе мы не будем этим заниматься, но, как правило, геометрическая модель сцены позднее заменяется моделями, созданными во внешних программах. Новая модель может выглядеть почти идентично старой, но предоставлять большую гибкость при назначении текстуре координат U и V.

СИСТЕМА АНИМАЦИИ ПЕРСОНАЖЕЙ MECANIM

Созданная нами модель статична. Но вы можете анимировать ее в приложении Blender и воспроизвести эту анимацию в Unity. Процесс создания анимации длинный и сложный, а эта книга посвящена совсем другой теме, поэтому здесь мы его обсуждать не будем. Как уже упоминалось, существует множество ресурсов для детального знакомства с трехмерной анимацией. Просто имейте в виду — это крайне обширная тема. Именно поэтому анимацией при разработке игр занимается отдельный специалист.

В Unity встроена сложная система управления анимацией моделей. Она называется Mecanim. Это новая усовершенствованная система, недавно добавленная в Unity взамен старой. Впрочем, старая версия пока доступна, хотя и может быть удалена при следующих обновлениях Unity.

В этой главе мы ничего анимировать не будем, а вот в главе 7 вам предстоит заняться воспроизведением анимации для модели персонажа.

4.6. Системы частиц

Кроме двухмерных изображений и трехмерных моделей у нас остался еще один тип визуального содержимого — системы частиц. Данное в начале этой главы определение поясняет, что системами частиц называются механизмы создания большого

количества движущихся объектов и управления ими. Системы частиц применяются при создании таких эффектов, как огонь, дым или водяные брызги. Например, огонь на рис. 4.18 получен именно при помощи системы частиц.

Если большинство других ресурсов создается во внешних приложениях и импортируется в проект, системы частиц генерируются непосредственно в Unity благодаря гибким и мощным инструментам для создания различных эффектов.

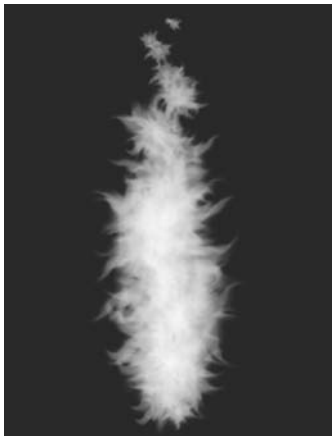


Рис. 4.18. Эффект огня, полученный при помощи системы частиц

ПРИМЕЧАНИЕ Почти как в случае с системой анимации Mecanim, в Unity соседствовали старая и более новая системы частиц. Последняя называлась Shuriken. В настоящее время старая система полностью вышла из употребления, так что отдельное имя новой системе уже не требуется.

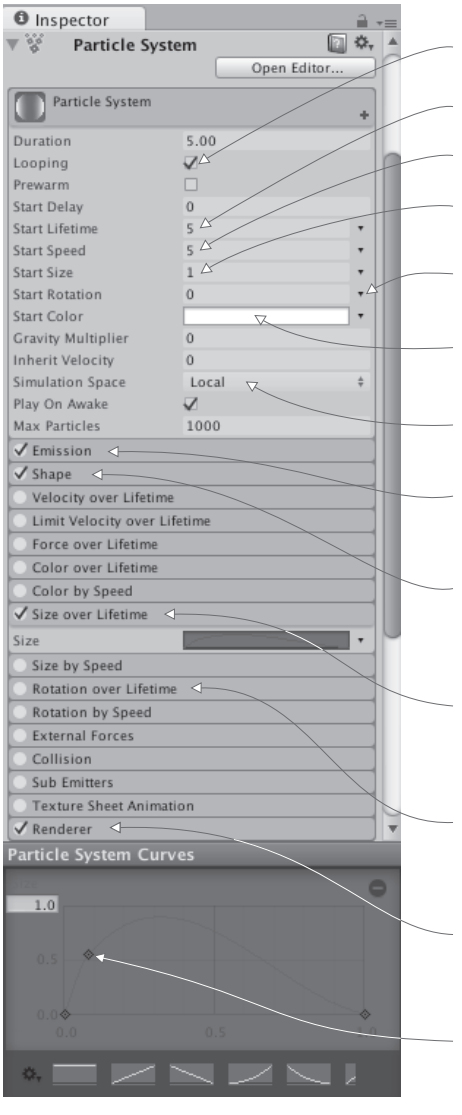
Для начала создайте систему частиц и выполните воспроизведение, чтобы посмотреть, как эффект выглядит по умолчанию. В меню GameObject выберите команду Particle System, и вы увидите, как из нового объекта полетят вверх белые пушинки. Для прекращения этого процесса достаточно снять с объекта выделение. При выделении системы частиц в нижнем правом углу экрана появляется панель воспроизведения эффекта, на которой, в частности, можно посмотреть, сколько времени работает система (рис. 4.19).

Рис. 4.19. Панель воспроизведения системы частиц

В принципе, эффект уже хорошо выглядит, но давайте рассмотрим список параметров, которыми вы можете пользоваться для его дополнительной настройки.

4.6.1. Редактирование параметров эффекта

Рисунок 4.20 демонстрирует полный список вариантов настройки системы частиц. Мы не будем рассматривать каждый параметр этого списка, а остановимся только на том, что требуется для создания эффекта пламени. Как только вы поймете принцип работы, остальное окажется достаточно очевидным. Каждая из надписей скрывает целый информационный свиток. Изначально раскрыт только первый из них; все остальные свернуты. Чтобы развернуть свиток, щелкните на его названии.



Looping: система частиц работает без остановки; оставьте установленный по умолчанию флажок

Lifetime: время жизни каждой частицы; уменьшите до 3

Speed: скорость перемещения частицы; уменьшите до 1

Size: размер частиц; оставьте значение по умолчанию

Rotation: ориентация частицы; щелкните на стрелке и выберите в меню вариант Between Constants, укажите значения 0 и 180

Color: цвет частиц. Нам требуется темно-оранжевый, например, со значениями RGB 182, 101, 58

Локальное пространство эффекта прекрасно подходит для статичных систем частиц, но для движущейся системы лучше выбрать значение World

Emission: скорость генерации новых частиц; оставьте значение по умолчанию

Shape: форма области, испускающей частицы. По умолчанию это широкий конус, нам же для получения компактного факела нужен куб (выберите вариант Box, все значения 0.2)

Size over Lifetime: в процессе движения частица увеличивается и уменьшается. По умолчанию эта настройка отключена. Включите ее и сформируйте кривую, которая быстро увеличивается от 0, а затем медленно уменьшается обратно до 0 (как и показано на рисунке)

Rotation over Lifetime: в процессе движения частица вращается. По умолчанию эта настройка отключена. Включите ее, выберите вариант Random Between и присвойте значения –80 и 80, чтобы частицы вращались в разных направлениях

Renderer: задает вид каждой частицы. Вы можете выбрать даже значение Mesh, но оставьте вариант Billboard и перетащите на ячейку новый материал (его мы сделаем позже)

Точки к кривой добавляются двойным щелчком или щелчком правой кнопкой мыши и выбором команды Add Key

Рис. 4.20. Панель Inspector отображает варианты настройки системы частиц (в данном случае — огня)

СОВЕТ Изрядное количество параметров редактируется с помощью кривой, отображаемой в нижней части панели Inspector. Она показывает изменение параметра во времени: левая сторона графика соответствует первому появлению частицы, правая — моменту ее исчезновения. Нижняя часть соответствует значению 0, а верхняя — максимально возможному значению. Меняйте форму кривой перетаскиванием точек, вставляя новые точки двойным щелчком или щелчком правой кнопки мыши.

Отредактируйте параметры системы частиц в соответствии с рис. 4.20, чтобы придать ей сходство с факелом.

4.6.2. Новая текстура для пламени

Теперь наша система частиц больше напоминает факел, но нужно придать частицам вид пламени, а не шариков. Для этого нам потребуется импортировать в Unity еще одно изображение. На рис. 4.21 показана нарисованная мной картинка: я поставил оранжевую точку и воспользовался инструментом Smudge для имитации языков пламени (затем я проделал то же самое с желтым цветом). Вне зависимости от того, возьмете ли вы изображение из примера проекта, нарисуете собственный вариант или скачаете подходящую картинку из Интернета, первым делом его нужно импортировать в Unity. Как я уже объяснял, перетащите картинку на панель Project или выберите в меню Assets команду Import New Asset.

Рис. 4.21. Изображение, используемое для частиц пламени

Системе частиц, как и трехмерной модели, невозможно назначить текстуру напрямую; поэтому добавьте эту текстуру к материалу и назначьте его системе частиц. Создайте новый материал и перетащите текстуру с вкладки Project на ячейку Albedo на панели Inspector. Это свяжет текстуру с материалом, после чего его можно будет добавить к системе частиц. Эта процедура показана на рис. 4.22. Выделите систему частиц, раскройте свиток Renderer в нижней части списка настроек и перетащите материал на ячейку Material.

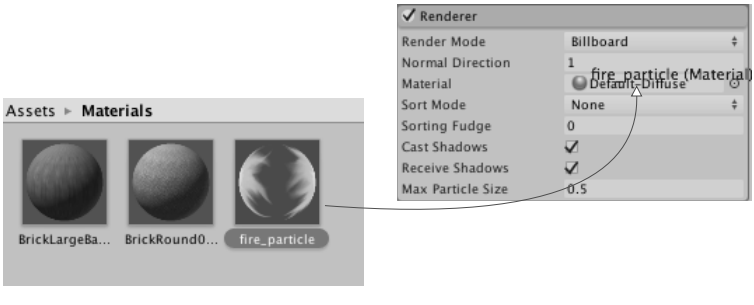


Рис. 4.22. Назначение материала системе частиц

Как и в случае материала для скайбокса, вам нужно выбрать другой вариант раскраски. Щелчком откройте меню Shader в верхней части настроек материала, чтобы увидеть список доступных раскрасок. Вместо используемого по умолчанию стандартного варианта материал для частиц нуждается в одном из вариантов раскраски, перечисленных в дополнительном меню Particles. Как показано на рис. 4.23, на этот раз мы выбираем вариант Additive (Soft). Это заставит частицы опалесцировать и освещать сцену, как настоящий огонь.

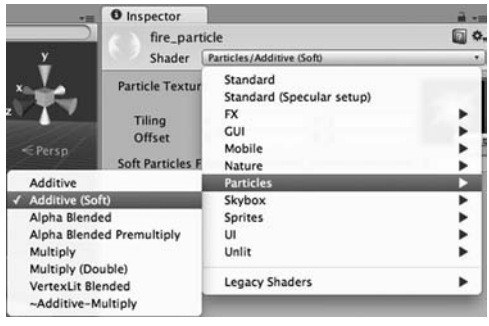


Рис. 4.23. Выбор раскраски материала для частиц пламени

ОПРЕДЕЛЕНИЕ Вариант раскраски Additive добавляет цвет частицы к цвету фона за ней, а не заменяет пикселы. Благодаря этому пикселы становятся ярче, а черный цвет делается невидимым. Противоположный эффект оказывает вариант Multiply, делающий все вокруг темнее; эти раскраски производят тот же самый эффект, что и эффекты слоя Additive и Multiply в приложении Photoshop.

После назначения системе частиц материала, имитирующего огонь, мы получили нужный эффект (см. рис. 4.18). Факел выглядит достаточно убедительно, причем он в состоянии работать не только в статичном положении. Чтобы убедиться в этом, давайте присоединим его к движущемуся объекту.

4.6.3. Присоединение эффектов частиц к трехмерным объектам

Создайте сферу (напоминаю, что в меню GameObject нужно выбрать команду 3D Object ▸ Sphere). Создайте новый сценарий с именем `BackAndForth`, показанный в следующем листинге, и присоедините его к сфере.

Листинг 4.1. Движение объекта взад и вперед по прямой

```
using UnityEngine;
using System.Collections;

public class BackAndForth : MonoBehaviour {
  public float speed = 3.0f;
  public float maxZ = 16.0f; ← Объект движется между этими точками.
  public float minZ = -16.0f;

  private int _direction = 1; ← В каком направлении объект движется в данный момент?

  void Update() {
    transform.Translate(0, 0, _direction * speed * Time.deltaTime);

    bool bounced = false;
    if (transform.position.z > maxZ || transform.position.z < minZ) {
      _direction = -_direction; ← Меняем направление на противоположное.
      bounced = true;
    }
    if (bounced) { ← Делаем дополнительное движение в этом кадре, если объект поменял направление.
      transform.Translate(0, 0, _direction * speed * Time.deltaTime);
    }
  }
}
```

Запустите этот сценарий, и сфера начнет двигаться взад и вперед по центральному коридору уровня. Теперь можно сделать систему частиц дочерней по отношению к сфере, и огонь начнет перемещаться вместе с ней. Точно так же, как вы поступали со стенами, на вкладке Hierarchy перетащите объект Particle system на объект Sphere.

ВНИМАНИЕ Обычно после того, как объект становится потомком другого объекта, его положение требуется обнулить. К примеру, нам нужно поместить систему частиц в точку 0, 0, 0 (относительно ее предка). В Unity сохраняется положение, которое объект имел до формирования иерархической связи.

Теперь система частиц двигается вместе со сферой, но пламя при этом не отклоняется, что выглядит неестественно. Это связано с тем, что по умолчанию частицы перемещаются корректно только в локальном пространстве собственной системы. Для завершения модели горящей сферы найдите в настройках системы частиц параметр Simulation Space (он находится в верхнем свитке, показанном на рис. 4.20) и измените его значение с Local на World.

ПРИМЕЧАНИЕ Наш объект перемещается по прямой, но обычно в играх используются более замысловатые траектории. В Unity поддерживаются сложные варианты навигации и пути; прочитать об этом можно на странице https://docs.unity3d.com/ru/current/Manual/Navigation.html.

Уверен, что вам не терпится реализовать собственные идеи и добавить в игру новые объекты. Займитесь этим сейчас — можете создать больше графических ресурсов или проверить свои способности, добавив в сцену разработанный в главе 3 механизм стрельбы. В следующей главе мы перейдем к другому игровому жанру и начнем создание новой игры. Впрочем, несмотря на подобные переходы, вся информация из первых четырех глав вполне применима и может оказаться вам полезной.

4.7. Заключение

- Термин «графические ресурсы» означает совокупность всей задействованной в проекте графики.
- Создание графической модели сцены является для проектировщика уровней первым шагом по разметке игрового пространства.
- Текстуры и двухмерные изображения отображаются на поверхности трехмерных моделей.
- Трехмерные модели создаются вне Unity и импортируются в виде FBX-файлов.
- Системы частиц позволяют создавать многочисленные визуальные эффекты (огонь, дым, воду и т. п.).

Часть II

ЖИЗНЬ НАЛАЖИВАЕТСЯ

Вы создали в Unity первый прототип игры и готовы расширить свои навыки на примерах других игровых жанров. К настоящему моменту вы должны полностью освоить приемы работы в Unity: создание сценариев с указанными функциями, перетаскивание объектов на ячейки панели Inspector и тому подобное. Я больше не буду подробно останавливаться на деталях интерфейса, так что в остальных главах вы не найдете повторения основ.

Давайте создадим еще несколько дополнительных проектов, которые научат вас множеству новых приемов разработки игр в Unity.

5 Игра Memory на основе новой 2D-функциональности

- ✓ Отображение двухмерной графики в Unity
- ✓ Создание интерактивных объектов
- ✓ Программная загрузка новых изображений
- ✓ Поддержка и отображение состояния с помощью текстового пользовательского интерфейса
- ✓ Загрузка игровых уровней и перезапуск игры

Пока вы работали только с трехмерной графикой. Но в Unity можно использовать и двухмерную графику, и в этой главе вы создадите двухмерную игру, чтобы понять, как это делается. Это будет классическая детская игра Memory: выкладываются карты рубашкой вверх, по щелчку переворачиваются, после чего считается количество совпадений. В процессе вы познакомитесь с основами разработки двухмерных игр в Unity.

Хотя изначально система Unity появилась как инструмент создания трехмерных игр, применяют ее не только для этого. В последние версии Unity (начиная с версии 4.3, появившейся в конце 2013 года) добавлена возможность отображения двухмерной графики, но и раньше с помощью этого инструмента разрабатывали двухмерные игры (особенно мобильные, в создании которых помогла кроссплатформенная природа Unity). В предыдущих версиях Unity для эмуляции двухмерной графики в трехмерных сценах разработчикам требовался сторонний фреймворк (например, 2D Toolkit от Unikron Software). В конечном счете основной редактор и игровой движок поменяли, встроив в него механизм поддержки двухмерной графики. Именно с этой новой функциональностью вы и познакомитесь в данной главе.

Рабочий процесс при создании двухмерной и трехмерной графики в Unity примерно одинаков и включает в себя импорт графических ресурсов, перетаскивание их в сцену

и написание сценариев, которые затем присоединяются к объектам. Основной вид ресурсов, необходимых для создания двухмерной графики, называется спрайтом.

ОПРЕДЕЛЕНИЕ Спрайты (sprites) представляют собой отображаемые непосредственно на экране двухмерные изображения, в то время как такие же изображения, отображаемые на поверхности трехмерных моделей, называются текстурами.

Импорт спрайтов по большей части напоминает импорт текстур (см. главу 4). С технической точки зрения спрайты представляют собой объекты в трехмерном пространстве, обладающие плоской поверхностью и ориентированные относительно оси *Z*. Так как все они повернуты в одном направлении, камеру можно нацелить прямо на них, в результате игрокам будут видимы только перемещения вдоль осей *X* и *Y* (то есть в плоскости).

В главе 2 мы обсудили координатные оси: для получения третьего измерения добавляется ось *Z*, перпендикулярная уже знакомым вам осям *X* и *Y*. Именно этими осями представлены два измерения, с которыми мы будем работать.

5.1. Подготовка к работе с двухмерной графикой

Мы собираемся воспроизвести классическую игру Memory. Опишем ее для тех, кто не знаком с правилами. Набор парных карт раскладывается рубашкой вверх. Расположение карт игроку неизвестно. Он может перевернуть любые две карты, пытаясь найти совпадающие. Если открытые карты не совпали, они переворачиваются назад, а игрок делает следующую попытку.

На рис. 5.1 показан макет этой игры; сравните его с планом, который мы составляли в главе 2.

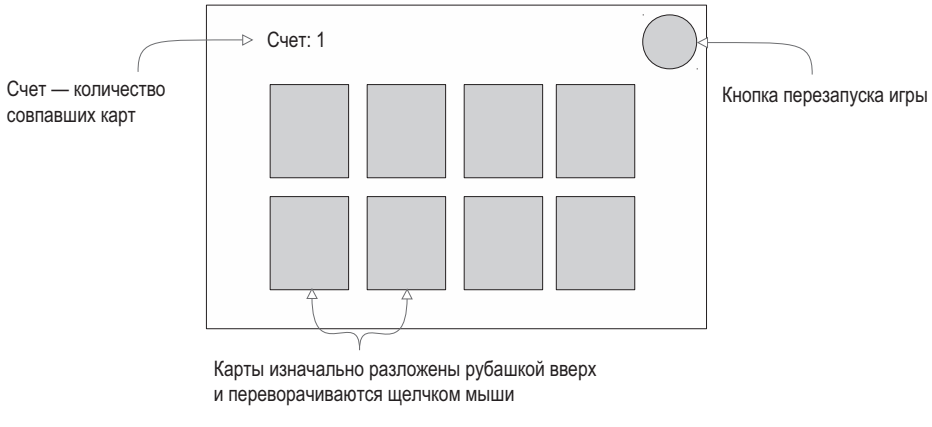


Рис. 5.1. Макет игры Memory

Обратите внимание, что на этот раз макет воспроизводит именно то, что увидит играющий (в то время как макет трехмерной сцены описывает пространство вокруг игрока и местоположение камеры, обеспечивающей возможность смотреть по сторонам).

Теперь, когда вы знаете, что мы собираемся создавать, пришло время приступить к делу!

5.1.1. Подготовка проекта

Первым делом нужно собрать и отобразить всю графику. Практически тем же способом, которым раньше создавался трехмерный демонстрационный ролик, мы начнем новую игру с помещения в сцену игровых объектов, после чего приступим к программированию их функциональности.

Это означает, что нужно создать все представленное на рис. 5.1: рубашку карт, их лицевую сторону, табло отображения количества набранных очков в одном углу игрового пространства и кнопку перезагрузки в другом. Кроме того, нам потребуется фон. Весь необходимый арсенал показан на рис. 5.2.

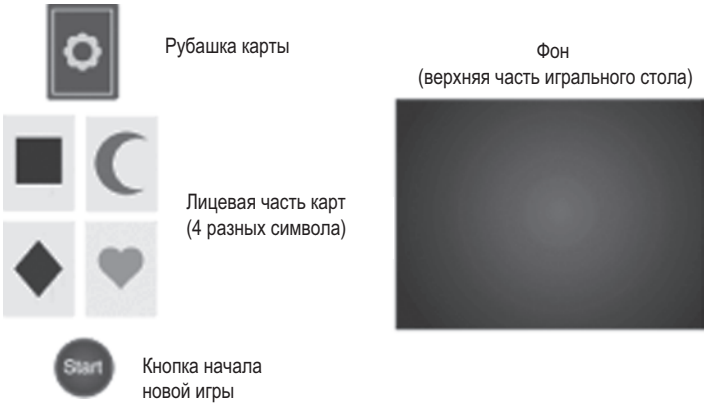


Рис. 5.2. Графические ресурсы для игры Memory

СОВЕТ Как обычно, готовую версию проекта со всеми графическими ресурсами можно скачать со страницы www.manning.com/hocking сайта, посвященного этой книге. Вы можете скопировать предоставленные изображения и использовать их в своем проекте.

Соберите все требуемые изображения и создайте в Unity новый проект. В нижней части появившегося окна вы увидите пару кнопок, позволяющих переключаться между режимами 2D и 3D (рис. 5.3). В предыдущих главах вы работали только с трехмерной графикой, а так как именно этот режим создания проектов используется по умолчанию, мы не обращали на данную настройку внимания. Сейчас же нам нужно переключиться в режим работы с двухмерной графикой.

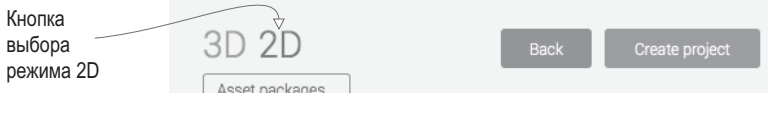
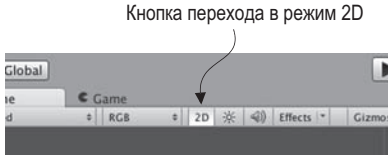


Рис. 5.3. Эти кнопки позволяют создавать как двухмерные, так и трехмерные проекты

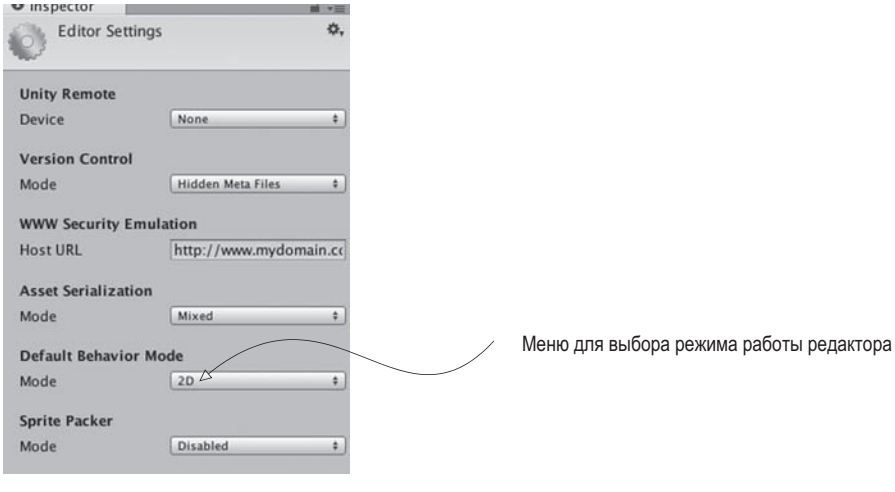
РЕЖИМ 2D ДЛЯ РЕДАКТОРА И ВКЛАДКИ SCENE

Кнопка 2D/3D при создании нового проекта меняет настройки редактора Unity, которые при желании можно поменять вручную. Это режим работы самого редактора и режим отображения вкладки Scene. Последний определяет способ показа сцены в Unity; кнопка перехода между режимами находится в верхней части вкладки Scene.



Кнопка, меняющая режим отображения на вкладке Scene

Для изменения режима работы редактора откройте меню Edit, наведите указатель мыши на строку Project Settings и выберите в дополнительном меню вариант Editor. Среди различных настроек на панели Inspector вы увидите раскрывающийся список Default Behavior Mode, в котором можно выбрать вариант 3D или 2D.



Список Default Behavior Mode, открываемый командой Edit ▸ Project Settings ▸ Editor

В режиме 2D все изображения импортируются как спрайты; в главе 4 вы видели, что в обычном режиме они превращаются в текстуры. Кроме того, в режиме 2D сцены лишены настроек освещения. Оно в данном случае просто не требуется. Если когда-нибудь вы захотите удалить освещение из трехмерной сцены, удалите автоматически создаваемый источник света и отключите скайбокс (щелкните на маленьком кружочке рядом с полем выбора файла и выберите в списке вариант None).

Итак, мы создали новый проект, выбрав для него режим 2D. Пришло время добавить в сцену наши изображения.

5.1.2. Отображение двухмерных изображений (спрайтов)

Выполните импорт всех изображений, перетащив их на вкладку Project; убедитесь, что изображения импортированы как спрайты, а не как текстуры. Для этого выделите любую картинку и посмотрите настройки ее импорта на панели Inspector. Нас интересует поле Texture Type. Теперь перетащите спрайт table_top (наше фоновое изображение) с вкладки Project в пустую сцену. Сохраните сцену. Как и в случае с сеточными объектами, на панели Inspector окажется компонент Transform для данного спрайта; присвойте его полям значения 0, 0, 5 для изменения положения фонового изображения.

СОВЕТ Есть еще одна настройка импорта, на которую следует обратить внимание, — это Pixels To Units. Так как в Unity раньше был трехмерный движок, к которому недавно добавили двухмерную графику, одна единица в Unity далеко не всегда соответствует одному пикселу изображения. Вы можете выбрать для этой настройки вариант 1:1, но лично я рекомендую оставить заданное по умолчанию значение 100:1 (так как физический движок при отображении варианта 1:1 работает не совсем корректно, а вариант, предлагаемый по умолчанию, обеспечивает лучшую совместимость с другим кодом).

АНИМИРОВАННЫЕ СПРАЙТЫ

Для данного проекта мы будем использовать только статичные изображения, но в двухмерных играх повсеместно распространены анимированные спрайты. Они создаются рисованием каждого кадра анимации и последующего воспроизведения полученной последовательности кадров в Unity.

Кадры можно импортировать в виде отдельных изображений, но обычно их выкладывают в виде одной картинки, которая называется листом спрайтов (sprite sheet). Листы спрайтов можно автоматически генерировать в Unity или пользоваться для их создания такими инструментами, как Texture Packer (см. приложение Б). При импорте листа в списке Sprite Mode настроек спрайта следует выбрать вариант Multiple.

Импорт листа спрайтов в Unity

Лист спрайтов появляется на вкладке Project как один ресурс, но щелчок на расположенной сбоку стрелке раскрывает его и дает возможность увидеть отдельные спрайты. Вместо того чтобы перетаскивать спрайты в сцену по одному за раз, можно перетащить их группой.

Понять, почему мы приравняли к 0 координаты *X* и *Y*, несложно — наш спрайт должен заполнить весь экран, поэтому его нужно расположить в центре, но присвоение значения 5 координате *Z* может показаться странным. Разве для двухмерной графики имеют значение какие-либо координаты, кроме *X* и *Y*? Разумеется, только эти координаты определяют положение плоского объекта на экране; а координата *Z* вступает в дело, если нам нужно положить объекты друг на друга. Меньшие значения координаты *Z* заставляют спрайты располагаться ближе к камере, поэтому спрайты с такой координатой оказываются поверх прочих объектов (рис. 5.4). Соответственно, у спрайта, служащего фоном, значение координаты *Z* должно быть максимальным. Мы присвоили ему положительную координату *Z*, а все остальные объекты будут иметь нулевую или отрицательную координату *Z*.

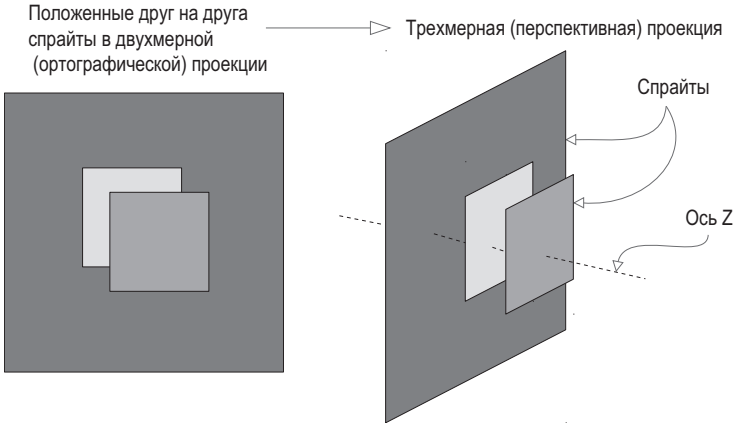


Рис. 5.4. Расположение спрайтов относительно оси Z

Положение остальных спрайтов будет указываться с точностью до двух знаков после запятой из-за упоминавшегося ранее параметра Pixels To Units. Соотношение 100:1 означает, что 100 пикселов изображения соответствуют 1 измерительной единице в Unity; или, если посмотреть с другой стороны, 1 пиксел равен 0,01 единицы. Однако перед размещением в сцене дополнительных спрайтов настроим камеру.

СОЗДАНИЕ АТЛАСОВ

Как упоминалось во врезке «Анимированные спрайты», набор спрайтов можно превратить в одно изображение. В случае, когда таким способом комбинируются кадры двухмерной анимации, такой набор называется листом спрайтов. Но для подобной структуры существует еще один, более общий термин — атлас (atlas).

Листы спрайтов используют для хранения кадров анимации, атласы же зачастую применяются для статичных изображений. Дело в том, что они двумя способами оптимизируют производительность: во-первых, уменьшая количество пустого пространства в изображениях путем их плотной упаковки; во-вторых, уменьшая количество вызовов прорисовки (draw calls) видеокарты (каждое новое загруженное изображение означает дополнительную работу видеокарты).

Для создания атласов спрайтов применяются внешние инструменты (после импорта в настройках спрайта нужно выбрать вариант Multiple), и это прекрасно работает. Но в Unity существует упаков-

щик спрайтов, автоматически соединяющий друг с другом наборы изображений. По умолчанию он отключен и включается в настройках редактора, для доступа к которым нужно выбрать в меню Edit команду Project Settings и затем — Editor. После этого следует указать имя в поле Packing Tag, находящемся среди настроек импорта спрайта; Unity упакует помеченные одним тегом спрайты в атлас. Более подробную информацию вы найдете по адресу http://docs.unity3d.com/ru/current/Manual/SpritePacker.html.

5.1.3. Переключение камеры в режим 2D

Отредактируем настройки основной камеры в сцене. Не думайте, что благодаря переключению вкладки Scene в режим работы с двухмерной графикой вы в процессе игры будете видеть то же самое, что видите в Unity. К сожалению, этот аспект не очевиден.

ВНИМАНИЕ Режим вкладки Scene не имеет никакого отношения к тому, что показывает камера запущенной игры.

Дело в том, что какой бы режим ни был выбран для вкладки Scene, игровая камера настраивается сама по себе. Такое положение вещей полезно во многих ситуациях, позволяя, например, вернуть вкладку Scene в режим 3D для работы над какими-то эффектами. Но одновременно оно означает, что вы видите в Unity вовсе не то же самое, что вы можете увидеть в игре, и новички об этом часто забывают.

Самая важная настройка камеры называется Projection (проекция). Скорее всего, проекция нашей камеры уже именно та, которая требуется, так как новый проект создавался в режиме 2D, но вы должны знать, как проверяются параметры камеры. Выделите камеру на вкладке Hierarchy, чтобы ее настройки появились на панели Inspector, как показано на рис. 5.5. Для работы с трехмерной графикой параметр Projection должен иметь значение Perspective, двухмерная же графика требует значения Orthographic.

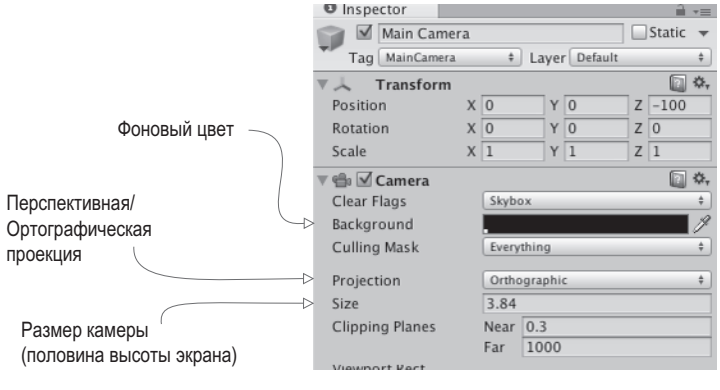


Рис. 5.5. Настройки камеры при работе с двухмерной графикой

ОПРЕДЕЛЕНИЕ Ортографической называется камера, у которой нет масштабного фактора для конечного изображения. Этим она отличается от перспективной камеры, в которой ближе расположенные объекты выглядят больше и уменьшаются с увеличением расстояния до камеры.

Впрочем, важный параметр Projection в редактировании не нуждается, чего не скажешь о прочих настройках камеры. Обратите внимание на параметр Size; он определяет размер поля зрения камеры от центра экрана до его верхнего края. Другими словами, этому параметру присваивается значение, равное половине желаемого размера экрана в пикселах. Если позднее вы укажете такое же разрешение для развертываемой игры, вы получите графику *пиксел в пиксел*.

ОПРЕДЕЛЕНИЕ Пиксел в пиксел (pixel-perfect) означает, что один экранный пиксел соответствует одному пикселу изображения (в противном случае видеокарта может сделать изображение слегка размытым, растянув его под размер экрана).

Предположим, что вы хотите попасть пиксел в пиксел при разрешении экрана 1024×768. Значит, высота камеры должна составить 384 пиксела. Поделите это число на 100 (с учетом соотношения пикселов и единиц измерения), и вы получите размер камеры, равный 3.84. То есть вычисления ведутся по формуле РАЗМЕР ЭКРАНА / 2 /100*f* (*f* указывает на значение типа `float`, а не типа `int`). Так как наше фоновое изображение имеет размер 1024 × 768 (для проверки выделите этот ресурс), очевидно, что значение 3.84 идеально подходит для нашей камеры.

Еще два параметра, которые нужно поменять на панели Inspector, — это фоновый цвет камеры и ее положение по координате *Z*. Как уже упоминалось при рассмотрении спрайтов, чем больше значение координаты *Z*, тем дальше расположен объект. Поэтому отодвинем камеру подальше в область отрицательных значений *Z*; например, 0, 0, –100. В качестве фонового цвета, наверное, лучше всего выбрать черный; по умолчанию фон имеет синий цвет, и если экран окажется шире фонового изображения (что вполне вероятно), синие полоски по сторонам будут выглядеть странно. Щелкните на образце цвета Background и выберите черный цвет.

Сохраните сцену под именем Scene и щелкните на кнопке Play — вы увидите игровое поле, заполненное спрайтом, изображающим поверхность стола. Думаю, вы согласитесь, что путь до этой точки был не совсем очевидным (еще раз напомню, что причиной этого является трехмерный игровой движок Unity, в который только недавно добавили возможность работы с двухмерной графикой). Итак, у нас есть пустая поверхность стола, теперь нужно выложить на нее карты.

5.2. Создание карт и превращение их в интерактивные объекты

Так как все изображения уже импортированы и готовы к использованию, давайте создадим объекты, представляющие карты, которые сформируют ядро нашей игры. В игре Memory карты выкладываются лицевой стороной вниз и переворачиваются только на время после выбора пары карт. Для реализации такой функциональности мы создадим объекты, состоящие из положенных друг на друга спрайтов. Затем мы напишем код, который заставит карты переворачиваться по щелчку мыши.

5.2.1. Создание объекта из спрайтов

Перетащите в сцену одну из карт. Используйте изображение лицевой стороны, так как мы добавим поверх другую карту, чтобы скрыть рисунок. С технической точки зрения положение карты пока не имеет значения, но в конечном счете она должна будет оказаться в определенной точке, поэтому присвойте полям Position значения –3, 1, 0. Затем перетащите в сцену спрайт card_back. Сделайте этот новый спрайт потомком предыдущего (напоминаю, что для этого на вкладке Hierarchy нужно перетащить дочерний объект на родительский) и установите для него положение 0, 0, –0.1. Помните, что это положение относительно предка, то есть данные координаты означают «Поместите рубашку карты в точку с теми же координатами *X* и *Y*, но ее чуть ближе по координате *Z*».

СОВЕТ Вместо инструментов Move, Rotate и Scale, которыми мы пользовались в режиме 3D, теперь нам доступен единый инструмент манипуляции Rect Tool. В режиме 2D этот инструмент активируется автоматически. Кроме того, вы можете нажать крайнюю правую кнопку на навигационной панели, расположенной в верхнем левом углу Unity. Когда этот инструмент активен, перетаскивание объектов приводит к выполнению всех трех операций (перемещения/поворота/масштабирования) в двух измерениях.

Поместив рубашку карты на место, как показано на рис. 5.6, вы подготовили все для создания интерактивного объекта, реагирующего на щелчок мыши.

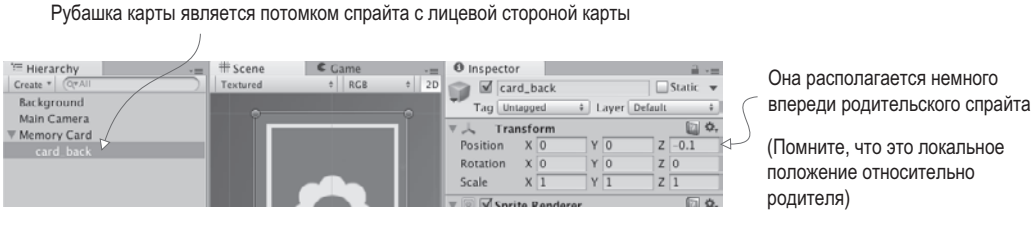


Рис. 5.6. Иерархическое связывание и расположение спрайта с рубашкой карты

5.2.2. Код ввода с помощью мыши

Чтобы карта могла реагировать на щелчки мыши, ей требуется такой компонент, как коллайдер. У новых спрайтов он отсутствует, поэтому щелкать на них бесполезно. Мы присоединим коллайдер к корневому объекту, а не к рубашке карты, поэтому именно лицевая сторона будет восприимчива к щелчкам. Для этого выделите корневой объект на вкладке Hierarchy (не имеет смысла щелкать на карте в сцене, так как сверху находится рубашка, и именно она будет выделена), а затем щелкните на кнопке Add Component в нижней части панели Inspector. Выделите строку Physics 2D (строка Physics относится к физике трехмерных игр) и выберите вариант Box collider.

Кроме коллайдера карте нужен сценарий, который заставит ее реагировать на действия игрока, поэтому давайте напишем его код. Создайте новый сценарий с именем `MemoryCard.cs` и присоедините его к корневой карте (а не к рубашке). Код

в следующем листинге заставляет карту после щелчка на ней сгенерировать отладочное сообщение.

Листинг 5.1. Генерация отладочных сообщений по щелчку

```
using UnityEngine;
using System.Collections;

public class MemoryCard : MonoBehaviour {
  public void OnMouseDown() { ← Эта функция вызывается после щелчка на объекте.
    Debug.Log("testing 1 2 3"); ← Пока мы только генерируем текстовое сообщение, выводимое на консоль.
  }
}
```

СОВЕТ Если вы еще не приобрели такой привычки, приучайте себя распределять ресурсы по папкам; создайте отдельную папку для сценариев и перетащите все сценарии туда непосредственно на вкладке Project. Главное — избегать имен, на которые реагирует Unity: Resources, Plugins, Editor и Gizmos. Позднее мы поговорим о том, какую функцию выполняют эти папки, а пока просто избегайте называть свои собственные папки перечисленными именами.

Итак, теперь мы можем щелкнуть на карте! Подобно методу `Update()`, функция `OnMouseDown()` также происходит от класса `MonoBehaviour`, именно она вызывает реакцию на щелчок мыши. Запустите игру и посмотрите, как на консоли появляется сообщение. Но вывод на консоль был сделан только с целью тестирования, мы же хотим открыть карту.

5.2.3. Открытие карты по щелчку

Перепишите код в соответствии со следующим листингом (пока запустить этот код не получится, но это не повод для беспокойства).

Листинг 5.2. Сценарий, скрывающий рубашку после щелчка на карте

```
using UnityEngine;
using System.Collections;
public class MemoryCard : MonoBehaviour {
  [SerializeField] private GameObject cardBack; ← Переменная, которая появляется на панели Inspector.
  public void OnMouseDown() {
    if (cardBack.activeSelf) { ← Просто запускаем код деактивации, если объект в данный момент является активным/видимым.
      cardBack.SetActive(false); ← Делаем объект неактивным/невидимым.
    }
  }
}
```

В этом сценарии есть два ключевых дополнения: ссылка на объект в сцене и метод `SetActive()`, деактивирующий данный объект. Первая часть, то есть ссылка на объект, аналогична тому, что мы делали в предыдущих главах: переменная помечается как сериализованная, а затем объект с вкладки Hierarchy перетаскивается на эту переменную на панели Inspector. После того как ссылка на объект установлена, код начинает влиять на объект сцены.

Вторым ключевым дополнением является команда `SetActive`. Она деактивирует любой объект `GameObject`, делая его невидимым. Если теперь перетащить

присутствующий в сцене объект card_back на переменную этого сценария на панели Inspector, в процессе игры карта начнет исчезать после щелчка на ней. Исчезновение рубашки откроет лицевую сторону карты; как видите, мы решили еще одну важную для построения игры Memory задачу! Но пока на стол выложена всего одна карта, давайте исправим этот недостаток.

5.3. Отображение различных карт

Мы написали программу, которая сначала отображает рубашку карты, а после щелчка на ней показывает лицевую сторону. Но это всего лишь одна карта, тогда как для игры требуется целый набор, по большей части с разными изображениями. Мы разложим карты, воспользовавшись как парой концепций из предыдущих глав, так и совершенно новыми пока для вас понятиями. В главе 3 вы научились, во-первых, применять невидимый компонент SceneController, во-вторых, создавать экземпляры объекта. На этот раз компонент SceneController позволит нам назначить различным картам разные изображения.

5.3.1. Программная загрузка изображений

В создаваемой нами игре есть четыре варианта изображений для карт. Все восемь лежащих на столе карт (по две для каждого символа) получаются клонированием одного и того же оригинала, поэтому изначально на всех картах будет один рисунок. Менять его нам придется при помощи сценария, загружая различные изображения программно.

Чтобы посмотреть на процесс программной загрузки изображений, напишем простой тестовый код (который позже будет заменен). Сначала добавьте код следующего листинга в сценарий `MemoryCard`.

Листинг 5.3. Тестовый код, демонстрирующий изменение картинки спрайта

```
...
[SerializeField] private Sprite image; ← Ссылка на загружаемый ресурс Sprite
void Start() {
  GetComponent<SpriteRenderer>().sprite = image; ← Сопоставляем этот спрайт компоненту SpriteRenderer.
}
...
```

После сохранения этого сценария на панели Inspector появилась новая переменная, так как она была указана как сериализованная. Перетащите какой-либо спрайт со вкладки Project (выберите одно из лицевых изображений, кроме того, которое уже есть в сцене) и перетащите на ячейку Image. Затем выполните воспроизведение сцены, и вы увидите на карте новое изображение.

Ключом к пониманию этого кода являются сведения о компоненте SpriteRenderer. Обратите внимание, что на рис. 5.7 у рубашки карты всего два компонента: стандартный компонент Transform, присутствующий у всех объектов сцены, и новый компонент SpriteRenderer. Именно он превращает объект в спрайт и определяет, какой ресурс будет на нем отображаться. Обратите внимание, что первое свойство этого компонента называется Sprite и связано с одним из спрайтов на вкладке Project; этим свойством можно управлять в коде, чем, собственно, и занимается наш сценарий.

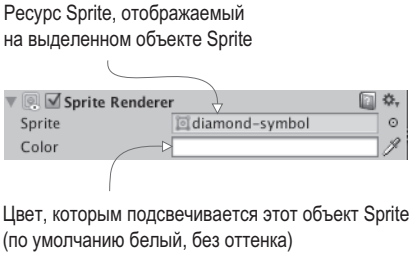


Рис. 5.7. Компонент SpriteRenderer, присоединенный к объекту-спрайту в сцене

Как вы видели в предыдущих главах на примере компонента CharacterController и написанных нами сценариев, метод `GetComponent()` возвращает другие компоненты для того же самого объекта, поэтому мы воспользуемся им для ссылки на объект SpriteRenderer. Свойству Sprite объекта SpriteRenderer можно присвоить любой ресурс, поэтому данный код присваивает указанному свойству объявленную в верхней части переменную `Sprite` (которую мы в редакторе заполнили одним из спрайтов).

Как видите, это не слишком сложно! Но это всего одно изображение, а у нас их четыре. Поэтому удалите новый фрагмент кода из листинга 5.3 (он был нужен только для демонстрации), чтобы подготовиться к следующему разделу.

5.3.2. Выбор изображения в невидимом компоненте SceneController

Вспомните, как в главе 3 мы создавали в сцене невидимый объект для управления процессом генерации объектов. Мы снова воспользуемся этим приемом, но на этот раз невидимый объект будет контролировать более абстрактные вещи, не связанные ни с одним из объектов сцены. Для начала создайте пустой объект GameObject (напоминаю, что нужно выбрать в меню GameObject команду Create Empty). Затем создайте на вкладке Project сценарий `SceneController.cs` и перетащите этот ресурс на контроллер GameObject. Перед тем как писать код нового сценария, добавьте содержимое следующего листинга в сценарий `MemoryCard` вместо того, что было в листинге 5.3.

Листинг 5.4. Новые открытые методы в сценарии MemoryCard.cs

```
...
[SerializeField] private SceneController controller;

private int _id;
public int id {
  get {return _id;} ← Добавление функции чтения (идиома, распространенная в таких языках, как C# и Java).
}

                                                    Открытый метод, которым могут пользоваться
                                                    другие сценарии для передачи указанному
public void SetCard(int id, Sprite image) { ←      объекту новых спрайтов.
  _id = id;
  GetComponent<SpriteRenderer>().sprite = image; ← Точно такая же строка SpriteRenderer,
}                                                   как в удаленном примере кода.
...
```

Основным отличием от предыдущего листинга является задание изображения спрайта методом `SetCard()` вместо метода `Start()`. Так как это открытый метод, принимающий спрайт в качестве параметра, его можно вызывать из других сценариев и устанавливать изображение для данного объекта. Обратите внимание, что метод `SetCard()` также принимает в качестве параметра значение `ID` и код его сохраняет. Хотя в настоящий момент этот параметр нам не требуется, скоро мы напишем код, сравнивающий карты, и это сравнение будет производиться именно на основе значения `ID`.

ПРИМЕЧАНИЕ Возможно, раньше вы пользовались языком программирования, в котором отсутствовали такие понятия, как метод чтения (getter) и метод задания (setter). Коротко говоря, это функции, которые запускаются, когда вы пытаетесь получить доступ к связанному с ними свойству (например, получить значение card.id). Эти методы применяются для разных целей, но в нашем случае свойство id предназначено только для чтения, поэтому вы можете только прочитать его значение, но не задать его.

Наконец, обратите внимание, что в этом коде присутствует переменная для контроллера; в то время как SceneController начинает клонировать карты для заполнения сцены, картам требуется ссылка на этот контроллер для вызова его открытых методов. Как обычно, когда код ссылается на объекты сцены, перетащите объект-контроллер из редактора Unity на ячейку переменной на панели Inspector. Достаточно сделать это для одной карты, а все появившиеся позже копии получат эту ссылку автоматически.

Теперь, когда мы добавили новый код в сценарий `MemoryCard`, введите код следующего листинга в сценарий `SceneController`.

Листинг 5.5. Первый проход компонента SceneController в игре Memory

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour {
  [SerializeField] private MemoryCard originalCard; ← Ссылка для карты в сцене.
  [SerializeField] private Sprite[] images; ← Массив для ссылок на ресурсы-спрайты.

  void Start() {
    int id = Random.Range(0, images.Length);
    originalCard.SetCard(id, images[id]); ← Вызов открытого метода, который
                                            мы добавили в сценарий MemoryCard.
  }
}
```

Это короткий фрагмент, демонстрирующий принцип управления картами контроллером SceneController. Большая часть представленного кода уже должна быть вам знакома (например, перетаскивание объекта-карты в редакторе Unity на ячейку переменной на панели Inspector), но с массивом изображений вы раньше не сталкивались. Как показано на рис. 5.8, на панели Inspector можно задать набор элементов. Введите 4 в поле размера массива, а затем перетащите спрайты с изображениями карт на ячейки элементов массива. Эти спрайты станут доступны в массиве, как и все остальные ссылки на объекты.

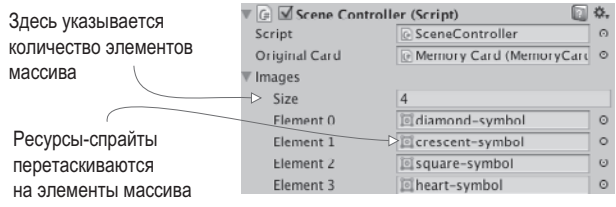


Рис. 5.8. Заполнение массива спрайтов

Вообще говоря, мы пользовались методом `Random.Range()` в главе 3, надеюсь, вы это помните. Точные граничные значения в данном случае не важны, но имейте в виду, что минимальное значение входит в диапазон и может быть возвращено, в то время как возвращаемое значение всегда меньше максимального.

Щелкните на кнопке Play для воспроизведения нового кода. Вы увидите, что при каждом запуске сцены изображение на открываемой карте меняется. Теперь нужно разложить по столу все остальные карты.

5.3.3. Создание экземпляров карт

У компонента SceneController уже есть ссылка на объект-карту, поэтому нам остается воспользоваться методом `Instantiate()` (см. следующий листинг) для получения набора карт, аналогично тому, как мы генерировали объекты в главе 3.

Листинг 5.6. Восьмикратное копирование карты и выкладывание карт на стол

```
using UnityEngine;
using System.Collections;

public class SceneController : MonoBehaviour {
  public const int gridRows = 2; ← Значения, указывающие количество ячеек сетки и их расстояние друг от друга.
  public const int gridCols = 4;
  public const float offsetX = 2f;
  public const float offsetY = 2.5f;

  [SerializeField] private MemoryCard originalCard;
  [SerializeField] private Sprite[] images;
  void Start() {
    Vector3 startPos = originalCard.transform.position; ← Положение первой карты; положение остальных карт отсчитывается от этой точки.

    for (int i = 0; i < gridCols; i++) {
      for (int j = 0; j < gridRows; j++) { ← Вложенные циклы, задающие как столбцы, так и строки нашей сетки.
        MemoryCard card; ← Ссылка на контейнер для исходной карты или ее копий.
        if (i == 0 && j == 0) {
          card = originalCard;
        } else {
          card = Instantiate(originalCard) as MemoryCard;
        }

        int id = Random.Range(0, images.Length);
        card.SetCard(id, images[id]);

        float posX = (offsetX * i) + startPos.x;
```

```
        float posY = -(offsetY * j) + startPos.y;
        card.transform.position = new Vector3(posX, posY, startPos.z); ← В двухмерной
графике нам нужно только смещение по осям X и Y; значение Z не меняется.
      }
    }
  }
}
```

Хотя этот сценарий намного длиннее предыдущих, объяснять тут особо нечего, потому что большинство дополнений представляют собой обычные объявления переменных и математические вычисления. Самым странным фрагментом кода является, вероятно, выражение, начинающее условную инструкцию `if (i == 0 && j == 0)`. Это выражение заставляет либо выбрать исходный объект-карту для первой ячейки нашей сетки, либо клонировать этот объект для остальных ячеек. Так как исходная карта в сцене уже есть, ее копирование на каждом шаге цикла приведет к появлению слишком большого количества карт. Карты раскладываются путем смещения в соответствии с количеством итераций цикла.

СОВЕТ Двухмерные объекты, как и трехмерные, можно двигать в различные места экрана, меняя значение transform.position, причем это значение может все время увеличиваться внутри метода Update(). Но как вы видели в процессе перемещения объекта-игрока, прямое редактирование параметра transform.position не позволяет добавить механизм распознавания столкновений. Аналогично, для перемещения двухмерных объектов с распознаванием столкновений нужно добавить к объекту компоненты группы Physics2D и редактировать, например, параметр rigidbody2D.velocity.

Запустите код, и в сцене появится сетка из восьми карт, показанная на рис. 5.9. Теперь нам осталось только разбить карты на пары.

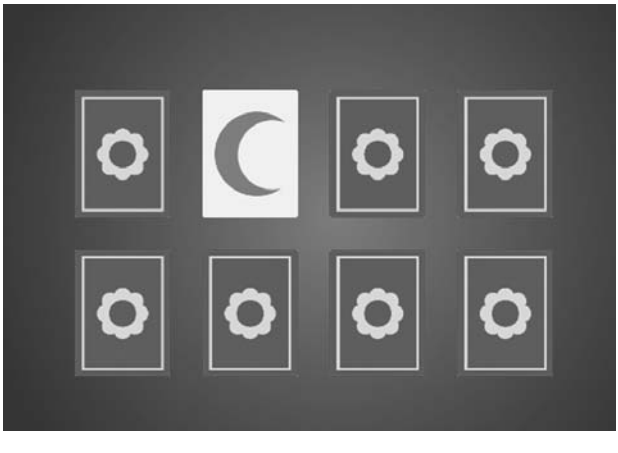


Рис. 5.9. Сетка из восьми карты, которые открываются щелчком мыши

5.3.4. Тасуем карты

Пока наши карты представляют собой не зависящие друг от друга объекты, но мы определим массив из их идентификаторов (чисел от 0 до 3, сопоставленных каждой

паре карт) и затем перемешаем элементы этого массива. Именно он послужит нам основой при выкладке карт. Код представлен в следующем листинге.

Листинг 5.7. Раскладка карт из перемешанного списка

```
...
void Start() { ← Большая часть листинга – это контекст, показывающий, куда вставлять дополнения.
  Vector3 startPos = originalCard.transform.position;

  int[] numbers = {0, 0, 1, 1, 2, 2, 3, 3}; ← Объявляем целочисленный массив с парами идентификаторов для всех четырех спрайтов с изображениями карт.

  numbers = ShuffleArray(numbers); ← Вызов функции, перемешивающей элементы массива.

  for (int i = 0; i < gridCols; i++) {
    for (int j = 0; j < gridRows; j++) {
      MemoryCard card;
      if (i == 0 && j == 0) {
        card = originalCard;
      } else {
        card = Instantiate(originalCard) as MemoryCard;
      }

      int index = j * gridCols + i;
      int id = numbers[index]; ← Получаем идентификаторы из перемешанного списка, а не из случайных чисел.
      card.SetCard(id, images[id]);

      float posX = (offsetX * i) + startPos.x;
      float posY = -(offsetY * j) + startPos.y;
      card.transform.position = new Vector3(posX, posY, startPos.z);
    }
  }
}

private int[] ShuffleArray(int[] numbers) { ← Реализация алгоритма тасования Кнута.
  int[] newArray = numbers.Clone() as int[];
  for (int i = 0; i < newArray.Length; i++ ) {
    int tmp = newArray[i];
    int r = Random.Range(i, newArray.Length);
    newArray[i] = newArray[r];
    newArray[r] = tmp;
  }
  return newArray;
}
...
```

Теперь после щелчка на кнопке Play сетка будет заполнена картами из перемешанного набора, в котором присутствует ровно по две карты каждого вида. Массив карт пропущен через алгоритм тасования Кнута (еще его называют алгоритмом Фишера—Йетса), который предлагает простой, но эффективный способ перемешать элементы массива. Он в цикле просматривает элементы массива и меняет местами каждый элемент с другим элементом, выбранным случайным образом.

Можно щелчками мыши открыть все карты, но в игре Memory карты должны открываться парами, поэтому нам нужен дополнительный код.

5.4. Совпадения и подсчет очков

Для создания полнофункциональной игры Memory нам не хватает проверки совпадений. Хотя у нас есть выложенные на стол карты, открывающиеся при щелчке на них, они никак не связаны друг с другом. Нам же нужно, чтобы каждая открытая пара проверялась на совпадение.

Эта абстрактная логическая схема — проверка совпадений и соответствующая реакция — требует, чтобы карты в ответ на щелчок уведомляли сценарий `SceneController`. Для этого в данный сценарий следует добавить код следующего листинга.

Листинг 5.8. Сценарий SceneController, следящий за открываемыми картами

```
...
private MemoryCard _firstRevealed;
private MemoryCard _secondRevealed;

public bool canReveal {
  get {return _secondRevealed == null;} ← Функция чтения, которая возвращает значение false, если вторая карта уже открыта.
}
...
public void CardRevealed(MemoryCard card) {
  // изначально пусто
}
...
```

Метод `CardRevealed()` мы сейчас напишем; просто пока нам нужен вспомогательный пустой метод, на который можно будет ссылаться в сценарии `MemoryCard.cs` без появления ошибок компиляции. Обратите внимание, что мы снова применяем функцию чтения, на этот раз чтобы определить, открыта ли вторая карта; возможность открыть карту дается игроку только при отсутствии в сцене двух открытых карт.

Теперь нам снова нужно отредактировать сценарий `MemoryCard.cs`, добавив туда вызов метода (пока пустого), который будет информировать компонент SceneController о том, что игрок щелкнул на карте. Внесите в код сценария `MemoryCard.cs` изменения в соответствии со следующим листингом.

Листинг 5.9. Отредактированный сценарий MemoryCard.cs для открытия карт

```
...
public void OnMouseDown() {
  if (cardBack.activeSelf && controller.canReveal) { ← Проверка свойства canReveal контроллера, позволяющая гарантировать, что одновременно могут быть открыты всего две карты.
    cardBack.SetActive(false);
    controller.CardRevealed(this); ← Уведомление контроллера при открытии этой карты.
  }
}

Открытый метод, позволяющий компоненту SceneController снова скрыть карту (вернув на место спрайт card_back).
public void Unreveal() { ←
  cardBack.SetActive(true);
}
...
```

Если поместить в метод `CardRevealed()` инструкцию отладки для проверки взаимодействия объектов, можно увидеть, что после щелчка на любой карте появляется тестовое сообщение. Но мы сначала обработаем одну открытую карту.

5.4.1. Сохранение и сравнение открытых карт

Объект-карта был передан в метод `CardRevealed()`, поэтому мы начнем следить за открываемыми картами. Введите код следующего листинга.

Листинг 5.10. Слежение за открываемыми картами в сценарии SceneController

```
...
public void CardRevealed(MemoryCard card) {
  if (_firstRevealed == null) {
    _firstRevealed = card;
  } else {
    _secondRevealed = card;
    Debug.Log("Match? " + (_firstRevealed.id == _secondRevealed.id));
  }
}
...
```

← Сохранение карт в одной из двух переменных в зависимости от того, какая из них свободна.

← Сравнение идентификаторов двух открытых карт.

Этот код сохраняет открытые карты в одной из двух переменных. Если первая переменная свободна, заполняется именно она; если ей уже присвоен другой объект-карта, заполняется вторая переменная, а затем проверяется, совпадают ли идентификаторы карт. В зависимости от результата инструкция отладки выводит на консоль значение `true` или `false`.

Пока наш код никак не реагирует на совпадения, он только проверяет их. Поэтому давайте запрограммируем ответ.

5.4.2. Скрытие несовпадающих карт

Мы снова воспользуемся сопрограммой, так как реакция на отсутствие совпадения должна включать в себя остановку выполнения, дающую игроку возможность рассмотреть открытые карты. Сопрограммы детально рассматривались в главе 3; коротко говоря, именно сопрограмма позволит нам сделать паузу в процессе проверки совпадения. Следующий листинг содержит код, который следует добавить в сценарий `SceneController`.

Листинг 5.11. Сценарий SceneController, подсчитывающий очки или скрывающий карты при отсутствии совпадения

```
...
private int _score = 0;
...
public void CardRevealed(MemoryCard card) {
  if (_firstRevealed == null) {
    _firstRevealed = card;
  } else {
    _secondRevealed = card;
    StartCoroutine(CheckMatch());
  }
}

private IEnumerator CheckMatch() {
  if (_firstRevealed.id == _secondRevealed.id) {
    _score++;
```

← Еще одна переменная, добавленная в список в верхней части сценария SceneController.

← Единственная отредактированная строка в этой функции вызывает сопрограмму после открытия двух карт.

← Увеличиваем счет на единицу, если идентификаторы открытых карт совпадают.

```
    Debug.Log("Score: " + _score);
  }
  else {
    yield return new WaitForSeconds(.5f);

    _firstRevealed.Unreveal(); ← Закрытие несовпадающих карт.
    _secondRevealed.Unreveal();
  }

  _firstRevealed = null; ← Очистка переменных вне зависимости от того, было ли совпадение.
  _secondRevealed = null;
}
...
```

Первым делом мы добавляем переменную `_score`, предназначенную для слежения; затем после открытия второй карты загружаем в метод `CheckMatch()` сопрограмму. В ней есть два варианта кода — выбор варианта зависит от того, произошло ли совпадение. В случае совпадения сопрограмма не останавливает выполнение кода; команда `yield` просто пропускается. В противном же случае происходит остановка на полсекунды, а потом для обеих карт снова вызывается скрывающий их метод `Unreveal()`. В конце, вне зависимости от совпадения, переменные, в которых хранятся карты, обнуляются, давая игроку возможность открыть следующие карты.

В процессе игры несовпадающие карты будут некоторое время демонстрироваться игроку. Подсчет совпадений выводится в виде отладочных сообщений, нужно же сделать так, чтобы они выводились на экран.

5.4.3. Текстовое отображение счета

Отображение сведений для игрока является одной из причин добавления в игру пользовательского интерфейса (вторая причина состоит в необходимости предоставить игроку способ ввода информации; UI-кнопки мы обсудим в следующем разделе).

ОПРЕДЕЛЕНИЕ Употребляя аббревиатуру UI (User Interface — пользовательский интерфейс), обычно имеют в виду графический интерфейс пользователя (Graphical User Interface, GUI), который означает визуальную часть интерфейса, то есть текст и кнопки.

В Unity существуют разные способы отображения текста. Можно, к примеру, создать в сцене трехмерный текстовый объект. Это специальный сеточный компонент, поэтому нам нужен пустой объект, к которому его можно будет присоединить. Выберите в меню GameObject команду Create Empty. Затем щелкните на кнопке Add Component и выберите в разделе Mesh вариант Text Mesh.

ПРИМЕЧАНИЕ Может показаться, что трехмерный текст несовместим с двухмерной игрой, но не следует забывать, что с технической точки зрения мы работаем с трехмерной сценой, которая выглядит плоской, так как демонстрируется через ортографическую камеру. Это означает, что при желании мы можем добавлять в игру трехмерные объекты — просто они будут отображаться как плоские.

Поместите этот объект в точку с координатами –4.75, 3.65, –10; то есть сдвиньте его на 475 пикселов влево и на 365 пикселов вверх, расположив в верхнем левом углу стола

и приблизив к камере, чтобы он отображался поверх других игровых объектов. Найдите в нижней части панели Inspector параметр Font; щелкните на маленьком кружке, чтобы вызвать окно выбора файлов, и щелчком выделите единственный доступный шрифт Arial. В поле Text введите слово `Score:`. Корректное позиционирование требует, чтобы параметр Anchor имел значение Upper Left (он контролирует, каким образом растягиваются вводимые буквы), поэтому отредактируйте его, если требуется. По умолчанию вы получите размытый текст, но это легко исправить, введя параметры, представленные на рис. 5.10.

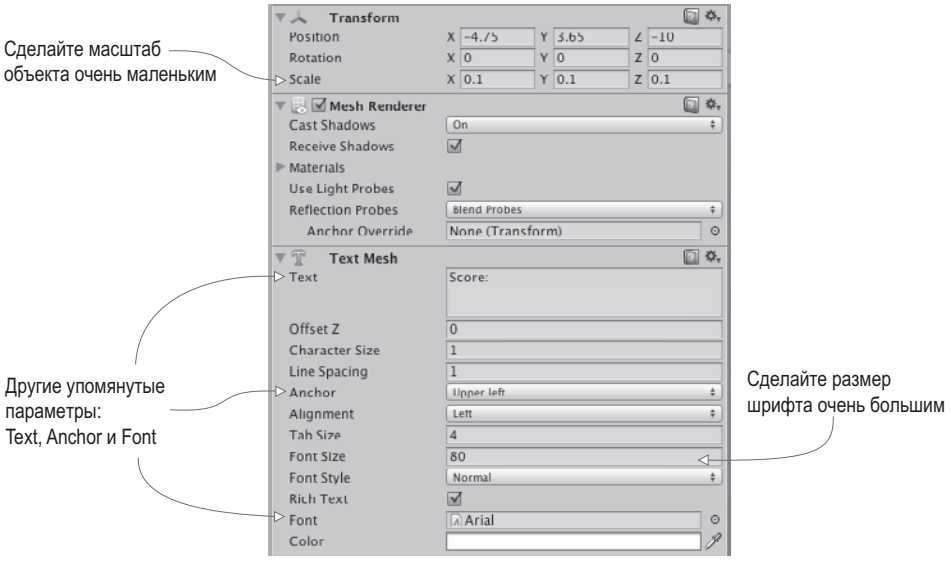


Рис. 5.10. Параметры текстового объекта на панели Inspector, позволяющие получить четкий и контрастный текст

Если импортировать в проект новый шрифт формата TrueType, можно будет им воспользоваться, но для наших целей вполне подойдет вариант, предлагаемый по умолчанию. Достаточно странно, что для получения четкой и контрастной надписи шрифтом, предлагаемым по умолчанию, требуется изменить его размер. Сначала мы присвоили параметру Font Size компонента Text Mesh очень большое значение (у меня это значение 80). Затем сделали масштаб этого компонента очень маленькими (например, 0.1, 0.1, 0.1). Увеличение размера шрифта добавило к отображаемому тексту множество пикселов, а масштабирование сгруппировало эти пикселы на меньшем пространстве. Для дальнейшего управления объектом нужно внести в код изменения, показанные в следующем листинге.

Листинг 5.12. Вывод счета при помощи текстового объекта

```
...
[SerializeField] private TextMesh scoreLabel;
...
private IEnumerator CheckMatch() {
```

```
  if (_firstRevealed.id == _secondRevealed.id) {
    _score++;
    scoreLabel.text = "Score: " + _score; ← Отображаемый текст – это задаваемое свойство текстовых объектов.
  }
...
```

Как видите, текст представляет собой свойство объекта, которому вы можете назначить новую строку. Перетащите расположенный в сцене текст на только что добавленную к компоненту SceneController переменную и щелкните на кнопке Play. После этого в процессе игры вы будете видеть, как при каждом следующем совпадении ваш счет увеличивается. Ура, наша игра работает!

5.5. Кнопка Restart

Наша игра Memory стала полнофункциональной. В нее уже можно играть. Отсутствует только один элемент, который рано или поздно потребуется любому игроку. Дело в том, что сейчас мы можем сыграть всего одну партию; после этого нужно будет выйти из игры и загрузить ее снова. Добавим на экран элемент управления, позволяющий игрокам начинать все заново без перезагрузки.

Для этого нужно решить две задачи: создать UI-кнопку и запрограммировать перезагрузку игры по щелчку на этой кнопке. После этого игра должна выглядеть так, как показано на рис. 5.11.

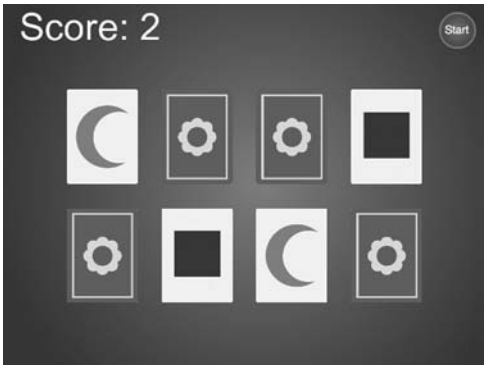


Рис. 5.11. Готовая игра Memory вместе с кнопкой Start

Кстати, ни одна из этих задач не относится к особенностям исключительно двухмерных игр; во всех играх требуется пользовательский интерфейс с кнопками, и все игры нуждаются в возможности перезагрузки. Именно решением этих задач мы закончим главу.

5.5.1. Добавление к компоненту UIButton метода SendMessage

Первым делом поместите в сцену спрайт с изображением кнопки, перетащив его с вкладки Project. Поместите его в точку с координатами 4.5, 3.25, –10; в результате изображение кнопки окажется в верхнем правом углу (то есть сместится на 450 пикселов

вправо и на 325 пикселов вверх) и подвинется ближе к камере, чтобы отобразиться поверх остальных игровых объектов. Так как нам нужна реакция этого объекта на щелчок мыши, добавьте к нему коллайдер (так же, как вы это делали с картами, щелкните на кнопке Add Component и выберите в разделе Physics 2D вариант Box Collider).

ПРИМЕЧАНИЕ Как уже было сказано в предыдущем разделе, Unity поддерживает разные механизмы создания UI-элементов, к которым относится и усовершенствованная система пользовательских интерфейсов, появившаяся в последних версиях Unity. Сейчас мы ограничимся созданием кнопки из стандартных экранных объектов. А вот следующая глава познакомит вас с усовершенствованной UI-функциональностью; мы изучим пользовательские интерфейсы для двухмерных и трехмерных игр, идеально встроенные в данную систему.

Создайте сценарий `UIButton.cs`, код которого показан в следующем листинге, и назначьте его объекту, изображающему кнопку.

Листинг 5.13. Создание кнопки многократного воспроизведения

```
using UnityEngine;
using System.Collections;

public class UIButton : MonoBehaviour {
  [SerializeField] private GameObject targetObject; ← Ссылка на целевой объект для информирования о щелчках.
  [SerializeField] private string targetMessage;
  public Color highlightColor = Color.cyan;

  public void OnMouseOver() {
    SpriteRenderer sprite = GetComponent<SpriteRenderer>();
    if (sprite != null) {
      sprite.color = highlightColor; ← Меняем цвет кнопки при наведении на нее указателя мыши.
    }
  }
  public void OnMouseExit() {
    SpriteRenderer sprite = GetComponent<SpriteRenderer>();
    if (sprite != null) {
      sprite.color = Color.white;
    }
  }

  public void OnMouseDown() {
    transform.localScale = new Vector3(1.1f, 1.1f, 1.1f); ← В момент щелчка размер кнопки слегка увеличивается.
  }
  public void OnMouseUp() {
    transform.localScale = Vector3.one;
    if (targetObject != null) {
      targetObject.SendMessage(targetMessage); ← Отправка сообщения целевому объекту в момент щелчка на кнопке.
    }
  }
}
```

Основная часть кода выполняется внутри набора функций вида `OnMouseЧтоТо`; подобно методам `Start()` и `Update()`, этот набор функций автоматически доступен всем компонентам сценариев в Unity. С функцией `MouseDown` вы уже встречались в разделе 5.2.2. Все эти функции в случае объектов, снабженных коллайдером, отвечают на

взаимодействия с мышью. Функции `MouseOver` и `MouseExit` представляют собой пару событий, возникающих при наведении на объект указателя мыши: первое — в момент, когда указатель впервые появляется над объектом, а второе — когда он убирается с объекта. Аналогично, функции `MouseDown` и `MouseUp` являются парой событий, возникающих в момент щелчка. Первое событие генерируется при физическом нажатии кнопки, второе — при ее отпускании.

Легко заметить, что при наведении указателя мыши на спрайт кнопки цвет этого спрайта слегка меняется, а в момент щелчка меняется еще и его размер. В обоих случаях вы видите, что эти изменения (цвета или масштаба) возникают после начала взаимодействия с мышью, затем свойство возвращается в исходное состояние (к белому цвету или единичному масштабу). Для масштабирования в коде используется стандартный компонент transform, присутствующий у всех объектов GameObject. А вот для изменения цвета применен компонент SpriteRenderer, присущий таким объектам, как спрайты; спрайту присваивается цвет, определенный в редакторе Unity при помощи общедоступной переменной.

В момент отпускания кнопки мыши происходит не только возвращение масштаба к единичному значению, но и вызов метода `SendMessage()`. Этот метод вызывает функцию с указанным именем во всех компонентах рассматриваемого объекта GameObject. Как целевой объект для рассылки, так и отправляемое сообщение определены через сериализованные переменные. Такой подход позволяет использовать один и тот же компонент UIButton для всех видов кнопок. Достаточно на панели Inspector сопоставить цели в виде различных кнопок разным объектам.

Обычно при написании программ на таком сильно типизированном языке, как C#, для взаимодействия с целевым объектом нужно знать его тип (к примеру, для вызова открытого метода объекта: `целевой-объект.SendMessage()`). Но сценарии для UI-элементов могут содержать множество различных типов целевых объектов, поэтому в Unity метод `SendMessage()` позволяет отправлять таким объектам заданные сообщения, даже если вы не знаете их тип.

ВНИМАНИЕ Метод SendMessage() менее рационально использует ресурсы процессора, чем открытые методы, вызываемые для известных типов (имеются в виду конструкции object. SendMessage(«Метод») вместо component.Method()), поэтому пользуйтесь этим методом только в случаях, когда это значительно упрощает понимание кода и работу с ним. Как правило, это ситуация, когда сообщение адресуется большому количеству объектов различных типов; в таких ситуациях отсутствие гибкости наследования или даже интерфейсов будет мешать процессу разработки игры и затруднять эксперименты.

Написав код, выполните связывание общедоступных переменных кнопки на панели Inspector. Выберите по своему вкусу цвет выделения (хотя заданный по умолчанию бирюзовый вполне подходит к синей кнопке). Кроме того, перетащите объект SceneController на ячейку целевого объекта и введите в поле для сообщения слово `Restart`.

Если теперь запустить игру, в верхнем правом углу игрового поля появится кнопка перезагрузки, меняющая цвет при наведении на нее указателя мыши и слегка выступающая в момент щелчка на ней. Но результатом щелчка станет сообщение об

ошибке; на консоли появится информация о том, что сообщение уходит «в никуда». Ведь метод `Restart()` в сценарии `SceneController` пока отсутствует. Давайте исправим этот недостаток.

5.5.2. Вызов метода LoadLevel в сценарии SceneController

Связанный с кнопкой метод `SendMessage()` пытается вызвать метод `Restart()` в сценарии `SceneController`, поэтому добавьте туда код следующего листинга.

Листинг 5.14. Код перезагрузки игрового уровня в сценарии SceneController

```
...
public void Restart() {
  Application.LoadLevel("Scene"); ← Эта команда загружает ресурс scene.
}
...
```

Как видите, единственным действием метода `Restart()` является вызов метода `Application.LoadLevel()`. Эта команда загружает такой ресурс, как сохраненная сцена (то есть файл, появившийся после команды Save Scene). Передайте в метод имя сцены, которую вы хотите загрузить; лично я сохранил ее под именем Scene, если же вы использовали другое имя, укажите его.

Щелкните на кнопке Play и посмотрите, что получилось. Откройте несколько карт и добейтесь нескольких совпадений; если после этого щелкнуть на кнопке Reset, игра начнется сначала, со скрытыми картами и нулевым счетом. Именно то, что и требовалось!

Как следует из имени метода `LoadLevel()`, он может загружать различные игровые уровни. Но что происходит в момент загрузки и почему игра при этом перезагружается? Просто все, что находилось на текущем уровне (все объекты сцены и присоединенные к ним сценарии), стирается из памяти, и загружается новая сцена. Так как в данном случае ее роль играет сцена, сохраненная перед началом игры, все удаляется из памяти и загружается с нуля.

СОВЕТ Можно пометить объекты, которые вы не хотите удалять из памяти при перезагрузке уровня. В Unity есть метод DontDestroyOnLoad(), сохраняющий объекты в разных сценах; вы увидите его как часть архитектуры кода в следующих главах.

Мы успешно создали еще одну игру! Разумеется, о «готовности» игры можно говорить только в относительном смысле — вы всегда можете добавлять новые функциональные возможности. Но все, что было намечено в начальном плане, мы реализовали. Многие концепции двухмерной графики применимы и к трехмерным играм, особенно связанные с проверкой состояния игры и с загрузкой игровых уровней. Пришло время изучить новую тему и выполнить новый проект.

5.6. Заключение

- Для отображения двухмерной графики в Unity применяется ортогональная камера.

- Для отображения графики пиксел в пиксел размер камеры должен быть равен половине высоты экрана.
- Чтобы спрайты реагировали на щелчки мыши, им следует назначить двухмерный коллайдер.
- Новые изображения спрайтов можно загружать программно.
- Текст для пользовательского интерфейса можно создавать с помощью трехмерных текстовых объектов.
- Загрузка игровых уровней позволяет перезагрузить сцену.

6 Двухмерный GUI для трехмерной игры

- ✓ Сравнение старой (до версии Unity 4.6) и новой GUI-систем
- ✓ Создание холста для интерфейса
- ✓ Позиционирование UI-элементов с помощью точек привязки
- ✓ Добавление к UI интерактивных элементов (кнопок, ползунков и т. п.)
- ✓ Рассылка и прием уведомлений о UI-событиях

В этой главе вам предстоит создать двухмерный пользовательский интерфейс для трехмерной игры. В процессе работы над демонстрационным роликом от первого лица вы концентрировались исключительно на самой виртуальной сцене. Но любая игра требует не только места, где происходит действие, но и средств отображения абстрактных взаимодействий и информации. Без этого невозможно представить себе ни одну игру: ни двухмерную, ни трехмерную, ни шутер от первого лица, ни головоломку.

Эти средства отображения абстрактных взаимодействий называют пользовательским интерфейсом (UI) или, точнее, графическим интерфейсом пользователя (GUI). Хотя с технической точки зрения аббревиатура GUI относится к визуальной части интерфейса, например к тексту и кнопкам (рис. 6.1), а аббревиатура UI — к физическим средствам управления, таким как клавиатура или джойстик, люди, говоря про «пользовательский интерфейс», как правило, подразумевают и графическую часть.

Хотя UI требуется любому программному обеспечению, потому что иначе пользователь просто не сможет контролировать работу приложения, GUI в играх зачастую функционирует несколько по-другому. Например, на веб-сайте GUI, по сути, представляет собой сам сайт (если говорить о визуальном представлении). В игре же текст и кнопки зачастую накладываются на игровое пространство с помощью так называемого *проекционного дисплея*.

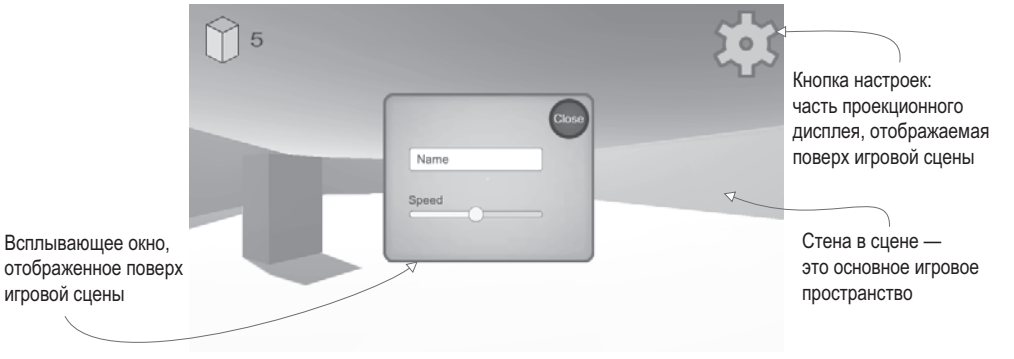


Рис. 6.1. Графический интерфейс (проекционный дисплей), который мы создадим для игры

ОПРЕДЕЛЕНИЕ Проекционным дисплеем (Heads-Up Display, HUD) называется индикатор для вывода важной информации непосредственно во время компьютерной игры. Его прообразом стал индикатор на лобовом стекле, который использовался изначально в военной авиации и позволял пилотам видеть навигационную и специальную информацию, не отвлекаясь на приборную панель.

В этой главе вы узнаете, как добавить в игру проекционный дисплей, используя новейшие инструменты Unity. В главе 5 вы узнали, что в Unity существуют разные способы создания элементов пользовательского интерфейса. Теперь же вам предстоит познакомиться с новой UI-системой, доступной начиная с версии Unity 4.6. Поговорим мы и о старой системе, обсудив на ее примере преимущества нововведений.

Изучать инструменты создания UI в Unity мы будем в рамках проекта шутера от первого лица, которым мы занимались в главе 3. Перечислим стоящие перед нами задачи:

1. Планирование интерфейса.
2. Размещение UI-элементов на экране.
3. Программирование взаимодействий с UI-элементами.
4. Программирование GUI-реакции на события в игре.
5. Программирование реакции сцены на действия с GUI.

ПРИМЕЧАНИЕ Эта глава по большому счету независима от проекта, который послужит рабочей основой — мы просто добавим графический интерфейс поверх существующего демонстрационного ролика. Все упражнения даются на примере шутера от первого лица, созданного в главе 3. Вы можете скачать готовую версию этого проекта или воспользоваться любым другим демонстрационным роликом игры.

Скопируйте проект из главы 3 и откройте его, чтобы приступить к работе. Как обычно, все необходимые вам ресурсы вы найдете в доступном для скачивания примере проекта. Итак, надеюсь, вы готовы к построению пользовательского интерфейса для нашей игры.

6.1. Перед тем как писать код...

Чтобы приступить к созданию проекционного дисплея, первым делом нужно понять, как функционирует UI-система. Unity предлагает разные подходы к построению проекционного дисплея, поэтому желательно выбрать наиболее подходящий для наших целей. После чего останется в общих чертах спланировать UI и подготовить необходимые нам графические ресурсы.

6.1.1. IMGUI или усовершенствованный 2D-интерфейс?

Уже в первой версии Unity появилась система GUI непосредственного режима (Immediate Mode GUI, IMGUI), дающая возможность легко поместить на экран интерактивную кнопку. Код такого размещения показан в листинге 6.1; достаточно присоединить этот сценарий к любому объекту сцены. В качестве еще одного примера UI непосредственного режима можно вспомнить курсор прицеливания, который мы создавали в главе 3. В основе такой системы GUI лежит исключительно код, работа в редакторе Unity не требуется.

ОПРЕДЕЛЕНИЕ Словосочетание «непосредственный режим» (immediate mode) означает явное выполнение команд рисования в каждом кадре, в отличие от режима удержания (retained mode), в котором все визуальные элементы задаются однократно, после чего система сама решает, что именно следует рисовать в каждом кадре.

Листинг 6.1. Реализация кнопки с помощью GUI непосредственного режима

```
using UnityEngine;
using System.Collections;

public class BasicUI : MonoBehaviour {
  void OnGUI() { ← Функция вызывается в каждом кадре после того, как все остальное уже визуализировано.
    if (GUI.Button(new Rect(10, 10, 40, 20), "Test")) { ← Параметры: положение по оси X,
      Debug.Log("Test button");                          положение по оси Y, ширина, высота,
    }                                                    текстовая подпись.
  }
}
```

Центральным местом кода этого листинга является метод `OnGUI()`. Подобно методам `Start()` и `Update()`, все представители класса `MonoBehaviour` автоматически отвечают на метод `OnGUI()`. Он запускается в каждом кадре после визуализации трехмерной сцены, предоставляя место для команд рисования GUI. В данном случае код рисует кнопку; обратите внимание, что команда рисования кнопки выполняется в каждом кадре (то есть в непосредственном режиме). Эта команда фигурирует внутри условной инструкции, срабатывающей при щелчке на кнопке.

Так как GUI непосредственного режима позволяет с минимальными усилиями получить на экране несколько кнопок, мы будем пользоваться им и в следующих главах (особенно в главе 8). Но это практически единственная вещь, которую данная система генерирует легко, именно поэтому в новейших версиях Unity появилась новая система интерфейса, основанная на компонуемой в редакторе двухмерной графике. Ее настройка требует некоторых усилий, но, скорее всего, в готовых играх вы предпочтете пользоваться именно этой системой, дающей более цельные результаты.

Новая система UI функционирует в режиме удержания, поэтому вся графика задается один раз и рисуется в каждом кадре без постоянных повторных определений компоновки. В этом случае размещение графических элементов UI осуществляется в редакторе Unity. Это дает два преимущества перед UI непосредственного режима: во-первых, вы можете оценить вид будущего интерфейса в процессе выкладывания его элементов; во-вторых, такая система позволяет без проблем воспользоваться собственной графикой.

Для работы с этой системой мы импортируем в редактор изображения и будем перетаскивать объекты в сцену. Теперь осталось подумать, как должен выглядеть наш UI.

6.1.2. Выбор компоновки

В большинстве игр проекционный дисплей состоит из нескольких UI-элементов, повторяющихся снова и снова. Словом, для изучения процесса создания UI вам не потребуется ничего сложного. Мы поместим в углы экрана поверх основного изображения сцены индикатор счета и кнопку настроек, как показано на рис. 6.2. Кнопка будет вызывать всплывающее окно с текстовым полем и ползунком.

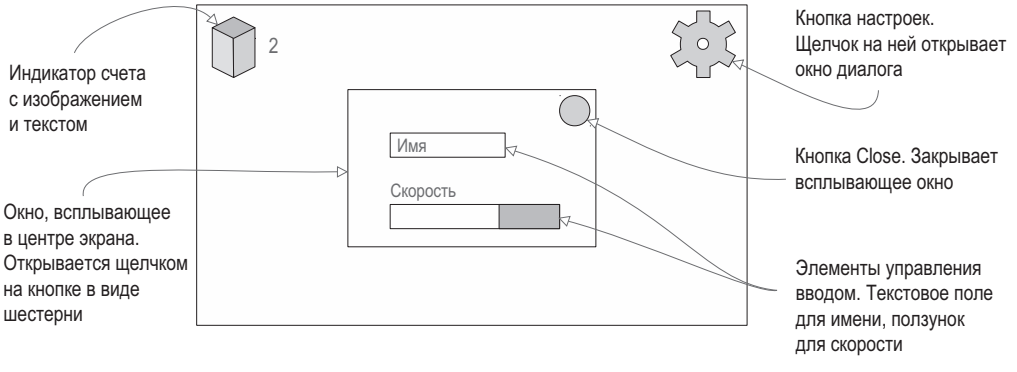


Рис. 6.2. Макет будущего графического интерфейса пользователя

В рассматриваемом примере эти элементы управления вводом послужат для указания имени игрока и выбора скорости движения, но по большому счету они могут управлять любыми относящимися к вашей игре настройками.

Как видите, макет крайне прост! Теперь нужно импортировать необходимые изображения.

6.1.3. Импорт изображений UI

Данный UI требует графики для отображения таких элементов, как, к примеру, кнопки. Мы воспользуемся двухмерными изображениями, как это было в главе 5, поэтому процедура будет вам уже знакома:

1. Импорт изображений (если нужно, определите их как спрайты).
2. Перетаскивание спрайтов в сцену.

Первым делом перетащите все изображения на вкладку Project, чтобы импортировать их в редактор, а затем на панели Inspector убедитесь, что у каждого из них параметр Texture Type имеет значение Sprite (2D And UI).

ВНИМАНИЕ Параметр Texture Type в трехмерных проектах по умолчанию принимает значение Texture, а в двухмерных — Sprite. Если вам требуются спрайты для трехмерного проекта, этот параметр нужно отредактировать вручную.

Все изображения, показанные на рис. 6.3, есть в скачанном примере проекта.

close

Это изображение станет кнопкой закрытия всплывающего окна

enemy

Это изображение станет индикатором счета в верхнем левом углу

gear

Это изображение станет кнопкой настроек в верхнем правом углу

popup

Это изображение после масштабирования послужит фоном всплывающего окна

Рис. 6.3. Изображения, необходимые для текущего проекта

Убедитесь, что все импортированные ресурсы являются спрайтами; вам, скорее всего, потребуется отредактировать у них параметр Texture Type.

Спрайты включают в себя кнопки, индикатор счета и всплывающее окно, которое вам предстоит создать. Теперь, когда они импортированы в проект, расположим графику на экране.

6.2. Настройка GUI

В качестве графических ресурсов выступает уже знакомая нам по главе 5 разновидность двухмерных спрайтов, просто на этот раз мы будем использовать их немного по-другому. В Unity есть специальные инструменты, которые вместо демонстрации изображений в составе сцены превращают их в проекционный дисплей, отображаемый поверх трехмерной сцены. Кроме того, есть особые приемы размещения UI-элементов, обусловленные тем фактом, что на разных экранах фрагменты интерфейса зачастую отображаются по-разному.

6.2.1. Холст для интерфейса

Одним из наиболее фундаментальных и при этом неочевидных аспектов функционирования системы UI является необходимость связать все изображения с *холстом*.

СОВЕТ Холст (canvas) представляет собой специальный вид объектов, который Unity визуализирует как UI для игр.

Откройте меню GameObject, чтобы увидеть перечень доступных для создания объектов, и в категории UI выберите вариант Canvas. В сцене появится холст (для ясности

присвойте ему имя HUD Canvas). Этот объект растянут на весь экран и сильно связан с трехмерной сценой, так как масштабирует один пиксел экрана в одну единицу измерения сцены.

ВНИМАНИЕ Вместе с холстом автоматически создается объект EventSystem. Он требуется для UI-взаимодействий, в остальных же случаях его можно просто игнорировать.

Переключитесь в режим 2D, как показано на рис. 6.4, и дважды щелкните на имени холста на вкладке Hierarchy, чтобы уменьшить масштаб этого объекта и увидеть его целиком. Двухмерный режим включается автоматически для всех 2D-проектов, но в случае 3D-проекта переход между UI и основной сценой осуществляется при помощи переключателя. Для возвращения к просмотру трехмерной сцены отключите режим 2D и дважды щелкните на строке Building, чтобы отмасштабировать объект.

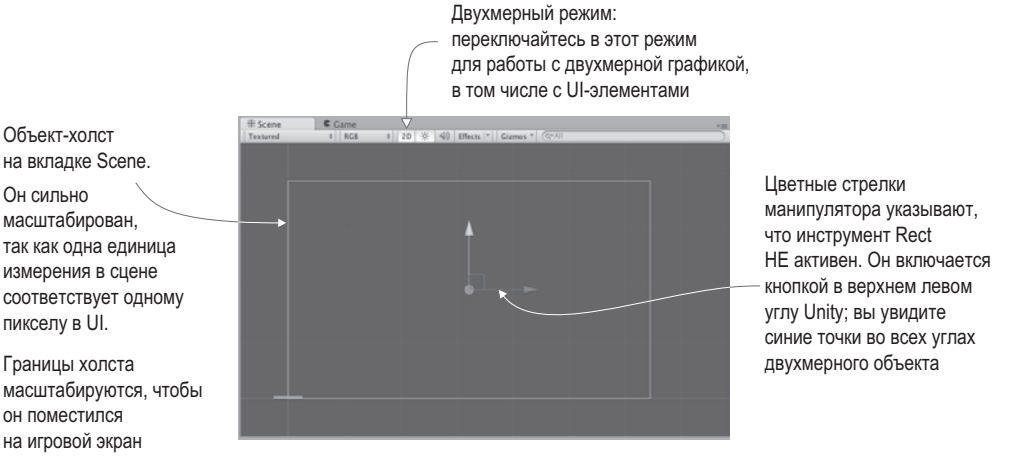


Рис. 6.4. Пустой холст на вкладке Scene

СОВЕТ Хочу напомнить вам совет из главы 4: в верхней части вкладки Scene располагаются кнопки, управляющие видимостью различных элементов, поэтому найдите кнопку Effects, чтобы отключить видимость скайбокса.

У холста есть ряд доступных для редактирования параметров. Прежде всего, это параметр Render Mode, которому следует оставить значение, предлагаемое по умолчанию. Для него возможны следующие значения:

- Screen Space-Overlay — визуализация элементов UI как наложенной поверх вида с камеры двухмерной графики (предлагается по умолчанию).
- Screen Space-Camera — элементы UI также визуализируются поверх вида с камеры, но могут поворачиваться, создавая эффекты перспективы.
- World Space — холст помещается в сцену, что делает элементы UI частью трехмерной сцены.

Последние два режима применяются для создания специальных эффектов, но имеют несколько более сложную реализацию по сравнению с режимом, предлагаемым по умолчанию.

Еще одной важной настройкой является флажок Pixel Perfect. После его установки положение изображений в процессе визуализации слегка корректируется с целью придать им четкий и контрастный вид (в отличие от размывания). Установите этот флажок, и холст готов к размещению спрайтов.

6.2.2. Кнопки, изображения и текстовые подписи

Объект-холст задает область, которая будет отображаться как UI, но туда следует добавить спрайты, соответствующие отдельным элементам. В соответствии с макетом на рис. 6.2 у нас есть изображение блока/врага в верхнем левом углу и рядом с ним текст, отображающий набранные очки, а также кнопка в виде шестерни в верхнем правом углу. В разделе UI меню GameObject есть команды, позволяющие создать изображение (Image), текст (Text) или кнопку (Button). Создайте по одному элементу каждого вида.

Для корректного отображения UI-элементы должны быть потомками объекта-холста. В Unity эта иерархическая связь возникает автоматически, но напомню, что подобные зависимости можно формировать и вручную, перетаскивая объекты на вкладке Hierarchy (рис. 6.5).

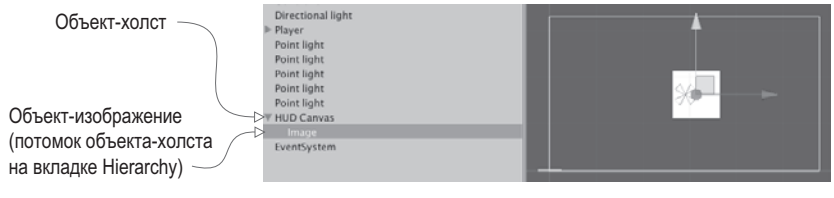


Рис. 6.5. Холст и связанное с ним изображение на вкладке Hierarchy

Между расположенными на холсте объектами также можно сформировать иерархические связи с целью их более удобного размещения. Например, можно перетащить текст на изображение, чтобы надпись перемещалась вместе с картинкой. Более того, у создаваемой по умолчанию кнопки есть дочерний по отношению к ней текстовый объект; в данном случае подпись на кнопке не требуется, поэтому просто удалите его.

Перетащите элементы UI на предназначенные им места. В следующем разделе мы зададим их точное положение, пока же это не имеет особого значения. Указателем мыши перетащите объект-изображение в верхний левый угол холста, а кнопку — в верхний правый.

СОВЕТ Как уже упоминалось в главе 5, в режиме 2D активируется инструмент Rect. Я описывал его как средство, включающее в себя все три преобразования: перемещение, поворот и масштабирование. В режиме 3D эти операции выполняются отдельными инструментами, но при переходе к работе с двухмерными объектами они объединяются, так как у нас пропадает одно измерение. В режиме 2D этот инструмент выбирается автоматически, кроме того, можно активировать его щелчком на кнопке в верхнем левом углу Unity.

Пока оба элемента пусты. Если выделить объект UI и посмотреть на панель Inspector, в верхней части свитка Image вы увидите поле Source Image. Как показано на рис. 6.6, перетащите на это поле спрайты (ни в коем случае не текстуры!) с вкладки Project. Назначьте спрайт с изображением врага объекту-изображению, а спрайт с изображением шестерни — объекту-кнопке. Для обеспечения корректного размера спрайтов щелкните на кнопке Set Native Size.

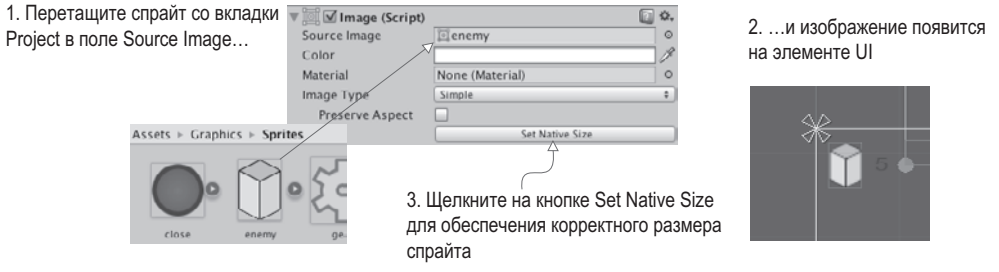


Рис. 6.6. Назначение двухмерных спрайтов свойству Image UI-элементов

Это обеспечит нужный внешний вид картинки с изображением врага и кнопки настроек. У текстового объекта также существует множество параметров, отображаемых на панели Inspector. Первым делом введите какое-нибудь число в большое поле Text; позднее это значение будет переопределено, пока же введенная информация выглядит в редакторе как индикатор набранных очков. Увеличьте размер текста, присвоив параметру Font Size значение 24 и выбрав в раскрывающемся списке Font Style вариант Bold. При этом горизонтально эта текстовая подпись должна быть выровнена по левому краю, а вертикально — по центру, как показано на рис. 6.7. Остальным параметрам мы пока оставим значения, предлагаемые по умолчанию.

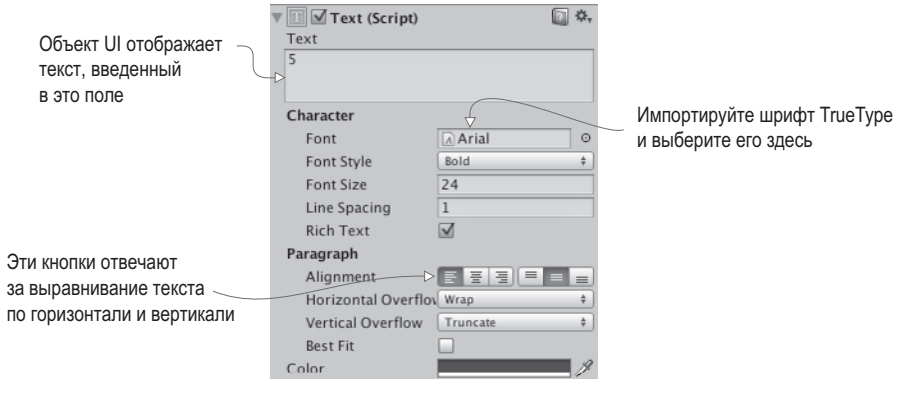


Рис. 6.7. Настройки текстового элемента UI

ПРИМЕЧАНИЕ Кроме содержимого поля Text и выравнивания чаще всего редактируется свойство Font (шрифт). В Unity можно импортировать шрифт TrueType и выбрать его на панели Inspector.

Теперь, когда мы назначили спрайты элементам UI и указали параметры текста, щелкните на кнопке Play, чтобы посмотреть, как выглядит проекционный дисплей, располагающийся поверх игровой сцены. Как показано на рис. 6.8, холст в редакторе Unity обозначает границы экрана, а элементы UI появляются на этом экране в заданных точках.

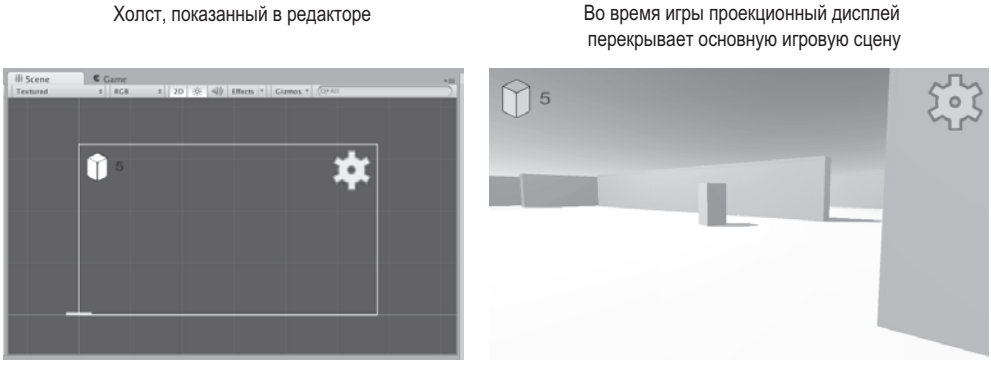


Рис. 6.8. Вид GUI в редакторе (слева) и в процессе игры (справа)

Видите, как здорово: вы добились отображения двухмерного проекционного дисплея поверх трехмерной игровой сцены! Осталось только расположить UI-элементы относительно холста.

6.2.3. Управление положением элементов UI

У всех объектов UI существует точка *привязки*, отображаемая в редакторе в виде крестика (рис. 6.9). Это гибкий инструмент позиционирования элементов интерфейса.

ОПРЕДЕЛЕНИЕ Привязкой (anchor) объекта называется точка его присоединения к холсту или экрану. Именно относительно этой точки указывается положение объекта.

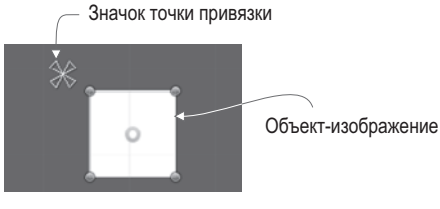


Рис. 6.9. Точка привязки изображения

Положение задается в таких категориях, как, к примеру, «50 пикселов по оси *X*». При этом возникает вопрос, от какой точки отсчитывать эти 50 пикселов? И тут нам на помощь приходит точка привязки. В то время как объект остается статичным относительно этой точки, сама она перемещается относительно холста. Точку привязки можно задать, например, как «центр экрана», и она будет оставаться в центре, когда

экран меняет свой размер. Аналогично привязка к правой стороне экрана позволит объекту оставаться справа даже при изменении размеров (к примеру, при игре на другом мониторе).

Понять, о чем я говорю, проще всего на примере. Выделите объект-изображение и посмотрите на панель Inspector. Настройки привязки вы найдете сразу под компонентом transform (рис. 6.10). По умолчанию в качестве точки привязки элементов UI выбран вариант Center, но нам для изображения требуется вариант Top Left; на рисунке демонстрируется процесс редактирования данного параметра в окне Anchor Presets.

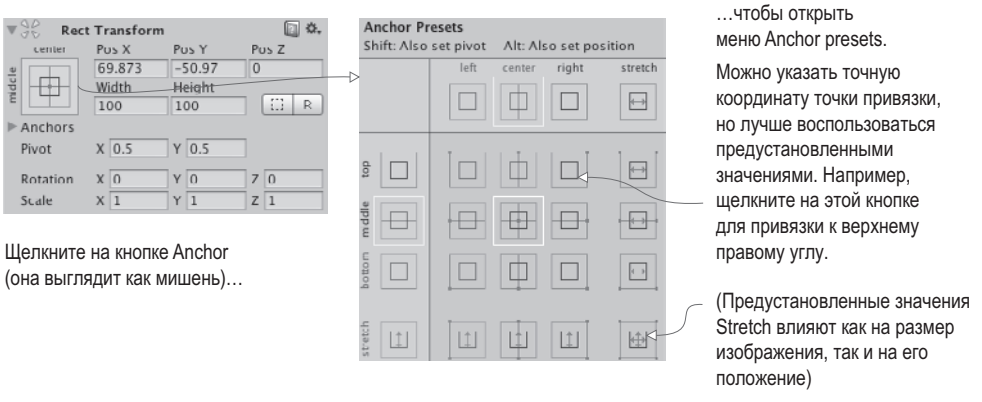


Рис. 6.10. Редактирование параметров привязки

Заодно поменяйте привязку кнопки настроек. Установите для нее значение Top Right, щелкнув на верхнем правом квадрате в меню Anchor Preset. Теперь попробуйте поменять размер экрана, перетаскивая его боковые края влево и вправо. Благодаря привязкам при изменении размеров холста объекты UI останутся на своих местах. Как показано на рис. 6.11, элементы UI перемещаются вместе с краями экрана.

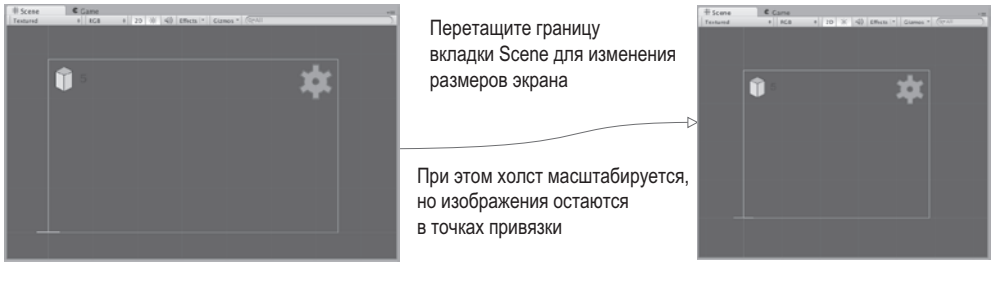


Рис. 6.11. Точки привязки остаются на месте при изменении размеров экрана

СОВЕТ Точки привязки позволяют подстраиваться не только под изменение положения, но и под изменение размера. В этой главе данная функциональность использоваться не будет, но каждый угол изображения можно связать с одним из углов экрана. На рис. 6.11 изображения сохраняют свой размер, но привязку можно отредактировать таким образом, что при увеличении размера экрана изображения будут растягиваться вместе с ним.

Итак, все визуальные элементы уже на своих местах, пришло время обеспечить их интерактивность.

6.3. Программирование интерактивного UI

Для взаимодействия с UI нам потребуется указатель мыши. Если помните, в этой игре его настройки редактируются в методе `Start()` сценария `RayShooter`, в котором мы блокировали и скрывали указатель мыши. Такое поведение прекрасно подходило для элементов управления в шутере от первого лица, но оно помешает работе с UI. Удалите эти строки из сценария `RayShooter.cs`, чтобы получить возможность щелкать на проекционном дисплее.

Кроме того, в сценарий `RayShooter.cs` нужно добавить строки, блокирующие возможность стрелять в процессе взаимодействия с GUI. Следующий листинг демонстрирует новую версию кода.

Листинг 6.2. Добавление команд взаимодействия с GUI в код сценария RayShooter.cs

```
using UnityEngine.EventSystems; ← Подключение библиотеки для UI-системы.
...
void Update() {
if (Input.GetMouseButtonDown(0) && ← Выделенный курсивом код в сценарии уже присутствовал; он приведен здесь для справки.
EventSystem.current.IsPointerOverGameObject(); ← Проверяем, что GUI не используется.
Vector3 point = new Vector3(
camera.pixelWidth/2, camera.pixelHeight/2, 0);
...
```

Теперь в процессе игры вы можете щелкать на кнопках, хотя пока это не дает результатов. Видно только, как меняется оттенок кнопки при наведении на нее указателя мыши и при щелчке. Это заданное по умолчанию изменение цвета, которое можно поменять для каждой кнопки, но пока мы этого делать не будем. Можно ускорить возвращение кнопки к обычному цвету; за это отвечает параметр Fade Duration в свитке Button, попробуйте уменьшить его до 0.01 и посмотрите, что получится.

СОВЕТ Иногда только что созданные элементы взаимодействия с UI могут мешать игре. Помните автоматически появившийся вместе с холстом объект EventSystem? Он контролирует в числе прочего и элементы интерфейса, по умолчанию используя для взаимодействия с GUI кнопки со стрелками. Имеет смысл отключить режим работы с этими кнопками для компонента EventSystem: выделите его на вкладке Hierarchy и на панели Inspector сбросьте флажок Send Navigation Event.

Однако щелчок на кнопке пока ни к чему не приводит, так как она не связана ни с каким кодом. Давайте исправим этот недостаток.

6.3.1. Программирование невидимого объекта UIController

В общем случае программирование взаимодействия с элементами UI сводится к стандартной процедуре, общей для всех элементов:

1. В сцене создается UI-объект (в нашем случае это созданная в предыдущем разделе кнопка).

2. Пишется сценарий, который будет вызываться при обращении к этому элементу UI.
3. Сценарий присоединяется к объекту в сцене.
4. Элементы UI (например, кнопки) связываются с объектом, к которому присоединен этот сценарий.

Кнопка у вас уже есть, осталось создать контроллер, который будет с ней связываться. Создайте сценарий с именем `UIController` (его код приведен в следующем листинге) и перетащите этот сценарий на объект-контроллер в сцене.

Листинг 6.3. Сценарий UIController, предназначенный для программирования кнопок

```
using UnityEngine;
using UnityEngine.UI; ← Импорт инфраструктуры для работы с кодом UI.
using System.Collections;

public class UIController : MonoBehaviour {
  [SerializeField] private Text scoreLabel; ← Объект сцены Reference Text, предназначенный
                                              для задания свойства text.

  void Update() {
    scoreLabel.text = Time.realtimeSinceStartup.ToString();
  }

  public void OnOpenSettings() { ← Метод, вызываемый кнопкой настроек.
    Debug.Log("open settings");
  }
}
```

СОВЕТ Скорее всего, вам интересно, зачем нам два объекта: SceneController и UIController. Ведь наша сцена настолько проста, что с управлением ее объектами и элементами интерфейса вполне мог бы справиться один контроллер. Но по мере ее усложнения вы убедитесь, что намного лучше иметь отдельные модули управления, взаимодействующие друг с другом косвенным образом. Этот принцип применим не только к играм, но и к программному обеспечению в целом; в среде разработчиков ПО такой подход называют разделением ответственности (separation of concerns).

Теперь нужно перетащить объекты на ячейки компонентов, чтобы связать их друг с другом. Перетащите созданный нами для отображения счета текстовый объект на поле Score Label объекта UIController. После этого код сценария `UIController` начнет определять отображаемый в данной подписи текст. В настоящее время отображаются значения времени — мы добавили таймер, чтобы протестировать, как все работает; чуть позже мы заменим значения времени набранными игроком очками.

Теперь нужно снабдить кнопку элементом `OnClick`, чтобы добавить ее к объекту-контроллеру. Выделите кнопку и найдите в нижней части панели Inspector поле OnClick; изначально оно пустое, но, как показано на рис. 6.12, можно щелкнуть на кнопке со значком + (плюс) и добавить туда элемент. Каждый элемент определяет одну функцию, вызываемую щелчком на кнопке; в листинге присутствуют как ячейка для объекта, так и меню для вызываемой функции. Перетащите на ячейку объект-контроллер, выделите в меню строку UIController и выберите в дополнительном меню вариант OnOpenSettings().

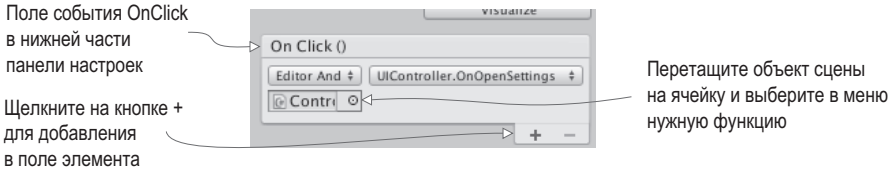


Рис. 6.12. Поле OnClick в нижней части панели с настройками кнопки

РЕАКЦИЯ НА ОСТАЛЬНЫЕ СОБЫТИЯ МЫШИ

Наша кнопка реагирует только на событие OnClick, в то время как UI-элементы могут отвечать на разные варианты взаимодействий. Для программирования взаимодействий, отличных от заданных по умолчанию, пользуйтесь компонентом EventTrigger.

Добавьте к кнопке новый компонент и найдите раздел Event в меню этого компонента. Выберите там вариант EventTrigger. Хотя событие OnClick кнопки отвечает только на полноценный щелчок (кнопка мыши нажимается, а затем отпускается), попробуем запрограммировать реакцию только на нажатие кнопки мыши. Последовательность действий будет той же самой, что и для события OnClick, просто на этот раз мы укажем реакцию на другое событие. Первым делом добавьте в сценарий UIController еще один метод:

```
...
public void OnPointerDown() {
  Debug.Log("pointer down");
}
...
```

Щелкните на кнопке Add New Event Type для добавления нового типа к компоненту EventTrigger. Выберите вариант Pointer Down. Появится пустое поле для этого события, полностью аналогичное полю для события OnClick. Щелкните на кнопке со значком + (плюс), чтобы добавить элемент, и перетащите на этот элемент объект-контроллер, после чего выберите в меню вариант OnPointerDown(). Все готово!

Запустите игру и щелкните на кнопке для вывода на консоль отладочных сообщений. В данном случае мы выводим информацию, не имеющую никакого отношения к игре, просто чтобы протестировать работу кнопки. По щелчку должно появляться всплывающее окно с настройками. Именно его созданием мы и займемся в следующем разделе.

6.3.2. Создание всплывающего окна

Наш интерфейс содержит кнопку, открывающую окно диалога, при этом само окно пока отсутствует. Его роль будет играть новый объект-изображение с присоединенными к нему элементами управления (такими, как кнопки и ползунки). Первым делом следует создать новое изображение, поэтому выберите в меню GameObject команду UI, а затем — команду Image. Как и раньше, на панели Inspector вы найдете ячейку Source Image. Перетащите на нее спрайт с именем popup.

По умолчанию спрайт масштабируется под размеры объекта-изображения; именно это происходило со спрайтами, предназначенными для отображения счета и кнопки настроек, и вы щелкали на кнопке Set Native Size, чтобы подогнать объект под размер картинки. Это стандартное поведение объектов-изображений, но у всплывающего окна другое назначение.

Как показано на рис. 6.13, у компонента image есть параметр Image Type. По умолчанию он имеет значение Simple, что раньше нас вполне устраивало. Но для всплывающего окна присвойте параметру Image Type значение Sliced.

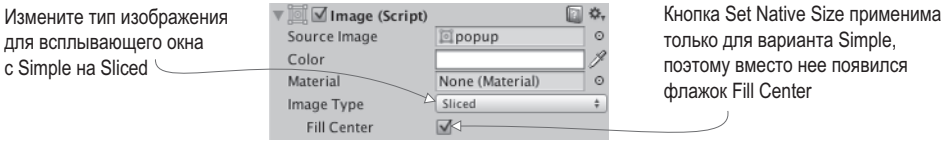


Рис. 6.13. Параметр Image Type компонента image

Переход к *фрагментированному изображению* может завершиться сообщением об ошибке, информирующим, что у изображения отсутствуют границы. Дело в том, что спрайт popup пока не разделен на девять частей. В этом случае выделите этот спрайт на вкладке Project и на панели Inspector щелкните на кнопке Sprite Editor, как показано на рис. 6.14. Откроется окно диалога Sprite Editor.

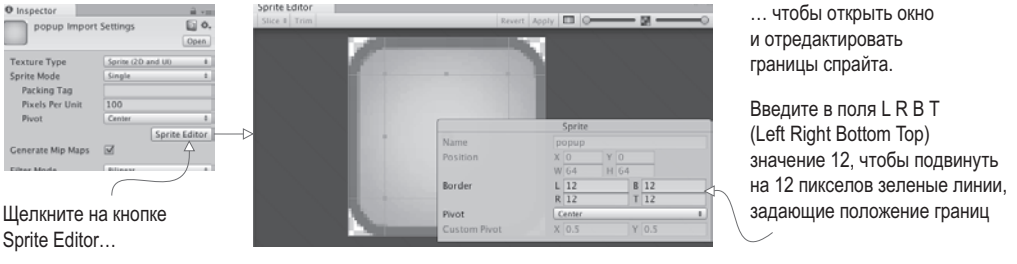


Рис. 6.14. Кнопка Sprite Editor на панели Inspector и открываемое ею окно

ОПРЕДЕЛЕНИЕ Фрагментированное изображение (sliced image) разбито на девять частей, которые масштабируются по-разному. Масштабируя края отдельно от середины, вы гарантируете сохранение четких и резких границ при любом изменении размеров. В других инструментах разработки имена таких изображений часто содержат цифру 9 (например, 9-slice, 9-patch, scale-9), что подчеркивает факт разделения на девять частей.

В окне Sprite Editor вы увидите зеленые линии, указывающие, каким образом будет осуществляться разбивка изображения. Изначально границ у нашего спрайта нет (то есть величина всех границ равна 0). Увеличьте ширину границ со всех сторон, чтобы получить показанный на рис. 6.14 результат. Так как все четыре параметра (Left, Right, Bottom и Top) имеют значение 12 пикселов, то пересекающиеся зеленые линии поделят изображение на девять частей. Закройте окно редактора и примените сделанные изменения.

Теперь, когда спрайт разделен на девять частей, параметр Image Type без проблем примет значение Sliced (а внизу появится флажок Fill Center). Перетащите находящийся в любом из углов изображения манипулятор синего цвета, чтобы выполнить масштабирование (если вы не видите манипуляторов, активируйте описанный в главе 5

инструмент Rect). Боковые фрагменты при этом сохранят свои размеры, в то время как центральная часть поменяет масштаб.

Это свойство боковых фрагментов сохраняет резкость границ изображения при любом изменении размеров, что идеально подходит для UI-элементов: различные окна могут иметь разные размеры, но выглядеть при этом должны одинаково. Сделайте ширину нашего окна равной 250, а высоту — 200, придав ему такой же вид, как на рис. 6.15 (заодно убедитесь, что оно находится в точке с координатами 0, 0, 0).

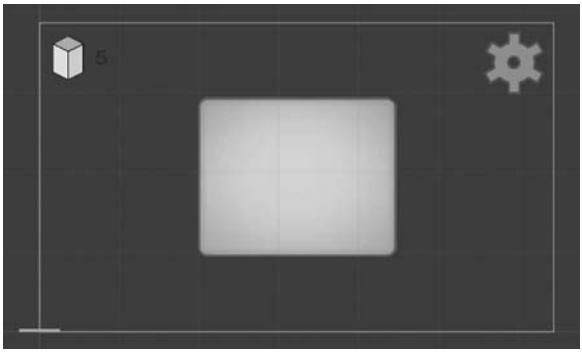


Рис. 6.15. Фрагментированное изображение отмасштабировано до размеров всплывающего окна

СОВЕТ Способ наложения элементов UI друг на друга определяется порядком их следования на вкладке Hierarchy. Расположите всплывающее окно поверх остальных элементов UI (разумеется, сохранив его связь с холстом). Теперь подвигайте это окно по сцене и посмотрите, каким образом перекрываются изображения. Затем перетащите окно в самый низ иерархического списка дочерних элементов холста, чтобы оно отображалось поверх всего остального.

Итак, всплывающее окно готово, пришла пора написать для него код. Создайте сценарий `SettingsPopup` (его код представлен в следующем листинге) и перетащите его на наше окно.

Листинг 6.4. Сценарий SettingsPopup для всплывающего окна

```
using UnityEngine;
using System.Collections;

public class SettingsPopup : MonoBehaviour {
  public void Open() {
    gameObject.SetActive(true); ← Активируйте этот объект, чтобы открыть окно.
  }
  public void Close() {
    gameObject.SetActive(false); ← Деактивируйте объект, чтобы закрыть окно.
  }
}
```

Теперь откройте сценарий `UIController.cs` и добавьте в него содержимое следующего листинга.

Листинг 6.5. Добавление в сценарий UIController возможности работы с окном

```
...
[SerializeField] private SettingsPopup settingsPopup;
void Start() {
  settingsPopup.Close(); ← Закрываем всплывающее окно в момент начала игры.
}
...
public void OnOpenSettings() {
  settingsPopup.Open(); ← Заменяем отладочный текст методом всплывающего окна.
}
...
```

Этот код добавляет ячейку для всплывающего окна, поэтому свяжите это окно с объектом UIController. После этого оно начнет закрываться в начале игры и открываться при щелчке на кнопке настроек.

Но пока у нас нет способа закрыть его вручную, значит, нужно добавить соответствующую кнопку. Последовательность действий будет практически такой же, как при создании предыдущей кнопки: выберите в меню GameObject команду UI ▸ Button, поместите новую кнопку в верхний правый угол всплывающего окна, перетащите спрайт close на ячейку Source Image данного UI-элемента и щелкните на кнопке Set Native Size, чтобы изображение приобрело нужный размер. Но в отличие от предыдущего случая нам требуется текстовая подпись, поэтому выделите дочерний компонент text и введите в текстовое поле слово `Close`, сделав его цвет белым. На вкладке Hierarchy перетащите эту кнопку на всплывающее окно, чтобы сформировать иерархическую связь. В качестве заключительного штриха присвойте параметру Fade Duration значение 0.01 и сделайте чуть темнее параметр Normal Color, присвоив ему значения 110, 110, 110, 255.

Чтобы кнопка начала закрывать окно, ей нужно сопоставить событие `OnClick`; щелкните на кнопке со значком + (плюс) поля OnClick, перетащите всплывающее окно на ячейку объекта и выберите в списке функций вариант Close(). Запустите игру и убедитесь, что созданная кнопка позволяет закрыть всплывающее окно.

Итак, мы добавили к нашему проекционному дисплею всплывающее окно. Но пока оно пустое, то есть нам нужно вставить в него элементы управления. Именно этим мы и займемся в следующем разделе.

6.3.3. Задание значений с помощью ползунка и поля ввода

Процедура добавления элементов управления к всплывающему окну настроек состоит из двух этапов, почти как уже знакомая вам процедура создания кнопок. Вы генерируете присоединенные к холсту элементы UI и связываете их со сценарием. Нам нужен ползунок, текстовое поле и текстовая подпись к ползунку. Выберите в меню GameObject команду UI ▸ Text, чтобы создать текстовый объект, затем — команду UI ▸ InputField, чтобы создать текстовое поле, а потом — команду UI ▸ Slider для создания ползунка (рис. 6.16).

Сделайте все три объекта потомками всплывающего окна путем их перетаскивания на вкладке Hierarchy, а затем расположите их в соответствии с рисунком, выровняв по центру всплывающего окна. Присвойте текстовому элементу значение Speed, сформировав

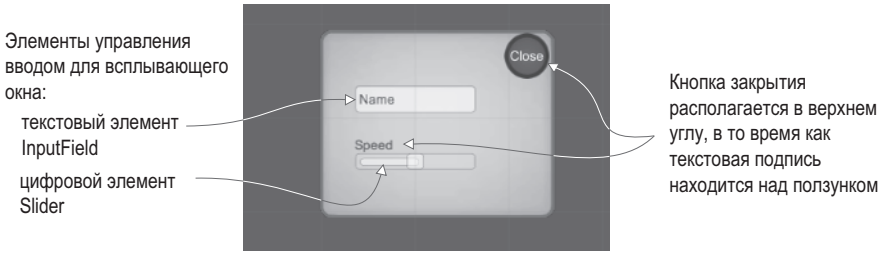


Рис. 6.16. Элементы управления вводом, добавленные к всплывающему окну

подпись к ползунку. Поле предназначено для ввода текста, и по умолчанию, пока игрок ничего не ввел, там отображается слово Text; замените его словом `Name`. Параметрам Content Type и Line Type можно оставить значения, предлагаемые по умолчанию. Впрочем, вы можете ограничить вводимую информацию только буквами или только целыми числами, изменив параметр Content Type, и разрешить ввод нескольких строк с помощью параметра Line Type.

ВНИМАНИЕ Вы не сможете пользоваться ползунком, если его перекрывает текстовая подпись. Сделайте так, чтобы на вкладке Hierarchy текстовый объект располагался над ползунком, что обеспечит расположение ползунка поверх подписи.

Параметры ползунка вы найдете в нижней части панели Inspector. Минимальное значение задает параметр Min Value, по умолчанию равный 0; его мы менять не будем. Параметр же Max Value по умолчанию имеет значение 1, но в нашем примере мы присвоим ему значение 2. Параметрам Value и Whole Numbers оставим значения, предлагаемые по умолчанию; первый устанавливает начальное положение ползунка, а второй ограничивает допустимые значения целыми числами (в данном случае такое ограничение не требуется).

Итак, мы задали параметры всех объектов, осталось написать код, управляющий их поведением; добавьте в сценарий `SettingsPopup.cs` методы из следующего листинга.

Листинг 6.6. Методы для элементов управления вводом всплывающего окна

```
...
public void OnSubmitName(string name) {
  Debug.Log(name);
}
public void OnSpeedValue(float speed) {
  Debug.Log("Speed: " + speed);
}
...
```

← Этот метод срабатывает в момент начала ввода данных в текстовое поле.

← Этот метод срабатывает при изменении положения ползунка.

Замечательно. Наши элементы управления обзавелись методами. Среди настроек поля ввода появилось поле End Edit; перечисленные в этом поле события наступают после завершения пользовательского ввода. Добавьте к этому полю элемент, щелкнув на кнопке со значком + (плюс), перетащите всплывающее окно на ячейку для объекта и выберите в списке функций вариант OnSubmitName().

ВНИМАНИЕ Функция OnSubmitName() присутствует как в верхнем списке Dynamic String, так и в нижнем Static Parameters. Но выбор варианта в нижнем списке даст возможность отправлять всего одну заранее заданную строку. А так как нам нужно, чтобы пересылалось любое введенное в поле значение, то есть динамическая строка, выберите функцию OnSubmitName() в списке Dynamic String.

Повторите эту последовательность действий для ползунка: найдите поле событий в нижней части панели Inspector (в данном случае поле называется OnValueChanged), щелкните на кнопке со значком + (плюс) для добавления элемента, перетащите на ячейку объекта всплывающее окно и выберите в списке динамических функций вариант OnSpeedValue().

Теперь оба элемента управления вводом связаны с кодом сценария для всплывающего окна. Запустите игру и понаблюдайте за консолью в процессе перемещения ползунка или нажатия клавиши Enter после ввода имени в текстовое поле.

СОХРАНЕНИЕ ПАРАМЕТРОВ МЕЖДУ ИГРАМИ

В Unity есть несколько методов сохранения данных на постоянной основе, простейшим из которых является PlayerPrefs. Вам предоставляется абстрагированный вариант, работающий на всех платформах и с разными файловыми системами (то есть вам не нужно беспокоиться о деталях реализации), позволяющий сохранять небольшие фрагменты информации. В случае с большими объемами данных метод PlayerPrefs вам не очень поможет (в главе 11 для сохранения данных в процессе игры мы воспользуемся другими методами), но для сохранения настроек он подходит просто идеально.

Метод PlayerPrefs дает вам простые команды чтения и задания именованных значений (его работа во многом напоминает хэш-таблицу или словарь). Например, для сохранения выбранной скорости достаточно добавить в метод OnSpeedValue() сценария SettingsPopup строку PlayerPrefs.SetFloat(«speed», speed);. Этот метод сохранит десятичное значение скорости в переменной speed.

Аналогичным образом происходит инициализация ползунка с использованием сохраненного значения. Добавьте в сценарий SettingsPopup следующий код:

using UnityEngine.UI;

```
...
[SerializeField] private Slider speedSlider;
void Start() {
  speedSlider.value = PlayerPrefs.GetFloat("speed", 1);
}
...
```

Обратите внимание, что команде get предоставляется как значение переменной speed, так и значение по умолчанию на случай, если сохраненная скорость отсутствует.

Наши элементы управления генерируют отладочные сообщения, но пока никак не влияют на происходящее в игре. Программированию взаимодействий проекционного дисплея с игрой посвящен последний раздел нашей главы.

6.4. Обновление игры в ответ на события

До текущего момента проекционный дисплей и игра существовали отдельно друг от друга, в то время как они должны взаимодействовать. Мы уже программировали взаимодействия между объектами с помощью ссылок в сценариях, но в данном случае этот вариант нам не подходит, так как он приводит к формированию сильной связи между сценой и проекционным дисплеем. Нам же нужно, чтобы они были

практически независимы друг от друга, обеспечивая возможность свободно редактировать игру, не беспокоясь о сохранности проекционного дисплея.

Для информирования UI о происходящем в сцене мы создадим систему широковещательной рассылки сообщений. Принцип ее работы иллюстрирует рис. 6.17. Сценарии могут подписаться на слушание события, сообщение о котором рассылает конкретный код. Давайте посмотрим на практике, как все это работает.

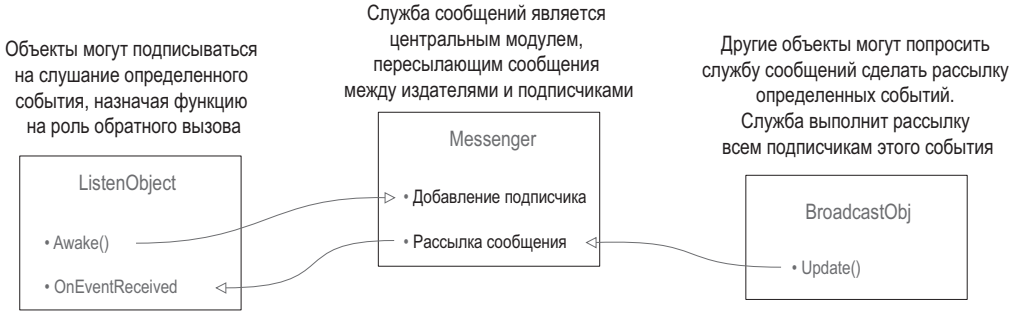


Рис. 6.17. Схема широковещательной рассылки сообщений

СОВЕТ В языке C# существует встроенная система обработки событий. Скорее всего, у вас возник вопрос, почему мы ею не пользуемся. Дело в том, что эта система работает с нацеленными сообщениями, в то время как нам требуется широковещательная рассылка. В нацеленной системе код должен заранее знать источник сообщения, в то время как рассылка работает с сообщениями произвольного источника.

6.4.1. Интегрирование системы сообщений

Для оповещения UI о происходящем в сцене мы создадим систему широковещательной рассылки сообщений. Хотя в Unity нет подходящей для выполнения такой задачи встроенной функции, мы можем скачать нужный сценарий. Вики-сообщества Unify представляют собой репозиторий бесплатного кода от различных разработчиков. Их система для службы сообщений дает несвязанный способ взаимодействия с остальной частью программы посредством событий. Рассылающий сообщения код может ничего не знать о подписчиках, что позволяет легко менять или добавлять взаимодействующие объекты.

Создайте сценарий с именем `Messenger` и скопируйте в него одноименный код со страницы http://wiki.unity3d.com/index.php/CSharpMessenger_Extended.

Далее вам понадобится еще один сценарий, который называется `GameEvent`. Его код приведен в следующем листинге.

Листинг 6.7. Сценарий GameEvent, который будет использоваться вместе со сценарием Messenger

```
public static class GameEvent {
  public const string ENEMY_HIT = "ENEMY_HIT";
  public const string SPEED_CHANGED = "SPEED_CHANGED";
}
```

Этот сценарий задает константу для пары сообщений о событиях, что позволяет систематизировать сообщения, одновременно избавляя вас от необходимости вводить строку сообщения в разных местах.

Система оповещений о событиях готова, давайте ею воспользуемся. Начнем мы с сообщений от сцены к проекционному дисплею, а затем запрограммируем взаимодействие в противоположном направлении.

6.4.2. Рассылка и слушание сообщений сцены

До текущего момента вместо набранных игроком очков отображались значения времени — мы использовали таймер для тестирования функциональности нашего текстового дисплея. Нам же нужно, чтобы там отображалось количество пораженных игроком врагов, поэтому давайте отредактируем код сценария `UIController`. Первым делом удалите метод `Update()`, так как именно там находился тестовый код. В момент своей смерти враг генерирует событие. Следующий листинг заставит сценарий `UIController` слушать это событие.

Листинг 6.8. Добавление подписчиков на событие в сценарий UIController

```
...
private int _score;

void Awake() {
  Messenger.AddListener(GameEvent.ENEMY_HIT, OnEnemyHit); ← Объявляем, какой метод отвечает на событие ENEMY_HIT.
}
void OnDestroy() {
  Messenger.RemoveListener(GameEvent.ENEMY_HIT, OnEnemyHit); ← При разрушении объекта удаляйте подписчика, чтобы избежать ошибок.
}

void Start() {
  _score = 0;
  scoreLabel.text = _score.ToString(); ← Присвоение переменной score начального значения 0.
  settingsPopup.Close();
}

private void OnEnemyHit() {
  _score += 1; ← Увеличение переменной score на 1 в ответ на данное событие.
  scoreLabel.text = _score.ToString();
}
...
```

Первым делом обратите внимание на методы `Awake()` и `OnDestroy()`. Как и методы `Start()` и `Update()`, все члены класса `MonoBehaviour` автоматически реагируют на активацию или удаление объекта. Подписчик добавляется в методе `Awake()` и удаляется в методе `OnDestroy()`. Будучи частью системы широковещательной рассылки сообщений, при получении данного сообщения он вызывает метод `OnEnemyHit()`, который увеличивает переменную `score` на 1 и выводит новое значение на текстовый дисплей.

Подписчик события задан в коде UI, поэтому каждое поражение врага должно сопровождаться рассылкой соответствующего сообщения. Реагирующий на смерть врага

код находится в сценарии `RayShooter.cs`, поэтому добавьте туда код отправки сообщения из следующего листинга.

Листинг 6.9. Рассылка сообщения о событии в сценарии RayShooter

```
...
if (target != null) {
  target.ReactToHit();
  Messenger.Broadcast(GameEvent.ENEMY_HIT); ← К реакции на попадания добавлена рассылка сообщения.
} else {
  ...
```

Запустите игру и убедитесь, что теперь текстовый дисплей отображает количество пораженных врагов. При каждом попадании счетчик должен увеличиваться на единицу. Мы успешно справились с программированием сообщений от трехмерной игры к двухмерному интерфейсу, теперь нам требуется рассылка в обратном направлении.

6.4.3. Рассылка и слушание сообщений проекционного дисплея

В предыдущем разделе сообщение о событии посылалось сценой и принималось проекционным дисплеем. Сходным образом UI-элементы могут посылать сообщения, которые будут слушаться как игроками, так и врагами. В результате параметры, которые игрок укажет во всплывающем окне, начнут влиять на настройки игры.

Откройте сценарий `WanderingAI.cs` и добавьте туда код следующего листинга.

Листинг 6.10. Добавление подписчика события в сценарий WanderingAI

```
...
public const float baseSpeed = 3.0f; ← Базовая скорость, меняемая в соответствии с положением ползунка.
...
void Awake() {
  Messenger<float>.AddListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
void OnDestroy() {
  Messenger<float>.RemoveListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
...
private void OnSpeedChanged(float value) { ← Метод, объявленный в подписчике для события SPEED_CHANGED.
  speed = baseSpeed * value;
}
...
```

Методы `Awake()` и `OnDestroy()` в данном случае тоже добавляют и удаляют подписчика, но на этот раз нам предстоит иметь дело еще и со значением, задающим скорость перемещения врага.

СОВЕТ В предыдущем разделе мы использовали обобщенное событие, в то время как система рассылки позволяет передавать не только сообщения, но и значения. Для этого к подписчику достаточно добавить определение типа; обратите внимание на дополнение <float> в команде задания подписчика.

Внесем аналогичные изменения в сценарий `FPSInput.cs`, чтобы повлиять на скорость перемещения игрока. Содержимое следующего листинга отличается от листинга 6.10 только значением переменной `baseSpeed` у игрока.

Листинг 6.11. Добавление подписчика события в сценарий FPSInput

```
...
public const float baseSpeed = 6.0f; ← Это значение отличается от указанного в листинге 6.10.
...
void Awake() {
  Messenger<float>.AddListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
void OnDestroy() {
  Messenger<float>.RemoveListener(GameEvent.SPEED_CHANGED, OnSpeedChanged);
}
...
private void OnSpeedChanged(float value) {
  speed = baseSpeed * value;
}
...
```

Напоследок нам требуется рассылка значений скорости из сценария `SettingsPopup` в ответ на изменение положения ползунка. Код этой рассылки приведен в следующем листинге.

Листинг 6.12. Рассылка сообщения сценарием SettingsPopup

```
public void OnSpeedValue(float speed) {
  Messenger<float>.Broadcast(GameEvent.SPEED_CHANGED, speed); ← Значение, заданное положением ползунка, рассылается как событие <float>.
  ...
```

Теперь при изменении положения ползунка скорость перемещения врага и игрока будет меняться. Щелкните на кнопке Play и убедитесь сами!

УПРАЖНЕНИЕ: ИЗМЕНЕНИЕ СКОРОСТИ ГЕНЕРИРУЕМЫХ ВРАГОВ

Пока что корректируется только скорость уже присутствующего в сцене врага. Новые враги создаются без учета настроек скорости. Попробуйте самостоятельно задать скорость перемещения генерируемых врагов. Подсказка: добавьте подписчика SPEED_CHANGED в сценарий SceneController, так как именно этот сценарий отвечает за появление новых врагов.

Теперь вы знаете, как создать графический интерфейс с помощью предлагаемых Unity инструментов. Это пригодится вам при работе над остальными проектами, а пока мы начнем знакомиться с другими игровыми жанрами.

6.5. Заключение

- В Unity поддерживается как система GUI непосредственного режима, так и более новая система, основой которой являются двухмерные спрайты.
- Применение двухмерных спрайтов для создания GUI требует наличия в сцене такого объекта, как холст.
- UI-элементы можно привязать к различным точкам настраиваемого холста.
- Свойство Active включает и выключает элементы UI.
- Несвязанная система передачи сообщений является замечательным средством передачи событий между интерфейсом и сценой.

7 Игра от третьего лица: перемещение и анимация игрока

- ✓ Добавление теней в реальном времени
- ✓ Облет камеры вокруг цели
- ✓ Плавное изменение вращения с помощью алгоритма линейной интерполяции
- ✓ Распознавание поверхности при прыжках, а также с учетом краев и склонов
- ✓ Назначение анимации антропоморфному персонажу и управление ею

В этой главе вы запрограммируете еще одну трехмерную игру, но на этот раз в другом жанре. Как вы помните, глава 2 посвящена созданию демонстрационного ролика для игры от первого лица. Сейчас нам предстоит создать еще один демонстрационный ролик, но на этот раз в центре внимания окажутся перемещения персонажа. Самым важным отличием будет положение камеры относительно игрока: в игре от первого лица игрок смотрит на сцену глазами персонажа, в то время как в игре от третьего лица камера находится *в стороне* от него. Скорее всего, такое представление знакомо вам по серии приключенческих игр Legend of Zelda или по более поздней серии игр Uncharted (виды сцены в игре от первого и от третьего лица сравниваются на рис. 7.3).

В этой главе вы создадите один из самых интересных в визуальном отношении проектов книги. Рисунок 7.1 демонстрирует макет будущей сцены. Сравните его с показанным на рис. 2.2 макетом игры от первого лица, с которой в главе 2 началось ваше знакомство с созданием игр в Unity.

Как видите, конструкция комнаты не изменилась, сходным осталось и применение сценариев. Коренные изменения претерпели только вид игрока и положение камеры. Еще раз упомяну аспекты, превращающие происходящее в игру от третьего лица. Это расположение камеры в стороне от персонажа и ее нацеленность в сторону персонажа. Мы воспользуемся вместо примитивной капсулы моделью, имеющей вид человека, так как игроки теперь могут себя видеть.

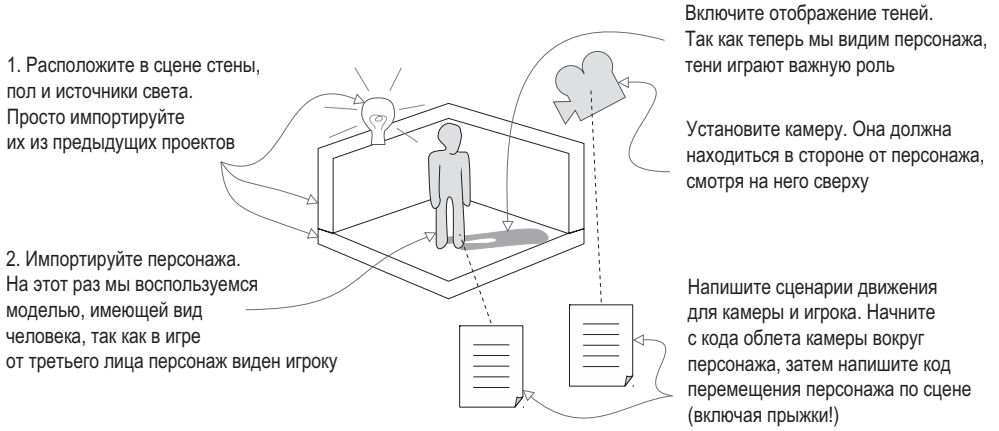


Рис. 7.1. Макет демонстрационного ролика игры от третьего лица

Помните, в главе 4 упоминались такие типы графических ресурсов, как трехмерные модели и анимация? Термин *трехмерная модель* является практически синонимом сеточного объекта — это статическая форма, определенная вершинами и полигонами (то есть сеточная геометрия). В случае антропоморфного персонажа этой геометрии придается форма головы, рук, ног и пр. (рис. 7.2).

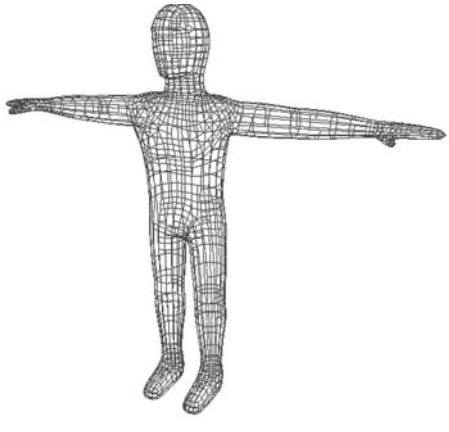


Рис. 7.2. Каркасное представление модели, с которой вы будете работать в этой главе

Как обычно, мы сфокусируемся на последнем пункте нашего плана: программировании объектов в сцене. Вот как выглядит последовательность наших действий:

1. Импорт модели персонажа в сцену.
2. Настройка элементов управления камерой и ее нацеливание на персонажа.
3. Написание сценария, позволяющего персонажу бегать по поверхности.
4. Добавление к сценарию движения возможности прыгать.
5. Воспроизведение анимации на модели на основе ее движений.

Скопируйте проект из главы 2, чтобы воспользоваться им как основой. Также можно открыть новый проект Unity (убедитесь, что вы создаете проект 3D, а не 2D, как в главе 5) и скопировать в него файл сцены из главы 2. Кроме того, вам понадобится папка scratch из материалов к данной главе. Именно в ней находится модель персонажа.

ПРИМЕЧАНИЕ Мы собираемся разрабатывать новый проект в интерьерах сцены из главы 2. Стены и источники света останутся теми же, замене подлежат только персонаж и все сценарии. Если вы хотите заранее иметь новые варианты для сравнения, скачайте примеры файлов к данной главе.

Если вы начали с готового проекта из главы 2 (я имею в виду демонстрационный ролик, а не более позднюю версию), удалите все, что нам не нужно. Первым делом разорвите связь камеры с игроком на вкладке Hierarchy (это делается простым перетаскиванием). Затем удалите объект player; если бы камера оставалась его дочерним объектом, она при этом тоже исчезла бы, в то время как нам нужно избавиться только от выступающей в роли игрока капсулы, сохранив камеру. Если же вы случайно удалили камеру, создайте новую, выбрав в меню GameObject команду Camera.

Заодно удалите все сценарии (для этого потребуется избавить камеру от компонента script и удалить все файлы сценариев на вкладке Project). В сцене должны остаться только стены, пол и источники света.

7.1. Корректировка положения камеры

Перед тем как приступить к написанию кода, управляющего перемещениями персонажа, следует поместить персонажа в сцену и настроить камеру на отслеживание его положения. Мы импортируем в проект антропоморфную модель без лица и расположим камеру сверху таким образом, чтобы она смотрела на эту модель под углом. На рис. 7.3 сравнивается сцена в игре от первого лица с тем, что мы увидим в игре от третьего лица (во втором случае вы видите также крупные блоки, которые мы добавим в проект в этой главе).

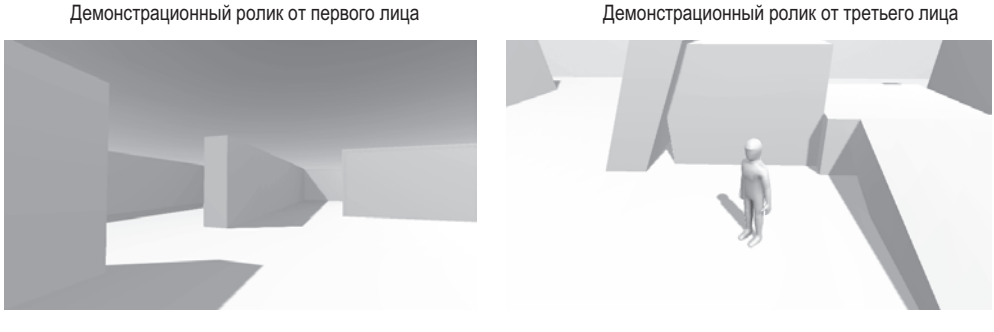


Рис. 7.3. Сравнительный вид сцены от первого и от третьего лица

Сцена уже полностью готова, осталось поместить в нее модель персонажа.

7.1.1. Импорт персонажа

Входящая в число материалов к данной главе папка scratch содержит как саму модель, так и текстуру; как вы узнали в главе 4, файл FBX — это модель, а файл TGA — это текстура. Импортируйте в проект файл FBX, перетащив его на вкладку Project или щелкнув на этой вкладке правой кнопкой мыши и выбрав в открывшемся меню команду Import New Asset. Теперь обратите внимание на панель Inspector, так как нам нужно скорректировать параметры импорта модели. Редактированием анимации вы займетесь чуть позже, а пока ограничимся парой параметров на вкладке Model. Первым делом присвойте параметру Scale Factor значение 10 (частично компенсируя слишком маленькое значение параметра File Scale), чтобы получить модель корректного размера.

Чуть ниже вы найдете параметр Normals (рис. 7.4). Он контролирует вид света и теней на модели, используя для этого такое математическое понятие, как *нормали*.

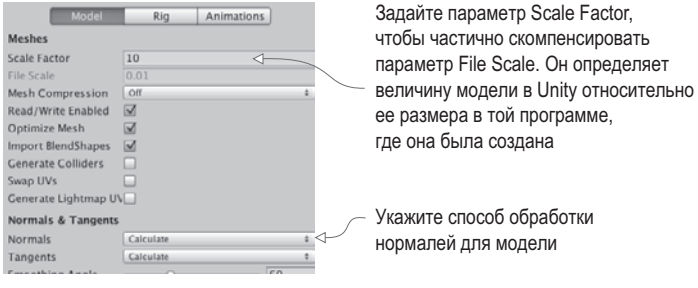


Рис. 7.4. Параметры импорта для модели персонажа

ОПРЕДЕЛЕНИЕ Нормалями (normals) называются перпендикулярные полигонам векторы направления, указывающие лицевую сторону полигонов. Именно это направление используется для вычисления освещенности.

По умолчанию параметр Normals имеет значение Import, что означает использование нормалей, задаваемых геометрией импортированной сетки. Однако нормали нашей модели определены некорректно, что дало бы странную реакцию на источники света. Поэтому мы присвоим данному параметру значение Calculate, заставив Unity вычислять вектор для лицевой стороны каждого полигона.

Завершив редактирование этих двух параметров, щелкните на кнопке Apply на панели Inspector. Затем импортируйте в проект файл TGA и сделайте полученное изображение текстурой материала. Для этого выделите материал player в папке Materials и на панели Inspector перетащите изображение текстуры на ячейку Albedo. Цвет модели после этого практически не изменится (эта текстура по большей части имеет белый цвет), но имеющиеся на текстуре тени улучшат внешний вид модели.

После назначения текстуры перетащите модель персонажа с вкладки Project в сцену. Введите в поля Position значения 0, 1.1, 0, чтобы персонаж оказался стоящим на полу в центре помещения. Итак, мы сделали первый шаг к созданию игры от третьего лица!

ПРИМЕЧАНИЕ Руки импортированного персонажа вытянуты в стороны, хотя естественнее было бы опустить их вниз. Это так называемая Т-образная поза, по умолчанию придаваемая всем анимируемым персонажам.

7.1.2. Добавление в сцену теней

Перед тем как двинуться дальше, хотелось бы осветить такой аспект, как отбрасываемая персонажем тень. Мы считаем появление теней само собой разумеющимся, но в виртуальном мире свои законы. К счастью, Unity в состоянии позаботиться о таких вещах, и источник света, по умолчанию присутствующий в любой новой сцене, приводит к формированию теней. Выделите на вкладке Hierarchy строку Directional Light и обратите внимание на параметр Shadow Type на панели Inspector. Он, как показано на рис. 7.5, имеет значение Soft Shadows (мягкие тени), при этом среди доступных значений есть и вариант No Shadows (без теней).

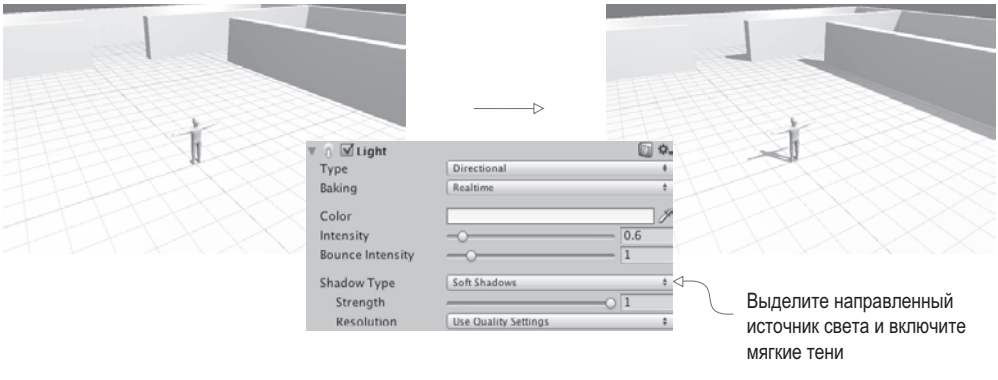


Рис. 7.5. Сцена до и после включения режима формирования теней от направленного источника света

Это все, что требуется для настройки теней в рамках данного проекта, но, разумеется, в целом создание теней в играх представляет собой куда более сложный процесс. Расчет теней в сцене является крайне затратной по времени операцией, поэтому для получения нужного эффекта зачастую идут на хитрости и различными способами имитируют нужные детали. Тени от персонажа относятся к так называемым *теням, визуализируемым в реальном времени* (real-time shadow), так как они вычисляются в процессе игры и двигаются вместе с объектом. Можно создать идеально реалистичное освещение, при котором все объекты будут отбрасывать тени и формировать тени на своих поверхностях в реальном времени, но обычно с целью ускорения расчетов в сценах накладывают ограничения как на вид теней, так и на количество источников света, приводящих к их формированию. Обратите внимание, что в нашей сцене тени формирует только направленный осветитель.

Другим распространенным приемом создания теней является применение *карт освещения*.

ОПРЕДЕЛЕНИЕ Картами освещения (lightmaps) называются текстуры, которые назначаются геометрии игровых уровней и в которых «запечено» изображение теней.

ОПРЕДЕЛЕНИЕ Рисование теней на текстуре модели называется запеканием теней (baking the shadows).

Так как получаемые изображения генерируются заранее (а не во время игры), они могут быть крайне детализированными и реалистичными. Разумеется, они полностью статичны и поэтому используются с неподвижной геометрией игрового уровня, но совершенно неприменимы к динамическим объектам, например персонажам. Карты освещения генерируются автоматически. Компьютер вычисляет, как имеющиеся в сцене источники света будут освещать уровень, оставляя в углах небольшое затемнение. Система визуализации карт освещения в Unity называется Enlighten. Используйте это название как ключевое поисковое слово в справочниках по Unity.

Делать выбор между визуализацией теней в реальном времени и картами освещения не нужно. Свойство источника света Culling Mask позволяет воспользоваться визуализацией в реальном времени только для определенных объектов, сымитировав детализированные тени от остальных объектов сцены с помощью карт освещения. Кроме того, хотя основной персонаж обычно в обязательном порядке отбрасывает тень, далеко не всегда нужно, чтобы на нем формировались тени от окружающих объектов. Поэтому для всех сеточных объектов можно включить функции отбрасывания и восприятия теней, как показано на рис. 7.6.

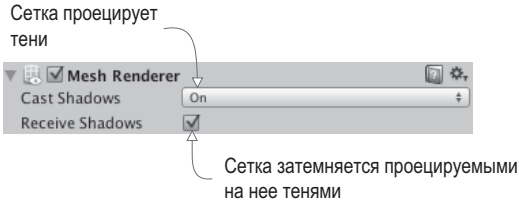


Рис. 7.6. Параметры Cast Shadows и Receive Shadows на панели Inspector

ОПРЕДЕЛЕНИЕ Термин «отбор» (culling) в общем случае означает исключение ненужных вещей. В посвященных компьютерной графике статьях он появляется в разных контекстах, в данном же случае свойство Culling mask определяет набор объектов, которые не будут отбрасывать тени.

Итак, вы познакомились с основами добавления теней в сцены. Освещение сцен и формирование теней — сама по себе обширная тема (в книгах, посвященных редактированию игровых уровней, ей зачастую посвящается не одна глава), но мы ограничимся визуализацией теней в реальном времени от одного источника света. И займемся настройкой камеры.

7.1.3. Облет камеры вокруг персонажа

В демонстрационном ролике от первого лица мы связали камеру с изображающим игрока объектом на вкладке Hierarchy, обеспечив их совместное вращение. В игре же от третьего лица персонаж будет поворачиваться в разные стороны независимо от камеры. Поэтому связывать камеру с персонажем на вкладке Hierarchy на этот раз не нужно. Вместо этого мы программно будем менять положение камеры вместе с положением персонажа, поворачивая ее независимо от последнего.

Первым делом выберите положение камеры относительно персонажа. Я ввел в поля position значения 0, 3.5, –3.75, расположив камеру чуть выше персонажа и за его спиной (в полях rotation при этом должны быть значения 0, 0, 0). После этого нужно создать сценарий `OrbitCamera` (его код показан в следующем листинге). Присоедините сценарий к камере в виде нового компонента, а затем перетащите персонажа player на ячейку Target. Запустите сцену, чтобы посмотреть, как работает код камеры.

Листинг 7.1. Сценарий вращения нацеленной на объект камеры вокруг объекта

```
using UnityEngine;
using System.Collections;

public class OrbitCamera : MonoBehaviour {
  [SerializeField] private Transform target; ← Сериализованная ссылка на объект, вокруг которого производится облет.

  public float rotSpeed = 1.5f;

  private float _rotY;
  private Vector3 _offset;

  void Start() {
    _rotY = transform.eulerAngles.y;
    _offset = target.position - transform.position; ← Сохранение начального смещения между камерой и целью.
  }

  void LateUpdate() {
    float horInput = Input.GetAxis("Horizontal");
    if (horInput != 0) { ← Медленный поворот камеры при помощи клавиш со стрелками...
      _rotY += horInput * rotSpeed;
    } else {
      _rotY += Input.GetAxis("Mouse X") * rotSpeed * 3; ← или быстрый поворот с помощью мыши.
    }

    Quaternion rotation = Quaternion.Euler(0, _rotY, 0);
    transform.position = target.position - (rotation * _offset); ← Поддерживаем начальное смещение, сдвигаемое в соответствии с поворотом камеры.
    transform.LookAt(target); ← Камера всегда направлена на цель, где бы относительно этой цели она ни располагалась.
  }
}
```

При изучении листинга обратите внимание на сериализованную переменную для целевого объекта. Код должен знать, вокруг какого объекта будет вращаться камера, поэтому данная переменная была сериализована, чтобы появиться в редакторе Unity и дать возможность связать с ней объект, изображающий персонаж. Следующая пара переменных связана с углами поворота и используется тем же способом, что и в коде управления камерой в главе 2. Еще код содержит объявление переменной `_offset`; в методе `Start()` ей присваивается разница в положении камеры и целевого объекта. Это позволяет в процессе выполнения кода сценария сохранять относительное положение камеры. Другими словами, камера все время будет оставаться на одном и том же расстоянии от целевого объекта, в какую бы сторону она ни поворачивалась. Остальная часть кода помещена в метод `LateUpdate()`.

СОВЕТ Метод LateUpdate() также относится к классу MonoBehaviour и подобно методу Update() запускается в каждом кадре. Как понятно из его имени, метод LateUpdate() вызывается для всех объектов после того, как с ними поработал метод Update(). Таким способом мы обеспечиваем обновление камеры только после перемещения целевого объекта.

Первым делом код увеличивает угол поворота в зависимости от элементов управления вводом. Так как рассматриваются элементы двух типов: клавиши с горизонтальными стрелками и горизонтальные перемещения указателя мыши, — для переключения между ними служит условная инструкция. Код проверяет нажатие клавиш со стрелками; в случае положительного результата проверки применяется этот тип ввода, в противном случае проверяется указатель мыши. Независимая проверка двух вариантов ввода позволяет в каждом случае задать собственную скорость вращения.

Затем код задает положение камеры на основе положения целевого объекта и угла поворота. Строка `transform.position`, скорее всего, является самым непонятным фрагментом этого кода, так как содержит важные математические вычисления, с которыми вы пока не сталкивались. Умножение вектора `position` на кватернион (обратите внимание, что угол поворота преобразован в кватернион методом `Quaternion.Euler`) дает нам новое положение, смещенное в соответствии с углом поворота. Этот новый вектор положения затем добавляется к смещению от положения персонажа, что дает нам положение камеры. Этапы и подробный механизм вычислений этой компактной строки кода иллюстрирует рис. 7.7.

Рис. 7.7. Этапы вычисления положения камеры

ПРИМЕЧАНИЕ Наиболее математически одаренные читатели могут подумать: «Нельзя ли воспользоваться преобразованиями от одной координатной системы к другой, о которых шла речь в главе 2?» Разумеется, мы можем преобразовать положение точки смещения, воспользовавшись углами Эйлера, но для этого сначала нужно будет перейти к другой системе координат. Намного проще без этого обойтись.

Завершает код метод `LookAt()`, направляющий камеру на целевой объект; он предназначен для нацеливания одного объекта (не обязательно камеры) на другой. Вычисленная ранее величина поворота используется для размещения камеры под нужным углом к целевому объекту, но при этом камера уже не поворачивается, меняется только ее положение. То есть без заключительной строки с методом `LookAt` камера летала бы вокруг персонажа, но при этом смотрела бы в разные стороны. Попробуйте превратить эту строку в комментарий и посмотреть, что получится.

Итак, мы написали сценарий, перемещающий камеру вокруг персонажа; пришла очередь кода, перемещающего персонаж по сцене.

7.2. Элементы управления движением, связанные с камерой

Теперь, когда мы импортировали в Unity модель персонажа и написали код, управляющий видом с камеры, пришло время создать элементы управления перемещением персонажа по сцене. Запрограммируем элементы управления, связанные с камерой, которые будут перемещать персонажа в разных направлениях при нажатии клавиш со стрелками, а также поворачивать его лицом в нужную сторону.

ЧТО ОЗНАЧАЕТ «СВЯЗАННЫЕ С КАМЕРОЙ»?

Принцип «связи с камерой» не вполне очевиден, но понять его крайне важно. Во многом он напоминает уже знакомую вам ситуацию с локальными и глобальными координатами, когда «лево» с точки зрения объекта может не совпадать с «лево» в общем смысле. Аналогичная ситуация возникает с выражением «персонаж бежит налево». Оно может подразумевать как левую сторону с точки зрения персонажа, так и левую сторону экрана.

В игре от первого лица камера располагается внутри персонажа и перемещается вместе с ним, поэтому понятие «лево» с точки зрения камеры и персонажа одно и то же. А вот в игре от третьего лица камера и персонаж существуют отдельно друг от друга, поэтому, к примеру, в ситуации, когда камера смотрит персонажу в лицо, лево с точки зрения камеры и с точки зрения персонажа располагается в противоположных направлениях. Именно поэтому нужно заранее выбрать настройки элементов управления.

Можно встретить разные варианты, но, как правило, в играх от третьего лица элементы управления настраиваются относительно камеры. При нажатии клавиши с левой стрелкой персонаж бежит по экрану налево. Многочисленные эксперименты с другими схемами элементов управления показали, что интуитивно понятным является именно вариант, когда «лево» означает «левую сторону экрана» (что далеко не всегда совпадает с левой стороной игрока).

Реализация связанных с камерой элементов управления состоит из двух этапов: сначала мы ориентируем персонажа в соответствии с положением элементов управления, а затем двигаем его вперед. Давайте напишем соответствующий код.

7.2.1. Поворот персонажа лицом в направлении движения

Начнем мы с написания кода, поворачивающего персонажа лицом в направлении клавиш со стрелками. Создайте сценарий `RelativeMovement` на C# (листинг 7.2). Перетащите этот сценарий на объект player и свяжите камеру со свойством Target компонента Script (таким же образом, как вы связывали персонажа со свойством Target сценария камеры). После этого персонаж при нажатии управляющих клавиш будет поворачиваться в разные стороны, выбирая направление относительно камеры, а при его вращении с помощью мыши он останется статичным.

Листинг 7.2. Поворот персонажа относительно камеры

```
using UnityEngine;
using System.Collections;

public class RelativeMovement : MonoBehaviour {
  [SerializeField] private Transform target; ← Сценарию нужна ссылка на объект, относительно которого будет происходить перемещение.

  void Update() {
    Vector3 movement = Vector3.zero; ← Начинаем с вектора (0, 0, 0), непрерывно добавляя компоненты движения.

    float horInput = Input.GetAxis("Horizontal");
    float vertInput = Input.GetAxis("Vertical");
    if (horInput != 0 || vertInput != 0) { ← Движение обрабатывается только при нажатии клавиш со стрелками.
      movement.x = horInput;
      movement.z = vertInput;

      Quaternion tmp = target.rotation; ← Сохраняем начальную ориентацию, чтобы вернуться к ней после завершения работы с целевым объектом.
      target.eulerAngles = new Vector3(0, target.eulerAngles.y, 0);
      movement = target.TransformDirection(movement); ← Преобразуем направление движения из локальных в глобальные координаты.
      target.rotation = tmp;

      transform.rotation = Quaternion.LookRotation(movement); ← Метод LookRotation() вычисляет кватернион, смотрящий в этом направлении.
    }
  }
}
```

Аналогично коду листинга 7.1, код этого листинга начинается с сериализованной переменной для целевого объекта. Если в предыдущем случае нам требовалась ссылка на объект, вокруг которого будет совершаться облет, то теперь нужна ссылка на объект, относительно которого будет выполняться перемещение. Затем мы переходим к методу `Update()`. Его первая строка объявляет, что `Vector3` имеет значение 0, 0, 0. Важно взять нулевой вектор и затем присвоить ему значения, а не просто создать вектор с вычисленными значениями перемещения. Дело в том, что величины перемещения по вертикали и горизонтали вычисляются на разных этапах, но при этом должны быть частями одного и того же вектора.

Далее мы проверяем элементы управления вводом, как делали это в предыдущих сценариях. Именно здесь вектору движения присваиваются значения координат X и Z, определяющие горизонтальные перемещения по сцене. Помните, что метод `Input.GetAxis()` возвращает значение 0, если ни одна клавиша не нажата, а при нажатии клавиш со стрелками его значение меняется от 1 до –1. Присваивание этого значения

вектору движения задает перемещение в положительном или отрицательном направлении оси (в случае оси *X* это перемещение влево и вправо, в случае оси *Z* — вперед и назад).

В следующих нескольких строках вектор движения корректируется относительно камеры. А именно метод `TransformDirection()` выполняет преобразование локальных координат в глобальные. Мы уже пользовались им для этой цели в главе 2, но на этот раз мы совершаем переход от системы координат целевого объекта, а не игрока. При этом код до и после строки с методом `TransformDirection()` выравнивает систему координат нужным нам образом: первым делом сохраняется ориентация целевого объекта, чтобы потом к ней можно было вернуться, а затем преобразование поворота корректируется таким образом, чтобы оно совершалось только относительно оси *Y*, а не всех трех осей. После чего мы выполняем преобразование и восстанавливаем ориентацию целевого объекта.

Весь этот код вычисляет направление движения в виде вектора. Последняя строка применяет полученный результат к персонажу, преобразуя `Vector3` в `Quaternion` методом `Quaternion.LookDirection()` и присваивая это значение. Попробуйте запустить игру и посмотреть, что получится!

ПЛАВНОЕ ВРАЩЕНИЕ С ПОМОЩЬЮ АЛГОРИТМА ЛИНЕЙНОЙ ИНТЕРПОЛЯЦИИ

В настоящее время наш персонаж меняет ориентацию скачками, мгновенно поворачиваясь лицом в направлении, заданном клавишей. Но плавное движение смотрелось бы куда лучше. В этом нам поможет линейный алгоритм интерполяции. Добавьте в сценарий такую переменную:

```
public float rotSpeed = 15.0f;
```

Затем замените строку transform.rotation... в конце листинга 7.2 следующим кодом:

```
...
      Quaternion direction = Quaternion.LookRotation(movement);
      transform.rotation = Quaternion.Lerp(transform.rotation,
      direction, rotSpeed * Time.deltaTime);
    }
  }
}
```

В результате вместо непосредственной привязки к значению, возвращаемому методом LookRotation(), это значение начнет использоваться в качестве целевого положения, в которое объект будет постепенно поворачиваться. Метод Quaternion.Lerp() выполняет плавный поворот из текущего положения в целевое (третий параметр метода контролирует скорость вращения).

Кстати, плавный переход от одного значения к другому называется интерполяцией; интерполировать можно значения любых типов. Термин Lerp расшифровывается как linear interpolation — линейная интерполяция. В Unity есть методы Lerp как для векторов, так и для значений типа float (то есть вы можете интерполировать положение объектов, цвета и другие параметры). Для кватернионов имеется также родственный метод Slerp (метод сферической линейной интерполяции). Для поворотов на маленькие углы преобразования Slerp зачастую подходят лучше, чем Lerp.

Пока что персонаж умеет только поворачиваться на месте; в следующем разделе мы напишем код, который заставит его перемещаться по сцене.

ПРИМЕЧАНИЕ Так как поворотом налево и направо управляют клавиши, отвечающие за облет камеры, поворот персонажа в сторону будет сопровождаться медленным вращением. Подобное дублирование функций элементов управления является желательным поведением для данного проекта.

7.2.2. Движение вперед в выбранном направлении

Как вы помните из главы 2, для перемещения персонажа по сцене к нему нужно добавить соответствующий контроллер. Выделите персонажа и выберите в меню Components команду Physics ▸ Character Controller. На панели Inspector уменьшите параметр Radius до 0.4, остальным же параметрам оставьте значения, предлагаемые по умолчанию, так как они вполне подходят для нашей модели персонажа.

Следующий листинг содержит код, который нужно добавить в сценарий `RelativeMovement`.

Листинг 7.3. Код, меняющий положение персонажа

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))] ← Окружающие строки показывают контекст размещения метода RequireComponent().
public class RelativeMovement : MonoBehaviour {
  ...
  public float moveSpeed = 6.0f;

  private CharacterController _charController;

  void Start() {
    _charController = GetComponent<CharacterController>(); ← Этот паттерн, знакомый вам по предыдущим главам, используется для доступа к другим компонентам.
  }

    void Update() {
      ...
      movement.x = horInput * moveSpeed; ← Переписываем существующие строки для X и Z, чтобы добавить скорость движения.
      movement.z = vertInput * moveSpeed;
      movement = Vector3.ClampMagnitude(movement, moveSpeed); ← Ограничиваем движение по диагонали той же скоростью, что и движение вдоль оси.
      ...
    }

    movement *= Time.deltaTime; ← Не забывайте умножать перемещения на значение deltaTime, чтобы они не зависели от частоты кадров.
    _charController.Move(movement);
  }
}
```

Запустив игру, вы увидите, как персонаж (в Т-образной позе) перемещается по сцене. Практически весь код этого листинга вы уже видели, поэтому я ограничусь кратким обзором.

В первую очередь, в верхней части кода появился метод `RequireComponent()`. Как объяснялось в главе 2, этот метод заставляет Unity проверять наличие у объекта GameObject компонента переданного в команду типа. Хотя эта строка не обязательна, без указанного компонента в сценарии появятся ошибки.

Затем следует объявление переменной для перемещений, после которого сценарий получает ссылку на контроллер персонажа. Как вы помните, метод `GetComponent()` возвращает остальные присоединенные к этому объекту компоненты, и если объект не указывается явным образом, подразумевается вариант `this.GetComponent()`, то есть объект, с которым работает сценарий.

Величины перемещений присваиваются с помощью элементов управления вводом. Этот прием фигурировал и в предыдущем листинге, просто сейчас мы учитываем еще и скорость перемещения. Мы умножаем обе оси, вдоль которых происходит движение, на его скорость, а затем методом `Vector3.ClampMagnitude()` ограничиваем модуль вектора этой скоростью; иначе движение по диагонали будет иметь больший вектор, чем движение вдоль осей (нарисуйте катеты и гипотенузу прямоугольного треугольника).

Напоследок мы умножаем значения перемещения на параметр `deltaTime`, чтобы сделать данное преобразование независимым от частоты кадров (напоминаю, что это означает перемещение персонажа с одной и той же скоростью на разных компьютерах с разной частотой кадров). Полученные значения передаются методу `CharacterController.Move()`, который и приводит персонажа в движение.

Этот код обеспечивает нам перемещения в горизонтальном направлении, нам же нужно заставить персонаж перемещаться еще и по вертикали.

7.3. Выполнение прыжков

В предыдущем разделе мы написали код перемещения персонажа по поверхности. Но во введении к данной главе упоминалась возможность совершать прыжки. Давайте ее добавим. Тем более что в большинстве игр от третьего лица есть соответствующий элемент управления. И даже при его отсутствии перемещение по вертикали поддерживается практически всегда, так как персонаж может сорваться вниз, например, в процессе преодоления пропасти. Мы добавим код как прыжка, так и падения. Точнее говоря, этот код будет учитывать силу тяжести, все время тянущую персонажа вниз, а в момент прыжка к нему будет применяться толчок вверх.

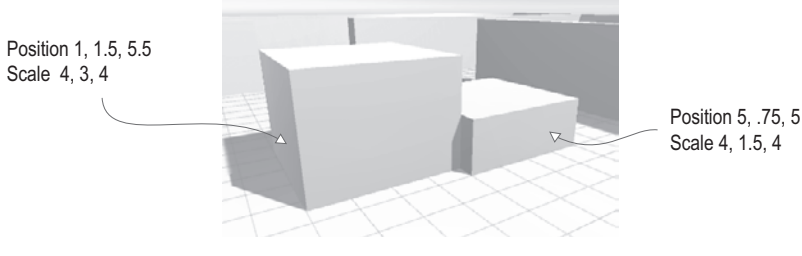


Рис. 7.8. Пара платформ, добавленных в сцену

Но сначала мы добавим в сцену пару возвышений. Ведь пока персонажу некуда запрыгивать и неоткуда падать! Создайте несколько кубов, поменяйте их размер по собственному вкусу и расположите в сцене. Лично я, как показано на рис. 7.8, добавил два куба со следующими параметрами:

- Position: 5, 0.75, 5, Scale: 4, 1.5, 4;
- Position: 1, 1.5, 5.5, Scale: 4, 3, 4.

7.3.1. Добавление вертикальной скорости и ускорения

Как я говорил, когда мы только начинали работать над сценарием `RelativeMovement`, значения перемещений вычисляются по отдельности и все время добавляются к вектору перемещения. Следующий листинг добавит к этому вектору движение по вертикали.

Листинг 7.4. Добавление движения по вертикали в сценарий RelativeMovement

```
...
public float jumpSpeed = 15.0f;
public float gravity = -9.8f;
public float terminalVelocity = -10.0f;
public float minFall = -1.5f;

private float _vertSpeed;
...
void Start() {
  _vertSpeed = minFall; ← Инициализируем скорость по вертикали, присваивая ей минимальную
                          скорость падения в начале существующей функции.
  ...
}

void Update() {
  ...
    if (_charController.isGrounded) { ← Свойство isGrounded компонента CharacterController проверяет,
                                        соприкасается ли контроллер с поверхностью.
      if (Input.GetButtonDown("Jump")) { ← Реакция на кнопку Jump при нахождении на поверхности.
        _vertSpeed = jumpSpeed;
      } else {
        _vertSpeed = minFall;
      }
    } else { ← Если персонаж не стоит на поверхности, применяем гравитацию, пока не будет достигнута предельная скорость.
      _vertSpeed += gravity * 5 * Time.deltaTime;
      if (_vertSpeed < terminalVelocity) {
        _vertSpeed = terminalVelocity;
      }
    }

    movement.y = _vertSpeed;
    movement *= Time.deltaTime; ← Конец листинга 7.3, чтобы вы могли понять, куда вставлять новый код.
    _charController.Move(movement);
  }
}
```

Как обычно, мы начинаем с добавления в верхнюю часть сценария новых переменных для различных значений скорости и их корректной инициализации. Весь код до большой инструкции `if`, задающей перемещения по горизонтали, мы оставляем, а потом добавляем еще одну большую инструкцию `if`, задающую перемещения по вертикали. Этот новый фрагмент кода проверяет, стоит ли персонаж на поверхности, так как именно от этого зависит изменение вертикальной скорости. Это нам позволит установить значение свойства `isGrounded` компонента `CharacterController`; свойство

имеет значение `true`, когда нижняя часть контроллера персонажа сталкивается в последнем кадре с любым объектом.

Если персонаж стоит на поверхности, значение вертикальной скорости (это закрытая переменная `_vertSpeed`) становится нулевым. Раз персонаж не падает, очевидно, что его вертикальная скорость равна нулю; если после этого персонаж спрыгнет с края платформы, мы получим вполне натуральное падение, так как его скорость начнет расти от нуля.

ПРИМЕЧАНИЕ На самом деле присваиваемое нами начальное значение вертикальной скорости равно не нулю, а значению minFall, означающему небольшое движение вниз, которое в процессе перемещений по горизонтали будет слегка придавливать персонажа к поверхности. Небольшая направленная вниз сила нужна для перемещений вверх и вниз по пересеченной местности.

Исключением является ситуация нажатия клавиши, отвечающей за прыжок. В этом случае вертикальная скорость должна увеличиться. Инструкция `if` проверяет результат метода `GetButtonDown()` — новой функции ввода, во многом аналогичной методу `GetAxis()`. Этот метод также возвращает состояние элемента управления вводом. Клавиша, отвечающая за прыжок, выбирается на панели Inspector, как и клавиши перемещения вдоль горизонтальной и вертикальной осей. Чтобы открыть нужный набор настроек, следует выбрать в меню Edit команду Project Settings ▸ Input (по умолчанию эта настройка имеет значение Space, то есть прыжок совершается путем нажатия клавиши пробела).

Если же персонаж не стоит на поверхности, вертикальная скорость должна непрерывно уменьшаться под действием силы тяжести. Обратите внимание, что код не присваивает значение скорости, а производит ее декремент; то есть, по сути, мы получаем направленное вниз ускорение, благодаря которому возникает реалистичное падение. Прыжки также происходят по дуге, так как скорость перемещения персонажа вверх постепенно уменьшается до нуля, и он начинает падать.

При этом код гарантирует, что скорость движения вниз не превысит предельного значения. Обратите внимание, что для этого используется оператор «меньше, чем», а не «больше чем», так как скорость движения вниз имеет отрицательное значение. После большой инструкции `if` рассчитанная вертикальная скорость присваивается оси *Y* вектора движения.

Это все, что требуется для моделирования реалистичного перемещения по вертикали! Применяя направленное вниз постоянное ускорение, когда персонаж не касается поверхности, и правильно корректируя скорость при его расположении на поверхности, код прекрасно имитирует падение. Но все это работает только при корректном распознавании поверхности. И здесь есть одна тонкость, о которой мы поговорим в следующем разделе.

7.3.2. Распознавание поверхности с учетом краев и склонов

Как вы узнали в предыдущем разделе, свойство `isGrounded` компонента `CharacterController` показывает, сталкивалась ли в последнем кадре нижняя часть контроллера персонажа с каким-либо объектом. В большинстве случаев этот подход дает прекрасные результаты, но, возможно, вы уже заметили, что при попытке сойти с края

платформы персонаж как бы зависает в воздухе. Дело в том, что область столкновений персонажа представляет собой окружающую его капсулу (это можно увидеть, выделив данный объект), и нижняя часть этой капсулы остается в контакте с поверхностью в начале движения за пределы платформы, как показано на рис. 7.9.

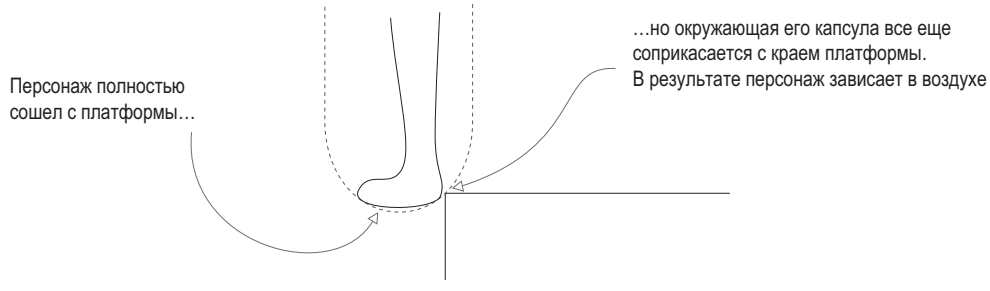


Рис. 7.9. Схема, демонстрирующая контакт капсулы контроллера и края платформы

Текущий вариант распознавания поверхности приводит к сходным проблемам и при попадании персонажа на наклонную плоскость. Создайте наклонный блок, опирающийся на одну из платформ. Например, у меня это был куб, для которого значения Position составили –1.5, 1.5, 5, Rotation — 0, 0, –25, а Scale — 1, 4, 4.

При попытке запрыгнуть на такой склон с поверхности ничто не помешает вам после первого прыжка совершить второй и оказаться наверху. Ведь склон соприкасается с фрагментом капсулы, а код в настоящее время рассматривает любые касания нижней части как наличие твердой точки опоры. Но это совершенно нереалистичное поведение; персонаж после первого прыжка должен скатиться по склону вниз.

ПРИМЕЧАНИЕ Персонаж должен скатываться вниз только с крутых склонов. Пологие склоны, например неровности почвы, следует пробегать, не обращая на них внимания. Если вы хотите получить такой вариант рельефа для тестирования, создайте куб и присвойте параметру Position значения 5.25, 0.25, 0.25, параметру Rotation — значения 0, 90, 75, параметру Scale — значения 1, 6, 3.

У всех этих проблем одна и та же причина: распознавание столкновений нижней частью персонажа не позволяет однозначно определить его положение по отношению к поверхности. Давайте вместо этого попробуем распознать поверхность методом бросания луча. В главе 3 именно так наш искусственный интеллект определял наличие перед ним препятствий; используем этот подход, чтобы выяснить тип поверхности под персонажем. Луч следует бросать вниз из точки, в которой он находится. Столкновение, зарегистрированное непосредственно под ногами персонажа, будет означать, что он стоит на поверхности.

В результате нам предстоит иметь дело с ситуацией, когда луч показывает, что поверхность под ногами персонажа отсутствует, а контроллер сталкивается поверхностью. На рис. 7.9 капсула все еще соприкасается с платформой, в то время как персонаж уже стоит за ее пределами. На рис. 7.10 в эту схему вводится метод бросания луча, в результате имеет место другой вариант развития событий: луч не сталкивается

с платформой, в то время как капсула касается ее края. Код должен каким-то образом обработать эту ситуацию.

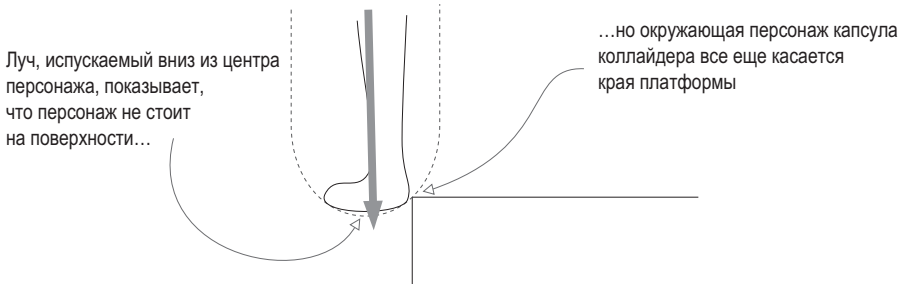


Рис. 7.10. Схема бросания лучей вниз при сходе с края платформы

В данном случае персонаж должен скользить по склону вниз. Он все еще будет падать (так как не стоит на поверхности), но при этом начнет отваливаться от точки столкновения (так как капсулу нужно отодвинуть от платформы, с которой она соприкасается). Соответственно, код распознает столкновение с помощью контроллера персонажа и отреагирует на него небольшим толчком в сторону.

Следующий листинг добавляет в вертикальное движение все упомянутые нами аспекты.

Листинг 7.5. Распознавание поверхности методом бросания луча

```
...
private ControllerColliderHit _contact; ← Нужно для сохранения данных о столкновении между функциями.
...
      bool hitGround = false;
      RaycastHit hit;
      if (_vertSpeed < 0 && ← Проверяем, падает ли персонаж.
      Physics.Raycast(transform.position, Vector3.down, out hit)) {
        float check = ← Расстояние, с которым производится сравнение (слегка выходит за нижнюю часть капсулы).
          (_charController.height + _charController.radius) / 1.9f;
        hitGround = hit.distance <= check;
      }

      if (hitGround) { ← Вместо проверки свойства isGrounded смотрим на результат бросания луча.
        if (Input.GetButtonDown("Jump")) {
          _vertSpeed = jumpSpeed;
        } else {
          _vertSpeed = minFall;
        }
      } else {
        _vertSpeed += gravity * 5 * Time.deltaTime;
        if (_vertSpeed < terminalVelocity) {
          _vertSpeed = terminalVelocity;
        }

        if (_charController.isGrounded) { ← Метод бросания луча не обнаруживает
                                            поверхности, но капсула с ней соприкасается.
```

```
          if (Vector3.Dot(movement, _contact.normal) < 0) {
            movement = _contact.normal * moveSpeed;
          } else {
            movement += _contact.normal * moveSpeed;
          }
        }
      }
      movement.y = _vertSpeed;

      movement *= Time.deltaTime;
      _charController.Move(movement);
    }

    void OnControllerColliderHit(ControllerColliderHit hit) {
    _contact = hit;
  }
}
```

Реакция слегка меняется в зависимости от того, смотрит ли персонаж в сторону точки контакта.

При распознавании столкновения данные этого столкновения сохраняются в методе обратного вызова.

Этот листинг содержит практически тот же самый код, что и предыдущий; новый код перемешан с существующим сценарием движения, а крупные фрагменты предыдущего листинга включены с целью задания контекста. Первая строка добавляет новую переменную в верхнюю часть сценария `RelativeMovement`. Эта переменная используется для сохранения данных о столкновении между вызовами функций.

Следующие несколько строк касаются бросания луча. Этот код следует вставить после кода движения по горизонтали, но перед инструкцией `if`, которая задает вертикальное движение. С вызовом метода `Physics.Raycast()` вы уже сталкивались, но на этот раз мы используем другой набор параметров. Хотя луч бросается из той же самой точки (местоположения персонажа), он направляется не вперед, а вниз. Затем мы смотрим на расстояние, пройденное до момента столкновения; если оно совпадает с расстоянием до ступней персонажа, значит, персонаж стоит на поверхности и свойству `hitGround` можно присвоить значение `true`.

ВНИМАНИЕ Процесс вычисления расстояния не вполне очевиден, поэтому рассмотрим его более подробно. Мы берем высоту контроллера персонажа (то есть его рост без скругленных углов) и добавляем к ней скругленные углы. Полученное значение делится пополам, так как луч бросается из центра персонажа и проходит расстояние до его стоп. Но мы хотим проверить чуть более длинную дистанцию, чтобы учесть небольшие неточности бросания луча, поэтому высота персонажа делится на 1.9, а не на 2, в итоге луч проходит несколько большее расстояние.

Далее в инструкции `if` для движения по вертикали вместо `isGrounded` используется свойство `hitGround`. Большая часть кода, управляющая перемещениями в вертикальном направлении, остается без изменений, мы просто добавляем туда код, обрабатывающий ситуацию, когда контроллер персонажа соприкасается с поверхностью, хотя персонаж на ней не стоит (то есть момент схода с края платформы). Здесь добавляется еще одна условная инструкция со свойством `isGrounded`, но обратите внимание, что она вложена в инструкцию проверки свойства `hitGround`. То есть свойство `isGrounded` проверяется только при условии, что свойство `hitGround` показало отсутствие поверхности.

Данные о столкновении включают в себя свойство `normal` (еще раз напомню, что нормалью называется вектор, определяющий лицевую сторону полигона), указывающее направление ухода от точки столкновения. Но есть один тонкий момент. Мы хотим, чтобы небольшой импульс в сторону от точки контакта обрабатывался в зависимости от того, в каком направлении двигался персонаж. Если предшествующее движение по горизонтали совершалось в сторону платформы, его следует заменить, чтобы персонаж не продолжал смещаться в некорректном направлении. Если же лицом он был повернут в сторону от края, предшествующее движение по горизонтали нужно добавить, чтобы сохранить толкающий вперед импульс. Направление вектора движения относительно точки столкновения определяется при помощи *скалярного произведения*.

ОПРЕДЕЛЕНИЕ Скалярным произведением (dot product) называется математическая операция, выполняемая с двумя векторами. Коротко говоря, результат такого произведения варьируется между –1 и 1, где 1 означает, что вектора смотрят в одном направлении, в то время как –1 указывает на диаметрально противоположные направления. Не следует путать скалярное произведение с векторным произведением — другой математической операцией, часто применяемой к векторам.

Объект `Vector3` обладает методом `Dot()`, вычисляющим скалярное произведение двух векторов. Скалярное произведение вектора движения и нормали плоскости столкновения будет отрицательным, если эти вектора смотрят в разные стороны, и положительным при их одинаковом направлении.

В конце листинга 7.5 появляется новый метод. Ранее мы проверяли нормаль плоскости столкновения, но откуда появлялась информация об этой нормали? Оказывается, о столкновениях с контроллером персонажа сообщалось через функцию обратного вызова `OnControllerColliderHit()`, предоставляемую классом `MonoBehaviour`; для программирования реакции на данные о столкновении в каком-либо фрагменте сценария следует сохранить эти данные во внешней переменной. Именно этим занимается наш метод: сохраняет данные о столкновении в переменной `_contact`, предоставляя нам возможность воспользоваться этими данными в методе `Update()`.

Итак, мы скорректировали поведение персонажа на краях платформ и на наклонной плоскости. Запустите игру и посмотрите, что происходит при сходе с края и при попытке запрыгнуть на крутой склон. Наш демонстрационный ролик почти готов. Персонаж нужным образом перемещается по сцене, осталось только придать ему более естественную позу.

7.4. Анимация персонажа

Кроме более сложной формы, определенной сеточной геометрией, наш антропоморфный персонаж нуждается еще и в анимации. В главе 4 вы узнали, что анимация представляет собой пакет информации, определяющий движение связанного с ним трехмерного объекта. В качестве примера мы рассматривали ходящего по сцене персонажа. Именно эту ситуацию вам сейчас предстоит воспроизвести! Персонажу следует назначить анимацию, которая заставит его руки и ноги двигаться взад и вперед, как показано на рис. 7.11.

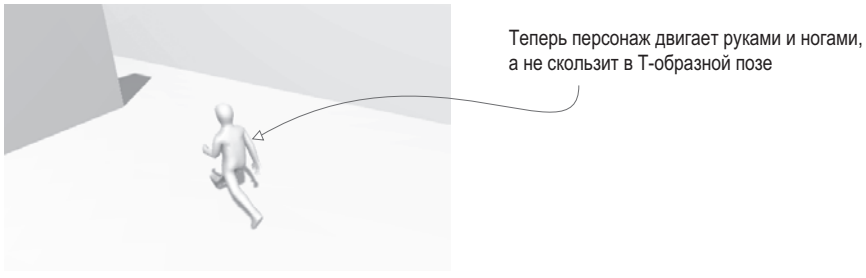


Рис. 7.11. Персонаж, двигающийся по сцене в процессе воспроизведения анимации

Хорошей аналогией трехмерной анимации является кукольный театр: трехмерные модели играют роль кукол, аниматор представляет собой кукловода, а анимация — это запись перемещений кукол. Существуют разные способы создания анимации; в большинстве современных игр для персонажей используется техника *скелетной анимации*.

ОПРЕДЕЛЕНИЕ Скелетной анимацией (skeletal animation) называется процедура связывания с моделью набора костей, которые и осуществляют движение. В процессе перемещения кости двигается также связанная с ней поверхность модели.

Принцип скелетной анимации легко понять интуитивно, представив имитацию скелета персонажа (как показано на рис. 7.12). Но при этом «скелет» является абстракцией, применимой в любой ситуации, в которой вам нужна гибкая и деформирующаяся модель с четко определенной структурой, программирующей ее перемещения (к примеру, представьте извивающееся щупальце осьминога). Сами кости перемещаются как недеформируемые объекты, в то время как окружающая их поверхность модели может сгибаться и менять свою форму.

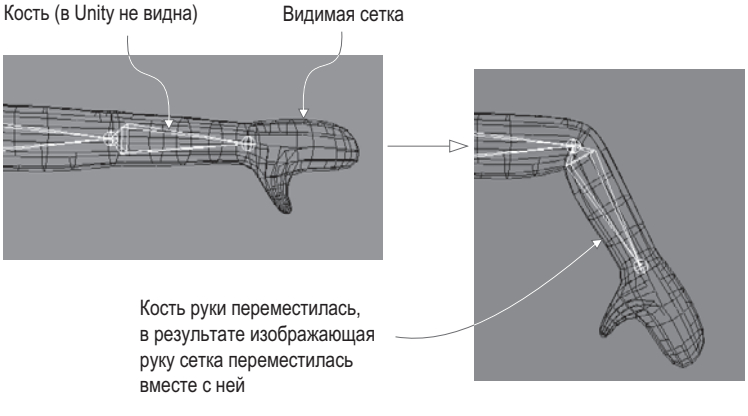


Рис. 7.12. Скелетная анимация антропоморфного персонажа

Получение результата, показанного на рис. 7.11, включает в себя несколько этапов: для начала мы определяем анимационные клипы в импортированном файле, затем настраиваем контроллер для воспроизведения этих клипов и, наконец, вставляем

этот контроллер в наш код. Анимация персонажа будет воспроизводиться в соответствии с написанными вами сценариями движения.

Разумеется, первое, с чего нужно начать, — это подключить систему анимации. Выделите модель персонажа на вкладке Project и на панели Inspector перейдите на вкладку Animations, где убедитесь в наличии флажка Import Animation. Затем перейдите на вкладку Rig и выберите в раскрывающемся списке Animation Type вариант Humanoid (ведь у нас же антропоморфный персонаж). Обратите внимание на вариант Legacy; до появления системы Mecanim настройки Generic и Humanoid были объединены.

СИСТЕМА АНИМАЦИИ MECANIM

В Unity встроена сложная система управления анимацией моделей, которая называется Mecanim. Ее основу составляет скелетная анимация, с которой вы познакомитесь в этой главе. Имя Mecanim указывает, что это более новая усовершенствованная система анимации, добавленная как замена старой версии. Последняя до сих пор доступна через настройку Legacy, но есть вероятность, что в следующих версиях Unity ее уже не будет.

Хотя анимация, которую мы будем задействовать, включена в тот же самый FBX-файл, что и модель персонажа, одним из основных преимуществ системы Mecanim является возможность наложения анимации из других FBX-файлов. Например, все враги могут пользоваться одним набором анимационных клипов. Такой подход имеет ряд преимуществ, в числе прочего предоставляя возможность структурированного хранения данных (модели сохраняются в одной папке, в то время как анимационные ролики — в другой) и позволяя экономить время за счет анимации всех персонажей разом.

Щелкните на кнопке Apply в нижней части панели Inspector, чтобы применить выбранные настройки к импортированной модели, и продолжите работу над определением анимационных клипов.

ВНИМАНИЕ На консоли может появиться предостережение (не ошибка) с текстом «conversion warning: spine3 is between humanoid transforms». На него можно не обращать внимания; он информирует о наличии в импортированной модели большего количества костей, чем ожидалось системой Mecanim.

7.4.1. Создание анимационных клипов для импортированной модели

Первым шагом по созданию анимации для персонажа будет подготовка отдельных клипов. Человек совершает разные движения в разное время. Аналогично и наш персонаж будет то бегать по сцене, то запрыгивать на платформы, то просто стоять с опущенными руками. Каждое из этих движений представляет собой «клип», который можно воспроизводить независимо.

Импортированная анимация зачастую выглядит как один длинный клип, который можно превратить в набор отдельных анимационных роликов. Это делается на вкладке Animations панели Inspector. Вы увидите там список Clips, показанный на рис. 7.13, — здесь перечислены все анимационные клипы, изначально входившие в состав единого импортированного клипа. Обратите внимание на кнопки со значками + (плюс) и – (минус) в нижней части этого списка; они предназначены для добавления и удаления клипов. Для нашего персонажа потребуется четыре клипа, поэтому добавляйте и удаляйте их по мере необходимости.

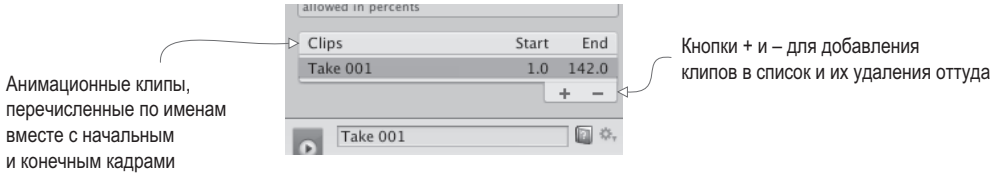


Рис. 7.13. Список Clips на вкладке Animation

После выбора клипа информация о нем появляется ниже, как показано на рис. 7.14. Первым идет поле с именем клипа, причем вы можете ввести новое имя. Присвойте нашему первому клипу имя idle. Теперь нужно задать параметры Start и End, определяющие первый и последний кадр выбранного клипа. Именно это позволит вырезать фрагмент из длинной импортированной анимации. Для клипа idle введите в поле Start значение 3, а в поле End — значение 141. Затем переходите к настройкам цикла.

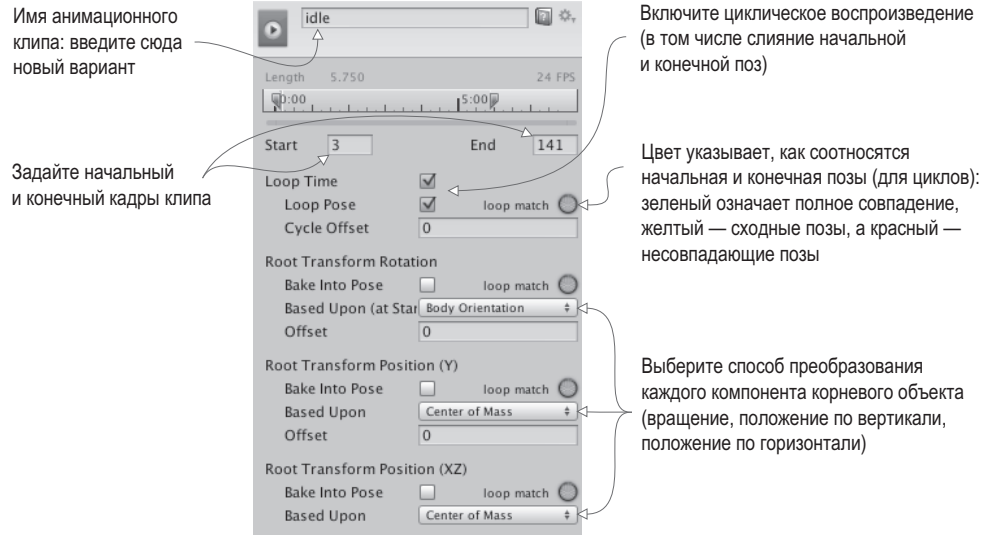


Рис. 7.14. Информация о выбранном анимационном клипе

ОПРЕДЕЛЕНИЕ Цикл (loop) означает запись, которая воспроизводится снова и снова. Зацикленный анимационный клип сразу же после завершения воспроизведения запускается снова.

Наш клип должен воспроизводиться в цикле, поэтому установите флажки Loop Time и Loop Pose. Кстати, зеленый цвет индикаторной точки указывает, что позы персонажа в начале и в конце клипа совпадают, что означает корректность циклического воспроизведения. В случае отдельных несовпадений индикатор становится желтым, красный же цвет указывает на совершенно разные позы.

Ниже находится набор параметров, связанных с преобразованиями корневого элемента. Слово *корневой* в случае скелетной анимации имеет то же самое значение, что

и в случае иерархических цепочек: корневым называется объект, с которым связаны все остальные объекты. Соответственно, корень анимации можно представить как основу персонажа, а все остальное двигается относительно этой основы. Для настройки этого объекта предоставляется несколько параметров, возможно, вы захотите поэкспериментировать, присваивая им различные значения в процессе работы над собственными вариантами анимации. Для наших же целей установите значения Body Orientation, Center Of Mass и Center Of Mass.

Щелчок на кнопке Apply добавит к персонажу анимационный клип idle. Выполните указанные операции еще два раза: клип walk должен начинаться в кадре 144 и заканчиваться в кадре 169, а клип run — начинаться в кадре 171 и заканчиваться в кадре 190. Все остальные параметры сделайте такими же, как у клипа idle, так как это тоже зацикленная анимация.

Четвертым станет анимационный клип jump, и его параметры будут несколько отличаться. Прежде всего, на этот раз нам нужен не цикл, а неподвижная поза, поэтому устанавливать флажок Loop Time не нужно. Введите в поля Start и End значения 190.5 и 191 соответственно; эта поза определена в одном кадре, но в Unity требуются различные значения параметров Start и End. Из-за введенных значений анимация в расположенном снизу окне предварительного просмотра будет выглядеть не совсем корректно, но на положении персонажа во время игры это не скажется.

Щелкните на кнопке Apply, чтобы подтвердить создание новых анимационных клипов, и переходите к следующему этапу — созданию контроллера анимации.

7.4.2. Создание контроллера для анимационных клипов

Теперь нам нужно создать контроллер-аниматор для персонажа. На этом этапе мы определяем состояния анимации и создаем между ними переходы. Во время каждого состояния воспроизводится свой набор клипов, а наши сценарии заставят контроллер переключаться между состояниями.

Может показаться странным, что мы пользуемся дополнительными средствами — помещаем между кодом и воспроизведением анимации абстракцию в виде контроллера. Скорее всего, раньше вы работали с системами, в которых анимация воспроизводилась непосредственно из кода; кстати, именно таким образом работала старая система Legacy, пользуясь вызовами вида `Play("idle")`. Но непрямой способ позволяет применять анимацию к разным моделям, не ограничиваясь воспроизведением анимации, непосредственно связанной с конкретной моделью. В этой главе мы не будем пользоваться данным преимуществом, поэтому просто запомните, что такая возможность существует. Получить готовую анимацию можно из разных источников, в том числе сотрудничая с аниматорами или покупая отдельные ролики в интернет-магазинах (например, в Asset Store для Unity).

Начните с создания нового контроллера-аниматора (в меню Assets выберите команду Create ▸ Animator Controller — не перепутайте с пунктом Animation, который создает ресурсы совсем другого типа). На вкладке Project появится значок, украшенный забавной сеткой линий (рис. 7.15); присвойте ему имя player. Выделите персонаж в сцене, и вы увидите, что у него появился новый компонент Animator; он есть у любой допускающей анимацию модели наряду с компонентом Transform и любыми другими

добавленными вами компонентами. У этого компонента есть поле Controller, предназначенное для привязки контроллера-аниматора. Перетащите на это поле только что созданный контроллер player (обязательно сбросьте флажок Apply Root Motion).

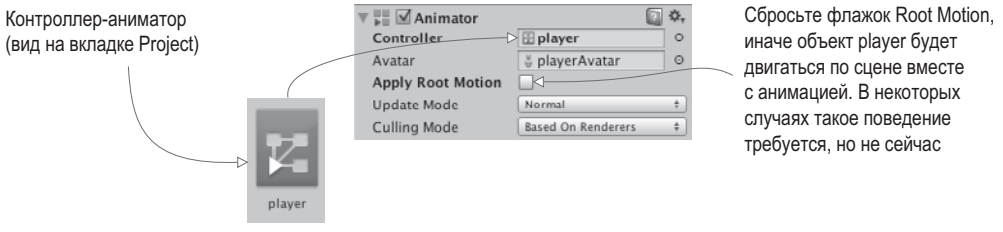


Рис. 7.15. Контроллер-аниматор и компонент Animator

Контроллер-аниматор представляет собой дерево связанных узлов (именно этим объясняется рисунок на значке данного ресурса), просмотр и управление которыми осуществляются на вкладке Animator. Эта вкладка, отображенная на рис. 7.16, показывает еще одно представление сцены, аналогично вкладкам Scene или Project, но по умолчанию она закрыта. Выберите в меню Window команду Animator (не перепутайте с окном Animation). На открывшейся вкладке вы увидите сеть узлов, принадлежащую выделенному в данный момент контроллеру (или контроллеру-аниматору выделенного персонажа).

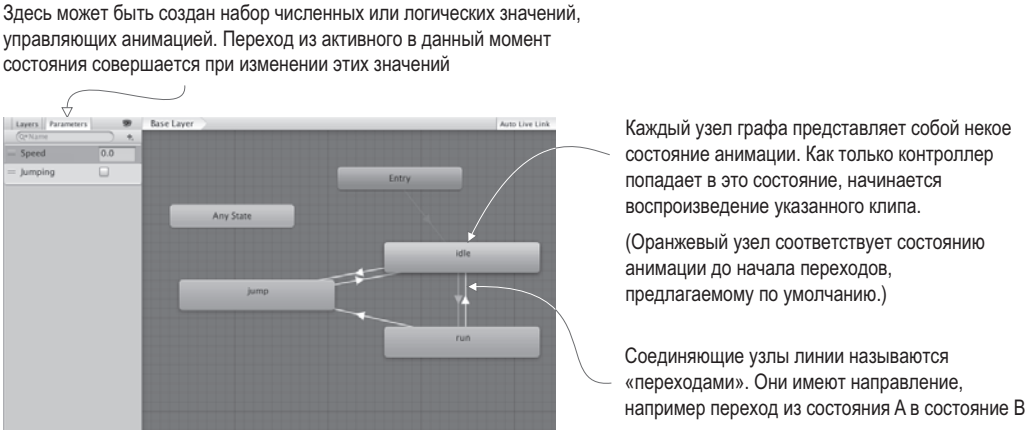


Рис. 7.16. Вид нашего контроллера-аниматора на вкладке Animator

СОВЕТ Напоминаю, что вы можете перемещать вкладки внутри редактора Unity, располагая их в нужных вам местах и формируя интерфейс по собственному вкусу. Мне нравится, когда вкладка Animator находится рядом с окнами Scene и Game.

Изначально мы видим только два существующих по умолчанию узла: Entry и Any State. Узлом Any State мы пользоваться не будем. Вместо этого создадим новые узлы, перетаскивая анимационные клипы. На вкладке Project щелкните на стрелке, расположенной

на значке модели персонажа, чтобы посмотреть содержимое этого ресурса. В числе прочего вы увидите и определенные вами анимационные клипы, как показано на рис. 7.17. Перетащите их на вкладку Animator. Клип walk пока не трогайте (он пригодится для других проектов), ограничьтесь клипами idle, run и jump.

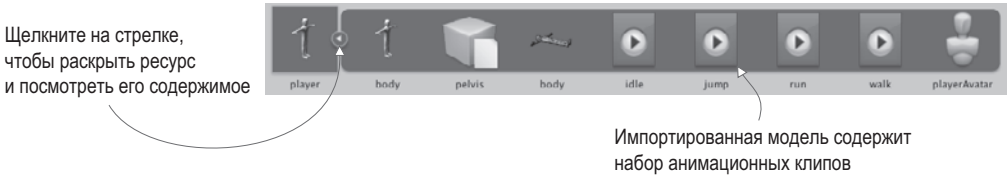


Рис. 7.17. Раскрытый ресурс на вкладке Project

Щелкните правой кнопкой мыши на узле Idle и выберите команду Set As Layer Default State. Узел станет оранжевым, в то время как все остальные останутся серыми; состояние анимации, предлагаемое по умолчанию, представляет собой место, в котором начинается сеть узлов, пока игра не внесет свои коррективы. Узлы нужно будет соединить друг с другом, обеспечивая переходы между состояниями анимации; щелкните на узле правой кнопкой мыши, выберите команду Make Transition и начните перетаскивать стрелку в направлении другого узла. Связь образуется щелчком на этом узле. Соедините узлы в соответствии с образцом на рис. 7.16 (в большинстве случаев вам требуется переход туда и обратно, исключением является переход от узла jump к узлу run). Эти линии переходов, определяющие связи состояний анимации друг с другом, во время игры будут контролировать переходы из одного состояния в другое.

ВНИМАНИЕ В процессе работы на вкладке Animator может появиться ошибка AnimationStateMachine.TransitionEditionContext.BuildNames. Просто перезапустите Unity; это не представляющий опасности сбой.

Переходы обусловливаются набором управляющих значений, которые нам нужно создать. В верхнем левом углу рис. 7.16 видна вкладка Parameters; перейдите на нее, чтобы увидеть панель добавления параметров с кнопкой в виде значка + (плюс). Добавьте параметр типа float с именем Speed и параметр типа Boolean с именем Jumping. Наш код будет менять эти значения, причем они послужат триггерами переходов между состояниями анимации. Выделите линию перехода, чтобы увидеть ее параметры на панели Inspector, как показано на рис. 7.18.

Именно здесь мы указываем, каким образом должны меняться состояния анимации при изменении параметров. Например, выделите переход Idle-Run, чтобы отредактировать условия его возникновения. В свитке Conditions выберите вариант Speed, Greater и 0.1. Сбросьте флажок Has Exit Time (он заставляет клип воспроизводиться на протяжении всей игры, вместо того чтобы завершиться сразу же после перехода). Затем щелкните на стрелке рядом с названием раздела Settings, чтобы увидеть меню целиком; другие переходы должны быть в состоянии прерывать этот, поэтому выберите в раскрывающемся списке Interruption Source вариант Current State. Повторите эту операцию для всех переходов, перечисленных в табл. 7.1.

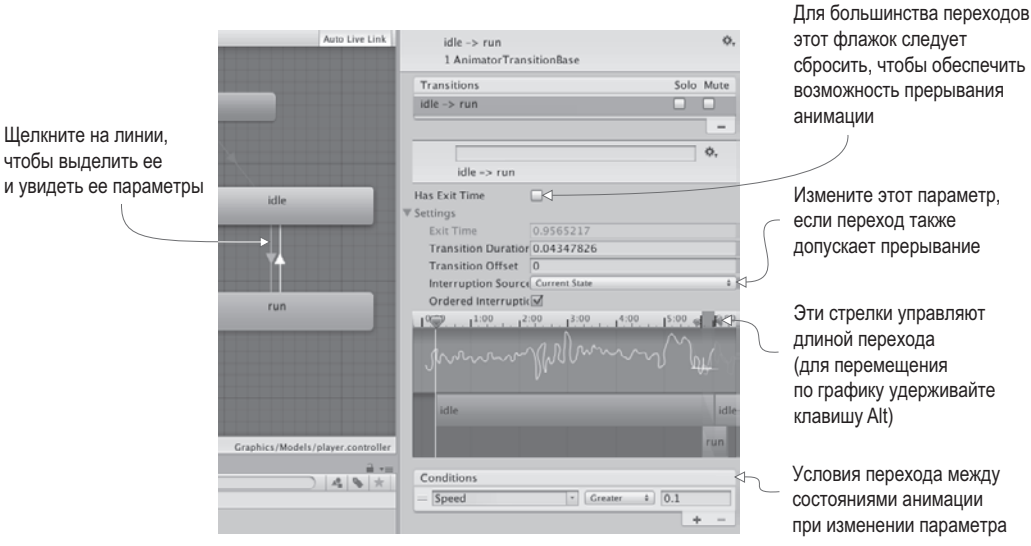


Рис. 7.18. Параметры перехода на панели Inspector

Таблица 7.1. Условия всех переходов для данного контроллера анимации

Переход	Условие	Прерывание
Idle-Run	Speed больше чем 0.1	Current State
Run-Idle	Speed меньше чем 0.1	None
Idle-Jump	Jumping = true	None
Run-Jump	Jumping = true	None
Jump-Idle	Jumping = false	None

В дополнение к параметрам, имеющим вид меню, вам предоставляется еще и визуальный интерфейс, расположенный, как показано на рис. 7.18, непосредственно над списком Condition. Этот график позволяет визуально корректировать продолжительность перехода. Заданное по умолчанию время перехода прекрасно подходит для переходов из Idle в Run и обратно, а вот переходы в состояние Jump и из него должны совершаться быстрее. Окрашенная область графика показывает, сколько времени займет переход; одновременное нажатие клавиши Alt и левой кнопки мыши даст возможность панорамирования графика, а клавиша Alt с правой кнопкой мыши позволит выполнить масштабирование (аналогичные способы управления применяются для представления Scene). Воспользуйтесь кнопками со стрелками над окрашенной областью, чтобы установить до менее 4 миллисекунд время всех трех связанных с прыжками переходов.

Напоследок можно усовершенствовать анимационную сеть, по очереди выделяя узлы и корректируя порядок переходов. На панели Inspector будет отображаться список всех переходов к выделенному узлу и от него; вы можете перетаскивать элементы

этого списка (манипуляторы управления находятся слева от их значков), меняя их порядок. Поместите переход Jump наверх списка для обоих узлов Idle и Run, дав ему приоритет над остальными. Если анимация кажется вам слишком медленной, поменяйте скорость воспроизведения (воспроизведение выглядит лучше всего при скорости 1.5).

Итак, мы настроили контроллер анимации, и теперь можем работать с ней через сценарий движения.

7.4.3. Код, управляющий контроллером-аниматором

Наконец, мы добавим методы в сценарий `RelativeMovement`. Как уже объяснялось, большая часть работы по настройке состояний анимации выполнена внутри контроллера; осталось написать небольшой код, управляющий богатой и гибкой системой анимации (см. следующий листинг).

Листинг 7.6. Задание значений в компоненте Animator

```
...
private Animator _animator;
...
_animator = GetComponent<Animator>(); ← Добавляется в метод Start().
...
_animator.SetFloat("Speed", movement.sqrMagnitude); ← Сразу после инструкции if для горизонтального движения.

if (hitGround) {
  if (Input.GetButtonDown("Jump")) {
    _vertSpeed = jumpSpeed;
  } else {
    _vertSpeed = -0.1f;
    _animator.SetBool("Jumping", false);
  }
} else {
  _vertSpeed += gravity * 5 * Time.deltaTime;
    if (_vertSpeed < terminalVelocity) {
      _vertSpeed = terminalVelocity;
    }
    if (_contact != null ) { ← Не вводите в действие это значение в самом начале уровня.
      _animator.SetBool("Jumping", true);
    }

    if (_charController.isGrounded) {
      if (Vector3.Dot(movement, _contact.normal) < 0) {
        movement = _contact.normal * moveSpeed;
      } else {
        movement += _contact.normal * moveSpeed;
      }
    }
  }
...
```

И снова большая часть кода повторяет предыдущие листинги; код анимации имеет множество пересечений с существующим сценарием движения. Выберите строки с переменной `_animator`, чтобы найти фрагменты, предназначенные для вставки в код.

Сценарию нужна ссылка на компонент Animator, после чего код присваивает контроллеру-аниматору значения (типа `float` или `Boolean`). Не совсем очевидно назначение условия (`_contact != null`) перед заданием логической переменной, отвечающей за прыжок. Это условие не дает контроллеру воспроизвести клип с прыжком сразу после начала действия. Хотя с технической точки зрения падение персонажа длится долю секунды, информация о столкновении появится только после его первого соприкосновения с поверхностью.

Итак, все готово! У вас есть демонстрационный ролик движения от третьего лица с элементами управления, связанными с камерой и анимированным персонажем.

7.5. Заключение

- Вид от третьего лица означает движение камеры вокруг персонажа, а не вместе с ним.
- Имитация теней путем визуализации в реальном времени и с помощью карт освещенности улучшает вид сцены.
- Действие элементов управления можно определить относительно камеры, а не персонажа.
- Метод бросания луча позволяет улучшить распознавание поверхности в Unity.
- Сложная анимация, настраиваемая в Unity с помощью контроллера-аниматора, позволяет моделировать антропоморфных персонажей.

8 Добавление в игру интерактивных устройств и элементов

- ✓ Программирование дверей, которые сможет открывать игрок (срабатывают при нажатии клавиши или при столкновении)
- ✓ Включение имитации физической среды для моделирования разлетающихся коробок
- ✓ Создание элементов, которые игрок сможет сохранять как инвентарь
- ✓ Применение кода для управления состоянием игры, например данными об инвентаре
- ✓ Подготовка и использование элементов инвентаря

Следующей темой, которую мы детально рассмотрим, станет реализация функциональных элементов. В предыдущих главах вы получили подробную информацию о различных аспектах готовой игры: перемещениях, врагах, пользовательском интерфейсе и т. п. До сих пор мы взаимодействовали только с врагами, и то это было достаточно ограниченное взаимодействие. В этой же главе вы научитесь создавать функциональные устройства, например двери. Также мы обсудим процесс сбора элементов, подразумевающий как взаимодействие с объектами игрового уровня, так и отслеживание состояния игры. В играх часто приходится следить за такими вещами, как статистика игрока, прохождение по этапам и т. п. Инвентарь игрока является примером состояния, поэтому вы создадите код, отслеживающий собранные игроком предметы. К концу главы вы построите динамическое пространство, которое будет выглядеть как настоящая игра!

Мы начнем с исследования устройств (таких, как двери), срабатывающих при нажатии игроком клавиш. После этого вы напишете код, распознающий столкновение персонажа с объектом игрового уровня и обеспечивающий такие взаимодействия, как перемещение предметов или сбор инвентаря. Затем вы разработаете архитектуру

кода в стиле MVC (Model-View-Controller — Модель-Представление-Контроллер) для управления данными о собранных предметах инвентаря. Наконец, вы запрограммируете интерфейс, позволяющий пользоваться инвентарем для игры, к примеру, открывать дверь при помощи найденного ранее ключа.

ВНИМАНИЕ Предыдущие главы были относительно независимы друг от друга и с технической точки зрения не требовали проектов из более ранних глав, теперь же ряд листингов будет получен редактированием сценариев из главы 7. Если вы перешли сразу к этой главе, скачайте пример проекта для главы 7, чтобы у вас была основа для работы.

Пример проекта будет содержать эти устройства и элементы, случайным образом распределенные по игровому уровню. Для создания полноценной игры потребуется тщательно распланировать размещение элементов на игровом уровне, но мы пока всего лишь тестируем данную функциональность. Впрочем, случайное расположение объектов не помешает вам получить представление о порядке внедрения различных элементов.

Как обычно, я объясню вам все этапы построения кода, но если вы хотите сразу получить готовую версию, скачайте с сайта пример проекта.

8.1. Создание дверей и других устройств

Игровые уровни по большей части состоят из статичных стен и предметов обстановки, к которым добавляется изрядная доля функциональных устройств. Я имею в виду объекты, с которыми может взаимодействовать и которыми может пользоваться игрок. Это такие вещи, как включающийся свет или начинающий вращаться вентилятор. Многообразие таких устройств, по большому счету, ограничено только вашим воображением, но практически все они пользуются одинаковым кодом активации устройства игроком. В этой главе мы реализуем пару вариантов, что даст вам возможность адаптировать данный код к устройствам других видов.

8.1.1. Открывание и закрывание дверей

Первый вид устройства, который нам предстоит запрограммировать, — это открывающаяся и закрывающаяся дверь. Начнем мы с управления дверью путем нажатия клавиши. В игру можно поместить множество различных устройств с самыми разными способами управления. При этом дверь является одним из самых распространенных интерактивных устройств, а управление посредством клавиатуры — наиболее очевидным подходом, поэтому мы начнем именно с них.

В стенах есть несколько промежутков, в которые можно поместить новый объект, блокирующий проход. Я создал куб, в поля Position которого ввел значения 2.5, 1.5, 17, а в поля Scale — значения 5, 3, 0.5, получив показанную на рис. 8.1 дверь.

Создайте сценарий на C# с именем `DoorOpenDevice` и назначьте его объекту door. Его код, показанный в следующем листинге, заставит объект функционировать как дверь.

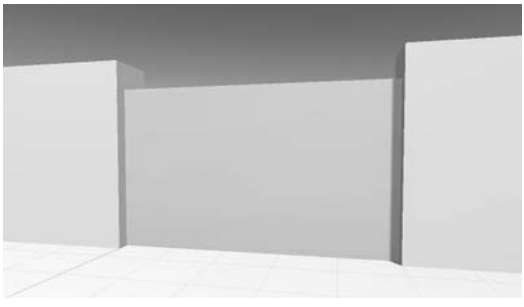


Рис. 8.1. Дверь, закрывающая проем в стене

Листинг 8.1. Сценарий, по команде закрывающий и открывающий дверь

```
using UnityEngine;
using System.Collections;

public class DoorOpenDevice : MonoBehaviour {
  [SerializeField] private Vector3 dPos; ← Смещение, применяемое при открывании двери.

  private bool _open; ← Переменная типа Boolean для слежения за открытым состоянием двери.

  public void Operate() {
    if (_open) { ← Открываем или закрываем дверь в зависимости от ее состояния.
      Vector3 pos = transform.position - dPos;
      transform.position = pos;
    } else {
      Vector3 pos = transform.position + dPos;
      transform.position = pos;
    }
    _open = !_open;
  }
}
```

Первая переменная определяет смещение, возникающее при открывании двери. Дверь сдвигается на указанное расстояние, когда ее открывают, а затем, чтобы закрыть дверь, это значение вычитается. Далее следует закрытая переменная логического типа, отслеживающая состояние двери. В методе `Operate()` выполняется преобразование объекта, меняющего его положение, путем добавления или вычитания величины смещения в зависимости от того, закрыта или открыта дверь; после этого состояние переменной `_open` меняется на противоположное.

Подобно другим сериализованным переменным, переменная `dPos` появляется на панели `Inspector`. Но так как она представляет собой структуру `Vector3`, вместо одного поля под одним именем переменной появятся три. Укажите относительное положение двери в открытом состоянии; я решил, что для этого она должна уходить вниз, поэтому смещение составило 0, –2.9, 0 (так как высота объекта door равна 3, смещение вниз на расстояние 2.9 оставит только небольшой порог, выступающий над полом).

Теперь нам нужен код, который должен вызывать функцию `Operate()`, открывающую и закрывающую дверь (обе ситуации будут обрабатываться одной и той же

функцией). Соответствующий сценарий для персонажа пока отсутствует; его написанием мы и займемся в следующем разделе.

ПРИМЕЧАНИЕ Преобразование совершается мгновенно, а вы, возможно, предпочитаете видеть, как открывается дверь. В главе 3 упоминалась такая методика, как анимация по начальной и конечной точкам, позволяющая получить плавное движение объектов. Английский термин tweening в разных контекстах имеет разные значения. В программировании игр он относится к командам кода, приводящим объекты в движение. Рекомендую обратить внимание на инструмент iTween, который можно скачать с официального сайта http://itween.pixelplacement.com.

8.1.2. Проверка расстояния и направления перед открытием двери

Создайте новый сценарий с именем `DeviceOperator`. Код из следующего листинга добавляет управляющую клавишу, которая воздействует на расположенные поблизости устройства.

Листинг 8.2. Клавиша контроля устройства для персонажа

```
using UnityEngine;
using System.Collections;

public class DeviceOperator : MonoBehaviour {
  public float radius = 1.5f; ← Расстояние, с которого персонаж может активировать устройства.

  void Update() {
    if (Input.GetButtonDown("Fire3")) { ← Реакция на кнопку ввода, заданную в настройках ввода в Unity.
      Collider[] hitColliders =
      Physics.OverlapSphere(transform.position, radius); ← Метод OverlapSphere() возвращает список ближайших объектов.
      foreach (Collider hitCollider in hitColliders) {
        hitCollider.SendMessage("Operate",
          SendMessageOptions.DontRequireReceiver); ← Метод SendMessage() пытается вызвать именованную функцию независимо от типа целевого объекта.
      }
    }
  }
}
```

Основная часть этого листинга вам уже знакома, но центральную часть кода занимает важный новый метод. Первым делом задается расстояние, на котором становится возможным управление устройством. После этого в методе `Update()` рассматривается клавиатурный ввод; так как клавиша прыжка уже задействована в сценарии `RelativeMovement`, на этот раз мы воспользуемся клавишей Fire3 (в настройках ввода проекта она определена как левая клавиша Command).

Вот мы и дошли до важного нового метода `OverlapSphere()`. Он возвращает массив всех объектов, расположенных не более чем на определенном расстоянии от текущего местоположения. Мы передаем в него положение персонажа и переменную `radius`, а он распознает все расположенные рядом с персонажем объекты. С полученным списком можно поступать как угодно (например, вы можете установить бомбу и применить к объектам силу взрыва), сейчас же мы хотим попытаться вызвать для всех этих объектов метод `Operate()`.

Для вызова этого метода вместо обычной точечной нотации используется метод `SendMessage()`. Этот подход вы уже встречали раньше, при работе с кнопками UI. В данном случае мы прибегаем к нему из-за отсутствия точных данных о типе целевого объекта. А метод `SendMessage()` работает со всеми объектами GameObjects. Правда, на этот раз мы снабдим его параметром `DontRequireReceiver`. Дело в том, что у большинства объектов, возвращаемых методом `OverlapSphere()`, метод `Operate()` отсутствует. Обычно, если объект не может получить сообщение, метод `SendMessage()` выводит сообщение об ошибке. Но сейчас это не требуется, так как мы и так знаем, что большая часть объектов проигнорирует рассылаемые сообщения.

Готовый сценарий нужно присоединить к объекту player. После этого для открытия и закрытия двери достаточно встать рядом с ней и нажать клавишу.

Осталась одна маленькая деталь. Пока что не имеет значения, как именно стоит персонаж, главное, чтобы он располагался достаточно близко к двери. Но можно сделать так, чтобы дверь открывалась только в случае, когда персонаж стоит к ней лицом. Напомню, что в главе 7 мы определяли направление движения персонажа, вычисляя скалярное произведение. Эта математическая операция с парой векторов возвращает значения в диапазоне от –1 до 1, где 1 означает одно и то же направление векторов, а –1 указывает на диаметрально противоположные направления. Следующий листинг демонстрирует код, который нужно добавить в сценарий `DeviceOperator`.

Листинг 8.3. Код, позволяющий открывать дверь, только стоя к ней лицом

```
...
foreach (Collider hitCollider in hitColliders) {
  Vector3 direction = hitCollider.transform.position - transform.position;
  if (Vector3.Dot(transform.forward, direction) > .5f) { ←  Сообщение отправляется только
    hitCollider.SendMessage("Operate",                      при корректной ориентации персонажа.
      SendMessageOptions.DontRequireReceiver);
    }
  }
...
```

Чтобы воспользоваться скалярным произведением, первым делом нужно определить направление, которое будет проверяться. Это направление от персонажа к объекту; нужный вектор создается вычитанием координат игрока из координат объекта. Затем вызывается метод `Vector3.Dot()`, которому передаются вычисленный вектор направления и вектор движения персонажа вперед. Если возвращенный методом результат близок к 1 (особенно при наличии в коде условия «больше чем 0.5»), значит, векторы имеют практически одно и то же направление.

После этих исправлений дверь перестанет открываться, когда персонаж стоит к ней спиной, даже если он находится рядом с ней. Этот подход применим к устройствам любого вида. Чтобы убедиться в его гибкости, давайте рассмотрим другой пример устройства.

8.1.3. Управление меняющим цвет монитором

Мы создали закрывающуюся и открывающуюся дверь, но аналогичная схема управления применима к устройствам любого вида. Построим еще одно устройство со

сходным управлением; на этот раз это будет меняющий цвет монитор, прикрепленный к стене.

Создайте новый куб и расположите его таким образом, чтобы одна из граней практически соприкасалась со стеной. Например, я ввел в поля Position значения 10.9, 1.5, –5. Теперь создайте сценарий `ColorChangeDevice`, добавьте в него код из следующего листинга и присоедините сценарий к монитору. Запустите игру, подойдите к монитору и нажмите ту же самую клавишу, которой открывали дверь; монитор начнет менять цвет, как показано на рис. 8.2.

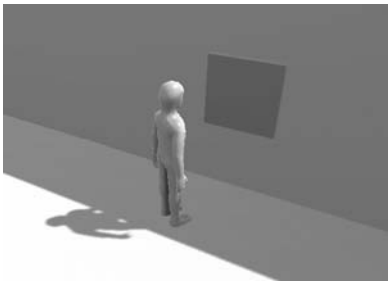


Рис. 8.2. Монитор, меняющий цвет

Листинг 8.4. Сценарий устройства, которое может менять цвет

```
using UnityEngine;
using System.Collections;

public class ColorChangeDevice : MonoBehaviour {
  public void Operate() { ← Объявление метода с таким же именем, как в сценарии для двери.
    Color random = new Color(Random.Range(0f,1f),   Эти числа представляют собой RGB-значения
      Random.Range(0f,1f), Random.Range(0f,1f)); ← в диапазоне от 0 до 1.
    GetComponent<Renderer>().material.color = random; ← Цвет задается в назначенном объекту материале.
  }
}
```

Первым делом мы объявляем функцию с тем же самым именем, что и в сценарии управления дверью. В сценариях управления устройствами всегда фигурирует функция с именем `Operate`, именно она вызывает срабатывание клавиши. В данном случае код этой функции присваивает материалу объекта случайный цвет (напоминаю, что цвет не является атрибутом самого объекта, объекту назначается материал, а у него уже есть цвет).

ПРИМЕЧАНИЕ Хотя в большинстве приложений для компьютерной графики цвет определяется компонентами Red, Blue и Green, параметры объекта Color в Unity лежат в диапазоне от 0 до 1, а не от 0 до 255, как это обычно бывает (в том числе и в интерфейсе выбора цвета в Unity).

Итак, вы познакомились с первым подходом к управлению устройствами в играх и даже создали пару таких демонстрационных устройств. Другим способом взаимодействия с элементами сцены является столкновение. Именно этим мы и займемся в следующем разделе.

8.2. Взаимодействие с объектами путем столкновений

В предыдущем разделе управление устройствами осуществлялось путем клавиатурного ввода, но это не единственный способ взаимодействия персонажа с элементами уровня. Другим крайне эффективным подходом является реакция на столкновение с персонажем. Большую часть операций при этом выполняет Unity благодаря встроенным в игровой движок механизмам распознавания столкновений и имитации физических явлений. При этом, хотя Unity поможет вам распознать столкновение, никто кроме вас не сможет запрограммировать реакцию объекта.

Мы рассмотрим три варианта реакции на столкновение, встречающиеся в играх:

- Толчок и падение.
- Срабатывание устройства.
- Исчезновение в момент контакта (при сборе элементов).

8.2.1. Столкновение с препятствиями, обладающими физическими свойствами

Первым делом мы создадим набор поставленных друг на друга коробок, которые будут разлетаться по сцене после того, как в них врежется персонаж. Моделирование подобной сцены требует сложных физических расчетов, но все они встроены в Unity, обеспечивая реалистичный вид разлетающихся коробок.

По умолчанию объекты в Unity не имеют физических свойств реальных объектов. Эта функция включается добавлением компонента Rigidbody. Собственно, именно с этого мы начали обсуждение проекта в главе 3, ведь бросаемым врагами огненным шарам был нужен этот компонент. Как я тогда объяснял, в Unity система имитации физических явлений работает только с объектами, оснащенными компонентом Rigidbody. Щелкните на кнопке Add Component и найдите вариант Rigidbody в разделе Physics.

Создайте новый куб и добавьте к нему компонент Rigidbody. Вам потребуется несколько таких кубов, положенных друг на друга. Например, я создал пять кубов и расположил их в виде двух рядов, как показано на рис. 8.3.

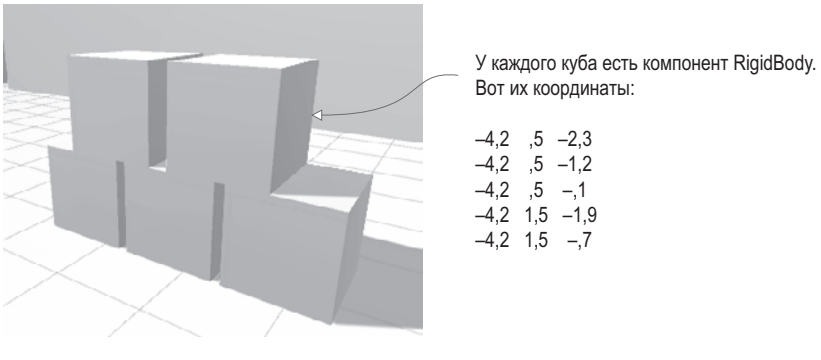


Рис. 8.3. Штабель из пяти коробок, в который мы будем врезаться

Кубы готовы реагировать на приложенную к ним силу. Чтобы источником этой силы стал персонаж, внесите дополнение из следующего листинга в присоединенный к персонажу сценарий `RelativeMovement` (это один из сценариев, написанных в предыдущей главе).

Листинг 8.5. Добавление физической силы в сценарий RelativeMovement

```
...
public float pushForce = 3.0f; ← Величина прилагаемой силы.
...
void OnControllerColliderHit(ControllerColliderHit hit) {
  _contact = hit;

  Rigidbody body = hit.collider.attachedRigidbody; ← Проверка, есть ли у участвующего в столкновении объекта компонент Rigidbody, обеспечивающий реакцию на приложенную силу.
  if (body != null && !body.isKinematic) {
    body.velocity = hit.moveDirection * pushForce; ← Назначение физическому телу скорости.
  }
}
...
```

Особых объяснений этот код не требует: при любом столкновении персонажа с объектом проверяется наличие у этого объекта компонента Rigidbody. В случае положительного результата проверки этому компоненту присваивается указанная скорость.

Запустите игру и заставьте персонажа бежать прямо на штабель — штабель вполне реалистично рассыплется. Как видите, имитация физического явления не потребовала от вас особых усилий! Благодаря встроенному в Unity механизму имитации физики, нам не приходится писать объемный код. Мы легко заставили объекты двигаться в ответ на столкновение, но есть и другой вариант реакции — генерация события срабатывания. Воспользуемся таким событием для управления дверью.

8.2.2. Управление дверью с помощью триггера

Если раньше управление дверью осуществлялось посредством нажатия клавиши, теперь она будет открываться и закрываться в ответ на столкновение персонажа с другим объектом сцены. Создайте еще одну дверь и поместите ее в другой проем (я сделал копию уже существующей двери и поместил ее в точку с координатами –2.5, 1.5, –17). Теперь создайте еще один куб, который будет играть роль триггера, и установите в свитке Box Collider флажок Is Trigger (вы уже делали подобное, работая над огненными шарами в главе 3). Кроме того, выберите в меню Layer в верхнем правом углу панели Inspector команду Ignore Raycast. Напоследок следует отключить у этого объекта возможность формирования теней (напоминаю, что это делается в свитке Mesh Renderer).

ВНИМАНИЕ Об этих небольших операциях легко забыть, меж тем как они крайне важны: вы не сможете использовать объект в качестве триггера, если не установлен флажок Is Trigger. Кроме того, обязательно укажите для слоя настройку Ignore Raycast, чтобы наш триггер не учитывался в операциях, связанных с бросанием лучей.

ПРИМЕЧАНИЕ Когда в главе 3 вы в первый раз столкнулись с объектами-триггерами, к ним требовалось добавить компонент Rigidbody. Теперь же он не требуется, так как триггеру не нужно реагировать на действия игрока (раньше у нас была реакция на столкновения со стенами). Сейчас же

для работы триггера нужно, чтобы у него или у входящего с ним в контакт объекта была включена система имитации физических явлений; под это требование подпадает не только компонент Rigidbody, но и назначенный персонажу компонент CharacterController.

Расположите объект так и придайте ему такой размер, чтобы он включал в себя дверь и некоторую область вокруг двери; я ввел в поля Position значения –2.5, 1.5, –17 (как у самой двери), а в поля Scale — значения 7.5, 3, 6. Кроме того, имеет смысл назначить этому объекту полупрозрачный материал, чтобы пространство триггера было легко отличить от остальных объектов. Создайте новый материал командой меню Assets и выделите его на вкладке Project. В верхней части панели Inspector находится раскрывающийся список Rendering Mode (там сейчас выбрано значение Opaque, предлагаемое по умолчанию), в котором нужно выбрать вариант Transparent.

Теперь щелкните на поле с каталогом цветов для вызова окна выбора цвета. В основной части окна выберите зеленый цвет и, используя нижний ползунок, уменьшите значение канала прозрачности. Перетащите материал со вкладки Project на объект. Рисунок 8.4 демонстрирует вид триггера после назначения материала.

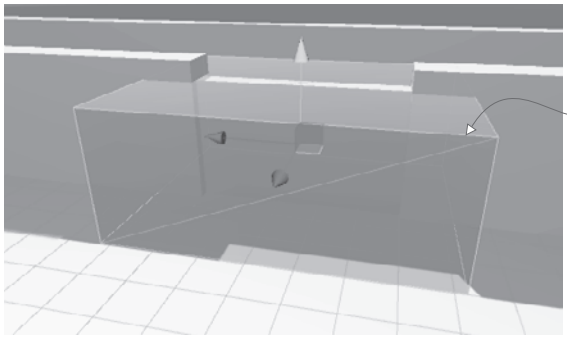


Рис. 8.4. Зона триггера, окружающая дверь, которой он управляет

ОПРЕДЕЛЕНИЕ Триггеры часто называются зонами, а не объектами, чтобы провести грань между твердыми телами и объектами, через которые вы можете пройти.

Запустите игру и убедитесь, что вы без проблем проходите через зону триггера; столкновения с объектом при этом регистрируются, но они больше не влияют на движение персонажа. Реакция на такого рода столкновения задается программно. В частности, нам нужно, чтобы триггер управлял дверью. Поэтому создайте новый сценарий `DeviceTrigger` и скопируйте в него код из следующего листинга.

Листинг 8.6. Код для триггера, контролирующего устройство

```
using UnityEngine;
using System.Collections;

public class DeviceTrigger : MonoBehaviour {
  [SerializeField] private GameObject[] targets; ← Список целевых объектов, которые будет
                                                   активировать данный триггер.
```

```
  void OnTriggerEnter(Collider other) {    ← Метод OnTriggerEnter() вызывается при попадании
    foreach (GameObject target in targets) { объекта в зону триггера...
      target.SendMessage("Activate");
    }
  }
                                                в то время как метод OnTriggerExit() вызывается
  void OnTriggerExit(Collider other) { ←  при выходе объекта из зоны триггера.
    foreach (GameObject target in targets) {
      target.SendMessage("Deactivate");
    }
  }
}
```

В этом листинге определяется массив целевых объектов; в большинстве случаев он будет состоять из одного элемента, но потенциально один триггер может управлять целым набором устройств. Цикл применяется для рассылки сообщений всем целевым объектам, причем как внутри метода `OnTriggerEnter()`, так и в методе `OnTriggerExit()`. Эти методы однократно вызываются при входе другого объекта в зону триггера и выходе из нее (вместо того, чтобы снова и снова вызываться, пока объект находится в зоне триггера).

Обратите внимание, что теперь рассылаются другие сообщения; на этот раз нам нужно определить для двери функции `Activate()` и `Deactivate()`. Добавьте в сценарий, присоединенный к двери, код следующего листинга.

Листинг 8.7. Добавление активирующей и деактивирующей функций в сценарий DoorOpenDevice

```
...
public void Activate() {
  if (!_open) { ← Открывает дверь, только если она пока не открыта.
    Vector3 pos = transform.position + dPos;
    transform.position = pos;
    _open = true;
  }
}
public void Deactivate() {
  if (_open) { ← Аналогично, закрывает дверь только при условии, что она уже не закрыта.
    Vector3 pos = transform.position - dPos;
    transform.position = pos;
    _open = false;
  }
}
...
```

Код новых методов `Activate()` и `Deactivate()` практически совпадает с кодом уже знакомого вам метода `Operate()`, просто теперь мы разделили функции открытия и закрытия двери, в то время как раньше у нас была одна функция, обрабатывающая обе операции.

Итак, у нас есть весь необходимый код, и теперь зону триггера можно заставить управлять состоянием двери. Свяжите сценарий `DeviceTrigger` с зоной триггера и свяжите дверь со свойством `Targets`; на панели Inspector сначала укажите размер

массива, а затем перетащите объекты с вкладки Hierarchy на появившиеся поля. Так как триггер предназначен для управления всего одной дверью, введите значение 1 в поле Size и перетащите объект door на поле Element 0.

Теперь запустите игру и посмотрите, что происходит с дверью, когда персонаж подходит к ней и отходит от нее. При входе персонажа в зону триггера и выходе из нее она автоматически открывается и закрывается.

Это еще один замечательный способ введения интерактивности в игровые уровни! Но триггеры применимы не только к таким устройствам, как двери; с их помощью можно добавить в игру инвентарь.

8.2.3. Сбор разбросанных по игровому уровню элементов

В играх часто встречаются элементы, которые персонаж может подобрать. В эту категорию попадают оборудование, пакеты для восстановления здоровья и бонусы. Базовый механизм столкновения с элементами для их сбора очень прост; сложности начинаются после того, как элементы собраны, но об этом мы поговорим чуть позже.

Создайте сферу и подвесьте ее примерно на уровне талии персонажа на открытом пространстве сцены. Уменьшите ее размеры, введя в поля Scale значения 0.5, 0.5, 0.5, в остальном установите те же параметры, что и в случае большой зоны триггера. Установите флажок Is Trigger в настройках коллайдера, выберите в настройках слоя вариант Ignore Raycast и создайте новый материал, чтобы присвоить объекту яркий цвет. Из-за малых размеров объекта мы не будем делать его полупрозрачным, поэтому ползунок, отвечающий за альфа-канал, трогать не нужно. Кроме того, в главе 7 упоминались настройки, управляющие отбрасыванием теней; использование теней в данном случае является делом вкуса, для небольших предметов, предназначенных для сбора, лично я предпочитаю их отключать.

Теперь создайте новый сценарий с именем `CollectibleItem`, введите в него код из следующего листинга и присоедините его к сфере.

Листинг 8.8. Сценарий, удаляющий элемент при контакте с персонажем

```
using UnityEngine;
using System.Collections;

public class CollectibleItem : MonoBehaviour {
  [SerializeField] private string itemName; ← Введите имя этого элемента на панели Inspector.

  void OnTriggerEnter(Collider other) {
    Debug.Log("Item collected: " + itemName);
    Destroy(this.gameObject);
  }
}
```

Это очень короткий и простой сценарий. Присвойте элементу значение `name`, чтобы в сцену можно было поместить несколько элементов. Метод `OnTriggerEnter()` вызывает исчезновение сферы. При этом на консоль выводится отладочное сообщение, которое мы впоследствии заменим кодом.

ВНИМАНИЕ Метод Destroy() должен вызываться для параметра this.gameObject, а не this! Не путайте эти вещи; ключевое слово this ссылается только на компонент сценария, в то время как выражение this.gameObject ссылается на объект, к которому присоединен сценарий.

Добавленная в код переменная должна появиться на панели Inspector. Введите в соответствующее поле имя, чтобы идентифицировать элемент; я выбрал для первого элемента имя energy. Затем создайте несколько копий элемента и поменяйте их имена; я использовал имена ore, health и key (точность написания имен крайне важна, так как впоследствии они появятся в коде). Заодно назначьте каждому элементу собственный материал, чтобы они имели разные цвета. Я выбрал голубой цвет для элемента energy, темно-серый — для элемента ore, розовый — для элемента health и желтый — для элемента key.

СОВЕТ Вместо имен, как в нашем случае, в более сложных играх у элементов зачастую имеются идентификаторы, используемые для поиска дальнейшей информации. Например, элементу может быть назначен идентификатор 301, который совпадает с определенным отображаемым именем, изображением, описанием и т. п.

Теперь превратите все элементы в шаблоны экземпляров, чтобы упростить их добавление в игровой уровень. В главе 3 вы узнали, что эта операция осуществляется перетаскиванием объектов со вкладки Hierarchy на вкладку Project.

ПРИМЕЧАНИЕ После этой операции имя объекта на панели Hierarchy выделяется синим цветом — это признак объектов, которые являются экземплярами данного шаблона экземпляров. Щелкните правой кнопкой мыши на имени такого объекта и выберите команду Select Prefab для выделения шаблона, экземпляром которого является объект.

Перетаскивая шаблоны, расположите элементы на открытых участках уровня; для тестирования создавайте по несколько экземпляров одного элемента. Запустите игру и «соберите» элементы. Все выглядит здорово, но пока ничего не происходит. Давайте начнем отслеживать собранные элементы; для этого нам нужно создать структуру кода, отвечающую за управление инвентарем.

8.3. Управление инвентаризационными данными и состоянием игры

Теперь, когда мы запрограммировали процедуру сбора элементов, требуется фоновый диспетчер данных (напоминающий веб-форму с ячейками) для игрового инвентаря. Код, который мы напишем, напоминает лежащую в основе многих веб-приложений MVC-архитектуру. Преимуществом такой архитектуры является разделение отображаемых на экране объектов и хранения данных, что упрощает эксперименты и итеративную разработку. Даже в случае сложноорганизованных данных и/или отображения объектов, редактирование части приложения не влияет на остальные его фрагменты.

В играх используются различные варианты таких структур. В каждой игре свои требования к управлению данными, поэтому правила вида «в основе любой игры должен

лежать вот такой паттерн проектирования» не имеют смысла. Разговор о таких вещах на слишком раннем этапе вообще приведет к снижению продуктивности, так как люди начнут подгонять свое будущее творение под вид рекомендованного паттерна.

В то же время, например, в ролевой игре требования к управлению данными высоки, поэтому, скорее всего, реализовать MVC-архитектуру целесообразно. А в головоломках подобных требований нет, и построение сложной несвязной структуры диспетчеров данных — напрасная трата времени и сил. Состояние такой игры без проблем отслеживается связанными со сценой объектами-контроллерами (более того, вы делали это в предыдущих главах).

Но сейчас перед нами стоит задача по управлению инвентарем игрока. Давайте запрограммируем соответствующую структуру кода.

8.3.1. Настраиваем диспетчеры игрока и инвентаря

Нам нужно разбить управление данными на отдельные хорошо продуманные модули, у каждого из которых своя зона ответственности. Для управления состоянием игрока (например, его здоровьем) мы создадим модуль `PlayerManager`, а для управления инвентарем — модуль `InventoryManager`. Диспетчеры данных будут вести себя как модели в MVC-архитектуре; контроллером в большинстве сцен послужит невидимый объект (в данном случае он не требовался, но вспомните объект SceneController из предыдущих глав), остальная же часть сцены выступит аналогом представления.

Создадим мы и расположенный на более высоком уровне «диспетчер диспетчеров», предназначенный для слежения за отдельными модулями. Кроме хранения списка диспетчеров, этот высокоуровневый диспетчер будет контролировать их жизненный цикл, в частности, занимаясь их инициализацией в начальный момент. Доступ всех прочих игровых сценариев к этим централизованным модулям должен осуществляться через главный диспетчер. В частности, для связи с нужным модулем остальной код может использовать ряд статических свойств в главном диспетчере.

ДОСТУП К ЦЕНТРАЛИЗОВАННЫМ МОДУЛЯМ СОВМЕСТНОГО ИСПОЛЬЗОВАНИЯ

За прошедшие годы для решения проблемы соединения частей программы с централизованными модулями общего доступа было создано множество паттернов проектирования. К примеру, паттерн одиночки (singleton) упоминался еще в первой книге «Банды четырех».

Однако этот паттерн разонравился многим разработчикам ПО, предпочитающим пользоваться такими альтернативами, как локатор служб (service locator) и инъекция зависимостей (dependency injection). Мой код представляет собой компромисс между простотой статических переменных и гибкостью локатора служб.

Представленный вашему вниманию проект объединяет простой в применении код с возможностью менять различные модули. Например, запрашивая InventoryManager с использованием паттерна одиночки, мы всегда будем ссылаться на один и тот же класс, плотно связывая код с этим классом; в то же время, запрос инвентаря через локатор служб позволяет вернуть как InventoryManager, так и DifferentInventoryManager. Иногда нам требуется возможность переключаться между слегка различающимися версиями одного модуля (например, при развертывании игры на разных платформах).

Чтобы главный диспетчер ссылался на другие модули одним и тем же способом, все эти модули должны унаследовать свойства от общей основы. Мы обеспечим

соблюдение этого условия с помощью интерфейса; многие языки программирования (в том числе C#) позволяют задавать своего рода сценарий, которому должны следовать остальные классы. Как `PlayerManager`, так и `InventoryManager` будут реализовывать общий интерфейс (он будет называться `IGameManager`), и основной объект `Managers` сможет воспринимать их как тип `IGameManager`. Иллюстрация этой схемы показана на рис. 8.5.

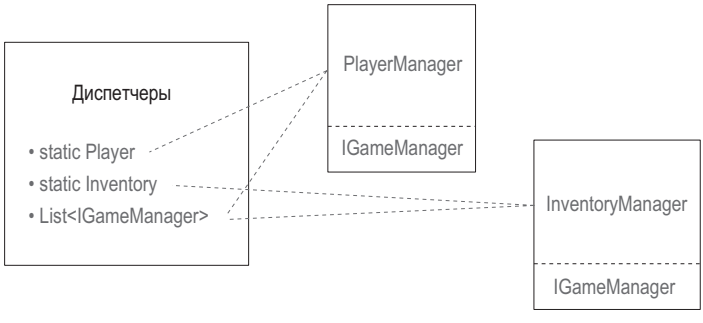


Рис. 8.5. Схема связи модулей

Кстати, хотя описанная мной архитектура состоит из невидимых модулей, существующих в фоновом режиме, для запуска этого кода Unity все равно требуются сценарии, связанные с объектами сцены. И как в случае с привязанными к сцене контроллерами в предыдущих проектах, мы создадим пустой объект GameObject, который будет связывать эти данные с диспетчерами.

8.3.2. Программирование диспетчеров

Итак, теперь, когда я объяснил все концепции, которыми мы будем пользоваться, пришло время написать код. Первым делом создайте новый сценарий `IGameManager` и скопируйте в него код следующего листинга.

Листинг 8.9. Базовый интерфейс, который будут реализовывать диспетчеры данных

```
public interface IGameManager {
  ManagerStatus status {get;} ← Это перечисление, которое нам нужно определить.

  void Startup();
}
```

Этот файл практически не содержит кода. Обратите внимание, что там отсутствует даже наследование от класса `MonoBehaviour`; сам по себе интерфейс ничего не делает и существует только затем, чтобы задавать структуру для других классов. Этот интерфейс объявляет одно свойство (переменную, у которой есть функция чтения) и один метод; как свойство, так и метод должны присутствовать у реализующего данный интерфейс класса. Свойство `status` сообщает остальной части кода, завершил ли модуль инициализацию. Метод `Startup()` предназначен для обработки процесса инициализации диспетчера, поэтому именно в нем выполняются все связанные с этим процессом задания, а функция задает состояние диспетчера.

Обратите внимание, что свойство относится к типу `ManagerStatus`; это пока еще не написанное нами перечисление. Создайте сценарий `ManagerStatus.cs` и скопируйте в него код из следующего листинга.

Листинг 8.10. ManagerStatus: возможные состояния для состояния IGameManager

```
public enum ManagerStatus {
  Shutdown,
  Initializing,
  Started
}
```

Это еще один файл с практически отсутствующим кодом. На этот раз мы перечисляем возможные состояния диспетчеров, принудительно устанавливая для свойства `status` одно из указанных значений.

Теперь, когда у нас есть интерфейс `IGameManager`, мы можем реализовывать его в других сценариях. Листинги 8.11 и 8.12 содержат код сценариев `PlayerManager` и `InventoryManager`.

Листинг 8.11. InventoryManager

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic; ← Импорт новых структур данных (используемых в листинге 8.14).

public class InventoryManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;} ← Свойство читается откуда угодно, но задается
                                                    только в этом сценарии.
  public void Startup() {                          Сюда идут все задачи запуска с долгим
    Debug.Log("Inventory manager starting..."); ← временем выполнения.
    status = ManagerStatus.Started; ← Для задач с долгим временем выполнения используем состояние 'Initializing'.
  }
}
```

Листинг 8.12. PlayerManager

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

public class PlayerManager : MonoBehaviour, IGameManager { ← Наследуем класс
                                                            и реализуем интерфейс.
  public ManagerStatus status {get; private set;}

  public int health {get; private set;}
  public int maxHealth {get; private set;}

  public void Startup() {
    Debug.Log("Player manager starting...");

    health = 50;       | Эти значения могут быть инициализированы
    maxHealth = 100;   | сохраненными данными.

    status = ManagerStatus.Started;
  }
```

```
  public void ChangeHealth(int value) { ← Другие сценарии не могут напрямую задавать переменную health,
    health += value;                       но могут вызывать эту функцию.
    if (health > maxHealth) {
      health = maxHealth;
    } else if (health < 0) {
      health = 0;
    }

    Debug.Log("Health: " + health + "/" + maxHealth);
  }
}
```

Пока что `InventoryManager` — это оболочка, заполнением которой мы займемся позднее, в то время как `PlayerManager` уже обладает всей необходимой для нашего проекта функциональностью. Оба диспетчера наследуют от класса `MonoBehaviour` и реализуют интерфейс `IGameManager`. Это означает, что оба диспетчера получили всю функциональность `MonoBehaviour`, но при этом обязаны реализовывать структуру, заданную интерфейсом `IGameManager`. Эта структура состоит из одного свойства и одного метода, которые и определяют наши диспетчеры.

Свойство `status` доступно для чтения из любого места (функция чтения общедоступна), но задается только внутри нашего сценария (функция записи закрыта). Кроме того, оба диспетчера определяют метод `Startup()`. Их инициализация завершается сразу же (`InventoryManager` пока ничего не делает, в то время как `PlayerManager` задает пару значений), поэтому состоянию присваивается значение `Started`. Но частью инициализации модулей данных могут оказаться длительные задания (например, загрузка сохраненных данных), тогда загрузкой этих заданий займется метод `Startup()`, а состоянию диспетчера будет присвоено значение `Initializing`. После завершения заданий измените состояние на `Started`.

Замечательно — мы наконец готовы связать все вместе внутри одного главного диспетчера! Создайте сценарий `Managers` и введите в него код следующего листинга.

Листинг 8.13. Диспетчер диспетчеров!

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

[RequireComponent(typeof(PlayerManager))] ← Гарантируем существование различных диспетчеров.
[RequireComponent(typeof(InventoryManager))]

public class Managers : MonoBehaviour {
  public static PlayerManager Player {get; private set;} ← Статические свойства, которыми остальной
  public static InventoryManager Inventory {get; private set;}   код пользуется для доступа к диспетчерам.

  private List<IGameManager> _startSequence; ← Список диспетчеров, который просматривается в цикле
                                               во время стартовой последовательности.
  void Awake() {
    Player = GetComponent<PlayerManager>();
    Inventory = GetComponent<InventoryManager>();

    _startSequence = new List<IGameManager>();
    _startSequence.Add(Player);
    _startSequence.Add(Inventory);
```

```
    StartCoroutine(StartupManagers()); ← Асинхронно загружаем стартовую последовательность.
  }

  private IEnumerator StartupManagers() {
    foreach (IGameManager manager in _startSequence) {
      manager.Startup();
    }

    yield return null;

    int numModules = _startSequence.Count;
    int numReady = 0;

    while (numReady < numModules) { ← Продолжаем цикл, пока не начнут работать все диспетчеры.
      int lastReady = numReady;
      numReady = 0;

      foreach (IGameManager manager in _startSequence) {
        if (manager.status == ManagerStatus.Started) {
          numReady++;
        }
      }

      if (numReady > lastReady)
      Debug.Log("Progress: " + numReady + "/" + numModules);
      yield return null; ← Остановка на один кадр перед следующей проверкой.
    }

    Debug.Log("All managers started up");
  }
}
```

Наиболее важной частью этого паттерна являются статические свойства на самом верху. Именно они позволяют другим сценариям использовать для доступа к различным модулям такой синтаксис, как `Managers.Player` или `Managers.Inventory`. Изначально эти свойства пусты, но они заполняются сразу же после запуска кода в методе `Awake()`.

СОВЕТ Подобно методам Start() и Update(), метод Awake() автоматически предоставляется классом MonoBehaviour. Как и метод Start(), он однократно запускается в начале выполнения кода. Но в принятой в Unity последовательности выполнения кода метод Awake() располагается перед методом Start(), отвечая за связанные с инициализацией задания, которые должны запускаться перед любыми другими модулями кода.

Кроме того, метод `Awake()` выводит последовательность запуска, а затем загружает сопрограмму, начинающую работу всех диспетчеров. А именно он создает объект `List` и прибегает к методу `List.Add()` для добавления всех диспетчеров.

ОПРЕДЕЛЕНИЕ Объект List представляет собой такую структуру данных, как коллекция из языка C#. Списки аналогичны массивам: в них хранятся последовательные наборы элементов определенного типа. Но размер списка может быть изменен в процессе работы, в то время как у массивов этот параметр не редактируется.

ВНИМАНИЕ Коллекции находятся в собственном пространстве имен, которое следует включить в сценарий; обратите внимание на дополнительную инструкцию using, появившуюся в верхней части листинга. Не забудьте добавить ее в свои сценарии!

Так как все диспетчеры реализуют интерфейс `IGameManager`, код может перечислить их как данный тип и вызвать определенный в каждом из них метод `Startup()`. Стартовая последовательность запускается как сопрограмма, то есть асинхронно с другими частями игры (например, анимированным индикатором выполнения на заставке).

Функция запуска сначала в цикле просматривает список диспетчеров и для каждого из них вызывает метод `Startup()`. Затем она входит в цикл, проверяющий, загрузился ли каждый из диспетчеров, и ожидает завершения этого процесса. После этого функция запуска уведомляет нас о загрузке и завершает свою работу.

СОВЕТ Инициализация написанных нами диспетчеров крайне проста и не связана с временем ожидания, но в общем случае эта стартовая последовательность на базе сопрограмм позволяет элегантно обрабатывать асинхронные задания запуска с долгим временем выполнения, такие как загрузка сохраненных данных.

Итак, наша структура кода готова. Вернитесь в редактор Unity и создайте пустой объект GameObject; как обычно, расположите его в точке с координатами 0,0,0 и присвойте ему значимое имя, например Game Managers. Свяжите с этим объектом сценарии `Managers`, `PlayerManager` и `InventoryManager`.

В процессе воспроизведения игры вы не заметите никаких изменений, но на консоли появится ряд сообщений, информирующих о выполнении стартовой последовательности. Как видите, диспетчеры запускаются корректно, значит, пришло время заняться программированием диспетчера инвентаря.

8.3.3. Сохранение инвентаря в виде коллекции: списки и словари

Собранные персонажем элементы можно сохранить в виде объекта `List`. Следующий листинг добавляет такой список в диспетчер инвентаря.

Листинг 8.14. Добавление элементов в InventoryManager

```
...
private List<string> _items;

public void Startup() {
  Debug.Log("Inventory manager starting...");

  _items = new List<string>(); ← Инициализируем пустой список элементов.

  status = ManagerStatus.Started;
}

private void DisplayItems() { ← Вывод на консоль сообщения о текущем инвентаре.
  string itemDisplay = "Items: ";
  foreach (string item in _items) {
    itemDisplay += item + " ";
  }
```

```
  Debug.Log(itemDisplay);
}

public void AddItem(string name) { ← Другие сценарии не могут напрямую управлять списком элементов, но могут вызывать этот метод.
  _items.Add(name);

  DisplayItems();
}
...
```

В сценарии `InventoryManager` появилось два важных дополнения. Во-первых, мы добавили объект `List`, предназначенный для хранения элементов. Во-вторых, появился открытый метод `AddItem()`, который может вызываться другим кодом. Эта функция добавляет элемент в список и выводит список на консоль. Теперь давайте слегка скорректируем сценарий `CollectibleItem`, добавив в него вызов нового метода `AddItem()`, как показано в следующем листинге.

Листинг 8.15. Применение нового диспетчера InventoryManager в сценарии CollectibleItem

```
...
void OnTriggerEnter(Collider other) {
  Managers.Inventory.AddItem(name);
  Destroy(this.gameObject);
}
...
```

Теперь в процессе сбора элементов персонажем вы увидите, как растет список инвентаря в выводимом на консоль сообщении. При этом всплывет недостаток такой структуры данных, как список: все собранные элементы одного типа (например, второй элемент Health) появятся в списке, как показано на рис. 8.6, хотя разумнее было бы подсчитывать количество таких элементов. Разумеется, можно себе представить вариант игры, в котором каждый элемент нужно отслеживать отдельно, но в большинстве игр подсчитывается количество копий одного элемента. Устранить этот недостаток вполне реально средствами списка, но более естественно и эффективно прибегнуть к другой структуре данных, которая называется словарем.

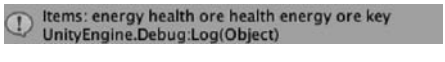


Рис. 8.6. Консольное сообщение, в котором однотипные элементы перечисляются несколько раз

ОПРЕДЕЛЕНИЕ Структура данных Dictionary также взята из языка C#. Доступ к записям словаря осуществляется по идентификатору (или ключу), а не по их местоположению. Словарь напоминает хэш-таблицу, но более гибок, так как ключами могут выступать данные практически любого типа (например, «верните записи для этого объекта GameObject»).

Отредактируйте код сценария `InventoryManager`, заменив структуру данных `List` структурой `Dictionary`. Для этого вставьте вместо кода из листинга 8.14 код следующего листинга.

Листинг 8.16. Словарь элементов в InventoryManager

```
...
private Dictionary<string, int> _items; ← При объявлении словаря указывается два типа: тип ключа и тип значения.

public void Startup() {
  Debug.Log("Inventory manager starting...");

  _items = new Dictionary<string, int>();

  status = ManagerStatus.Started;
}

private void DisplayItems() {
  string itemDisplay = "Items: ";
  foreach (KeyValuePair<string, int> item in _items) {
    itemDisplay += item.Key + "(" + item.Value + ") ";
  }
  Debug.Log(itemDisplay);
}

public void AddItem(string name) {
  if (_items.ContainsKey(name)) { ← Проверка существующих записей перед вводом новых данных.
    _items[name] += 1;
  } else {
    _items[name] = 1;
  }
  DisplayItems();
}
...
```

По большому счету этот код выглядит так же, как и раньше, но появилось несколько отличий. Если раньше вы не сталкивались с такими структурами данных, как `Dictionary`, обратите внимание, что при ее объявлении указывается два типа. Если структура `List` объявляется только с одним типом (типом сохраняемых в нее значений), то `Dictionary` объявляет как тип ключей (то есть идентификаторов, по которым будет вестись поиск), так и тип значений.

В методе `AddItem()` появился дополнительный алгоритм. Если раньше каждый элемент просто добавлялся в список, теперь первым делом требуется проверить наличие такого же элемента в словаре; именно этим занимается метод `ContainsKey()`. Если перед нами новый элемент, счет начинается с единицы, если же такой элемент уже существует, на единицу увеличивается сохраненное значение. Воспроизведите сцену, и вы убедитесь, что теперь в сообщениях о сборе инвентаря элементы одного типа объединяются, как показано на рис. 8.7.

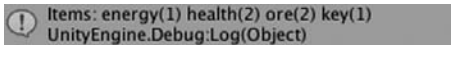


Рис. 8.7. Консольное сообщение, в котором элементы одного типа объединяются

Наконец-то мы систематизировали в инвентаре игрока собранные элементы! Скорее всего, вам кажется, что для решения такой простой задачи мы написали слишком

много кода. Если бы это была единственная стоявшая перед нами цель, решение действительно можно было бы назвать излишне сложным. Но смысл всей этой тщательно разрабатываемой архитектуры кода в том, чтобы сохранять все данные в отдельных гибких модулях. Полезность данного паттерна становится очевидной по мере усложнения игры. Сейчас мы напишем пользовательский интерфейс, и управлять отдельными частями кода станет намного проще.

8.4. Интерфейс для использования и подготовки элементов

В игре коллекцией собранных элементов можно пользоваться по-разному, но в любом случае вам потребуется какой-то пользовательский интерфейс, чтобы игрок мог видеть имеющийся у него инвентарь. После этого мы добавим к этому интерфейсу интерактивность, позволив игрокам применять элементы. Мы снова рассмотрим пару конкретных примеров (подготовку ключа и применение пакетов здоровья), после чего вы самостоятельно сможете адаптировать этот код для элементов других типов.

ПРИМЕЧАНИЕ Как упоминалось в главе 6, в Unity есть как более старый GUI непосредственного режима, так и более новая система UI на основе спрайтов. В этой главе мы задействуем GUI непосредственного режима, поскольку этот интерфейс быстрее в реализации и требует меньшего количества настроек, что хорошо подходит для упражнений. При этом UI на основе спрайтов имеет более завершенный вид, поэтому этот интерфейс выбирают для итогового варианта игры.

8.4.1. Отображение элементов инвентаря в UI

Для отображения информации об элементах инвентаря в UI нужно добавить пару методов в сценарий `InventoryManager`. В настоящее время список элементов является закрытым и доступен только в соответствующем диспетчере; чтобы сделать его видимым, потребуются открытые методы доступа к данным. Добавьте их из следующего листинга.

Листинг 8.17. Добавление в сценарий InventoryManager методов доступа к данным

```
...
public List<string> GetItemList() {
  List<string> list = new List<string>(_items.Keys); ← Возвращаем список всех ключей словаря.
  return list;
}

public int GetItemCount(string name) { ← Возвращаем количество указанных элементов в инвентаре.
  if (_items.ContainsKey(name)) {
    return _items[name];
  }
  return 0;
}
...
```

Метод `GetItemList()` возвращает список элементов инвентаря. У вас может появиться вопрос, зачем мы ранее тратили столько сил на преобразование списка элементов инвентаря в другой формат. Дело в том, что теперь элемент каждого типа появляется

в списке всего один раз. Например, при наличии в инвентаре двух пакетов здоровья, имя «health» в списке будет всего одно. Ведь этот список был создан из ключей элементов словаря, а не из отдельных элементов.

Метод `GetItemCount()` возвращает количество элементов указанного типа в инвентаре. Например, вызов `GetItemCount("health")` представляет собой вопрос «Сколько пакетов здоровья содержит инвентарь?». Это даст возможность отображать в UI не только все элементы, но и их количество.

Благодаря наличию этих методов в сценарии `InventoryManager`, мы можем создать пользовательский интерфейс. Давайте отобразим все его элементы в виде горизонтальной панели в верхней части экрана. Каждому элементу будет соответствовать свой значок, поэтому нужно импортировать в проект соответствующие изображения. Ресурсы, находящиеся в папке Resources, Unity обрабатывает особым способом.

СОВЕТ Ресурсы из папки Resources доступны для загрузки методом Resources.Load(). В остальных случаях для вставки их в сцену пользуйтесь редактором Unity.

Рисунок 8.8 демонстрирует четыре значка вместе с адресом папки, в которой они хранятся. Создайте папку Resources, а затем внутри нее — папку Icons.

Рис. 8.8. Графические ресурсы для значков оборудования в папке Resources

После этого создайте пустой объект GameObject с именем Controller и свяжите с ним новый сценарий `BasicUI` (его код представлен в следующем листинге).

Листинг 8.18. Сценарий BasicUI, отображающий инвентарь

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

public class BasicUI : MonoBehaviour {
  void OnGUI() {
    int posX = 10;
    int posY = 10;
    int width = 100;
    int height = 30;
    int buffer = 10;

    List<string> itemList = Managers.Inventory.GetItemList();
    if (itemList.Count == 0) { ← Отображает сообщение, информирующее об отсутствии инвентаря.
      GUI.Box(new Rect(posX, posY, width, height), "No Items");
    }
    foreach (string item in itemList) {
      int count = Managers.Inventory.GetItemCount(item);
```

```
        Texture2D image = Resources.Load<Texture2D>("Icons/"+item);
        GUI.Box(new Rect(posX, posY, width, height),
        new GUIContent("(" + count + ")", image));
        posX += width+buffer;
    }
  }
}
```

← Метод, загружающий ресурсы из папки Resources.

← При каждом прохождении цикла сдвигаемся в сторону.

Этот листинг отображает собранные элементы в виде горизонтальной полосы, показанной на рис. 8.9, показывая заодно их количество. Как упоминалось в главе 3, все представители класса `MonoBehaviour` автоматически отвечают на метод `OnGUI()`, после визуализации сцены запускаемый в каждом кадре.

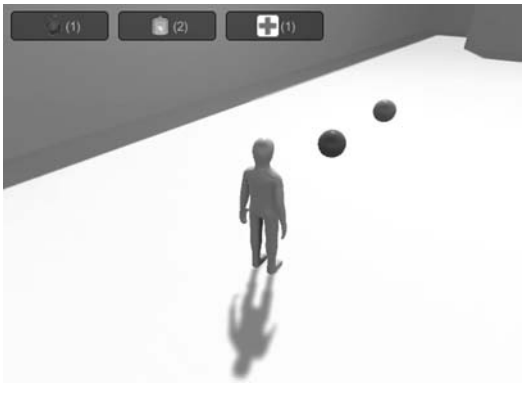


Рис. 8.9. Отображение инвентаря средствами UI

Внутри метода `OnGUI()` первым делом определяется набор значений для размещения UI-элементов. Эти значения увеличиваются в процессе циклического просмотра всех элементов, что позволяет расположить их в ряд. Рисуемые UI-элементы — это в данном случае `GUI.Box`; они представляют собой неинтерактивные поля, в которых отображаются текст и значки.

Метод `Resources.Load()` загружает ресурсы из папки Resources. Это удобный способ загрузки ресурсов по именам; обратите внимание, что имя каждого элемента передается как параметр. Нужно указать тип загружаемых ресурсов, в противном случае метод вернет обобщенные объекты.

Пользовательский интерфейс показывает нам собранный инвентарь. Теперь мы можем им воспользоваться.

8.4.2. Подготовка ключа для открытия двери

Рассмотрим пару примеров применения инвентаря, чтобы вы смогли потом по аналогии работать с элементами любых типов. Начнем мы с подготовки ключа, который потребуется нам для открытия двери.

В настоящее время сценарий `DeviceTrigger` нашего инвентаря не видит (ведь он был написан до появления связанного с инвентарем кода). Коррективы, которые следует внести в сценарий, показаны в следующем листинге.

Листинг 8.19. Запрос ключа в сценарии DeviceTrigger

```
...
public bool requireKey;

void OnTriggerEnter(Collider other) {
  if (requireKey && Managers.Inventory.equippedItem != "key") {
    return;
  }
...
```

Как видите, вам потребовались всего лишь новая открытая переменная и условная инструкция для поиска подготовленного к использованию ключа. Логическая переменная `requireKey` появляется в виде флажка на панели Inspector, что позволяет требовать ключ только от определенных триггеров. Условие в начале метода `OnTriggerEnter()` проверяет наличие подготовленного ключа в диспетчере `InventoryManager`; соответственно, нужно добавить в сценарий `InventoryManager` код следующего листинга.

Листинг 8.20. Код подготовки инвентаря для сценария InventoryManager

```
...
public string equippedItem {get; private set;}
...
public bool EquipItem(string name) {
  if (_items.ContainsKey(name) && equippedItem != name) {
    equippedItem = name;
    Debug.Log("Equipped " + name);
    return true;
  }

  equippedItem = null;
  Debug.Log("Unequipped");
  return false;
}
...
```

Проверяем наличие в инвентаре указанного элемента и тот факт, что он еще не подготовлен к использованию.

В верхнюю часть сценария добавлено свойство `equippedItem`, которое проверяется другим фрагментом кода. Затем мы добавили открытый метод `EquipItem()`, позволяющий другому коду менять подготовленные элементы. Этот код готовит элемент, если он еще не готов к применению, или кладет его обратно в инвентарь, если пользоваться им уже можно.

Напоследок, чтобы у игрока появилась возможность выполнять подготовку элементов, указанная функциональность добавляется в UI. Это делается в следующем фрагменте кода.

Листинг 8.21. Добавление к сценарию BasicUI функции подготовки элементов

```
...
    foreach (string item in itemList) {
      int count = Managers.Inventory.GetItemCount(item);
      GUI.Box(new Rect(posX, posY, width, height), item +
        "(" + count + ")");
      posX += width+buffer;
    }
```

Написанный курсивом код в сценарии уже присутствовал, тут он фигурирует в качестве точки отсчета.

```
    string equipped = Managers.Inventory.equippedItem;
    if (equipped != null) { ← Отображение подготовленного элемента.
      posX = Screen.width - (width+buffer);
      Texture2D image = Resources.Load("Icons/"+equipped) as Texture2D;
      GUI.Box(new Rect(posX, posY, width, height),
      new GUIContent("Equipped", image));
    }

    posX = 10;
    posY += height+buffer;

    foreach (string item in itemList) { ← Просмотр всех элементов в цикле для создания кнопок.
      if (GUI.Button(new Rect(posX, posY, width, height),
        "Equip "+item)) { ← Запуск вложенного кода при щелчке на кнопке.
        Managers.Inventory.EquipItem(item);
      }
      posX += width+buffer;
    }
  }
}
```

Для отображения подготовленного элемента мы снова пользуемся методом `GUI.Box()`. Но так как он не является интерактивным, ряд кнопок Equip мы нарисуем с помощью метода `GUI.Button()`. Этот метод создает кнопку, которая при щелчке выполняет код внутри инструкции `if`.

Теперь, когда мы написали весь нужный код, выделите триггер в сцене и установите в свитке Device Trigger флажок Require Key, после чего запустите игру. Попробуйте войти в зону триггера без ключа — ничего не произойдет. Теперь поднимите ключ и щелкните на кнопке, чтобы подготовить его к использованию; после этого вход в зону триггера откроет дверь.

Из спортивного интереса поместите ключ в точку с координатами –11, 5, –14 и попробуйте сообразить, как его можно достать. А пока мы рассмотрим процесс использования пакетов здоровья.

8.4.3. Восстановление здоровья персонажа

Восстановление здоровья персонажа является еще одним распространенным примером применения инвентаря. Нам потребуется внести в код два изменения: добавить новый метод в сценарий `InventoryManager` и новую кнопку в пользовательский интерфейс — для этого служат листинги 8.22 и 8.23 соответственно.

Листинг 8.22. Новый метод в сценарии InventoryManager

```
...
public bool ConsumeItem(string name) {
  if (_items.ContainsKey(name)) { ← Проверка наличия элемента среди инвентаря.
    _items[name]--;
    if (_items[name] == 0) { ← Удаление записи, если количество становится равным нулю.
      _items.Remove(name);
    }
    } else { ← Реакция в случае отсутствия в инвентаре нужного элемента.
      Debug.Log("cannot consume " + name);
      return false;
    }
```

```
    DisplayItems();
    return true;
  }
...
```

Листинг 8.23. Добавление кнопки здоровья в базовый UI

```
...
      foreach (string item in itemList) { ← Выделенный курсивом код в сценарии уже был,
        if (GUI.Button(new Rect(posX, posY, width, height),   здесь он приведен в качестве точки отсчета.
          "Equip "+item)) {
          Managers.Inventory.EquipItem(item);
      }

      if (item == "health") { ← Начало нового кода.
        if (GUI.Button(new Rect(posX, posY + height+buffer, width,
          height), "Use Health")) { ← Запуск вложенного кода при щелчке на кнопке.
          Managers.Inventory.ConsumeItem("health");
          Managers.Player.ChangeHealth(25);
        }
      }

      posX += width+buffer;
    }
  }
}
```

Новый метод `ConsumeItem()`, по сути, диаметрально противоположен методу `AddItem()`; он проверяет наличие указанного элемента в инвентаре и уменьшает значение, если таковой не обнаружен. Умеет он реагировать и на ситуацию, когда количество элементов уменьшается до нуля. Код UI вызывает этот новый метод работы с инвентарем, который, в свою очередь, вызывает метод `ChangeHealth()`, с самого начала присутствовавший в сценарии `PlayerManager`.

Если вы соберете несколько пакетов здоровья, а затем ими воспользуетесь, на консоли появятся соответствующие сообщения. Вот вы и узнали еще один вариант использования элементов инвентаря!

8.5. Заключение

- Управлять устройствами можно как посредством нажатия клавиш, так и с помощью триггеров столкновения.
- Объекты, для которых включена имитация реальных физических явлений, могут реагировать на силу столкновения и зоны триггеров.
- Сложными игровыми состояниями управляют через специальные объекты общего доступа.
- Коллекции объектов можно систематизировать с помощью таких структур данных, как `List` или `Dictionary`.
- Отслеживание подготовленных состояний элементов инвентаря позволяет влиять на другие части игры.

Часть III
УВЕРЕННЫЙ ФИНИШ

Вы уже много знаете о Unity. Вы умеете программировать элементы управления персонажем, создавать блуждающих по сцене врагов и добавлять в игру интерактивные устройства. Вы даже способны пользоваться при создании игр как двухмерной, так и трехмерной графикой! Всего этого *почти* достаточно для разработки полноценной игры. Почти, но не совсем. Вам нужно изучить еще ряд вещей, таких как добавление в игру звука, кроме того, вы должны суметь собрать воедино все разрозненные фрагменты, с которыми мы до этого работали.

Вы уже на финишной прямой, осталось всего четыре главы!

9 Подключение игры к Интернету

- ✓ Генерирование графики для неба с помощью скайбокса
- ✓ Скачивание данных с помощью веб-объектов в сопрограммах
- ✓ Анализ распространенных форматов данных, таких как XML и JSON
- ✓ Отображение скачанных из Интернета изображений
- ✓ Отправка данных на веб-сервер

В этой главе мы рассмотрим способы отправки и получения данных по сети. Все созданные нами проекты, представляющие различные игровые жанры, располагались исключительно на вашей машине. А как известно, подключение к Интернету и обмен данными становятся все более важным аспектом игр всех жанров. Многие игры существуют исключительно в Интернете с постоянным подключением к сообществу игроков; для их обозначения используется аббревиатура MMO (Massively Multiplayer Online — массовая многопользовательская онлайновая игра). Шире всего известны так называемые массовые многопользовательские ролевые онлайновые игры (Massively Multiplayer Online Role-Playing Games — MMORPG). Даже если постоянное подключение для игры не требуется, современные видеоигры оснащены такими функциями, как, например, отправка сведений о набранных очках в глобальный список лучших результатов. В Unity они тоже поддерживаются, так что мы подробно рассмотрим данную функциональность.

В Unity поддерживаются разные варианты передачи данных по сети, так как существуют разные варианты задач. Но в этой главе будет рассматриваться в основном наиболее общий вид сетевых взаимодействий — отправка HTTP-запросов.

ЧТО ТАКОЕ HTTP-ЗАПРОСЫ?

Я думаю, что большинство читателей уже знают, что представляют собой HTTP-запросы, но на всякий случай дам небольшое пояснение: HTTP — это протокол отправки запросов к веб-серверам

и получения оттуда ответов. Например, когда вы щелкаете на расположенной на веб-странице ссылке, ваш браузер (клиент) отправляет запрос на определенный адрес, а расположенный по этому адресу сервер возвращает новую страницу. Такие запросы выполняются различными методами, например существуют методы GET и POST, используемые для получения и отправки данных соответственно.

Популярность HTTP-запросов объясняется их надежностью. При проектировании как самих запросов, так и предназначенной для их обработки инфраструктуры, закладывается устойчивость к сбоям и способность справляться с самыми разными видами неполадок.

Разрабатываемая в Unity онлайновая игра на базе HTTP-запросов будет, по сути, толстым клиентом (thick client), взаимодействующим с сервером в стиле Ajax. Для сравнения вспомните, как работают современные одностраничные веб-приложения (в отличие от старых добрых веб-страниц, генерируемых на стороне сервера). Хорошее знание принципов старой школы порой заставляет опытных разработчиков делать неверные шаги. Видеоигры зачастую имеют куда более строгие требования к производительности, чем веб-приложения, и именно эта разница в требованиях может влиять на проектные решения.

ВНИМАНИЕ Для веб-приложений и видеоигр понятие времени может иметь разный смысл. При обновлении скачиваемой страницы половина секунды, как правило, вообще ничего не значит, в то время как в разгар активных действий в игре подобная задержка способна испортить все удовольствие. Понятие «быстро» целиком и полностью зависит от ситуации.

Онлайновые игры, как правило, подключаются к специально предназначенному для этой цели серверу; но в целях обучения мы будем подключаться к открытым источникам данных, включая источники прогнозов погоды и доступных для скачивания изображений. Последний раздел этой главы потребует от вас настройки собственного веб-сервера; вы можете просто пропустить этот раздел, хотя я, естественно, на примере программного обеспечения с открытым исходным кодом предлагаю вам максимально простой способ настройки.

В процессе работы над упражнениями этой главы вам много раз придется прибегать к HTTP-запросам, чтобы понять принцип их работы в Unity:

1. Настройка натурной сцены (а именно, создание неба, реагирующего на данные метеонаблюдений).
2. Написание кода, запрашивающего прогноз погоды из Интернета.
3. Анализ ответа и корректировка сцены в соответствии с полученной информацией.
4. Скачивание и отображение графики.
5. Отправка данных на ваш собственный сервер (в нашем случае это будут записи журнала о погоде за прошедшее время).

Игра, на примере которой будет демонстрироваться проект этой главы, не имеет значения. Мы будем добавлять к проекту новые сценарии, никак не затрагивая существующего кода. Например, я воспользовался демонстрационным роликом из главы 2, так как там есть возможность от первого лица наблюдать за изменениями неба. Проект этой главы не имеет непосредственной связи с игровым процессом, но очевидно, что возможность передачи данных по сети пригодится в большинстве

создаваемых вами игр (например, генерировать врагов можно на базе ответов сервера).

Поэтому приступим к работе!

9.1. Создание натурной сцены

Так как мы собираемся скачать информацию о погоде, первым делом следует настроить сцену, в которой мы сможем наблюдать за погодными условиями. Сложнее всего создать небо, поэтому для начала мы займемся назначением текстуры камня геометрии игрового уровня.

Как и в главе 4, я скачал пару изображений с сайта www.textures.com, чтобы назначить их стенам и полу уровня. Напоминаю, что скачанные изображения нужно отредактировать таким образом, чтобы их размер выражался степенями двойки, например 256×256. Затем импортируйте их в проект Unity, создайте материалы и назначьте им изображения (то есть перетащите каждое из них на соответствующую ячейку в редакторе параметров материала). Перетащите материалы на стены и пол в сцене и увеличьте параметр Tiling (попробуйте значение 8 или 9 в одном или в обоих направлениях), чтобы изображение перестало ужасным образом растягиваться по всей поверхности.

Теперь пришло время заняться небом.

9.1.1. Генерирование неба с помощью скайбокса

Начните с импорта изображений для скайбокса, как вы делали в главе 4: скачать подходящие изображения можно с сайта www.93i.de. На этот раз выберем в дополнение к набору TropicalSunnyDay набор DarkStormy (в этом проекте мы создадим более сложный вариант неба). Перетащите эти текстуры на вкладку Project и (как объяснялось в главе 4) выберите в раскрывающемся списке Wrap Mode вариант Clamp.

Теперь создайте материал, который будет использоваться скайбоксом. В верхней части настроек откройте меню Shader, чтобы увидеть список доступных вариантов раскраски. Наведите указатель мыши на раздел Skybox и выберите в дополнительном меню вариант 6-Sided. После этого у материала появятся шесть ячеек для текстур (вместо одной ячейки Albedo, характерной для стандартной раскраски).

Перетащите изображения типа SunnyDay на ячейки для текстур нового материала. Окончания имен текстур соответствуют именам ячеек, на которые их нужно поместить (top, front и т. д.). После этого новый материал можно использовать в качестве скайбокса для сцены.

Перетащите новый материал на поле Skybox в окне Lighting (чтобы его открыть, выберите в меню Window команду Lighting). Щелкните на кнопке Play, и вы увидите нечто напоминающее рис. 9.1.

Замечательно, теперь у нас есть природная сцена! Скайбокс позволяет легко имитировать окружающую среду, в которой оказывается игрок. Однако встроенная в Unity раскраска для скайбокса имеет один серьезный недостаток — изображения нельзя менять, что дает нам совершенно статичное небо. Для решения этой проблемы мы создадим свой вариант раскраски.

Рис. 9.1. Сцена с фоновыми изображениями неба

9.1.2. Настройка управляемой кодом атмосферы

Изображения из набора TropicalSunnyDay дают нам великолепную иллюзию солнечного дня, но как быть, если мы хотим сымитировать переход от солнечной погоды к облачности? Нам потребуется второй набор снимков (демонстрирующих облачное небо), а значит, и новая раскраска для скайбокса.

Как объяснялось в главе 4, раскраска реализуется короткой программой, указывающей, каким образом должно визуализироваться изображение. Это означает, что вы можете программировать новые варианты раскрасок, что нам в данном случае и требуется. Мы создадим новую раскраску, в которой используется два варианта изображений для скайбокса и переход между ними. Впрочем, нужный нам вариант уже существует в коллекции сценариев Unify-сообщества по адресу http://wiki.unity3d.com/index.php?title=SkyboxBlended.

В редакторе Unity создайте новый сценарий раскраски. Для этого вызовите меню с командой Create, как при создании новых сценариев на языке C#, но на этот раз выберите в нем вариант Shader. Присвойте новому ресурсу имя SkyboxBlended и дважды щелкните на нем, чтобы получить доступ к редактированию сценария. Скопируйте код с вики-страницы и вставьте его в наш сценарий. Команда `Shader "Skybox/Blended"` в верхней строке заставляет Unity добавить новую раскраску в категорию Skybox (в которой находится и обычный скайбокс).

ПРИМЕЧАНИЕ Мы не будем углубляться в детали программирования раскрасок. Это достаточно сложная тема, выходящая за рамки темы данной книги. Если вы захотите самостоятельно ее изучить, начать можно отсюда: http://docs.unity3d.com/Manual/ShadersOverview.html.

Теперь для нашего материала можно выбрать раскраску Skybox Blended. Появятся 12 полей текстуры, два набора по шесть изображений. Первым шести полям, как и раньше, следует назначить изображения из набора TropicalSunnyDay, а для остальных текстур используйте набор DarkStormy.

В верхней части настроек новой раскраски находится ползунок Blend. Он контролирует отображаемую долю изображений; когда вы перетаскиваете его из крайнего левого положения в крайнее правое, скайбокс переходит от солнечной к облачной погоде. Для тестирования меняйте положение ползунка и запускайте воспроизведение.

Разумеется, во время игры никто так делать не будет, поэтому давайте напишем код, меняющий вид неба.

Создайте пустой объект и присвойте ему имя Controller. Создайте сценарий и присвойте ему имя `WeatherController`. Перетащите сценарий на пустой объект и введите в него код следующего листинга.

Листинг 9.1. Сценарий перехода от солнечной погоды к облачности

```
using UnityEngine;
using System.Collections;

public class WeatherController : MonoBehaviour {
  [SerializeField] private Material sky; ← Ссылаться можно не только на объекты сцены,
  [SerializeField] private Light sun;      но и на материал на вкладке Project.

  private float _fullIntensity;

  private float _cloudValue = 0f;

  void Start() {
    _fullIntensity = sun.intensity; ← Исходная интенсивность света считается «полной».
  }

  void Update() {
    SetOvercast(_cloudValue);
    _cloudValue += .005f; ← Для непрерывности перехода увеличивайте значение в каждом кадре.
  }

  private void SetOvercast(float value) { ← Корректируем как значение Blend
    sky.SetFloat("_Blend", value);          материала, так и интенсивность света.
    sun.intensity = _fullIntensity - (_fullIntensity * value);
  }
}
```

Я обратил внимание на несколько аспектов данного кода, но его центром является находящийся почти в конце метод `SetFloat()`. До этого момента все уже более-менее знакомо, а с этим методом вы пока не сталкивались. Он задает для материала числовое значение, которое указывается в качестве его первого параметра. В рассматриваемом случае материал обладает свойством Blend (обратите внимание, что свойства материала в коде начинаются с нижнего подчеркивания).

В остальной части кода мы определили несколько переменных, в том числе для материала и света. В случае материала нам потребовалась ссылка на только что созданный нами составной материал для скайбокса, а как обстоят дела с освещением? Дело в том, что при переходе от солнечной к облачной погоде сцена становится темнее; при увеличении параметра Blend мы уменьшаем интенсивность освещения. В качестве основного осветителя сцены выступает направленный источник света; выделите его для доступа к параметрам на панели Inspector.

ПРИМЕЧАНИЕ Усовершенствованная система освещения в Unity (которая называется Enlighten) задействует скайбокс для достижения реалистичных результатов. Но этот подход некорректно работает в случае меняющегося скайбокса, поэтому имеет смысл отключить эту систему. В нижней части окна диалога Lighting можно сбросить флажок Continuous Baking для визуализатора; после

этого обновление будет происходить только после щелчка на кнопке. Установите ползунок Blend для скайбокса в центральное положение, чтобы получить среднюю освещенность сцены, и щелкните на кнопке Build в нижней части окна Lighting, чтобы запечь карты освещенности (к проекту будет добавлена папка Scene; трогать ее не нужно).

В начале работы сценарий инициализирует интенсивность света. Это начальное значение он сохраняет, считая его за «полную» интенсивность. Позднее интенсивность света появится в сценарии, когда нам потребуется приглушить свет.

Затем код на постоянную величину увеличивает значение каждого кадра и использует его для корректировки неба. А именно, вызывает в каждом кадре метод `SetOvercast()`, который инкапсулирует все внесенные в сцену изменения. Назначение метода `SetFloat()` я уже объяснял, так что снова мы на этом останавливаться не будем, последняя же строка меняет интенсивность света.

Теперь выполните воспроизведение сцены. Вы должны получить результат, показанный на рис. 9.2: через пару секунд солнечный день в сцене превратится в пасмурный и облачный.

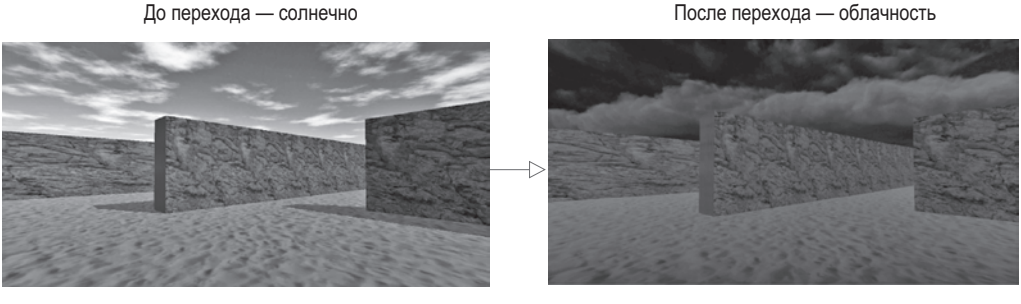


Рис. 9.2. До и после: переход сцены от солнца к облачности

ВНИМАНИЕ Неожиданной особенностью Unity является фиксация в материале изменения параметра Blend. После остановки игры Unity возвращает объекты сцены в исходное состояние, но ресурсы, добавленные непосредственно с вкладки Project (как материал для нашего скайбокса), изменяются насовсем. Такое происходит только в редакторе Unity (изменения не переносятся из игры в игру после развертывания игры за пределами редактора) и может привести к крайне неприятным ошибкам, если забыть о данной особенности.

Здорово наблюдать за переходом от солнца к облачности. Но все это является лишь подготовкой к решению основной задачи: синхронизации погоды в игре в соответствии с реальными погодными условиями. Для этого нам потребуется скачать сводку погоды из Интернета.

9.2. Скачивание сводки погоды из Интернета

Теперь, когда у нас есть сцена на открытом воздухе, можно написать код, скачивающий из Интернета информацию о погоде и корректирующий сцену в соответствии с полученными данными. Эта задача является хорошим примером выборки

данных с помощью HTTP-запросов. В качестве бесплатного источника метеоданных я использую сайт OpenWeatherMap — перейдя по адресу http://openweathermap.org/api, можно воспользоваться прикладным программным интерфейсом (Application Programming Interface, API), который обеспечивает доступ к службе посредством кода, а не графического интерфейса.

ОПРЕДЕЛЕНИЕ Веб-службой, или веб-API, называется подключенный к Интернету сервер, возвращающий данные по запросу. С технической точки зрения никакой разницы между API и веб-узлом не существует; веб-узел является службой, возвращающей данные веб-странице, а браузеры интерпретируют HTML-данные, превращая их в видимые документы.

Код, который вам предстоит написать, структурирован вокруг уже знакомой вам по главе 8 архитектуры диспетчеров. На этот раз у нас будет класс `WeatherManager`, инициализируемый центральным диспетчером диспетчеров. Именно этот класс отвечает за получение и сохранение метеорологических данных, но для этого ему требуется взаимодействовать с Интернетом.

Поэтому мы создадим вспомогательный класс `NetworkService`, который позаботится о подключении к Интернету и HTTP-запросах. После чего класс `WeatherManager` сможет заставить класс `NetworkService` отправлять запросы и возвращать полученные ответы. Принцип функционирования такой структуры кода показан на рис. 9.3.

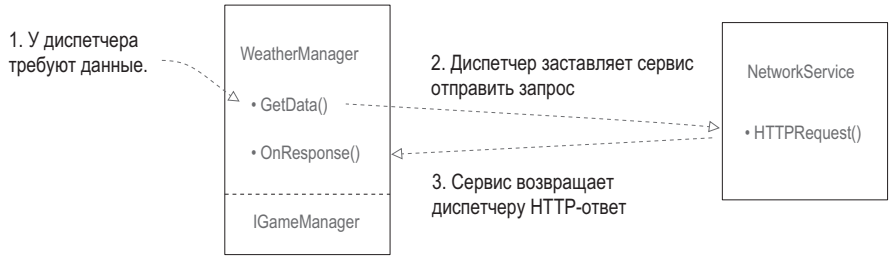


Рис. 9.3. Структура кода, передающего данные по сети

Очевидно, что этот механизм будет работать только при наличии у класса `WeatherManager` доступа к объекту `NetworkService`. Для решения этой задачи мы создадим объект в сценарии `Managers` и в момент инициализации различных диспетчеров будем вставлять в них объект `NetworkService`. В результате ссылку на этот объект получит не только диспетчер `WeatherManager`, но и все диспетчеры, которые вы создадите после этого.

Воспроизводить архитектуру из главы 8 мы начнем с копирования сценариев `ManagerStatus` и `IGameManager` (напоминаю, что `IGameManager` — это интерфейс, который должны реализовывать все диспетчеры, в то время как `ManagerStatus` — это перечисление, которым пользуется `IGameManager`). В сценарий `IGameManager` следует внести небольшие изменения, связанные с появлением класса `NetworkService`, поэтому создайте новый сценарий `NetworkService` (пока оставьте его пустым, код мы напишем позже), а затем отредактируйте сценарий `IGameManager` в соответствии с листингом 9.2.

Листинг 9.2. Добавление в сценарий IGameManager класса NetworkService

```
public interface IGameManager {
  ManagerStatus status {get;}

  void Startup(NetworkService service); ← Функция Startup теперь принимает один параметр – вставленный объект.
}
```

Далее создадим сценарий `WeatherManager` для реализации нашего слегка отредактированного интерфейса. Добавьте в новый сценарий содержимое следующего листинга.

Листинг 9.3. Начальный код сценария WeatherManager

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

public class WeatherManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  // Сюда добавляется значение облачности (см. листинг 9.8)
  private NetworkService _network;

  public void Startup(NetworkService service) {
    Debug.Log("Weather manager starting...");

    _network = service; ← Сохранение вставленного объекта NetworkService.

    status = ManagerStatus.Started;
  }
}
```

Этот начальный вариант сценария `WeatherManager` не выполняет никаких действий. Пока это всего лишь минимальное количество кода, необходимое сценарию `IGameManager` для реализации класса: здесь объявляется свойство интерфейса `status` и выполняется функция `Startup()`. Этот пустой каркас мы заполним в следующих разделах. А пока скопируйте сценарий `Managers` из главы 8 и отредактируйте его таким образом, чтобы он запускал сценарий `WeatherManager` (как показано в следующем листинге).

Листинг 9.4. Сценарий Managers, инициализирующий сценарий WeatherManager

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

[RequireComponent(typeof(WeatherManager))] ← Вместо диспетчеров персонажа и инвентаря требуется новый диспетчер погоды.

public class Managers : MonoBehaviour {
  public static WeatherManager Weather {get; private set;}

  private List<IGameManager> _startSequence;

  void Awake() {
    Weather = GetComponent<WeatherManager>();
```

```
    _startSequence = new List<IGameManager>();
    _startSequence.Add(Weather);
    StartCoroutine(StartupManagers());
  }

  private IEnumerator StartupManagers() {
    NetworkService network = new NetworkService(); ← Создаются экземпляры объекта NetworkService для вставки во все диспетчеры.

    foreach (IGameManager manager in _startSequence) {
      manager.Startup(network); ← Во время загрузки диспетчеров передаем им сетевую службу.
    }

    yield return null;

    int numModules = _startSequence.Count;
    int numReady = 0;

    while (numReady < numModules) {
      int lastReady = numReady;
      numReady = 0;

      foreach (IGameManager manager in _startSequence) {
        if (manager.status == ManagerStatus.Started) {
          numReady++;
        }
      }

      if (numReady > lastReady)
      Debug.Log("Progress: " + numReady + "/" + numModules);

      yield return null;
    }

    Debug.Log("All managers started up");
  }
}
```

Это всё, что нужно для архитектуры, заданной сценарием `Managers`. Как и в предыдущей главе, создайте в сцене пустой объект, который будет играть роль диспетчера, и присоедините к нему сценарии `Managers` и `WeatherManager`. При корректной настройке он будет выводить на консоль сообщения о загрузке, чем его функция пока и ограничится.

Все, на этом мы завершаем все наши подготовительные дела, сводящиеся в основном к копированию, и можем приступать к написанию кода для передачи данных по сети.

9.2.1. Запрос веб-данных через сопрограмму

Сценарий `NetworkService` пока пуст, и в него нужно ввести код реализации HTTP-запросов. Основным классом, сведения о котором вам потребуются, является `WWW`. В Unity этот класс обеспечивает взаимодействие с Интернетом. Создание экземпляра объекта `WWW` с использованием URL-адреса приводит к отправке запроса по этому адресу.

Сопрограммы позволяют классу `WWW` ждать завершения запроса. В первый раз сопрограммы упоминаются в главе 3, где мы использовали их для остановки кода на определенный период времени. Напомню определение: сопрограммами называются специальные функции, которые запускаются в фоновом режиме основной программы, в цикле выполняют код и возвращают результат в программу. Вместе с методом `StartCoroutine()` мы использовали ключевое слово `yield`, которое заставляло сопрограмму на время остановиться, вернуть управление программе, а в следующем кадре снова начать свою работу.

В главе 3 сопрограмма возвращала метод `WaitForSeconds()`, что останавливало работу функции на указанное количество секунд. В случае с классом `WWW` выполнение функции будет прервано до завершения сетевого запроса. Работа программы в данном случае напоминает асинхронные Ajax-вызовы в веб-приложениях: сначала вы посылаете запрос, потом продолжаете выполнение остальной части программы, а через некоторое время получаете ответ.

Это была теория, давайте, наконец, писать код!

Первым делом откройте сценарий `NetworkService` и замените присутствующий там шаблон содержимым следующего листинга.

Листинг 9.5. Выполнение HTTP-запросов в сценарии NetworkService

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

public class NetworkService {
  private const string xmlApi = ← URL-адрес для отправки запроса.
    "http://api.openweathermap.org/data/2.5/weather?q=Chicago,us&mode=xml";

  private bool IsResponseValid(WWW www) { ← Проверка ответа на наличие ошибок.
    if (www.error != null) {
      Debug.Log("bad connection");
      return false;
    }
    else if (string.IsNullOrEmpty(www.text)) {
      Debug.Log("bad data");
      return false;
    }
    else { // все хорошо
      return true;
    }
  }

  private IEnumerator CallAPI(string url, Action<string> callback) {
    WWW www = new WWW(url); ← HTTP-запрос, отправленный путем создания веб-объекта.
    yield return www; ← Пауза в процессе скачивания.

    if (!IsResponseValid(www))
    yield break; ← Прерывание сопрограммы в случае ошибки.

    callback(www.text); ← Делегат может быть вызван так же, как и исходная функция.
```

```
  }

  public IEnumerator GetWeatherXML(Action<string> callback) {
    return CallAPI(xmlApi, callback); ← Каскад ключевых слов yield в вызывающих друг друга методах
сопрограммы.
  }
}
```

Надеюсь, вы помните объяснение особенностей данного проекта: диспетчер `WeatherManager` заставит сценарий `NetworkService` извлечь нужные данные. Фактически, весь этот код пока не запускается; вы написали код, который позднее будет вызван сценарием `WeatherManager`. Объяснять смысл этого листинга я начну снизу.

Включенные друг в друга методы сопрограммы

Помещенный в сопрограмму метод `GetWeatherXML()` заставляет сценарий `NetworkService` создать HTTP-запрос. Обратите внимание, что в качестве типа возвращаемого значения указан тип `IEnumerator`, который объявляется во всех методах, используемых в сопрограмме.

Отсутствие ключевого слова `yield` в методе `GetWeatherXML()` на первый взгляд может показаться странным. Ведь именно это ключевое слово останавливает выполнение сопрограммы, а значит, его наличие предполагается по умолчанию. Но дело в том, что у нас есть набор вставленных друг в друга методов. Если первый метод сопрограммы вызывает какой-то другой метод, который останавливается, выполнив только часть своего кода, остановка и возобновление работы сопрограммы будут осуществляться внутри второго метода. То есть в нашем случае ключевое слово `yield` в методе `CallAPI()` прерывает сопрограмму, начавшуюся в методе `GetWeatherXML()`. Этот принцип работы иллюстрирует рис. 9.4.

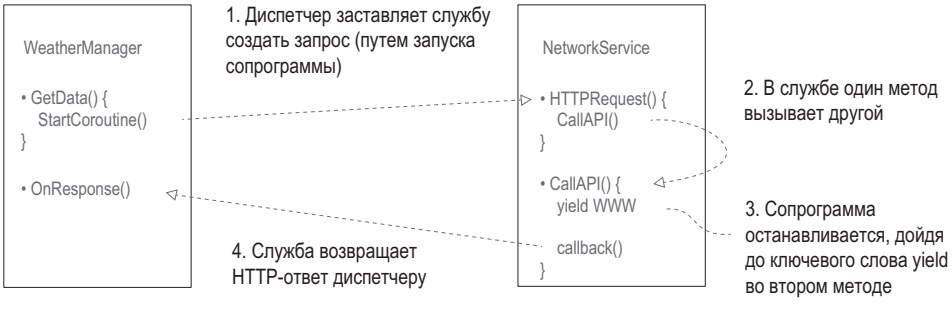


Рис. 9.4. Схема работы сопрограммы, управляющей передачей данных по сети

Следующей потенциально непонятной вещью является параметр `callback` типа `Action`.

Как работает обратный вызов

В начале сопрограммы вызывается метод с параметром `callback`, который принадлежит типу `Action`. Но что это за тип?

ОПРЕДЕЛЕНИЕ Тип Action является делегатом (в C# есть ряд объяснений этому понятию, но я изложу самый простой). Делегаты представляют собой ссылки на какой-то другой метод/функцию. Делегат позволяет сохранить функцию (или, точнее, указатель на нее) в переменной и передать эту переменную в качестве параметра другой функции.

Если вы раньше не сталкивались с понятием делегата, представьте, что они позволяют передавать функции точно так же, как числа или строки, и вызывать их позже. Без делегатов мы могли бы вызывать функции только напрямую. Делегаты дают возможность сообщить коду о других методах, которые можно вызвать позже. Такое поведение требуется во многих случаях, особенно при реализации функций *обратного вызова*.

ОПРЕДЕЛЕНИЕ Обратным вызовом (callback) называется вызов функции, используемой для обмена данными с вызывающим объектом. Объект A может сообщить объекту B об одном из своих методов. Позднее объект B может вызвать этот метод для обмена данными с объектом A.

Например, в данном случае обратный вызов позволил нам, дождавшись завершения HTTP-запроса, переслать обратно полученный ответ. Код метода `CallAPI()` сначала отправляет HTTP-запрос, затем останавливается, ожидая завершения этого запроса, и наконец с помощью метода `callback()` возвращает полученный ответ.

Обратите внимание на синтаксис `<>`, используемый с ключевым словом `Action`; указанный в угловых скобках тип требуется параметрам, чтобы подойти к этому делегату. Другими словами, функция, на которую указывает делегат `Action`, должна иметь параметры указанного типа. В данном случае параметром является единственная строка, поэтому у метода обратного вызова должна быть примерно такая сигнатура:

```
MethodName(string value)
```

Возможно, вы лучше поймете концепцию обратного вызова, посмотрев пример его применения в листинге 9.6. Надеюсь, мои объяснения помогут вам понять, что именно произойдет, когда вы увидите этот дополнительный код.

Остаток листинга 9.5 достаточно очевиден. Метод `IsResponseValid()` проверяет наличие ошибок в HTTP-ответе. Возможны ошибки двух типов: сбой запроса из-за проблем с интернет-подключением и некорректность возвращенных данных. Константа `const` объявляется с URL-адресом, по которому код будет отправлять запросы. (Если хотите, можете поменять этот URL-адрес для получения сведений о погоде из другого места.)

Применение кода передачи данных по сети

Этот сценарий включает в себя код `NetworkService`. Воспользуемся им в сценарии `WeatherManager` — следующий листинг демонстрирует дополнения, которые туда нужно внести.

Листинг 9.6. Добавление в сценарий WeatherManager кода для работы с NetworkService

```
...
public void Startup(NetworkService service) {
  Debug.Log("Weather manager starting...");
```

```
  _network = service;
  StartCoroutine(_network.GetWeatherXML(OnXMLDataLoaded)); ← Начинаем загрузку данных из Интернета.

  status = ManagerStatus.Initializing; ← Меняем состояние со Started на Initializing.
}

public void OnXMLDataLoaded(string data) { ← Метод обратного вызова сразу после загрузки данных.
  Debug.Log(data);

  status = ManagerStatus.Started;
}
...
```

В код этого диспетчера были внесены три основных изменения: запуск сопрограммы для скачивания данных из Интернета, задание другого состояния загрузки и определение метода обратного вызова для получения ответа.

С началом сопрограммы все просто. Все сложные аспекты работы с сопрограммами реализованы в сценарии `NetworkService`, поэтому сейчас вам остается только вызвать метод `StartCoroutine()`. Потом вы меняете состояние загрузки, так как на самом деле инициализация диспетчера не завершена; сначала он должен получить данные из Интернета.

ВНИМАНИЕ Методы передачи данных по сети всегда нужно начинать с функции StartCoroutine(); обычный вызов в их случае неприменим. Об этом легко забыть, так как создание веб-объектов за пределами сопрограммы не приводит к ошибкам компиляции.

Вызов метода `StartCoroutine()` должен сопровождаться активацией. То есть нужно добавить скобки, а не просто указать имя функции. В нашем случае в качестве одного из параметров методу сопрограммы требуется функция обратного вызова, которую нам следует задать. Для обратного вызова мы воспользуемся функцией `OnXMLDataLoaded()`; обратите внимание, что ее параметр относится к типу `string`, что совпадает с объявлением `Action<string>` в сценарии `NetworkService`. Пока что от функции обратного вызова нам много не требуется — она просто проверяет корректность полученных данных и выводит их на консоль. После этого последняя строчка функции меняет состояние загрузки диспетчера, уведомляя о том, что теперь он полностью загружен.

Щелкните на кнопке Play для воспроизведения кода. При наличии хорошего интернет-подключения на консоли быстро появятся данные. Это всего лишь длинная строка, но отформатированная особым образом, что дает нам возможность воспользоваться содержащейся в ней информацией.

9.2.2. Парсинг текста в формате XML

Существующие в виде длинных строк данные обычно состоят из отдельных битов информации. Эти биты информации извлекаются методом синтаксического разбора, или, как его еще называют, *парсинга*.

ОПРЕДЕЛЕНИЕ Парсингом (parsing) называется процесс анализа фрагментов кода и его разделения на отдельные фрагменты данных.

Для парсинга строки она должна быть отформатирована таким образом, чтобы у вас (точнее, у кода-анализатора) была возможность идентификации отдельных фрагментов. Существует пара стандартных форматов передачи данных через Интернет; наиболее распространенным из них является XML.

ОПРЕДЕЛЕНИЕ Язык XML (Extensible Markup Language — расширяемый язык разметки) задает правила структурированного кодирования документов аналогично HTML-страницам.

К счастью, Unity (точнее, Mono — встроенная в Unity среда разработки) предлагает функциональность для анализа XML-кода. Запрошенный нами прогноз погоды имеет формат XML, поэтому добавим в сценарий `WeatherManager` код, анализирующий ответ и извлекающий из него информацию об облачности. Код вы найдете в Интернете, он довольно длинный, но нас интересует только фрагмент, содержащий, к примеру, такие данные, как `<clouds value="40" name="scattered clouds"/>`.

Мы не только добавим код, анализирующий XML, но и воспользуемся системой сообщений, знакомой нам по главе 6. Дело в том, что нам нужно уведомить сцену о скачанных и проанализированных данных. Создайте сценарий с именем `Messenger` и вставьте в него код со страницы http://wiki.unity3d.com/index.php/CSharpMessenger_Extended.

После этого нужно будет создать сценарий с именем `GameEvent` (его код приведен в следующем листинге). Как объяснялось в главе 6, эта система сообщений дает нам замечательный несвязанный способ информировать остальную часть программы о событиях.

Листинг 9.7. Код сценария GameEvent

```
public static class GameEvent {
  public const string WEATHER_UPDATED = "WEATHER_UPDATED";
}
```

Теперь, когда у нас есть система сообщений, скорректируйте сценарий `WeatherManager`, как показано в следующем листинге.

Листинг 9.8. Парсинг XML-кода в сценарии WeatherManager

```
...
using System;
using System.Xml; ← Обязательно добавьте необходимые инструкции using.
...
public float cloudValue {get; private set;} ← Облачность редактируется внутренне, в остальных местах это свойство предназначено только для чтения.
...
public void OnXMLDataLoaded(string data) {
  XmlDocument doc = new XmlDocument();
  doc.LoadXml(data); ← Разбиваем XML-код на структуру с возможностью поиска.
  XmlNode root = doc.DocumentElement;

  XmlNode node = root.SelectSingleNode("clouds"); ← Извлекаем из данных один узел.
  string value = node.Attributes["value"].Value;
  cloudValue = Convert.ToInt32(value) / 100f; ← Преобразуем значение в число типа float в диапазоне от 0 до 1.
  Debug.Log("Value: " + cloudValue);
```

```
    Messenger.Broadcast(GameEvent.WEATHER_UPDATED); ← Рассылка сообщения для информирования
                                                      остальных сценариев.
    status = ManagerStatus.Started;
}
...
```

Как видите, самые важные изменения появились внутри метода `OnXMLDataLoaded()`. Раньше он всего лишь выводил данные на консоль, позволяя нам убедиться в корректности их передачи. Теперь же мы добавили в метод команды, анализирующие XML-код.

Мы начинаем с создания нового пустого XML-документа; он послужит контейнером для разбираемой XML-структуры. Следующая строка разбивает строку данных, превращая ее в структуру из XML-документа. После чего мы начинаем с корня XML-дерева, чтобы в последующем коде можно было выполнять поиск по этому дереву.

На данном этапе в XML-структуре уже можно искать узлы, извлекая в итоге отдельные биты информации. Нас интересует только узел `<clouds>`. Первым делом мы ищем его в XML-документе, затем извлекаем из него атрибут `value`. Этот атрибут задает облачность в виде целого числа в диапазоне от 0 до 100, нам же для последующей корректировки вида сцены нужно число типа `float` в диапазоне от 0 до 1. Преобразование в данном случае осуществляется простой математической операцией.

Наконец, после извлечения из полных данных информации об облачности мы рассылаем сообщение об обновлении погодных данных. Пока что никто этого сообщения не слышит, но издатель не обязан ничего знать о подписчиках (собственно, в этом и состоит смысл несвязанной системы рассылки сообщений). Позднее мы добавим в сцену подписчиков.

Замечательно! Мы написали код, анализирующий XML-данные! Но, перед тем как воспользоваться полученным значением для изменения вида сцены, я хотел бы рассмотреть еще один формат передачи данных.

9.2.3. Парсинг текста в формате JSON

Перед тем как перейти к следующему этапу проекта, давайте познакомимся с альтернативным форматом передачи данных. Этот формат распространен так же, как XML, и называется JSON.

ОПРЕДЕЛЕНИЕ JSON (JavaScript Object Notation) — это текстовый формат обмена данными на основе JavaScript, назначение которого аналогично XML. Собственно, формат JSON проектировался как облегченная альтернатива XML. Хотя JSON-синтаксис является производным от языка JavaScript, на конкретный язык этот формат не ориентирован и легко используется с самыми разными языками программирования.

В отличие от XML-парсера, в среде разработки Mono парсер формата JSON отсутствует. Впрочем, есть ряд доступных для скачивания парсеров, например MiniJSON (https://gist.github.com/darktable/1411710). Создайте сценарий с именем `MiniJSON` и вставьте туда код парсера. Теперь вы можете пользоваться этой библиотекой для анализа данных в формате JSON. Данные в формате XML мы получали от сервиса

OpenWeatherMap, но иногда часть данных оттуда поступает в формате JSON. Для получения таких данных отредактируйте сценарий `NetworkService` в соответствии со следующим листингом.

Листинг 9.9. Заставляем сценарий NetworkService запрашивать JSON-, а не XML-данные

```
...
private const string jsonApi = ← URL-адрес на этот раз выглядит чуть иначе.
  "http://api.openweathermap.org/data/2.5/weather?q=Chicago,us";
...
public IEnumerator GetWeatherJSON(Action<string> callback) {
  return CallAPI(jsonApi, callback);
}
...
```

Фактически, у нас тот же самый код для скачивания XML-данных, просто в нем фигурирует немного другой URL-адрес. Этот запрос возвращает те же самые данные, что и раньше, но в другом формате. И на этот раз мы уже будем искать фрагмент кода `"clouds":{"all":40}`.

Большое количество дополнительного кода теперь не потребуется. Ведь мы выделили код запросов в аккуратные отдельные функции, поэтому ничего сложного в добавлении последующих HTTP-запросов нет. Видите, как здорово! Давайте отредактируем сценарий `WeatherManager`, заставив его запрашивать данные в формате JSON, а не XML. Необходимый для этого код показан в следующем листинге.

Листинг 9.10. Заставляем сценарий WeatherManager запрашивать JSON-данные

```
...
using MiniJSON; ← Обязательно добавьте инструкцию using.
...
public void Startup(NetworkService service) {
  Debug.Log("Weather manager starting...");

  _network = service;
  StartCoroutine(_network.GetWeatherJSON(OnJSONDataLoaded)); ← Измененный сетевой запрос.

  status = ManagerStatus.Initializing;
}
...
public void OnJSONDataLoaded(string data) {
  Dictionary<string, object> dict; ← Разбираем не созданный нами XML-контейнер, а содержимое словаря.
  dict = Json.Deserialize(data) as Dictionary<string,object>;

  Dictionary<string, object> clouds =
    (Dictionary<string,object>)dict["clouds"];
  cloudValue = (long)clouds["all"] / 100f;
  Debug.Log("Value: " + cloudValue);

  Messenger.Broadcast(GameEvent.WEATHER_UPDATED);
  status = ManagerStatus.Started;
}
...
```

Синтаксис изменился, но функциональность кода осталась прежней.

Как видите, код для работы с данными в формате JSON напоминает код для работы с данными в формате XML. Основное отличие состоит в том, что парсер JSON-данных работает со стандартным словарем, а не с нестандартным XML-контейнером. В коде встречается процедура *десериализации*, возможно, вы не знаете, что это такое.

ОПРЕДЕЛЕНИЕ Десериализация (deserialization) представляет собой процесс, обратный сериализации. То есть данные преобразуются в форму, доступную для передачи и сохранения, например, в JSON-строку.

В сценарии изменился только синтаксис, а все операции остались неизменными. Извлекаем из фрагмента данных значение (по какой-то причине на этот раз оно содержится в переменной с именем `all`, но это странности API) и выполняем все то же несложное математическое преобразование к значению типа `float` в диапазоне от 0 до 1.

И вот теперь настало время применить полученное значение к нашей сцене.

9.2.4. Изменение вида сцены на базе данных о погоде

Независимо от того, каким именно способом отформатированы данные, как только из полученного ответа извлечено значение облачности, мы можем использовать его в методе `SetOvercast()` сценария `WeatherController`. Строка данных в формате XML или JSON в конечном итоге превратилась в набор слов и чисел. И одно из таких чисел метод `SetOvercast()` принимает в качестве параметра. В разделе 9.1.2 мы использовали число, увеличивавшееся в каждом кадре, но также легко мы можем вставить в код значение, которое нам вернул сервис с сайта прогнозов погоды.

Следующий листинг демонстрирует сценарий `WeatherController` после внесения в него изменений.

Листинг 9.11. Сценарий WeatherController, реагирующий на скачанные метеорологические данные

```
using UnityEngine;
using System.Collections;

public class WeatherController : MonoBehaviour {
  [SerializeField] private Material sky;
  [SerializeField] private Light sun;

  private float _fullIntensity;

  void Awake() { ← Добавляем/удаляем подписчиков на событие.
    Messenger.AddListener(GameEvent.WEATHER_UPDATED, OnWeatherUpdated);
  }
  void OnDestroy() {
    Messenger.RemoveListener(GameEvent.WEATHER_UPDATED, OnWeatherUpdated);
  }

  void Start() {
    _fullIntensity = sun.intensity;
  }

  private void OnWeatherUpdated() {
```

```
    SetOvercast(Managers.Weather.cloudValue); ←  Используем значение облачности
  }                                              из сценария WeatherManager.

  private void SetOvercast(float value) {
    sky.SetFloat("_Blend", value);
    sun.intensity = _fullIntensity - (_fullIntensity * value);
  }
}
```

Обратите внимание, что изменения сводятся не только к дополнениям; удалены несколько фрагментов тестового кода, а именно, мы удалили локальное значение облачности, увеличивавшееся в каждом кадре; оно нам больше не требуется, так как мы будем пользоваться значением из сценария `WeatherManager`.

Подписчики добавляются и удаляются в методах `Awake()` и `OnDestroy()` — это методы класса `MonoBehaviour`, вызываемые, когда объект активируется или удаляется. Каждый такой подписчик является частью системы рассылки сообщений. При получении сообщения он вызывает метод `OnWeatherUpdated()`, получающий значение облачности из сценария `WeatherManager` и, в свою очередь, вызывающий метод `SetOvercast()`, использующий это значение. В результате вид сцены контролируется скачанными из Интернета метеорологическими данными.

Выполните воспроизведение сцены и убедитесь, что небо обновляется в соответствии с информацией об облачности из прогноза погоды. Скорее всего, вы увидите, что запрос погоды занимает некоторое время; в настоящей игре до завершения загрузки неба имело бы смысл спрятать сцену за экраном загрузки.

ДРУГИЕ СПОСОБЫ ПЕРЕДАЧИ ДАННЫХ ПО СЕТИ В ИГРАХ

При всей устойчивости к сбоям и надежности HTTP-запросов задержка в получении ответа для большинства игр является слишком существенным недостатком. Поэтому эти запросы хороши при относительно медленной передаче сообщений на сервер (например, значений перемещения в играх со сменой хода или данных о набранных очках), но для таких игр, как сетевой шутер от первого лица, требуется другой подход к передаче данных по сети.

Существуют различные технологии взаимодействия, а также приемы, позволяющие скомпенсировать запаздывание. К примеру, в Unity применяется сетевая библиотека RakNet в системе удаленного вызова процедур (Remote Procedure Call, RPC).

Передовые технологии для создания сетевых игр — сложная тема, рассмотрение которой выходит за рамки темы данной книги. Вы можете самостоятельно заняться ее изучением, начав со страницы http://docs.unity3d.com/ru/current/Manual/NetworkReferenceGuide.html.

Теперь, когда вы знаете, как получить из Интернета числовые и строковые данные, давайте рассмотрим способы работы с изображениями.

9.3. Добавление рекламного щита

Ответы от веб-сервиса почти всегда представляют собой текстовые строки в формате XML или JSON, но по сети передается и множество других данных. Кроме текстовой информации чаще всего запрашиваются изображения. Объект `WWW` в Unity применяется в числе прочего и для их скачивания.

Способ решения этой задачи вы изучите на примере создания рекламного щита, на котором отображается скачанное из Интернета изображение. Создаваемый код будет

решать две задачи: скачивание изображения из Интернета и назначение этого изображения рекламному щиту. Третьей задачей станет редактирование кода, позволяющее сохранить изображение для его показа на нескольких рекламных щитах.

9.3.1. Загрузка изображений из Интернета

Первым делом напишем код загрузки изображения. Для тестирования мы скачаем свободно распространяемые фотографии пейзажей. Пример такой фотографии показан на рис. 9.5. Загруженного изображения на рекламном щите вы пока не увидите; сценарий отображения картинки в сцене будет представлен в следующем разделе, но до этого мы напишем код получения графического файла.

Рис. 9.5. Озеро Морейн в национальном парке Банф, Канада

Структура кода, предназначенного для скачивания данных и графики, практически одинакова. За загрузку изображений будет отвечать помещенный в отдельный модуль очередной диспетчер (он называется `ImagesManager`). Напомню, что за подключение к Интернету и отправку HTTP-запросов отвечает сценарий `NetworkService`, к которому и будет обращаться наш новый диспетчер. Первым делом отредактируем код сценария `NetworkService`. Следующий листинг добавляет туда возможность скачивания изображений.

Листинг 9.12. Скачивание изображений в сценарии NetworkService

```
...
private const string webImage = ← Эта константа с другими URL-адресами помещается в верхнюю часть сценария.
  "http://upload.wikimedia.org/wikipedia/commons/c/c5/
  Moraine_Lake_17092005.jpg";
...
public IEnumerator DownloadImage(Action<Texture2D> callback) { ← Этот обратный вызов вместо строки принимает объекты типа Texture2D.
  WWW www = new WWW(webImage);
  yield return www;
  callback(www.texture);
}
...
```

Код скачивания изображений выглядит практически идентично коду скачивания данных. Основным отличием является тип метода обратного вызова; отметьте, что на этот раз обратный вызов вместо строк работает с объектами типа Texture2D. Дело в типе возвращаемого ответа: раньше скачивалась строка данных, теперь — изображение. Следующий листинг содержит код нового диспетчера. Создайте сценарий `ImagesManager` и скопируйте туда код.

Листинг 9.13. Диспетчер ImagesManager, запрашивающий и сохраняющий изображения

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

public class ImagesManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  private NetworkService _network;

  private Texture2D _webImage; ← Переменная для сохранения скачанного изображения.

  public void Startup(NetworkService service) {
    Debug.Log("Images manager starting...");

    _network = service;

    status = ManagerStatus.Started;
  }

  public void GetWebImage(Action<Texture2D> callback) {
    if (_webImage == null) { ← Проверяем, нет ли уже сохраненного изображения.
      StartCoroutine(_network.DownloadImage(callback));
    }
    else {
      callback(_webImage); ← При наличии сохраненного изображения сразу
    }                        активируется обратный вызов (без скачивания).
  }
}
```

Самой интересной частью этого кода является метод `GetWebImage()`; все остальное в сценарии представляет собой стандартные свойства и методы, реализующие интерфейс диспетчера. Именно метод `GetWebImage()` возвращает (через функцию обратного вызова) скачанное из сети изображение. Первым делом мы проверяем, не содержит ли уже переменная `_webImage` сохраненное изображение: в случае отрицательного результата проверки активируется сетевой вызов для скачивания. Если же в переменной `_webImage` уже есть сохраненное изображение, метод `GetWebImage()` возвращает его (чтобы не скачивать заново).

ПРИМЕЧАНИЕ Пока мы еще не загрузили и не сохранили ни одного изображения, то есть переменная _webImage пуста. При этом уже есть код, указывающий, что делать при наличии сохраненного изображения, поэтому в следующих разделах вам останется только слегка его отредактировать. Процесс редактирования вынесен в отдельный раздел, так как он включает в себя несколько хитрых приемов.

Разумеется, диспетчер `ImagesManager`, как и все остальные модули диспетчеров, нужно добавить в сценарий менеджеров; этот процесс демонстрируется в следующем листинге.

Листинг 9.14. Добавление нового диспетчера в сценарий Managers.cs

```
...
[RequireComponent(typeof(ImagesManager))]
...
public static ImagesManager Images {get; private set;}
...
void Awake() {
  Weather = GetComponent<WeatherManager>();
  Images = GetComponent<ImagesManager>();

  _startSequence = new List<IGameManager>();
  _startSequence.Add(Weather);
  _startSequence.Add(Images);

  StartCoroutine(StartupManagers());
}
...
```

В отличие от настроек диспетчера `WeatherManager`, при начальной загрузке диспетчера `ImagesManager` метод `GetWebImage()` автоматически не вызывается. Вместо этого код ждет процесса активации, который мы запрограммируем в следующем разделе.

9.3.2. Вывод изображения на щите

Только что написанный нами диспетчер `ImagesManager` ничего не сделает, пока мы его не активируем. Поэтому создадим рекламный щит, который будет вызывать методы в сценарии `ImagesManager`. Создайте куб и поместите его в центре сцены, введя в поля Position значения 0, 1.5, –5, а в поля Scale — значения 5, 3, 0.5. Результат показан на рис. 9.6.

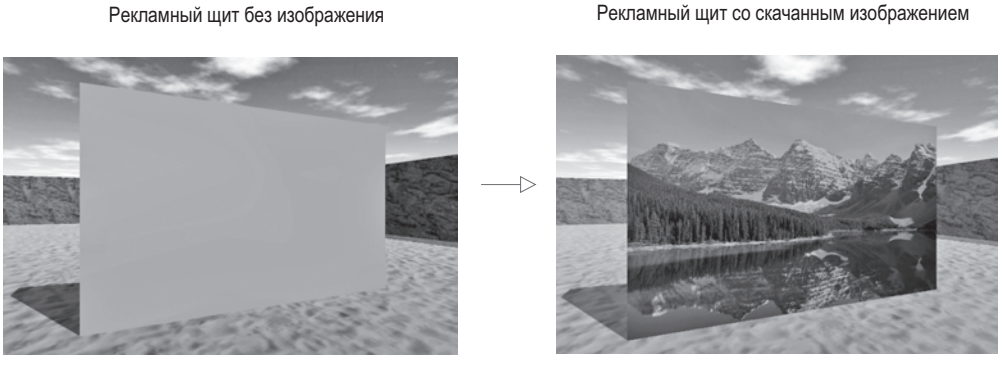


Рис. 9.6. Рекламный щит до и после вывода скачанного из Интернета изображения

Мы создадим устройство, управляющееся так же, как меняющий цвета монитор из главы 8. Скопируйте сценарий `DeviceOperator` и свяжите его с объектом player.

Возможно, вы помните, что этот сценарий срабатывает рядом с устройствами при нажатии клавиши Fire3 (она задается в настройках ввода проекта и в данном случае ей соответствует левая клавиша Command). Кроме того, создайте для устройства Billboard сценарий с именем `WebLoadingBillboard`, свяжите его с рекламным щитом и скопируйте в него код следующего листинга.

Листинг 9.15. Сценарий WebLoadingBillboard для устройства

```
using UnityEngine;
using System.Collections;

public class WebLoadingBillboard : MonoBehaviour {
  public void Operate() {
    Managers.Images.GetWebImage(OnWebImage); ← Вызов метода в сценарии ImagesManager.
  }

  private void OnWebImage(Texture2D image) {
    GetComponent<Renderer>().material.mainTexture = image; ← Скачанное изображение назначается материалу во время обратного вызова.
  }
}
```

Этот код решает две основные задачи: в процессе эксплуатации устройства вызывает метод `ImagesManager.GetWebImage()` и применяет изображение из функции обратного вызова. Текстуры назначены материалам, так что вы можете менять их в материале рекламного щита. Вид щита во время игры показан на рис. 9.6.

СКАЧИВАНИЕ ДРУГИХ РЕСУРСОВ

Скачивание изображений с помощью объекта WWW является достаточно простой операцией, а как обстоят дела с остальными видами ресурсов, такими как сеточные объекты и шаблоны экземпляров? Объект WWW обладает свойствами, связанными с текстом и графикой, поэтому в случае других ресурсов ситуация несколько усложняется.

Впрочем, в Unity есть механизм AssetBundles, позволяющий скачивать что угодно. Коротко говоря, вы сначала упаковываете ресурсы в пакет (bundle), который затем скачивается, и Unity извлекает их из пакета. Подробное рассмотрение процесса создания и скачивания пакетов выходит за рамки темы данной книги; но при желании изучить эту тему вы можете начать с руководства, которое найдете по адресу http://docs.unity3d.com/ru/current/Manual/AssetBundlesIntro.html.

Потрясающе! Скачанное нами изображение появилось на рекламном щите. Но мы можем оптимизировать код, добавив в него возможность работы с несколькими щитами. Именно этим мы и займемся в следующем разделе.

9.3.3. Кэширование скачанного изображения

Как упоминалось в разделе 9.3.1, сценарий `ImagesManager` пока не сохраняет скачанное изображение. Поэтому для нескольких рекламных щитов изображение придется скачивать несколько раз. Это неэффективно, ведь мы каждый раз делаем уже выполненную работу. Чтобы решить проблему, заставим сценарий `ImagesManager` *кэшировать* скачанные изображения.

ОПРЕДЕЛЕНИЕ Кэшировать (cache) означает сохранить на локальной машине. Наиболее распространенный (но не единственный!) контекст касается изображений, скачанных из Интернета.

Ключом к решению этой задачи станет функция обратного вызова в сценарии `ImagesManager`, сначала сохраняющая изображение, а затем выполняющая обратный вызов из сценария `WebLoadingBillboard`. Решение в данном случае является нетривиальным (в отличие от текущего кода, в котором фигурирует обратный вызов из сценария `WebLoadingBillboard`), потому что наш код заранее не знает, каким будет обратный вызов из сценария `WebLoadingBillboard`. Другими словами, мы не можем добавить в сценарий `ImagesManager` метод, вызывающий определенный метод в сценарии `WebLoadingBillboard`, так как коду неизвестно, что это за метод. Найти выход из положения нам поможет *лямбда-функция*.

ОПРЕДЕЛЕНИЕ Лямбда-функцией (lambda function), или анонимной функцией, называется функция, не имеющая имени. Обычно лямбда-функции создаются «на лету» внутри других функций.

Лямбда-функции представляют собой нетривиальный инструмент кодирования, поддерживаемый рядом языков программирования, в том числе C#. Воспользовавшись лямбда-функцией для обратного вызова в сценарии `ImagesManager`, мы сможем создать функцию обратного вызова «на лету», взяв метод, переданный из сценария `WebLoadingBillboard`. Это избавит нас от необходимости заранее знать, какой из методов будет вызываться, ведь лямбда-функции изначально не существует! Следующий листинг демонстрирует, как это все выглядит на практике.

Листинг 9.16. Обратный вызов с помощью лямбда-функции в сценарии ImagesManager

```
...
using System;
...
public void GetWebImage(Action<Texture2D> callback) {
  if (_webImage == null) {
    StartCoroutine(_network.DownloadImage((Texture2D image) => {
      _webImage = image; ← Сохраняем скачанное изображение.
      callback(_webImage); ← Обратный вызов используется в лямбда-функции, а не отправляется
    }));                     непосредственно в сценарий NetworkService.
  }
  else {
    callback(_webImage);
  }
}
...
```

Основные изменения претерпела функция, передаваемая в метод `NetworkService.DownloadImage()`. Раньше код передавался через один и тот же метод обратного вызова из сценария `WebLoadingBillboard`. Но после редактирования отправляемый в сценарий `NetworkService` обратный вызов превратился в отдельную лямбда-функцию,

объявленную в момент вызова метода из сценария `WebLoadingBillboard`. Обратите внимание на синтаксис объявления лямбда-функции: `() => {}`.

Реализация обратного вызова в виде отдельной функции не только дала возможность вызвать метод в сценарии `WebLoadingBillboard`; именно лямбда-функция сохраняет локальную копию скачанного изображения. В итоге методу `GetWebImage()` достаточно скачать изображение один раз; во всех последующих вызовах будет использоваться уже локальная копия.

Так как данная оптимизация применяется к последующим вызовам, ее эффект появится только при наличии нескольких рекламных щитов. Давайте сделаем копию щита, чтобы в сцене появилась хотя бы пара объектов. Выделите щит, выберите команду Duplicate (в меню Edit или в меню, вызываемом щелчком правой кнопки мыши) и переместите копию в сторону (например, измените координату *X* на 18).

Запустите игру и посмотрите, что получилось. В управлении первым щитом отмечается значительная пауза, связанная со скачиванием изображения из Интернета. Зато при подходе ко второму щиту изображение появляется сразу же, ведь оно уже загружено.

Это важная оптимизация процесса скачивания изображений (именно поэтому у браузеров режим кэширования графики включен по умолчанию). Теперь у нас осталась всего одна задача, с решением которой я хочу вас познакомить: отправка данных на сервер.

9.4. Отправка данных на веб-сервер

Мы рассмотрели несколько примеров скачивания данных, но пока ни разу не выполняли обратный процесс. Для выполнения заданий в этом разделе вам потребуется сервер, на который будут отправляться данные. Если возможность доступа к серверу у вас отсутствует, раздел можно просто пропустить. Кроме того, желающие могут легко скачать ПО с открытым исходным кодом и настроить тестовый сервер. Я рекомендую для этой цели сервер XAMPP. Для его скачивания посетите страницу https://www.apachefriends.org/ru/index.html. После установки у вас появился доступ к папке htdocs с адресом http://localhost/, как будто вы работаете с сервером в Интернете. Создайте в этой папке папку ch9 — именно в ней мы будем сохранять работающие на стороне сервера сценарии.

Вне зависимости от того, с чем вы работаете — с XAMPP или с собственным веб-сервером, задача будет одна: отправить на сервер информацию о погоде, как только игрок дойдет до контрольной точки. В качестве такой точки мы поместим в сцену зону триггера, напоминающую дверь из главы 8. Создайте куб, расположите его где-нибудь в сторонке, укажите, что его коллайдер используется как триггер, и назначьте ему полупрозрачный материал, как и в предыдущем случае (напоминаю, что вам нужно настроить режим визуализации материала). Итоговый вид этого объекта показан на рис. 9.7.

Теперь, когда в сцене появился триггер, напишем код, который он будет активировать.

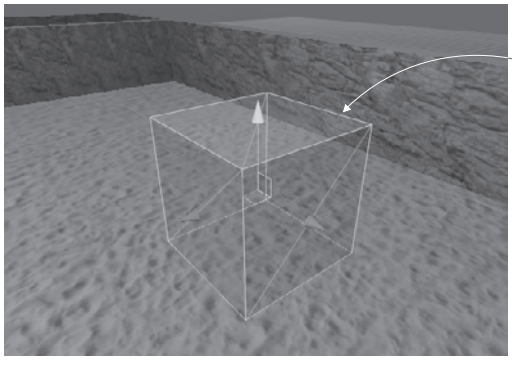


Рис. 9.7. Контрольная точка, вызывающая отправку данных на сервер

9.4.1. Слежение за погодой: отправка запросов POST

Код, активируемый в контрольной точке, будет последовательно включаться в нескольких сценариях. Как и в случае со скачиванием данных, при отправке данных сценарий `WeatherManager` заставляет сценарий `NetworkService` сделать запрос, причем последний полностью отвечает за передачу данных через сеть по протоколу HTTP. Следующий листинг демонстрирует коррективы, которые нужно внести в сценарий `NetworkService`.

Листинг 9.17. Сценарий NetworkService с возможностью отправки данных

```
...
private const string localApi = "http://localhost/ch9/api.php"; ← Обращение к сценарию на стороне сервера; при необходимости вы можете его поменять.
...
private IEnumerator CallAPI(string url, Hashtable args, Action<string>
callback) { ← Аргументы, добавленные к параметрам метода CallAPI().
  WWW www;
  if (args == null) {
    www = new WWW(url);
  } else {
    WWWForm form = new WWWForm(); ← Отправляем аргументы вместе с объектом WWW с помощью объекта WWWForm.
    foreach(DictionaryEntry arg in args) {
      form.AddField(arg.Key.ToString(), arg.Value.ToString());
    }
    www = new WWW(url, form); ← Объект WWWForm автоматически меняет запрос с GET на POST.
  }

  yield return www;
...
}

public IEnumerator GetWeatherXML(Action<string> callback) {
  return CallAPI(xmlApi, null, callback); ← Вызовы, измененные из-за измененных параметров.
}
public IEnumerator GetWeatherJSON(Action<string> callback) {
  return CallAPI(jsonApi, null, callback);
}
```

```
public IEnumerator LogWeather(string name, float cloudValue, Action<string>
callback) {
  Hashtable args = new Hashtable(); ← Определяем таблицу отправляемых аргументов.
  args.Add("message", name);
  args.Add("cloud_value", cloudValue);
  args.Add("timestamp", DateTime.UtcNow.Ticks); ← Отправляем метку времени вместе
                                                  с информацией об облачности.

  return CallAPI(localApi, args, callback);
}
...
```

Прежде всего обратите внимание на новый параметр метода `CallAPI()`. Это таблица аргументов, отправляемая вместе с HTTP-запросом. В соответствии с этой таблицей в методе `CallAPI()` может быть создан объект `WWWForm`. Обычно объект `WWW` отправляет запросы GET, а объект `WWWForm` меняет их на запросы POST. Все остальные изменения в коде вызваны этим основным нововведением (например, модификация кода метода `GetWhatever()` из-за параметров метода `CallAPI()`).

Следующий листинг демонстрирует код, который нужно добавить в сценарий `WeatherManager`.

Листинг 9.18. Добавление к сценарию WeatherManager кода отправки данных

```
...
public void LogWeather(string name) {
  StartCoroutine(_network.LogWeather(name, cloudValue, OnLogged));
}
private void OnLogged(string response) {
  Debug.Log(response);
}
...
```

И наконец, мы задействуем этот код, добавив к зоне триггера в сцене сценарий, управляющий контрольной точкой. Создайте сценарий `CheckpointTrigger`, свяжите его с зоной триггера и скопируйте в него код из следующего листинга.

Листинг 9.19. Сценарий CheckpointTrigger для зоны триггера

```
using UnityEngine;
using System.Collections;

public class CheckpointTrigger : MonoBehaviour {
  public string identifier;

  private bool _triggered; ← Проверяем, срабатывала ли уже контрольная точка.

  void OnTriggerEnter(Collider other) {
    if (_triggered) {return;}

    Managers.Weather.LogWeather(identifier); ← Активируем отправку данных.
    _triggered = true;
  }
}
```

На панели Inspector появится параметр Identifier; назовите его, например, checkpoint1. Запустите код, и при входе в зону триггера на сервер начнут отправляться данные. Но в качестве ответа вы получите сообщение об ошибке, так как на сервере пока отсутствует сценарий, получающий запрос. Именно его созданием и завершится данный раздел.

9.4.2. Серверный код в PHP-сценарии

Серверу требуется сценарий, который будет получать отправляемые игрой данные. Вопрос написания таких сценариев выходит за рамки темы данной книги, поэтому детально мы его рассматривать не будем. Мы просто напишем на скорую руку PHP-сценарий, так как это самый простой подход к решению задачи. Создайте в папке htdocs (или там, где располагается ваш веб-сервер) текстовый файл `api.php` и скопируйте в него код из следующего листинга.

Листинг 9.20. Серверный сценарий на языке PHP, получающий наши данные

```
<?php

  $message = $_POST[‘message’]; ← Извлекаем отправленные данные в переменные.
  $cloudiness = $_POST['cloud_value'];
  $timestamp = $_POST['timestamp'];
  $combined = $message." cloudiness=".$cloudiness." time=".$timestamp."\n";

  $filename = "data.txt"; ← Определяем имя файла, в который будет выполняться запись.
  file_put_contents($filename, $combined, FILE_APPEND | LOCK_EX); ← Записываем файл.

  echo "Logged";

?>
```

Обратите внимание, что сценарий записывает полученные данные в файл `data.txt`, соответственно, на сервере нужно создать файл с таким именем. Как только сценарий `api.php` будет готов, вы увидите, что в файле `data.txt` при достижении контрольной точки в игре появляются данные о погоде. Великолепно!

9.5. Заключение

- Скайбокс предназначен для отображения неба, которое визуализируется позади всех остальных объектов сцены.
- В Unity есть объект `WWW`, предназначенный для скачивания данных.
- Распространенные форматы данных, такие как XML и JSON, легко доступны для парсинга.
- Материалы могут отображать фотографии, скачанные из Интернета.
- Объект `WWW` позволяет также отправлять данные на веб-сервер.

10 Звуковые эффекты и музыка

- ✓ Импорт и воспроизведение аудиоклипов с созданием различных звуковых эффектов
- ✓ Использование двухмерных звуков для UI и трехмерных звуков в сцене
- ✓ Регулирование уровня громкости всех звуков в процессе воспроизведения
- ✓ Воспроизведение фоновой музыки в процессе игры
- ✓ Плавный переход от одного музыкального трека к другому

Когда речь заходит о видеоиграх, основное внимание уделяется графике, между тем как звуковое сопровождение также является важным аспектом. В большинстве игр играет фоновая музыка и присутствуют звуковые эффекты. Соответственно, Unity предлагает необходимую функциональность. Вы можете импортировать в Unity и воспроизводить аудиофайлы различных форматов, регулировать громкость звука и даже обрабатывать звуки, исходящие из определенной точки сцены.

Эту главу мы начнем с рассмотрения звуковых эффектов. Они представляют собой короткие аудиоклипы, воспроизводимые во время определенных действий (например, звук выстрела, который сопровождает стрельбу игрока по врагам). Звуковые клипы с музыкой более продолжительные (часто речь идет о минутах), а их воспроизведение не привязано к конкретным событиям в игре. В конечном счете все сводится к одному виду аудиофайлов и одинаковому коду их воспроизведения, но тот факт, что файлы с музыкой обычно намного продолжительнее клипов со звуковыми эффектами (более того, часто оказывается, что это самые большие файлы в игре!), заслуживает, чтобы их выделили в отдельный раздел.

В этой главе мы планируем взять игру без звукового сопровождения и сделать следующее:

1. Импортировать аудиофайлы для звуковых эффектов.
2. Добавить звуковые эффекты к врагам и процессу стрельбы по ним.

3. Запрограммировать диспетчер управления звуком, чтобы контролировать громкость звука.
4. Оптимизировать загрузку музыки.
5. Сделать отдельную регулировку громкости для музыки и звуковых эффектов, в том числе для переходящих один в другой музыкальных треков.

ПРИМЕЧАНИЕ По большому счету, эта глава не привязана к конкретному проекту; мы просто добавляем звук к существующему демонстрационному ролику. Все упражнения демонстрируются на примере шутера от первого лица, созданного в главе 3. Вы можете скачать соответствующий фрагмент проекта или воспользоваться любым другим демонстрационным роликом по своему вкусу.

Надеюсь, у вас уже открыт демонстрационный ролик и можно сделать первый шаг: импортировать звуковые эффекты.

10.1. Импорт звуковых эффектов

Чтобы у вас появилась возможность воспроизведения звуков, нужно импортировать аудиофайлы в Unity-проект. Процедура начинается с подбора файлов нужного формата, которые затем переносятся в Unity и настраиваются под ваши цели.

10.1.1. Поддерживаемые форматы файлов

При рассмотрении графических ресурсов в главе 4 вы видели, что Unity поддерживает различные форматы, каждый со своими достоинствами и недостатками. В табл. 10.1 представлены поддерживаемые форматы аудиофайлов.

Таблица 10.1. Форматы аудиофайлов, поддерживаемые в Unity

Тип файла	Достоинства и недостатки
WAV	Формат, предлагаемый в Windows по умолчанию. Несжатый звуковой файл
AIF	Формат, предлагаемый в Mac по умолчанию. Несжатый звуковой файл
MP3	Сжатый звуковой файл; до некоторой степени жертвуется качество ради уменьшения размера
OGG	Сжатый звуковой файл; до некоторой степени жертвуется качество ради уменьшения размера
MOD	Формат для музыкальных трекеров. Специализированная разновидность эффективной цифровой музыки
XM	Формат для музыкальных трекеров. Специализированная разновидность эффективной цифровой музыки

Основным фактором, отличающим форматы друг от друга, является сжатие. Оно позволяет уменьшить размер файла за счет удаления из него некоторой части информации. Сжатие реализовано таким образом, что удалению подвергается наименее важная информация, поэтому качество звука остается вполне приемлемым. Несмотря на то что потеря качества незначительна, для коротких звуковых клипов нужно

выбирать форматы без сжатия. Для более длинных звуковых клипов (особенно с музыкой) желателен формат со сжатием, в противном случае размер файла становится недопустимо большим.

Впрочем, при работе в Unity на выбор влияет еще один фактор...

СОВЕТ В Unity есть возможность сжимать аудиофайлы после импорта. Поэтому в процессе разработки игры имеет смысл пользоваться форматами без сжатия даже для длинных музыкальных клипов, а не импортировать сжатые аудиофайлы.

КАК РАБОТАЕТ ЦИФРОВОЙ ЗВУК

В общем случае аудиофайлы сохраняют форму волны, которая создается в наушниках при прослушивании музыки. Звук представляет собой набор распространяющихся в воздухе волн, соответственно, различные звуки состоят из звуковых волн различных размеров и частот. В аудиофайлах эти волны записываются с созданием множества их замеров через короткие интервалы времени и сохранением состояния волны в каждом замере.

Чем чаще записываются замеры волн, тем более точной получается запись изменений волны во времени — тем меньше интервалы между изменениями. Но это означает большее количество сохраняемых данных и, как следствие, больший размер файла. При сжатии размер файла уменьшается за счет различных приемов, в число которых входит, к примеру, удаление данных на звуковых частотах, неразличимых человеческим ухом.

Музыкальные трекеры представляют собой особый тип программных музыкальных секвенсоров для создания музыки. Если в традиционных аудиофайлах звук хранится в виде обычных волн, секвенсеры сохраняют нечто, больше напоминающее ноты: файл-трекер содержит последовательность нот с такой информацией, как интенсивность и частота каждой ноты. Эти «ноты» состоят из небольших волн, но общее количество сохраненных данных невелико, так как каждая нота используется в последовательности много раз. Написанная таким способом музыка при всей своей эффективности представляет собой крайне специализированные аудиофайлы.

Благодаря возможности сжатия после импорта всегда выбирайте файлы в формате WAV или AIF. Настройки импорта коротких звуковых эффектов и музыки, естественно, будут различаться (особенно в части сжатия), но исходный файл всегда должен быть без сжатия.

Существуют различные способы создания музыкальных файлов (например, в приложении Б упоминается инструмент Audacity, позволяющий записывать звук при помощи микрофона), но для нашего проекта мы скачаем образцы звуков с одного из многочисленных бесплатных сайтов. В качестве источника ресурсов я предлагаю сайт www.freesound.org. Нам нужны клипы в формате WAV.

ВНИМАНИЕ «Бесплатные» аудиофайлы предлагаются под различными вариантами лицензий, поэтому всегда проверяйте, разрешено ли задействовать конкретный клип нужным вам способом. К примеру, многие бесплатные образцы музыки предназначены только для некоммерческого использования.

В моем примере проекта фигурируют следующие звуковые эффекты общего пользования (разумеется, вам ничто не мешает скачать собственные варианты клипов; не забудьте проверять отсутствие лицензионных ограничений):

- thump от пользователя hy96;
- ding от пользователя Daphne_in_Wonderland;

- swish bamboo pole от пользователя ra_gun;
- fireplace от пользователя leosalom.

Следующим шагом после скачивания файлов станет их импорт в Unity.

10.1.2. Импорт аудиофайлов

Собранную вами коллекцию аудиофайлов нужно импортировать в Unity. В главе 4 вы узнали, что любые ресурсы нужно вставить в проект, и только после этого их можно будет использовать в игре.

Простой механизм импорта работает со всеми видами ресурсов: вы перетаскиваете файлы из папки на компьютере на вкладку Project в Unity (создайте папку с именем Sound FX для перетаскивания в нее файлов). Как видите, все очень просто! Но как и в случае с остальными ресурсами, у аудиофайлов есть настройки импорта, которые задаются на панели Inspector (рис. 10.1).

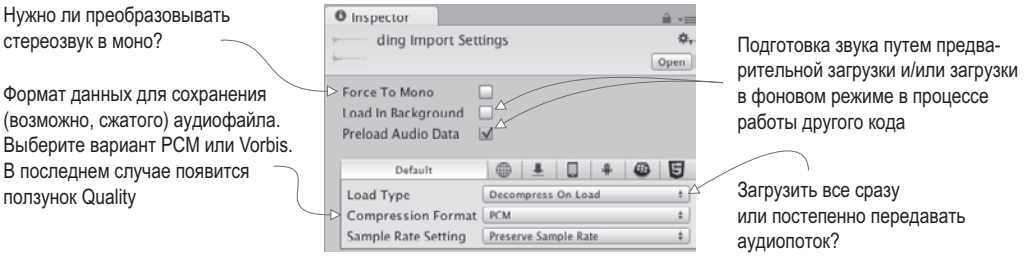


Рис. 10.1. Настройки импорта аудиофайлов

Флажок Force To Mono устанавливать не нужно. Он позволяет выбрать между моно- и стереозвуком; зачастую звук записывается в стерео, то есть в файле присутствуют одновременно две волны, одна для левого, другая для правого наушника или колонки. Чтобы уменьшить размер файла, можно убрать половину звуковой информации. После этого на обе колонки будет посылаться одна волна вместо двух.

Следом идут флажки Load In Background (загрузка в фоновом режиме) и Preload Audio Data (предварительная загрузка аудиоданных). Настройки предварительной загрузки относятся к балансировке производительности воспроизведения и использования памяти; заранее загруженный аудиофайл занимает память, ожидая своей очереди, зато в процессе его воспроизведения не приходится тратить время на загрузку. Загрузка звука в фоновом режиме позволяет программе продолжить свою работу; этот вариант в общем случае подходит для длинных музыкальных клипов, но исключает возможность немедленного воспроизведения звука. Для коротких звуковых клипов флажок Load In Background обычно не устанавливается, что гарантирует их полную загрузку перед воспроизведением. Так как сейчас мы импортируем короткие звуковые эффекты, устанавливать этот флажок не нужно.

Наконец, наиболее важными параметрами являются Load Type (загружаемый тип) и Compression Format (формат сжатия). Последний отвечает за формат сохраненных аудиоданных. Как упоминалось в предыдущем разделе, музыка должна быть сжатой,

поэтому для нее выбирайте вариант Vorbis (это название аудиоформата со сжатием). Короткие звуковые клипы в сжатии не нуждаются, поэтому выбирайте для них вариант PCM (Pulse Code Modulation — импульсно-кодовая модуляция). Третий вариант — ADPCM — является вариацией PCM и дает несколько более качественный звук.

Раскрывающийся список Load Type позволяет указать, каким образом компьютер будет загружать данные из файла. Так как объем памяти у компьютера ограничен, а аудиофайлы могут быть очень большими, иногда имеет смысл запустить воспроизведение звука в процессе его передачи в память, избавляя компьютер от необходимости единовременно загружать весь файл. Однако такая передача звука требует затрат вычислительных ресурсов, поэтому быстрее всего воспроизводятся файлы, предварительно загруженные в память. Но даже в этом случае вы можете выбрать, будут ли данные представлены в сжатой форме или их следует распаковать для более быстрого воспроизведения. Так как наши аудиоклипы имеют небольшой размер, воспроизведение в процессе загрузки нам не требуется, поэтому можно выбрать вариант Decompress On Load.

Итак, в нашем проекте появились импортированные и готовые к использованию аудиофайлы.

10.2. Воспроизведение звуковых эффектов

Теперь, когда вы добавили к проекту звуковые файлы, вы, разумеется, хотите, чтобы они звучали. Код активации звуковых эффектов понять несложно, но аудиосистема в Unity состоит из разных частей, которые должны работать согласованно.

10.2.1. Система воспроизведения: клипы, источник, подписчик

Возможно, вы ожидаете, что для проигрывания звука в Unity достаточно указать, какие клипы нужно воспроизводить, но на самом деле для этого нужно задать три компонента: AudioClip, AudioSource и AudioListener. Подобное разделение связано с поддержкой в Unity трехмерного звука: различные компоненты сообщают Unity информацию о местоположении, которая используется для управления трехмерным звуком.

2D- И 3D-ЗВУК

Звук в играх делится на двухмерный и трехмерный. С первым вы уже знакомы, это стандартный звук, воспроизводимый обычным образом. Выражение «2D-звук» в большинстве случаев означает «не 3D-звук».

3D-звук характерен для трехмерного моделирования, и, возможно, вы с ним еще не сталкивались — это звук, исходящий из конкретных мест сцены. На его громкость и тон влияет положение слушателя. Например, звуковой эффект, включенный с большого расстояния, будет слышен крайне слабо.

В Unity поддерживаются оба вида звуков, и вы сами выбираете вид звука для каждого источника. Звук для таких вещей, как музыка, должен быть двухмерным, в то время как трехмерный звук в большинстве звуковых эффектов создает ощущение присутствия в сцене.

В качестве аналогии представьте магнитолу, проигрывающую компакт-диск в комнате. Зашедший в комнату человек четко услышит звук. Уходя из комнаты, он начнет

слышать его все тише, и в конце концов звук вообще исчезнет. При перемещении магнитолы по комнате вы будете слышать, как меняется громкость звука. Эту аналогию иллюстрирует рис. 10.2. Компакт-диск — это аналог компонента AudioClip, магнитола — компонента AudioSource, а человек — компонента AudioListener.

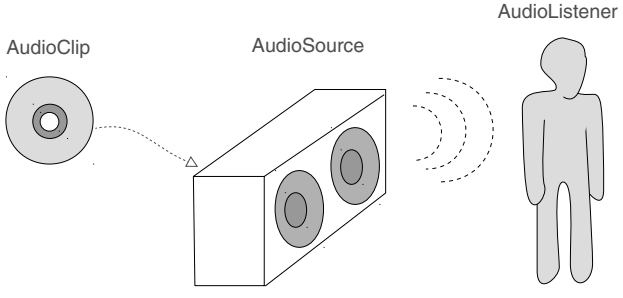


Рис. 10.2. Схема компонентов, управляющих аудиосистемой в Unity

Первым компонентом является аудиоклип. Он связан с реальным звуковым файлом, который мы импортировали в предыдущем разделе. Эти необработанные данные о форме сигнала являются основой всего, что делает аудиосистема, но сам по себе аудиоклип не выполняет никаких действий.

Следующий вид объекта — это источник звука. Именно он воспроизводит аудиоклипы. Это абстракция того, что на самом деле делает аудиосистема, но именно она делает более понятной концепцию трехмерного звука. 3D-звук, воспроизводимый конкретным источником, расположен в той же самой точке, что и этот источник; и хотя 2D-звуки также воспроизводятся источником, их местоположение не имеет значения.

Третьим видом объекта в составе аудиосистемы в Unity является слушатель звука. Как понятно из его названия, это объект, который слышит звуки, проецируемые из источников. Это еще одна абстракция реальных операций аудиосистемы (ведь очевидно, что реальным слушателем является играющий в игру человек!), но аналогично тому, как положение источника звука определяет координаты места, из которого исходит звук, положение слушателя звука определяет координаты места, в котором слышен звук.

УСОВЕРШЕНСТВОВАННАЯ СИСТЕМА УПРАВЛЕНИЯ ЗВУКОМ

В Unity 5 появилась функция Audio Mixers (аудиомикшеры). Аудиомикшеры позволяют обрабатывать аудиосигналы и накладывать на клипы различные эффекты. Более подробно с этой функцией можно познакомиться в документации к Unity: http://docs.unity3d.com/ru/current/Manual/AudioMixer.html.

Такие компоненты, как AudioClip и AudioSource, нужно назначать, а вот компонентом AudioListener оснащена камера, которая по умолчанию появляется в каждой новой сцене. Как правило, вам нужно, чтобы 3D-звуки реагировали на положение наблюдателя.

10.2.2. Присваивание зацикленного звука

Итак, давайте настроим наш первый звук в Unity! Аудиоклипы уже импортированы, у используемой по умолчанию камеры есть компонент AudioListener, так что нам остается назначить только компонент AudioSource. Добавим треск огня к шаблону Enemy — это персонаж, который хаотично перемещается по сцене.

ПРИМЕЧАНИЕ Так как враг будет звучать так, как будто он охвачен пламенем, можно связать с ним систему частиц, чтобы придать ему соответствующий звуку вид. Можно скопировать систему частиц, созданную в главе 4, превратив объект Particle в шаблон экземпляра, а затем выбрав в меню Asset команду Export Package. В качестве альтернативы можно повторить процедуру из главы 4, создав новый объект с нуля (перетащите шаблон Enemy в сцену для редактирования, а затем выберите в меню GameObject команду Apply Changes To Prefab).

Обычно для редактирования шаблона экземпляра его нужно перетащить в сцену, но в данном случае эту процедуру можно выполнить напрямую сразу же после добавления компонента к объекту. Выделите шаблон Enemy, чтобы его свойства появились на панели Inspector. Затем добавьте новый компонент, выбрав вариант Audio ▸ Audio Source. На панели Inspector появится компонент AudioSource.

Укажите источнику аудиоклип, который нужно воспроизвести. Перетащите файл со звуком со вкладки Project на ячейку Audio Clip панели Inspector; для этого примера мы используем звуковой эффект fireplace (рис. 10.3).

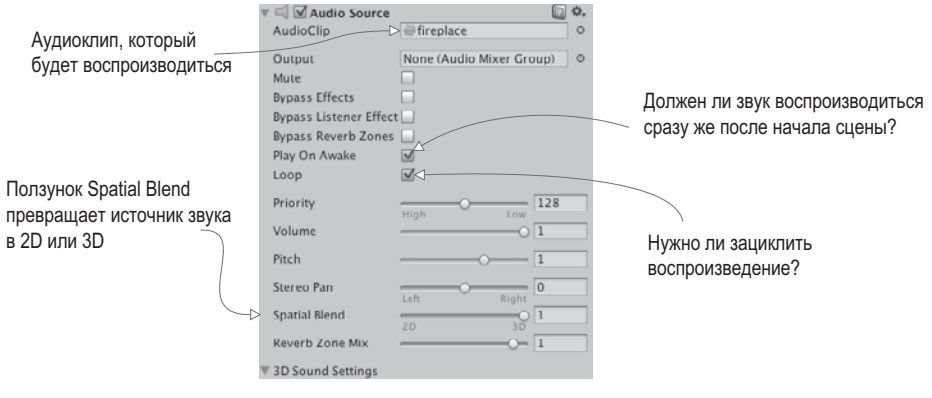


Рис. 10.3. Настройки компонента AudioSource

Установите флажки Play On Awake и Loop (обязательно убедитесь, что флажок Mute сброшен). Флажок Play On Awake заставляет источник звука начать воспроизведение сразу же после загрузки сцены (в следующем разделе вы узнаете, как активировать звуки вручную во время игры). Флажок Loop заставляет источник звука играть непрерывно, повторяя клип снова и снова.

Мы хотим, чтобы этот источник аудио испускал 3D-звуки. Как уже объяснялось, 3D-звук имеет в сцене определенное положение. Это свойство источника звука регулируется ползунком Spatial Blend. Именно он отвечает за превращение звука из двухмерного в трехмерный; установите его в крайнее правое положение.

Теперь проверьте, включен ли звук у вас в колонках, и запустите игру. Вы услышите исходящее от врага потрескивание, которое ослабевает по мере его удаления от персонажа, так как в сцене используется источник 3D-звука.

10.2.3. Активация звуковых эффектов из кода

Настройка компонента AudioSource на автоматическое воспроизведение хорошо подходит для циклических звуков, но в большинстве случаев нужно, чтобы звуковые эффекты возникали в ответ на команды кода. Для этого нам все равно потребуется компонент AudioSource, но теперь звук будет воспроизводиться не непрерывно, а по команде.

Добавьте компонент AudioSource к объекту player (а не к камере). Связывать с компонентом конкретный аудиоклип на этот раз нет нужды, так как он определяется в коде. Сбросьте флажок Play On Awake, ведь звук будет возникать в ответ на команду. Ползунок Spatial Blend установите в положение 3D, так как эффект привязан к положению в сцене.

Затем добавьте код из следующего листинга в сценарий `RayShooter`, отвечающий за стрельбу.

Листинг 10.1. Звуковые эффекты, добавленные в сценарий RayShooter

```
...
[SerializeField] private AudioSource soundSource;
[SerializeField] private AudioClip hitWallSound;
[SerializeField] private AudioClip hitEnemySound;
...
if (target != null) {
  target.ReactToHit();
  soundSource.PlayOneShot(hitEnemySound);
} else {
  StartCoroutine(SphereIndicator(hit.point));
  soundSource.PlayOneShot(hitWallSound);
}
...
```

Ссылки на два звуковых файла, которые нужно воспроизвести.

Если переменная target не равна null, значит, игрок выстрелил во врага, поэтому...

... вызываем метод PlayOneShot() для воспроизведения звука Hit An Enemy или...

...метод PlayOneShot() для воспроизведения звука Hit A Wall, если игрок промазал.

В новом варианте кода в верхней части сценария появилось несколько сериализованных переменных. Выделите камеру и перетащите объект player (обладающий компонентом AudioSource) на ячейку soundSource панели Inspector. Затем перетащите на ячейки Hit Wall Sound и Hit Enemy Sound аудиоклипы, которые будут воспроизводиться в каждом случае (swish при попадании в стену и ding при поражении врага).

Остальные две добавленные строки представляют собой методы `PlayOneShot()`. Именно они заставляют источник звука воспроизвести указанный клип. Методы добавляются в условную инструкцию для переменной `target` и воспроизводят разные звуки при попадании в разные цели.

ПРИМЕЧАНИЕ Можно было задать клип в компоненте AudioSource и вызвать для его воспроизведения метод Play(). Но несколько звуков будут обрывать друг друга, поэтому мы воспользовались методом PlayOneShot(). Чтобы понять суть проблемы, замените метод PlayOneShot() вот таким кодом и сделайте несколько быстрых выстрелов:

```
soundSource.clip=hitEnemySound; soundSource.Play();
```

Запустите игру и сделайте несколько выстрелов. Теперь в нашей игре появилось несколько звуковых эффектов. Аналогичным образом добавляются все остальные виды звуковых эффектов. Но для полноценной звуковой системы в игре требуется несколько больше, чем набор несвязанных звуков; как минимум, игрок должен иметь возможность контролировать громкость.

Этот элемент управления мы реализуем в следующем разделе с помощью специального модуля.

10.3. Интерфейс управления звуком

Продолжая доработку архитектуры, созданной в предыдущих главах, добавим туда диспетчер `AudioManager`. Напомню вам, что объект `Managers` обладает списком различных модулей кода, используемых в игре, таких как, к примеру, диспетчер игрового инвентаря. На этот раз будет создан диспетчер управления звуком, который мы и добавим в список. Этот центральный модуль позволит регулировать громкость звука в игре и даже выключать звук совсем. Изначально он будет работать только со звуковыми эффектами, но в следующих разделах мы добавим объекту `AudioManager` возможность регулировать громкость музыки.

10.3.1. Настройка центрального диспетчера управления звуком

Для настройки диспетчера `AudioManager` первым делом нужно восстановить фреймворк `Managers`. Из проекта главы 9 скопируйте сценарии `IGameManager`, `ManagerStatus` и `NetworkService`; их мы редактировать не будем. Напоминаю, что `IGameManager` — это интерфейс, который должны реализовывать все диспетчеры, а `ManagerStatus` — перечисление, которым пользуется `IGameManager`. Сценарий `NetworkService`, отвечающий за подключение к Интернету, в этой главе нам не потребуется.

ПРИМЕЧАНИЕ Скорее всего, Unity выведет на экран предупреждение, ведь сценарий NetworkService назначен, но не задействован. Просто проигнорируйте его; мы хотим предоставить фреймворку возможность подключения к Интернету, просто в этой главе данная функциональность не требуется.

Также скопируйте файл `Managers`, который мы отредактируем с учетом нашего нового диспетчера управления звуком. Пока оставьте его без изменений (или, если вас раздражают появляющиеся сообщения об ошибках компиляции, просто закомментируйте строки, приводящие к появлению ошибки!). Создайте новый сценарий `AudioManager`, на который может ссылаться код сценария `Managers`. Код нового сценария представлен в следующем листинге.

Листинг 10.2. Скелетный код сценария AudioManager

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

public class AudioManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}
```

```
  private NetworkService _network;

  // Сюда добавляются элементы управления громкостью (см. листинг 10.4)

  public void Startup(NetworkService service) {
    Debug.Log("Audio manager starting...");

    _network = service;

   // Здесь инициализируются источники музыки (см. листинг 10.10) ←
                                                                    Здесь будут выполняться
                                                                    все длительные задания,
                                                                    запускаемые при старте.
    status = ManagerStatus.Started; ← При наличии длительных заданий, запускаемых
                                      при старте, присваиваем состоянию значение
                                      Initializing.
  }
}
```

Этот сценарий напоминает диспетчеры из предыдущих глав — это минимум кода, необходимый интерфейсу `IGameManager` для реализации классом. Теперь можно добавить новый диспетчер в сценарий `Managers`.

Листинг 10.3. Сценарий Managers после добавления диспетчера AudioManager

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

[RequireComponent(typeof(AudioManager))]

public class Managers : MonoBehaviour {
  public static AudioManager Audio {get; private set;}

  private List<IGameManager> _startSequence;

  void Awake() {
    Audio = GetComponent<AudioManager>(); ← В этом проекте в списке AudioManager,
                                            а не PlayerManager, и т. п.

    _startSequence = new List<IGameManager>();
    _startSequence.Add(Audio);

    StartCoroutine(StartupManagers());
  }

  private IEnumerator StartupManagers() {
    NetworkService network = new NetworkService();

    foreach (IGameManager manager in _startSequence) {
      manager.Startup(network);
    }

    yield return null;

    int numModules = _startSequence.Count;
    int numReady = 0;

    while (numReady < numModules) {
```

```
      int lastReady = numReady;
      numReady = 0;

      foreach (IGameManager manager in _startSequence) {
        if (manager.status == ManagerStatus.Started) {
          numReady++;
        }
      }

      if (numReady > lastReady)
      Debug.Log("Progress: " + numReady + "/" + numModules);

      yield return null;
    }

    Debug.Log("All managers started up");
  }
}
```

Как и в предыдущих главах, создайте пустой объект, который будет играть роль диспетчера, и свяжите с ним сценарии `Managers` и `AudioManager`. При воспроизведении игры вы увидите на консоли сообщения о запуске диспетчеров, но пока наш диспетчер управления звуком не несет никакой функциональной нагрузки.

10.3.2. UI для управления громкостью

У нас уже есть скелет сценария `AudioManager`, пришло время добавить туда средства контроля громкости. Отвечающие за эту функциональность методы затем будут использоваться визуальными UI-элементами, позволяя выключать звуковые эффекты и регулировать громкость.

Мы воспользуемся новыми инструментами создания пользовательского интерфейса, с которыми вы уже встречались в главе 6. Будет создано показанное на рис. 10.4 всплывающее окно с кнопкой и ползунком, отвечающим за громкость звука.

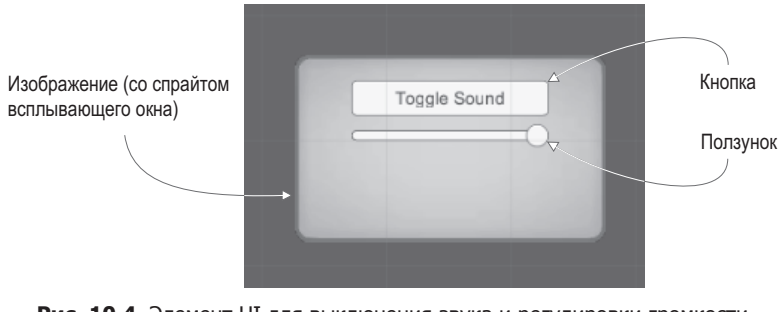


Рис. 10.4. Элемент UI для выключения звука и регулировки громкости

Я перечислю этапы построения такого окна, не вдаваясь в детали; их вы можете вспомнить самостоятельно, обратившись к главе 6.

1. Импортируем изображение popup.png как спрайт (параметру Texture Type присваиваем значение Sprite).

2. В окне диалога Sprite Editor формируем со всех сторон границу размером 12 пикселов (не забудьте применить сделанные изменения).
3. Создаем в сцене холст (GameObject ▸ UI ▸ Canvas).
4. Устанавливаем для холста флажок Pixel Perfect.
5. (По желанию.) Присваиваем объекту имя HUD Canvas и переключаемся в режим работы с двухмерной графикой.
6. Создаем связанное с холстом изображение (GameObject ▸ UI ▸ Image).
7. Присваиваем новому объекту имя Settings Popup.
8. Перетаскиваем на ячейку Source Image этого объекта спрайт popup.
9. В раскрывающемся списке Image Type выбираем вариант Sliced и устанавливаем флажок Fill Center.
10. Располагаем изображение всплывающего окна в точке с координатами 0, 0.
11. Меняем размеры всплывающего окна до 250 по ширине и 150 по высоте.
12. Создаем кнопку (GameObject ▸ UI ▸ Button).
13. Делаем кнопку дочерним объектом по отношению к всплывающему окну (путем перетаскивания на вкладке Hierarchy).
14. Помещаем кнопку в точку с координатами 0, 40.
15. Раскрываем иерархический список кнопки, чтобы выделить связанную с ней текстовую метку.
16. Меняем текст на Toggle Sound.
17. Создаем ползунок (GameObject ▸ UI ▸ Slider).
18. Делаем ползунок дочерним объектом всплывающего окна и помещаем его в точку с координатами 0, 15.

Теперь, когда у нас есть всплывающее окно, напишем для него код. Нам понадобится сценарий как для самого окна, так и для функции управления громкостью, которую этот сценарий будет вызывать. Для начала отредактируйте код сценария `AudioManager` в соответствии со следующим листингом.

Листинг 10.4. Добавление в сценарий AudioManager средств регулировки громкости звука

```
...
public float soundVolume { ← Свойство с функцией чтения и функцией доступа для громкости.
  get {return AudioListener.volume;}  | Реализуем функцию чтения/функцию доступа
  set {AudioListener.volume = value;} | с помощью AudioListener.
}

public bool soundMute { ← Добавляем аналогичное свойство для выключения.
  get {return AudioListener.pause;}
  set {AudioListener.pause = value;}
}

public void Startup(NetworkService service) { ← Выделенный курсивом код уже был в сценарии,
  Debug.Log("Audio manager starting..."); тут он показан для справки.
```

```
    _network = service;

    soundVolume = 1f; ← Инициализация значения (в диапазоне от 0 до 1; 1 соответствует полной громкости).

    status = ManagerStatus.Started;
  }
  ...
```

В сценарий `AudioManager` добавлены свойства `soundVolume` и `soundMute`. Функция чтения и задающая функция для этих свойств реализованы с помощью глобальных переменных класса `AudioListener`. Класс `AudioListener` может регулировать громкость всех звуков, получаемых всеми экземплярами `AudioListener`. Задание свойства `soundVolume` в сценарии `AudioManager` оказывает такой же эффект, как задание громкости в компоненте AudioListener. Главным преимуществом такого подхода является инкапсуляция: все, что имеет отношение к звуку, обрабатывается одним диспетчером, и внешнему коду даже не нужно знать детали реализации. После добавления этих методов в сценарий `AudioManager` можно написать сценарий для всплывающего окна. Создайте сценарий с именем `SettingsPopup` и добавьте туда содержимое следующего листинга.

Листинг 10.5. Сценарий SettingsPopup с элементами управления громкостью

```
using UnityEngine;
using System.Collections;

public class SettingsPopup : MonoBehaviour {

  public void OnSoundToggle() { ← Кнопка переключает свойство mute диспетчера управления звуком.
    Managers.Audio.soundMute = !Managers.Audio.soundMute;
  }

  public void OnSoundValue(float volume) { ← Ползунок регулирует свойство volume
    Managers.Audio.soundVolume = volume;      диспетчера управления звуком.
  }
}
```

В этом сценарии мы видим два метода, влияющие на свойства объекта `AudioManager`: метод `OnSoundToggle()` задает свойство `soundMute`, а метод `OnSoundValue()` — свойство `soundVolume`. Как обычно, перетащите сценарий `SettingsPopup` на объект Settings Popup в пользовательском интерфейсе.

Затем для получения возможности вызывать эти функции с помощью кнопки и ползунка свяжите всплывающее окно с событиями взаимодействия этих элементов управления. На панели Inspector для кнопки найдите поле OnClick. Щелкните на кнопке со знаком + (плюс), чтобы добавить к этому событию новый элемент. Перетащите объект Settings Popup на ячейку для объекта в новом элементе и найдите в меню вариант SettingsPopup; чтобы кнопка начала вызывать данную функцию, выберите вариант OnSoundToggle().

Этот способ связи с функцией применим и к ползунку. Первым делом найдите событие взаимодействия в настройках ползунка — в данном случае оно будет называться OnValueChanged. Щелкните на кнопке со знаком + (плюс) для добавления

нового элемента и перетащите на ячейку для объекта объект Settings Popup. В меню функций найдите сценарий SettingsPopup и выберите в разделе Dynamic Float вариант OnSoundVolume().

ВНИМАНИЕ Помните, что нам нужна функция из раздела Dynamic Float, а не Static Parameter! Указанный метод присутствует в обоих разделах, но в последнем случае он сможет получить только одно заранее введенное значение.

Теперь элементы управления работают, но в проект нужно внести небольшие коррективы. Дело в том, что сейчас всплывающее окно все время закрывает экран. Давайте сделаем так, чтобы оно открывалось только при нажатии клавиши M. Создайте новый сценарий `UIController`, свяжите его с контроллером в сцене и введите код следующего листинга.

Листинг 10.6. Сценарий UIController, вызывающий и скрывающий всплывающее окно

```
using UnityEngine;
using System.Collections;

public class UIController : MonoBehaviour {
  [SerializeField] private SettingsPopup popup; ← Ссылки на всплывающее окно в сцене.

  void Start() {
    popup.gameObject.SetActive(false); ← Инициализируем всплывающее окно в скрытом состоянии.
  }

  void Update() {
    if (Input.GetKeyDown(KeyCode.M)) { ← Вызываем и скрываем всплывающее окно при помощи клавиши M.
      bool isShowing = popup.gameObject.activeSelf;
      popup.gameObject.SetActive(!isShowing);

      if (isShowing) {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
      } else {
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
      }
    }
  }
}
```

Вместе со всплывающим окном вызываем курсор.

Для подключения этой ссылки на объект перетащите всплывающее окно настроек на ячейку сценария. Теперь запустите игру и попытайтесь подвигать ползунок (напоминаю, что UI активируется нажатием клавиши M), стреляя по сторонам, чтобы слышать звуковые эффекты; вы убедитесь, что их громкость меняется в соответствии с положением ползунка.

10.3.3. Воспроизведение звуков UI

Внесем еще одно дополнение в сценарий `AudioManager`, добавив к щелчкам на кнопках UI звуковое сопровождение. Эта задача сложнее, чем кажется на первый взгляд.

Ведь Unity требуется компонент AudioSource. В случае звуков, испускаемых объектами сцены, место его присоединения очевидно. Но звуковые эффекты UI не являются частью сцены, поэтому вам нужно настроить специальный компонент `AudioSource` для диспетчера `AudioManager`. Он будет использоваться, если в сцене отсутствуют другие источники звука.

Создайте новый пустой объект GameObject и сделайте его дочерним по отношению к основному игровому диспетчеру. Так как у этого нового объекта будет компонент AudioSource, используемый диспетчером управления звуком, присвойте ему имя Audio. Добавьте к нему компонент AudioSource (на этот раз оставьте ползунок Spatial Blend в положении 2D, так как пользовательский интерфейс не имеет определенного положения в сцене) и добавьте в сценарий `AudioManager` код следующего листинга.

Листинг 10.7. Воспроизведение звуковых эффектов в диспетчере AudioManager

```
...
[SerializeField] private AudioSource soundSource; ← Ячейка переменной на панели Inspector для ссылки на новый источник звука.
...
public void PlaySound(AudioClip clip) { ← Воспроизводим звуки, не имеющие другого источника.
  soundSource.PlayOneShot(clip);
}
...
```

На панели Inspector появится новая ячейка переменной; перетащите на нее объект Audio. Теперь добавим звуковой эффект UI к сценарию всплывающего окна.

Листинг 10.8. Добавление звуковых эффектов в сценарий SettingsPopup

```
...
[SerializeField] private AudioClip sound; ← Ячейка на панели Inspector для ссылки на звуковой клип.
...
public void OnSoundToggle() {
  Managers.Audio.soundMute = !Managers.Audio.soundMute;
  Managers.Audio.PlaySound(sound); ← Воспроизводим звуковой эффект при нажатии кнопки.
}
...
```

Перетащите звуковой эффект UI на ячейку переменной; я использовал 2D-звук thump. В результате щелчок на кнопке UI сопровождается этим звуком (если, конечно, вы не уменьшили его громкость до нуля!). И хотя сам UI не имеет источника звука, диспетчер `AudioManager` воспроизводит нужный звуковой эффект через свой источник.

Замечательно, мы настроили все звуковые эффекты! Пришло время обратить внимание на музыку.

10.4. Фоновая музыка

Теперь нам нужно вставить в игру фоновую музыку, а для этого ее нужно добавить в диспетчер `AudioManager`. Как объяснялось во введении к данной главе, музыкальные клипы, по сути, не отличаются от звуковых эффектов. Цифровой звук функционирует посредством звуковых волн тем же самым способом, команды его воспроизведения тоже по большей части одни и те же. Основное отличие состоит в размере клипа, но именно из этого вытекает ряд последствий.

Во-первых, для музыкальных треков, как правило, требуется много компьютерной памяти, и этот расход необходимо оптимизировать. Нужно отслеживать два аспекта: загрузка музыки в память до того, как она начнет использоваться, и потребление слишком большого объема памяти при загрузке.

Оптимизация *в процессе* загрузки выполняется методом `Resources.Load()`, с которым вы познакомились в главе 8. Как вы узнали, этот метод позволяет загружать ресурсы по имени — это удобный инструмент, но есть и другие причины загружать ресурсы из папки Resources. Одним из ключевых факторов является отложенная загрузка; обычно все ресурсы в Unity загружаются вместе со сценой, но это не касается ресурсов из папки Resources, ожидающих извлечения с помощью кода. В этом разделе нас интересует *загрузка* музыкальных клипов *по требованию*. В противном случае музыка займет слишком много места в памяти, причем еще до того, как ею начнут пользоваться.

ОПРЕДЕЛЕНИЕ Загрузка по требованию (lazy-loading) означает, что загрузка файла заранее не делается, а откладывается до момента, когда этот файл понадобится. Обычно данные реагируют быстрее (например, немедленно начинается воспроизведение звука), когда они загружены заранее, но загрузка по требованию экономит память в ситуациях, когда быстрая реакция не имеет особого значения.

Второй способ решения проблемы с потреблением памяти связан с потоковой передачей музыки с диска. Как объяснялось в разделе 10.1.2, потоковая передача звука позволяет компьютеру обойтись без загрузки файла целиком. Тип загрузки указывается на панели Inspector импортированного аудиоклипа.

В конечном счете процедура подготовки к воспроизведению музыки делится на несколько этапов, в том числе включающих в себя оптимизацию потребления памяти.

10.4.1. Воспроизведение музыкальных циклов

Для воспроизведения музыки потребуется уже знакомая вам последовательность шагов, которой мы придерживались при программировании звуковых эффектов пользовательского интерфейса (фоновая музыка также является 2D-звуком, то есть не имеет источника в сцене), поэтому мы просто повторим ее:

1. Импортируем аудиоклипы.
2. Настраиваем компонент AudioSource для использования диспетчером `AudioManager`.
3. Пишем код для воспроизведения аудиоклипа в диспетчере `AudioManager`.
4. Добавляем UI-элементы управления музыкой.

Каждый шаг будет слегка модифицирован, так как теперь мы работаем с музыкой, а не со звуковыми эффектами. Рассмотрим первый шаг.

Шаг 1. Импорт аудиоклипов

Скачайте или запишите музыкальные треки. В прилагаемом к книге примере проекта я скачал с сайта www.freesound.org следующие композиции:

- loop от пользователя by Xythe/Ville Nousiainen;
- Intro Synth от пользователя noirenex.

Перетащите файлы в Unity и отредактируйте их настройки импорта на панели Inspector. Как уже упоминалось, настройки аудиоклипов со звуковыми эффектами и с музыкой в общем случае различаются. Прежде всего, для звука со сжатием следует выбрать формат Vorbis. Напомню, что в результате сжатия размер файлов значительно уменьшается. Одновременно слегка ухудшается качество звука, но для длинных музыкальных клипов это вполне допустимый компромисс; установите ползунок Quality на отметку 50 %.

Затем нужно выбрать параметр Load Type. Еще раз напомню, что нам нужно чтение музыки с диска, а не ее полная загрузка. Выберите в меню Load Type вариант Streaming. Установите флажок Load In Background, чтобы в процессе загрузки музыки игра не останавливалась и не замедлялась.

И даже после всех этих настроек для корректной загрузки нужно еще поместить аудиофайл в нужное место. Надеюсь, вы помните, что метод `Resources.Load()` загружает только содержимое папки Resources. Создайте новую папку с этим именем, а внутри нее — папку Music и перетащите туда аудиофайлы, как показано на рис. 10.5.

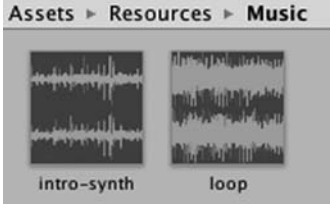


Рис. 10.5. Музыкальные аудиоклипы, помещенные в папку Resources

Это даст возможность выполнить шаг 1.

Шаг 2. Настройка компонента audiosource для использования диспетчером управления звуком

Теперь нам нужно создать новый компонент AudioSource для воспроизведения музыки. Создайте пустой объект GameObject, присвойте ему имя Music 1 (единица появилась потому, что позднее мы добавим объект Music 2) и сделайте его дочерним по отношению к объекту Audio.

Добавьте к объекту Music 1 компонент AudioSource и отредактируйте его параметры. Сбросьте флажок Play On Awake, но установите флажок Loop; если звуковой эффект обычно проигрывается один раз, музыка воспроизводится в цикле. Ползунок Spatial Blend оставьте в положении 2D, так как музыка не привязана к конкретной точке сцены.

Возможно, вы захотите уменьшить параметр Priority. Для звуковых эффектов мы оставили заданное по умолчанию значение 128, но для музыки имеет смысл снизить его примерно до 60. При наложении друг на друга различных звуков этот параметр указывает Unity, какие из звуков имеют больший приоритет; как ни странно, более низкие значения указывают на более высокий приоритет. При одновременном

воспроизведении слишком большого количества звуков аудиосистема начинает пропускать некоторые из них; увеличив приоритет музыки, мы гарантируем, что она будет звучать, даже если одновременно запустится множество звуковых эффектов.

Шаг 3. Программирование механизма воспроизведения аудиоклипов в диспетчере управления звуком

Мы настроили источник аудио `Music`, поэтому нужно добавить код следующего листинга в сценарий `AudioManager`.

Листинг 10.9. Воспроизведение музыки в диспетчере AudioManager

```
...
[SerializeField] private AudioSource music1Source;
[SerializeField] private string introBGMusic;
[SerializeField] private string levelBGMusic;
...
public void PlayIntroMusic() {
  PlayMusic(Resources.Load("Music/"+introBGMusic) as AudioClip);
}
public void PlayLevelMusic() {
  PlayMusic(Resources.Load("Music/"+levelBGMusic) as AudioClip);
}

private void PlayMusic(AudioClip clip) {
  music1Source.clip = clip;
  music1Source.Play();
}

public void StopMusic() {
  music1Source.Stop();
}
...
```

Эти строки указывают имена музыкальных клипов.

Загрузка музыки intro из папки Resources.

Загрузка основной музыки из папки Resources.

Воспроизведение музыки при помощи параметра AudioSource.clip.

Как обычно, при выборе игрового диспетчера на панели Inspector можно увидеть новые сериализованные переменные. Перетащите на ячейку для источника аудио клип Music 1. Затем введите в две строковые переменные имена музыкальных файлов: intro-synth и loop.

Остальная часть нового кода вызывает команды загрузки и воспроизведения музыки (или в последнем добавленном методе — остановки музыки). Команда `Resources.Load()` загружает именованный ресурс из папки Resources (учитывая, что нужные файлы расположены во вложенной папке Music). Эта команда возвращает обобщенный объект, но его можно преобразовать к конкретному типу (в данном случае — к типу `AudioClip`), воспользовавшись ключевым словом `as`.

Затем загруженный аудиоклип передается в метод `PlayMusic()`, который настраивает клип в компоненте `AudioSource` и вызывает метод `Play()`. Как я уже объяснял, звуковые эффекты лучше воспроизводятся методом `PlayOneShot()`, но установка клипа в компоненте `AudioSource` является более надежным подходом, позволяющим кроме всего прочего на время или совсем останавливать воспроизведение музыки.

Шаг 4. Добавление в UI элементов управления музыкой

Чтобы новые методы воспроизведения музыки выполняли какие-либо действия, их нужно откуда-то вызвать. Добавим к нашему UI дополнительные кнопки, щелчки на которых позволят включать разную музыку. Сейчас я опять перечислю этапы этого процесса с краткими пояснениями (подробности вы сможете найти в главе 6):

1. Измените ширину всплывающего окна на 350 (чтобы на нем поместились дополнительные кнопки).
2. Создайте новую кнопку и сделайте ее дочерним объектом по отношению к всплывающему окну.
3. Ширину кнопки сделайте равной 100 и поместите ее в точку с координатами 0, –20.
4. Раскройте иерархический список кнопки, выделите текстовую метку и поменяйте текст на Level Music.
5. Повторите эту последовательность еще два раза, чтобы создать две дополнительные кнопки.
6. Одну из них поместите в точку с координатами –105, –20, вторую — в точку с координатами 105, –20 (чтобы она появилась с другой стороны).
7. Текстовую метку первой кнопки поменяйте на Intro Music, второй — на No Music.

Теперь у нас есть три кнопки для воспроизведения различной музыки. Скопируйте метод из следующего листинга в сценарий `SettingsPopup`, который мы свяжем с каждой из кнопок.

Листинг 10.10. Добавление элементов управления музыкой в сценарий SettingsPopup

```
...
public void OnPlayMusic(int selector) { ← Этот метод получает от кнопки численный параметр.
  Managers.Audio.PlaySound(sound);

  switch (selector) { ← Вызываем для каждой кнопки свою музыкальную функцию в диспетчере AudioManager.
    case 1:
      Managers.Audio.PlayIntroMusic();
      break;
    case 2:
      Managers.Audio.PlayLevelMusic();
      break;
    default:
      Managers.Audio.StopMusic();
      break;
  }
}
...
```

Обратите внимание, что на этот раз функция принимает параметр типа `int`; как правило, связанные с кнопками методы лишены параметров и просто срабатывают по щелчку. Но сейчас нам нужно различие между кнопками, именно поэтому каждой присваивается свой номер.

При помощи стандартной процедуры свяжем кнопку с этим кодом: добавьте элемент в список OnClick на панели Inspector, перетащите всплывающее окно на ячейку объекта

и выберите в меню подходящую функцию. На этот раз нам предоставят текстовое поле для ввода числа, так как метод `OnPlayMusic()` принимает в качестве параметра число. Введите 1 для Intro Music, 2 для Level Music и произвольное число для No Music (у меня это был ноль). Инструкция `switch` в методе `OnMusic()` воспроизводит клип intro или level в зависимости от числа или останавливает воспроизведение, если число не равно ни 1, ни 2.

Щелкая на кнопках в процессе игры, вы сможете включать музыку. Великолепно! Код загружает аудиоклипы из папки Resources. Производительность воспроизведения оптимизирована. Осталось доработать две вещи: сделать отдельные элементы управления громкостью и обеспечить плавный переход при переключении музыки.

10.4.2. Отдельная регулировка громкости

В игре уже есть элемент управления громкостью, который сейчас влияет в числе прочего и на фоновую музыку. Но в большинстве игр громкость звуковых эффектов и музыки контролируется по отдельности. Давайте посмотрим, как этого достичь.

Первым делом нужно сделать так, чтобы компоненты AudioSources музыки игнорировали настройки компонента AudioListener. Мы хотим, чтобы регулятор громкости и кнопка отключения звука, связанные с глобальным компонентом AudioListener, продолжали влиять на все звуковые эффекты, но не затрагивали бы фоновую музыку. Листинг 10.11 содержит код, заставляющий источник музыки игнорировать громкость, задаваемую компонентом AudioListener. Заодно этот код создает регулятор громкости и возможность выключения музыки. Добавьте его к диспетчеру аудио.

Листинг 10.11. Раздельное управление громкостью музыки в диспетчере управления звуком

```
...
private float _musicVolume; ← Непосредственный доступ к закрытой переменной
public float musicVolume {      невозможен, только через функцию задания свойства.
  get {
    return _musicVolume;
  }
  set {
    _musicVolume = value;

    if (music1Source != null) { ← Непосредственно регулируем громкость источника звука.
      music1Source.volume = _musicVolume;
    }
  }
}
...
public bool musicMute {
  get {
    if (music1Source != null) {
      return music1Source.mute;
    }
    return false; ← Значение предлагается по умолчанию, если AudioSource отсутствует.
  }
  set {
```

```
    if (music1Source != null) {
      music1Source.mute = value;
    }
  }
}

public void Startup(NetworkService service) {   ← Выделенный курсивом код уже был в сценарии, тут он приведен для справки.
  Debug.Log("Audio manager starting...");

  _network = service;

  music1Source.ignoreListenerVolume = true;   ┐
  music1Source.ignoreListenerPause = true;    ┘ Эти свойства заставляют компонент AudioSource игнорировать громкость компонента AudioListener.

  soundVolume = 1f;
  musicVolume = 1f;

  status = ManagerStatus.Started;
}
...
```

Ключевой в этом коде является возможность непосредственной регулировки громкости в компоненте AudioSource с одновременным игнорированием источником глобальных параметров громкости, заданных в компоненте AudioListener. Существуют свойства для управления громкостью и включения/выключения звука отдельного источника музыки.

Метод `Startup()` инициализирует источник музыки со свойствами `ignoreListenerVolume` и `ignoreListenerPause`, имеющими значение `true`. Как следует из их названий, они заставляют источник звука игнорировать глобальные настройки громкости в компоненте AudioListener.

Вы можете щелкнуть на кнопке Play и удостовериться, что существующий элемент регулировки громкости на громкость музыки больше не влияет. Значит, нам нужен в UI еще один элемент управления, который будет отвечать за громкость музыки; отредактируйте сценарий `SettingsPopup` в соответствии со следующим листингом.

Листинг 10.12. Элементы управления громкостью в сценарии SettingPopup

```
...
public void OnMusicToggle() {
  Managers.Audio.musicMute = !Managers.Audio.musicMute;   ← Снова воспроизводим элемент управления включением звука, но на этот раз используя musicMute.
  Managers.Audio.PlaySound(sound);
}

public void OnMusicValue(float volume) {
  Managers.Audio.musicVolume = volume;   ← Снова воспроизводим элемент управления громкостью звука, но на этот раз используя musicVolume.
}
...
```

Особых пояснений этот код не требует — по большей части он повторяет элементы управления громкостью звука. Бросается в глаза замена свойств объекта `AudioManager` с `soundMute`/`soundVolume` на `musicMute`/`musicVolume`.

В редакторе создайте кнопку и ползунок. Вот последовательность ваших действий:

1. Измените высоту всплывающего окна на 225 (чтобы появилось место для дополнительных элементов управления).
2. Создайте UI-кнопку.
3. Сделайте ее потомком по отношению к всплывающему окну.
4. Поместите кнопку в точку с координатами 0, –60.
5. Раскройте иерархию кнопки и выделите текстовую метку.
6. Измените ее текст на Toggle Music.
7. Создайте ползунок (командой того же самого меню UI).
8. Сделайте ползунок потомком всплывающего окна и поместите его в точку с координатами 0, –85.

Свяжите эти UI-элементы с кодом в сценарии `SettingsPopup`. Найдите список OnClick/OnValueChanged в настройках UI-элементов, щелкните на кнопке со знаком + (плюс) для добавления новой записи и перетащите всплывающее окно на ячейку для объекта, после чего выберите в меню функцию. В первом случае это будет вариант OnMusicToggle(), во втором — OnMusicValue(). Обе функции находятся в разделе Dynamic Float.

Теперь запустите код и убедитесь, что на звуковые эффекты и фоновую музыку влияют разные элементы управления. Но на самом деле, это еще не все. Остался последний штрих — плавное микширование музыкальных треков.

10.4.3. Переход между песнями

В качестве финального штриха давайте заставим диспетчер управления звуком постепенно увеличивать и уменьшать громкость звука при переходе от одной мелодии к другой. Сейчас в процессе переключения мелодия просто обрывается, и начинается новый трек, что несколько режет ухо. Мы можем сгладить этот переход, сделав так, чтобы громкость предыдущего трека быстро затухала, в то время как громкость нового увеличивалась от нуля. Это простой, но хорошо продуманный код, объединяющий изученные вами методы регулировки громкости с сопрограммой, постепенно меняющей громкость.

Листинг 10.13 содержит код сценария `AudioManager`. Смысл происходящего сводится к простой концепции: теперь, когда у нас есть два отдельных источника звука, мы пользуемся ими для воспроизведения отдельных музыкальных треков и пошагово увеличиваем громкость одного источника, одновременно пошагово уменьшая громкость другого (как обычно, выделенный курсивом код уже присутствует в сценарии и показан для справки).

Листинг 10.13. Плавный переход между треками в сценарии AudioManager

```
...
[SerializeField] private AudioSource music2Source; ← Второй компонент AudioSource (первый тоже сохраняем).
private AudioSource _activeMusic; ← Следим за тем, какой из источников активен, а какой нет.
private AudioSource _inactiveMusic;
```

```
public float crossFadeRate = 1.5f;
private bool _crossFading; ← Переключатель, позволяющий избежать ошибок в процессе перехода.
...
public float musicVolume {
  ...
  set {
    _musicVolume = value;

    if (music1Source != null && !_crossFading) {
      music1Source.volume = _musicVolume;
      music2Source.volume = _musicVolume; ← Регулировка громкости обоих источников музыки.
    }
  }
}
...
public bool musicMute {
  ...
  set {
    if (music1Source != null) {
      music1Source.mute = value;
      music2Source.mute = value;
    }
  }
}

public void Startup(NetworkService service) {
  Debug.Log("Audio manager starting...");

  _network = service;

  music1Source.ignoreListenerVolume = true;
  music2Source.ignoreListenerVolume = true;
  music1Source.ignoreListenerPause = true;
  music2Source.ignoreListenerPause = true;

  soundVolume = 1f;
  musicVolume = 1f;

  _activeMusic = music1Source; ← Инициализируем один из источников как активный.
  _inactiveMusic = music2Source;

  status = ManagerStatus.Started;
}
...
private void PlayMusic(AudioClip clip) {
  if (_crossFading) {return;}
  StartCoroutine(CrossFadeMusic(clip)); ← При изменении музыкальной композиции вызываем сопрограмму.
}
private IEnumerator CrossFadeMusic(AudioClip clip) {
  _crossFading = true;

  _inactiveMusic.clip = clip;
  _inactiveMusic.volume = 0;
  _inactiveMusic.Play();

  float scaledRate = crossFadeRate * _musicVolume;
```

```
  while (_activeMusic.volume > 0) {
    _activeMusic.volume -= scaledRate * Time.deltaTime;
    _inactiveMusic.volume += scaledRate * Time.deltaTime;

    yield return null; ← Эта инструкция yield останавливает операции на один кадр.
  }

  AudioSource temp = _activeMusic; ← Временная переменная, используемая, когда мы меняем местами переменные _active и _inactive.

  _activeMusic = _inactiveMusic;
  _activeMusic.volume = _musicVolume;

  _inactiveMusic = temp;
  _inactiveMusic.Stop();

  _crossFading = false;
}

public void StopMusic() {
  _activeMusic.Stop();
  _inactiveMusic.Stop();
}
...
```

Первым дополнением стала переменная для второго источника музыки. Вам нужно сохранить первый объект AudioSource, но создать его копию (убедитесь, что настройки остались теми же — установите флажок Loop), а затем перетащить его на ячейку панели Inspector. При этом код задает переменные для активного и неактивного состояний источника звука, но это закрытые переменные, которые не появляются на панели Inspector. Именно они определяют, музыка из какого источника воспроизводится в конкретный момент времени.

Теперь в процессе воспроизведения музыки код вызывает сопрограмму. Она заставляет второй объект `AudioSource` начать воспроизведение нового музыкального трека, в то время как старый воспроизводится на втором источнике звука. После этого сопрограмма пошагово увеличивает громкость новой музыки, одновременно уменьшая громкость старой. После завершения перехода (то есть в момент, когда уровни громкости поменялись местами) сопрограмма меняет местами «активный» и «неактивный» источники.

Потрясающе! Мы добавили фоновую музыку в аудиосистему игры.

FMOD: ЗВУКОВОЙ ИНСТРУМЕНТ ДЛЯ ИГР

Аудиосистема в Unity приводится в действие популярной программной аудиобиблиотекой FMOD. Ее можно скачать с сайта www.fmod.org, но в Unity она уже встроена, хотя и без наиболее нетривиальных функциональных возможностей (о них вы можете узнать на сайте библиотеки).

Эти нетривиальные функциональные возможности предлагает подключаемый к Unity модуль FMOD Studio, но для выполнения упражнений этой главы предоставленного по умолчанию набора функциональных возможностей более чем достаточно. В этот набор входят самые необходимые для создания аудиосистемы в играх компоненты. Они удовлетворяют нуждам большинства разработчиков, дополнительный же модуль требуется только для добавления в игры сложных звуковых эффектов.

10.5. Заключение

- Для звуковых эффектов следует использовать несжатые аудиофайлы, а для фоновой музыки — сжатые, но в качестве исходного материала в обоих случаях можно использовать файлы формата WAV, так как Unity умеет сжимать импортированный звук.
- Для аудиоклипов возможны как 2D-звук, всегда воспроизводимый одинаково, так и 3D-звук, меняющийся в зависимости от положения слушателя.
- Громкость звуковых эффектов можно легко регулировать на глобальном уровне с помощью компонента AudioListener.
- Можно по отдельности регулировать громкость отдельных источников звука.
- Вы можете выполнять переходы между треками фоновой музыки, задавая громкость отдельных источников звука.

11 Объединение фрагментов в готовую игру

- ✓ Сборка объектов и кода из других проектов
- ✓ Программирование элементов наведения и щелчка
- ✓ Обновление UI при переходе от старой системы к новой
- ✓ Загрузка новых уровней как реакция на достижение поставленных целей
- ✓ Настройка условий выигрыша/проигрыша
- ✓ Сохранение и загрузка текущего состояния игры

В этой главе мы соберем воедино созданные нами проекты. Большинство предыдущих глав содержало относительно разрозненный материал, и необходимости рассматривать игру в целом просто не возникало. Но сейчас мы рассмотрим процесс объединения разработанных по отдельности фрагментов, чтобы вы знали, каким образом строится игра. Обсудим мы и обобщенную структуру игры, в том числе переход с одного уровня на другой и процесс завершения (например, появление надписи Game Over, когда персонаж умирает, или надписи Success, когда он доходит до выхода). Я научу вас сохранять игру, потому что по мере увеличения объема игры растет и важность сохранения полученных игроком результатов.

ВНИМАНИЕ В этой главе по большей части рассматриваются задачи, подробности решения которых объяснялись в предыдущих главах, поэтому я ограничусь общими рекомендациями. Если окажется, что вы не понимаете каких-то вещей, перечитайте соответствующую главу (например, главу 6, если у вас возникли сложности с элементами UI).

В качестве проекта мы рассмотрим ролевой боевик. В таких играх камера располагается сверху и смотрит четко вниз, как показано на рис. 11.1, а направления перемещений персонажа указываются щелчками мыши. Возможно, вы знакомы с игрой *Diablo*, которая представляет собой как раз ролевой боевик. Я специально перешел к новому жанру, чтобы познакомить вас с как можно большим числом различных типов игр!

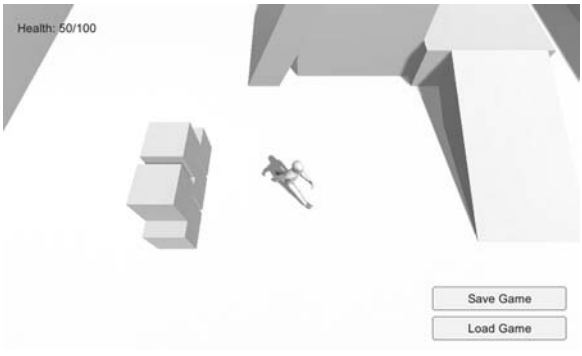


Рис. 11.1. Снимок при наблюдении сверху вниз

С такими масштабными играми, как в этой главе, вы еще не работали. Мы рассмотрим следующие темы:

- Вид на сцену сверху и перемещения методом наведения и щелчка.
- Возможность управлять устройствами путем щелчка на них.
- Разбросанные элементы, которые можно собирать.
- Инвентарь, отображаемый в окне UI.
- Бродящие по уровню враги.
- Возможность сохранять игру и возобновлять ее с прерванной точки.
- Три уровня, которые нужно завершать по очереди.

Как видите, вам предстоит насыщенная программа; к счастью, это практически последняя глава!

11.1. Построение ролевого боевика изменением назначения проектов

Основой нашего ролевого боевика послужит проект из главы 8. Скопируйте его папку и откройте в Unity. Если вы пропустили данную главу, просто скачайте соответствующий пример проекта.

Мы выбрали в качестве основы проект из главы 8, так как он наиболее полно отвечает нашим целям, а значит, требует наименьшего количества модификаций (в сравнении с остальными проектами). В конечном счете мы сведем вместе ресурсы из разных глав, так что с технической точки зрения нет никакой разницы, откуда начинать.

Вот краткий список фрагментов проекта из главы 8:

- Персонаж с уже настроенным контроллером анимации.
- Камера, следующая за персонажем.
- Уровень с полом, стенами и наклонными поверхностями.
- Источники света и тени.
- Работающие устройства, в том числе монитор, меняющий цвет.

- Инвентарь, который можно собирать.
- Фреймворк из интерфейсных диспетчеров.

Как видите, большая часть работы по созданию демонстрационной версии ролевой игры уже выполнена, но остался еще ряд деталей, которые требуется отредактировать или добавить.

11.1.1. Сборка ресурсов и кода из разных проектов

Первым делом нам нужно обновить фреймворк диспетчеров и добавить в проект врагов, управляемых компьютером. Первую задачу мы уже решали в главе 9, добавляя в фреймворк из главы 8 новые детали. Врагов же мы программировали в главе 3.

Обновление фреймворка диспетчеров

Обновить диспетчеры достаточно просто, поэтому эту задачу мы решим первой. В главе 9 мы редактировали интерфейс `IGameManager` (см. следующий листинг).

Листинг 11.1. Скорректированный интерфес IGameManager

```
public interface IGameManager {
  ManagerStatus status {get;}

  void Startup(NetworkService service);
}
```

В коде этого листинга появилась ссылка на сценарий `NetworkService`, значит, вам обязательно нужно скопировать его в проект; перетащите в новый проект файл со сценарием из проекта главы 9 (напоминаю, что каждый Unity-проект представляет собой папку на диске, то есть вам нужно взять файл из папки, в которой он хранится). Теперь отредактируйте сценарий `Managers.cs`, так как мы поменяли интерфейс, с которым он работает (см. следующий листинг).

Листинг 11.2. Изменения в сценарии Managers

```
...
private IEnumerator StartupManagers() { ← Исправления в начале метода.
  NetworkService network = new NetworkService();

  foreach (IGameManager manager in _startSequence) {
    manager.Startup(network);
  }
...
```

Напоследок отредактируйте сценарии `InventoryManager` и `PlayerManager`, учтя в них внесенные в интерфейс изменения. Следующий листинг демонстрирует исправления в сценарии `InventoryManager`; аналогичные правки, но с другими именами, вносятся и в сценарий `PlayerManager`.

Листинг 11.3. Изменения в сценарии InventoryManager с учетом модификаций в IGameManager

```
...
private NetworkService _network;
```

```
public void Startup(NetworkService service) {
  Debug.Log("Inventory manager starting..."); ← Одни и те же правки в обоих сценариях,
                                                просто с разными именами.
  _network = service;

  _items = new Dictionary<string, int>();
  ...
```

После внесения всех этих небольших изменений все остальное должно функционировать так же, как и раньше. Мы скорректировали операции, происходящие невидимо для игрока, в игре же никаких изменений не произошло. Это была самая простая часть редактирования, дальше будет сложнее.

Копирование врагов, оснащенных искусственным интеллектом

Наша задача сводится не только к редактированию сценария `NetworkServices` из главы 9, но и к вставке в проект нашего врага с искусственным интеллектом из главы 3. Реализация такого персонажа требует целого набора сценариев и графических ресурсов, которые сейчас нужно импортировать.

Первым делом скопируйте все сценарии (напоминаю, что в сценариях `WanderingAI` и `ReactiveTarget` программировалось поведение врага, сценарий `Fireball` определял снаряд, которым враг атаковал компонент `PlayerCharacter`, а сценарий `SceneController` отвечал за порождение врагов):

- `PlayerCharacter.cs`;
- `SceneController.cs`;
- `WanderingAI.cs`;
- `ReactiveTarget.cs`;
- `Fireball.cs`.

Заодно импортируйте в проект материал Flame и шаблоны Fireball и Enemy. Тем, кто предпочитает врага из главы 10, а не из главы 3, понадобится также материал fire particle.

В процессе импорта обычно нарушаются связи между ресурсами, поэтому их нужно восстановить, чтобы все снова начало работать. В частности, сценарии, скорее всего, некорректно связаны с шаблонами экземпляров. Например, для шаблона Enemy на панели Inspector вы увидите отсутствие двух сценариев. Чтобы исправить ошибку, щелкните на кнопке в виде окружности, как показано на рис. 11.2, и выберите в списке сценариев варианты WanderingAI и ReactiveTarget.

Рис. 11.2. Связывание сценария с компонентом

Аналогичным образом проверьте шаблон Fireball, и если нужно, повторно соедините его со сценарием. После этого проверьте ссылки на материалы и текстуры.

Теперь добавьте к объекту-контроллеру сценарий `SceneController.cs` и перетащите шаблон Enemy на одноименную ячейку компонента панели Inspector. Возможно, потребуется перетащить шаблон Fireball на компонент сценария объекта Enemy (выделите шаблон Enemy и на панели Inspector посмотрите поле WanderingAI). Кроме того, свяжите сценарий `PlayerCharacter.cs` с объектом player, чтобы враги начали атаковать игрока.

Запустите игру и посмотрите, как вокруг вас перемещается враг. Он кидает в игрока огненные шары, пока не причиняя особого вреда; выделите шаблон Fireball и присвойте его параметру Damage значение 10.

ПРИМЕЧАНИЕ Пока что слежение за игроком и попытки его поразить враг выполняет с небольшой точностью. Я начал бы исправление ситуации с расширения сектора обзора врага (воспользовавшись для этого скалярным произведением, как было показано в главе 8). В конечном счете разработчики проводят много времени над доработками игры. К таким доработкам относится и поведение врагов. Этот процесс имеет решающее значение для выхода окончательный версии, но в книге мы им заниматься не будем.

Когда в главе 3 мы писали код, здоровье игрока фигурировало только в качестве некоего тестового атрибута. Но теперь в игре появился диспетчер игрока, а значит, можно отредактировать сценарий `PlayerCharacter` в соответствии со следующим листингом, добавив в него средства для работы со здоровьем в данном диспетчере.

Листинг 11.4. Добавляем в сценарий PlayerCharacter возможность использовать здоровье в диспетчере игрока

```
using UnityEngine;
using System.Collections;

public class PlayerCharacter : MonoBehaviour {
  public void Hurt(int damage) {
    Managers.Player.ChangeHealth(-damage); ←
  }
}
```

Используйте в диспетчере PlayerManager это значение вместо переменной в объекте PlayerCharacter.

Теперь у вас есть демонстрационный ролик, фрагменты которого собраны из ранее выполненных проектов. В сцену был добавлен враг, что сделало игру более захватывающей. Но элементы управления и угол обзора до сих пор те же самые, что и в демонстрационном ролике от третьего лица. Нам нужно создать элементы наведения и щелчка (point-and-click controls) для нашей ролевой игры.

11.1.2. Элементы наведения и щелчка

Нашему демонстрационному ролику требуется камера, нацеленная сверху вниз, и управление перемещениями персонажа с помощью мыши (см. рис. 11.1). В настоящее время мышь управляет камерой, в то время как перемещения игрока контролируются с клавиатуры (это мы запрограммировали в главе 7), то есть это диаметрально противоположно тому, что нам нужно. Кроме того, мы отредактируем меняющий цвета монитор, заставив его реагировать на щелчки мыши. Впрочем, в обоих случаях огромного количества правок код не требует, поэтому давайте просто возьмем и внесем необходимые коррективы в сценарии движения и устройств.

Обзор сцены сверху вниз

Первым делом присвойте координате *Y* камеры значение 8, чтобы поднять ее над сценой. Кроме того, мы отредактируем сценарий `OrbitCamera`, убрав оттуда управление с помощью мыши и оставив в качестве элементов управления только клавиши со стрелками (см. следующий листинг).

Листинг 11.5. Удаление средств управления из сценария OrbitCamera с помощью мыши

```
...
void LateUpdate() {
  _rotY -= Input.GetAxis("Horizontal") * rotSpeed; ← Меняем направление на обратное.
  Quaternion rotation = Quaternion.Euler(0, _rotY, 0);
  transform.position = target.position - (rotation * _offset);
  transform.LookAt(target);
}
...
```

БЛИЖНЯЯ/ДАЛЬНЯЯ ПЛОСКОСТИ ОТСЕЧКИ КАМЕРЫ

Так как дело дошло до настройки камеры, я хотел бы упомянуть о такой вещи, как ближняя/дальняя плоскости отсечки. Раньше эти параметры не рассматривались, так как нам прекрасно подходили их значения, предлагаемые по умолчанию, но в каких-то других проектах они могут вам потребоваться.

Выделите камеру в сцене и обратите внимание на настройку Clipping Planes на панели Inspector; именно здесь указываются оба значения, Near и Far. Они задают переднюю и заднюю границы, внутри которых происходит визуализация сеток: полигоны, оказавшиеся ближе, чем задано значением Near, и дальше, чем задано значением Far, отсекаются.

Значения параметров Near/Far должны быть, с одной стороны, как можно ближе друг к другу, с другой — отстоять друг от друга достаточно для визуализации сцены в целом. При слишком большом расстоянии между этими плоскостями (ближняя располагается слишком близко, а дальняя — слишком далеко) алгоритм визуализации теряет возможность различать, какие полигоны ближе. В результате возникает ошибка визуализации, называемая z-конфликтом (z-fighting), когда полигоны мерцают один поверх другого.

Так как мы подняли камеру повыше, при воспроизведении игры сцена будет демонстрироваться сверху. Но движение все еще управляется с клавиатуры, поэтому давайте напишем сценарий для перемещений путем наведения и щелчка.

Написание кода движения

Основная идея этого кода (проиллюстрированная на рис. 11.3) сводится к автоматическому перемещению персонажа в указанную точку. Эта точка задается щелчком мыши. При этом код, перемещающий игрока, напрямую на мышь не реагирует, но косвенным образом управляет перемещением персонажа.

ПРИМЕЧАНИЕ Этот же алгоритм перемещения можно использовать для персонажей с искусственным интеллектом. Однако в этом случае целевая точка не задается щелчками мыши, а просто находится на заданной для персонажа траектории.

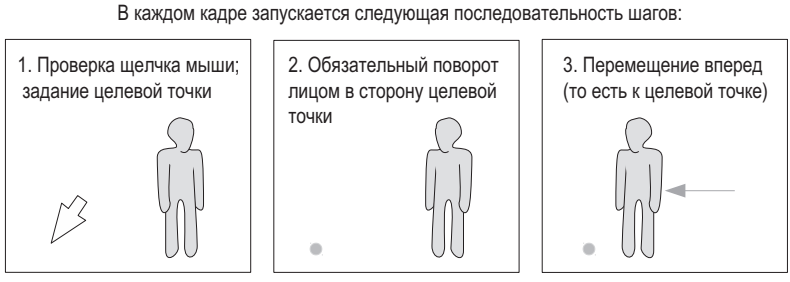


Рис. 11.3. Схема работы элементов наведения и щелчка

Создайте новый сценарий `PointClickMovement` и скопируйте в него код сценария `RelativeMovement` (ведь нам нужно, чтобы он реализовывал падения и анимацию). Замените компонент RelativeMovement объекта player. Затем отредактируйте код нового сценария в соответствии со следующим листингом.

Листинг 11.6. Новый код движения в сценарии PointClickMovement

```
...
public class PointClickMovement : MonoBehaviour {  ← Исправьте имя после вставки кода.
  ...
  public float deceleration = 20.0f;
  public float targetBuffer = 1.5f;
  private float _curSpeed = 0f;
  private Vector3 _targetPos = Vector3.one;
  ...
  void Update() {
    Vector3 movement = Vector3.zero;

    if (Input.GetMouseButton(0)) {  ← Задаем целевую точку по щелчку мыши.
      Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);  ← Испускаем луч в точку щелчка мышью.
      RaycastHit mouseHit;
      if (Physics.Raycast(ray, out mouseHit)) {
        _targetPos = mouseHit.point;  ← Устанавливаем цель в точке попадания луча.
        _curSpeed = moveSpeed;
      }
    }

    if (_targetPos != Vector3.one) {  ← Перемещаем при заданной целевой точке.
      Vector3 adjustedPos = new Vector3(_targetPos.x,
      transform.position.y, _targetPos.z);
      Quaternion targetRot = Quaternion.LookRotation(
        adjustedPos - transform.position);
      transform.rotation = Quaternion.Slerp(transform.rotation,
        targetRot, rotSpeed * Time.deltaTime);  ← Поворачиваем по направлению к цели.

      movement = _curSpeed * Vector3.forward;
      movement = transform.TransformDirection(movement);

      if (Vector3.Distance(_targetPos, transform.position) < targetBuffer) {
        _curSpeed -= deceleration * Time.deltaTime;  ← Снижаем скорость до нуля при приближении к цели.
```

```
        if (_curSpeed <= 0) {
          _targetPos = Vector3.one;
        }
      }
    }
    _animator.SetFloat("Speed", movement.sqrMagnitude); ← С этого момента код не меняется.
    ...
```

Мы практически полностью убрали начало метода `Update()`, так как этот код отвечал за управление перемещением с клавиатуры. Обратите внимание, что новый код содержит две основные инструкции `if`: одна выполняется при щелчке мышью, вторая — после задания целевой точки.

Целевая точка задается в месте щелчка мышью. Именно тут нам снова пригодится метод испускания луча: он позволит определить, какая точка сцены попала под указатель. И место попадания луча фиксируется как целевая точка.

Вторая условная инструкция первым делом обеспечивает поворот персонажа лицом к целевой точке. Метод `Quaternion.Slerp()` выполняет этот поворот плавно, а не скачком. Затем мы преобразуем направление движения от локальных координат персонажа к глобальным координатам (для того, чтобы пойти вперед). Последним проверяется расстояние между персонажем и целью: если персонаж уже почти достиг нужной точки, его скорость постепенно уменьшается, а в конце происходит удаление целевой точки.

УПРАЖНЕНИЕ: ОТКЛЮЧЕНИЕ УПРАВЛЕНИЯ ПРЫЖКАМИ

Сейчас в нашем сценарии есть элемент управления прыжками, скопированный из сценария RelativeMovement. При нажатии клавиши пробела персонаж подпрыгивает, между тем как при управлении движением методом наведения и щелчка подобного быть не должно. Подсказка: отредактируйте код в условной инструкции 'if (hitGround)'.

Итак, мы запрограммировали перемещение персонажа с помощью мыши. Запустите игру, чтобы посмотреть, как это выглядит на практике. Теперь нужно сделать так, чтобы на щелчки мыши стали реагировать и наши устройства.

Управление устройствами с помощью мыши

В главе 8 управление устройствами осуществлялось с клавиатуры. Нам же нужно, чтобы они управлялись мышью. Для этого создадим сценарий, от которого будут наследовать все устройства; именно туда мы поместим процедуру управления посредством мыши. Присвойте новому сценарию имя `BaseDevice` и скопируйте в него код следующего листинга.

Листинг 11.7. Сценарий BaseDevice, срабатывающий по щелчку мыши

```
using UnityEngine;
using System.Collections;

public class BaseDevice : MonoBehaviour {
  public float radius = 3.5f;
```

```
  void OnMouseDown() { ← Функция, запускаемая щелчком.
    Transform player = GameObject.FindWithTag("Player").transform;
    if (Vector3.Distance(player.position, transform.position) < radius) {
      Vector3 direction = transform.position - player.position;
      if (Vector3.Dot(player.forward, direction) > .5f) {
        Operate(); ← Вызов метода Operate(), если персонаж находится рядом и повернут лицом к устройству.
      }
    }
  }

  public virtual void Operate() { ← Ключевое слово virtual указывает на метод, который можно переопределить после наследования.
    // поведение конкретного устройства
  }
}
```

Большая часть операций выполняется внутри метода `OnMouseDown()`, так как именно его вызывает класс `MonoBehaviour` после щелчка на объекте. Первым делом проверяется расстояние до персонажа, а затем с помощью скалярного произведения определяется, повернут ли он в сторону устройства. Метод `Operate()` пока представляет собой пустую оболочку, которая будет заполняться кодом устройств, наследующих данный сценарий.

ПРИМЕЧАНИЕ Этот код ищет в сцене объект с тегом Player, поэтому назначьте данный тег объекту player. Раскрывающийся список Tag находится в верхней части панели Inspector; можно задать свой тег, но среди тегов, предлагаемых по умолчанию, есть нужный нам вариант Player. Выделите объект player и затем выберите для него в меню тег Player.

Теперь, когда у нас появился сценарий `BaseDevice`, можно внести изменения в сценарий `ColorChangeDevice` в соответствии со следующим листингом.

Листинг 11.8. Добавляем в сценарий ColorChangeDevice код наследования от сценария BaseDevice

```
using UnityEngine;
using System.Collections;

public class ColorChangeDevice : BaseDevice { ← Наследование от BaseDevice, а не от MonoBehaviour.
  public override void Operate() {
    Color random = new Color(Random.Range(0f,1f),
    Random.Range(0f,1f), Random.Range(0f,1f));
    GetComponent<Renderer>().material.color = random;
  }
}
```

Наследуя от класса `BaseDevice`, а не от `MonoBehaviour`, этот сценарий получает функциональность управления мышью. Затем он переопределяет пустой метод `Operate()`, добавляя туда поведение, меняющее цвет монитора.

Теперь устройство управляется щелчками мыши. Кроме того, мы убираем у персонажа компонент сценария `DeviceOperator`, так как этот сценарий задает управление устройством с клавиатуры.

Новый вариант управления устройством, к сожалению, конфликтует с элементами управления перемещениями: целевая точка задается щелчком мыши, но мы не хотим, чтобы она появлялась в момент щелчка на устройствах. Эту проблему помогают решить слои; аналогично тому, как мы присвоили персонажу тег, объекты можно распределить по разным слоям, а код будет проверять это обстоятельство. Давайте добавим в сценарий `PointClickMovement` (см. следующий листинг) проверку слоя, к которому принадлежит объект.

Листинг 11.9. Корректировка кода, обрабатывающего щелчки мышью, в сценарии PointClickMovement

```
...
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
RaycastHit mouseHit;
if (Physics.Raycast(ray, out mouseHit)) {
  GameObject hitObject = mouseHit.transform.gameObject;
  if (hitObject.layer == LayerMask.NameToLayer("Ground")) {
    _targetPos = mouseHit.point;
    _curSpeed = moveSpeed;
  }
}
...
```

Добавленный код; остальное приведено для справки.

Этот листинг добавляет в код обработки щелчков мыши условную инструкцию, проверяющую, принадлежит ли объект, на котором был сделан щелчок, слою Ground. Раскрывающийся список Layers (как и список Tags) находится в верхней части панели Inspector; раскройте его, чтобы посмотреть доступные варианты. Как и в случае с тегами, несколько слоев заданы по умолчанию. Нам в данном случае требуется новый слой, поэтому выберите в меню вариант Add Layer. Введите в пустое поле название Ground (например, в поле 8; присутствующий в коде метод `NameToLayer()` преобразует имена в номера слоев, что дает возможность использовать имя вместо номера).

Теперь, когда в меню появился слой Ground, поместите в него все объекты, по которым может ходить персонаж, то есть пол здания вместе с наклонными поверхностями и платформами. Выделите все эти объекты и выберите в меню Layers вариант Ground.

Запустите игру и убедитесь, что щелчки на меняющем цвет мониторе не приводят персонаж в движение. Великолепно, мы завершили работу над элементами наведения и щелчка! Теперь в проект из предыдущих проектов осталось добавить только импортированный пользовательский интерфейс.

11.1.3. Замена старого GUI новым

В главе 8 мы использовали старый графический пользовательский интерфейс непосредственного режима, так как его было проще программировать. Но этот UI выглядит не так красиво, как интерфейс из главы 6, поэтому давайте воспользуемся новой системой. Более новый UI визуально лучше доработан, чем старый GUI. Рисунок 11.4 демонстрирует интерфейс, который вам предстоит создать.

Начнем мы с настройки графики для UI. Как только все изображения элементов UI окажутся в сцене, можно связать с объектами UI соответствующие сценарии. Я пере-

Рис. 11.4. Вид UI для текущего проекта

числю этапы создания, не вдаваясь в детали; их вы можете вспомнить самостоятельно, обратившись к главе 6:

1. Импортируйте изображение popup.png как спрайт (выберите нужный вариант в списке Texture Type).
2. В окне диалога Sprite Editor задайте со всех сторон границы размером 12 пикселов (не забудьте применить изменения).
3. Создайте в сцене холст (GameObject ▸ UI ▸ Canvas).
4. Установите для холста флажок Pixel Perfect.
5. По желанию: присвойте объекту имя HUD Canvas и переключитесь в режим отображения 2D.
6. Создайте связанный с холстом объект Text (GameObject ▸ UI ▸ Text).
7. Задайте для объекта Text привязку к верхнему левому углу и положение 100, –40.
8. В качестве текста метки введите Health.
9. Создайте связанное с холстом изображение (GameObject ▸ UI ▸ Image).
10. Присвойте новому объекту имя Inventory Popup.
11. Назначьте спрайт всплывающего окна ячейке Source Image изображения.
12. Выберите в меню Image Type вариант Sliced и установите флажок Fill Center.
13. Расположите изображение всплывающего окна в точке с координатами 0, 0 и сделайте его ширину равной 250, а высоту — 150.

ПРИМЕЧАНИЕ Напоминаю, что для перехода от просмотра трехмерной сцены к просмотру двухмерного интерфейса нужно нажать кнопку режима 2D view и дважды щелкнуть на объекте Canvas или Building, чтобы отмасштабировать этот объект.

Теперь у нас есть метка Health в углу и большое всплывающее окно голубого цвета в центре. Запрограммируем эти элементы перед тем, как углубиться в UI-функцио-

нальность. Код интерфейса будет пользоваться уже знакомой вам по главе 6 системой диспетчеров, поэтому скопируйте сценарий `Messenger`. Затем создайте сценарий `GameEvent` и введите в него код следующего листинга.

Листинг 11.10. Сценарий GameEvent, который будет использоваться с системой диспетчеров

```
public static class GameEvent {
  public const string HEALTH_UPDATED = "HEALTH_UPDATED";
}
```

Пока у нас определено только одно событие; постепенно мы добавим еще несколько. Разошлите сообщение об этом событии из сценария `PlayerManager`, как показано в следующем листинге.

Листинг 11.11. Рассылка сообщения health из сценария PlayerManager

```
...
public void ChangeHealth(int value) {
  health += value;
  if (health > maxHealth) {
    health = maxHealth;
  } else if (health < 0) {
    health = 0;
  }

  Messenger.Broadcast(GameEvent.HEALTH_UPDATED); ← Добавляем строку в конец этой функции.
}
...
```

Это сообщение рассылается каждый раз, когда метод `ChangeHealth()` завершает свою работу, сообщая остальной программе об изменении параметра `health`. В качестве реакции на это событие должна меняться метка `health`, поэтому создайте сценарий `UIController` и введите в него код следующего листинга.

Листинг 11.12. Сценарий UIController, обслуживающий интерфейс

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

public class UIController : MonoBehaviour {
  [SerializeField] private Text healthLabel; ← Ссылка на UI-объект в сцене.
  [SerializeField] private InventoryPopup popup;

  void Awake() { ← Задаем подписчика для события обновления здоровья.
    Messenger.AddListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
  }
  void OnDestroy() {
    Messenger.RemoveListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
  }

  void Start() {
    OnHealthUpdated(); ← Вызов функции вручную при загрузке.
    popup.gameObject.SetActive(false); ← Всплывающее окно инициализируется как скрытое.
```

```
  }

  void Update() {
    if (Input.GetKeyDown(KeyCode.M)) { ← Отображение всплывающего окна нажатием клавиши M.
      bool isShowing = popup.gameObject.activeSelf;
      popup.gameObject.SetActive(!isShowing);
      popup.Refresh();
    }
  }

  private void OnHealthUpdated() { ← Подписчик события вызывает функцию для обновления метки health.
    string message = "Health: " + Managers.Player.health + "/" +
      Managers.Player.maxHealth;
    healthLabel.text = message;
  }
}
```

Присоедините этот сценарий к объекту Controller и удалите сценарий `BasicUI`. Кроме того, создайте сценарий `InventoryPopup` (добавьте в него пустой открытый метод `Refresh()`; остальной код мы напишем позже) и свяжите его со всплывающим окном (это объект Image). Теперь можно перетащить всплывающее окно на ячейку для ссылки в компоненте Controller; свяжите с этим компонентом еще и метку health.

Эта метка меняется при ранении персонажа и при использовании им пакетов здоровья, а нажатие клавиши `M` делает всплывающее окно видимым. Но нужно скорректировать одну маленькую деталь. Пока что щелчок на всплывающем окне вызывает движение персонажа, как и в случае с устройствами, но нам не нужно, чтобы щелчок на UI-элементе приводил к заданию целевой точки. Внесите показанные в следующем листинге изменения в сценарий `PointClickMovement`.

Листинг 11.13. Проверка UI в сценарии PointClickMovement

```
using UnityEngine.EventSystems;
...
void Update() {
  Vector3 movement = Vector3.zero;
  if (Input.GetMouseButton(0) &&
    !EventSystem.current.IsPointerOverGameObject()) {
    ...
```

Обратите внимание на условную инструкцию, проверяющую местоположение указателя мыши в момент щелчка. На этом работу над общей структурой интерфейса можно считать выполненной, поэтому давайте перейдем к всплывающему окну с инвентарем.

Реализация всплывающего списка инвентаря

Пока у нас есть только пустое всплывающее окно, в то время как на нем должен отображаться список игрового инвентаря, как показано на рис. 11.5. Вот последовательность создания этих объектов UI:

1. Создайте четыре изображения и сделайте их потомками всплывающего окна (путем перетаскивания на вкладке Hierarchy).

2. Создайте четыре текстовых метки и сделайте их потомками всплывающего окна.
3. Расположите изображения в точках с координатой *Y*, равной 0, и координатами *X*, равными –75, –25, 25 и 75.
4. Расположите текстовые метки в точках с координатой *Y*, равной 50, и координатами *X*, равными –75, –25, 25 и 75.
5. Выберите для текста (не привязка!) выравнивание по горизонтали Center, выравнивание по вертикали, а высоту сделайте равной 60.
6. В папке Resources превратите все значки инвентаря в спрайты (изначально они являются текстурами).
7. Перетащите эти спрайты на ячейку Source Image объектов Image (заодно щелкните на кнопке Set Native Size).
8. Введите x2 для всех текстовых меток.
9. Добавьте еще одну текстовую метку и две кнопки, сделав их потомками всплывающего окна.
10. Расположите текстовую метку в точке с координатами –120, –55 и выберите для выравнивания по горизонтали вариант Right.
11. В качестве текста метки введите Energy.
12. Присвойте параметру Width обеих кнопок значение 60, а затем расположите в точках с координатой *Y*, равной –50, и координатами *X*, равными 0 и 70.
13. На одной кнопке напишите Equip, на другой — Use.

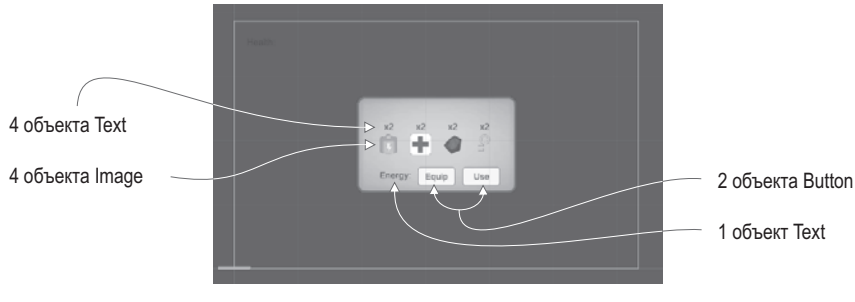


Рис. 11.5. Схема UI для отображения инвентаря

Это визуальные элементы для всплывающего окна со списком инвентаря, а сейчас мы напишем код. Введите содержимое следующего листинга в сценарий `InventoryPopup`.

Листинг 11.14. Полный сценарий InventoryPopup

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

public class InventoryPopup : MonoBehaviour {
```

```
[SerializeField] private Image[] itemIcons;
[SerializeField] private Text[] itemLabels;
```
Массивы для ссылки на четыре изображения и текстовые метки.

```
[SerializeField] private Text curItemLabel;
[SerializeField] private Button equipButton;
[SerializeField] private Button useButton;

private string _curItem;

public void Refresh() {
  List<string> itemList = Managers.Inventory.GetItemList();

  int len = itemIcons.Length;
  for (int i = 0; i < len; i++) {
    if (i < itemList.Count) {
      itemIcons[i].gameObject.SetActive(true);
      itemLabels[i].gameObject.SetActive(true);

      string item = itemList[i];

      Sprite sprite = Resources.Load<Sprite>("Icons/"+item);
      itemIcons[i].sprite = sprite;
      itemIcons[i].SetNativeSize();

      int count = Managers.Inventory.GetItemCount(item);
      string message = "x" + count;
      if (item == Managers.Inventory.equippedItem) {
        message = "Equipped\n" + message;
      }
      itemLabels[i].text = message;

      EventTrigger.Entry entry = new EventTrigger.Entry();
      entry.eventID = EventTriggerType.PointerClick;
      entry.callback.AddListener((BaseEventData data) => {
        OnItem(item);
      });
      EventTrigger trigger = itemIcons[i].GetComponent<EventTrigger>();
      trigger.delegates.Clear();
      trigger.delegates.Add(entry);
    }
    else {
      itemIcons[i].gameObject.SetActive(false);
      itemLabels[i].gameObject.SetActive(false);
    }
  }

  if (!itemList.Contains(_curItem)) {
    _curItem = null;
  }
  if (_curItem == null) {
    curItemLabel.gameObject.SetActive(false);
    equipButton.gameObject.SetActive(false);
    useButton.gameObject.SetActive(false);
  }
```

Проверка списка инвентаря в процессе циклического просмотра всех изображений элементов UI.

Загрузка спрайта из папки Resources.

Изменение размеров изображения под исходный размер спрайта.

На метке может появиться не только количество элементов, но и слово «Equipped».

Превращаем значки в интерактивные объекты.

Лямбда-функция, позволяющая по-разному активировать каждый элемент.

Сброс подписчика, чтобы начать с чистого листа.

Добавление функции-подписчика к классу EventTrigger.

Скрываем изображение/текст при отсутствии элементов для отображения.

Скрываем кнопки при отсутствии выделенных элементов.

```
    else { ← Отображение выделенного в данный момент элемента.
      curItemLabel.gameObject.SetActive(true);
      equipButton.gameObject.SetActive(true);
      if (_curItem == "health") {
        useButton.gameObject.SetActive(true);
      } else {
        useButton.gameObject.SetActive(false);
      }

      curItemLabel.text = _curItem+":";
    }
  }

  public void OnItem(string item) { ← Функция, вызываемая подписчиком события щелчка мыши.
    _curItem = item;
    Refresh(); ← Актуализируем отображение инвентаря после внесения изменений.
  }

  public void OnEquip() {
    Managers.Inventory.EquipItem(_curItem);
    Refresh();
  }

  public void OnUse() {
    Managers.Inventory.ConsumeItem(_curItem);
    if (_curItem == "health") {
      Managers.Player.ChangeHealth(25);
    }
    Refresh();
  }
}
```

Это был огромный сценарий! Пришло время связать все элементы интерфейса. Компонент сценария теперь обладает ссылками на различные объекты, включая два массива; раскройте оба массива и сделайте их длину равной 4, как показано на рис. 11.6. Четыре изображения перетащите на ячейки массива значков, а четыре текстовых метки — на ячейки массива меток.

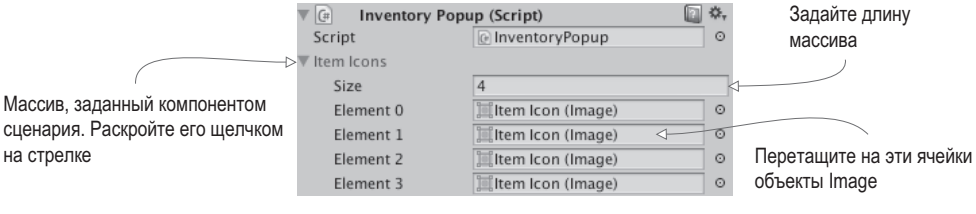


Рис. 11.6. Отображение массива на панели Inspector

ПРИМЕЧАНИЕ Если вы не уверены, какой объект нужно связывать с определенной ячейкой (они все выглядят одинаково), щелкните на ячейке и посмотрите, какой объект будет выделен на вкладке Hierarchy.

Аналогичным образом добавьте в ячейки компонента ссылки на текстовую метку и кнопки в нижней части всплывающего окна. После связывания этих объектов

добавим к обеим кнопкам подписчика на событие OnClick. Свяжите эти события со всплывающим окном и выберите подходящий вариант — OnEquip() или OnUse().

Наконец, добавим ко всем четырем изображениям элементов компонент EventTrigger. Сценарий `InventoryPopup` редактирует этот компонент для каждого значка, поэтому лучше, чтобы во всех случаях он присутствовал! Компонент EventTrigger находится в разделе Event (напомню, что для добавления нужно щелкнуть на кнопке Add Component). Возможно, вам будет удобнее добавить его методом копирования. Для этого нужно щелкнуть на маленькой кнопке с изображением шестерни справа от имени компонента, выбрать в появившемся меню команду Copy Component, выделить объект, в который его нужно скопировать, и воспользоваться командой Paste As New. Выполните эту операцию, но не назначайте подписчиков события, так как это уже сделано в коде сценария `InventoryPopup`.

На этом работа над UI для отображения инвентаря завершается! Запустите игру, чтобы посмотреть, что происходит со всплывающим окном, когда вы собираете инвентарь и щелкаете на кнопках. Мы закончили сборку фрагментов предыдущих проектов; теперь я объясню, как создать более масштабную игру с нуля.

11.2. Разработка общей игровой структуры

Теперь, когда у нас есть функционирующая демонстрационная версия ролевой игры, нужно создать для нее общую игровую структуру. Под этими словами я подразумеваю наличие различных уровней и прохождение игры путем поочередного прохождения каждого уровня. Проект из главы 8 состоял из одного уровня, в то время как в нашем плане работ фигурируют три уровня.

Для этого нам потребуется еще сильнее уменьшить связь сцены с отвечающими за ее обработку диспетчерами, в результате придется рассылать сообщения об их состояниях (точно так же, как диспетчер `PlayerManager` рассылает информацию об обновлениях состояния здоровья персонажа). Создайте новый сценарий `StartupEvent` (листинг 11.15); происходящее при загрузке мы выделяем в отдельный сценарий, так как эти события идут в связке с постоянно использующейся системой диспетчеров, в то время как класс `GameEvent` связан с конкретной игрой.

Листинг 11.15. Сценарий StartupEvent

```
public static class StartupEvent {
  public const string MANAGERS_STARTED = "MANAGERS_STARTED";
  public const string MANAGERS_PROGRESS = "MANAGERS_PROGRESS";
}
```

Теперь можно приступать к редактированию диспетчеров, добавляя к ним средства рассылки сообщений о новых событиях!

11.2.1. Управление ходом миссии и набором уровней

Пока что проект состоит из одной сцены, в которой в числе прочего находится главный диспетчер. Проблема в том, что у каждой сцены будет собственный набор игровых диспетчеров, в то время как нам требуется один набор, доступный всем сценам.

Для этого мы создадим отдельную сцену Startup, которая будет инициализировать диспетчеры, а затем давать к ним доступ остальным игровым сценам.

Также нам нужен новый диспетчер, обрабатывающий процесс прохождения игры. Создайте сценарий `MissionManager` и скопируйте в него содержимое следующего листинга.

Листинг 11.16. Сценарий MissionManager

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

public class MissionManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  public int curLevel {get; private set;}
  public int maxLevel {get; private set;}

  private NetworkService _network;

  public void Startup(NetworkService service) {
    Debug.Log("Mission manager starting...");

    _network = service;

    curLevel = 0;
    maxLevel = 1;

    status = ManagerStatus.Started;
  }

  public void GoToNext() {
    if (curLevel < maxLevel) { ← Рассылаем аргументы вместе с объектом WWW, используя объект WWWForm.
      curLevel++;
      string name = "Level" + curLevel;
      Debug.Log("Loading " + name);
      Application.LoadLevel(name); ← Проверяем, достигнут ли последний уровень.
    } else {
      Debug.Log("Last level");
    }
  }
}
```

По большей части ничего необычного в этом листинге не происходит. Обратить внимание имеет смысл на расположенный в конце метод `LoadLevel()`; я уже упоминал про него в главе 5, но до текущего момента он нас не особо интересовал. Этот метод служит в Unity для загрузки файла сцены; в главе 5 он использовался для перезагрузки одной и той же сцены, теперь же мы с его помощью начнем загружать произвольные сцены, передавая имена соответствующих файлов.

Свяжите этот сценарий с объектом Game Managers. Заодно добавьте к сценарию `Managers` новый компонент (см. следующий листинг).

Листинг 11.17. Добавление нового компонента в сценарий Managers

```
...
[RequireComponent(typeof(MissionManager))]

public class Managers : MonoBehaviour {
  public static PlayerManager Player {get; private set;}
  public static InventoryManager Inventory {get; private set;}
  public static MissionManager Mission {get; private set;}
  ...
  void Awake() {
    DontDestroyOnLoad(gameObject); ← Команда Unity для сохранения объекта между сценами.

    Player = GetComponent<PlayerManager>();
    Inventory = GetComponent<InventoryManager>();
    Mission = GetComponent<MissionManager>();

    _startSequence = new List<IGameManager>();
    _startSequence.Add(Player);
    _startSequence.Add(Inventory);
    _startSequence.Add(Mission);

    StartCoroutine(StartupManagers());
  }

  private IEnumerator StartupManagers() {
    ...
    if (numReady > lastReady) {
      Debug.Log("Progress: " + numReady + "/" + numModules);
      Messenger<int, int>.Broadcast(
      StartupEvent.MANAGERS_PROGRESS, numReady, numModules); ← Событие загрузки рассылается без параметров.
    }

    yield return null;
  }

  Debug.Log("All managers started up");
  Messenger.Broadcast(StartupEvent.MANAGERS_STARTED); ← Событие загрузки рассылается вместе с относящимися к нему данными.
}
...
```

Большая часть этого кода вам уже знакома (процесс добавления диспетчера `MissionManager` ничем не отличается от процесса добавления всех прочих диспетчеров), но появились и два новых фрагмента. Во-первых, событие, рассылаемое с двумя целочисленными значениями. Раньше вы видели обобщенные события без значений и сообщения с одним числом, но данный синтаксис позволяет рассылать произвольное количество численных значений.

Вторым новым фрагментом кода является метод `DontDestroyOnLoad()`. В Unity он обеспечивает сохранение объекта между сценами. Обычно при загрузке новой сцены все объекты удаляются, но метод `DontDestroyOnLoad()` позволяет перенести объект в следующую сцену.

Разделение сцен для загрузки и игрового уровня

Так как объект Game Managers существует во всех сценах, следует разделить диспетчеры и игровые уровни. На вкладке Project создайте копию файла сцены (Edit ▸ Duplicate) и присвойте двум файлам соответствующие имена: Startup и Level1. Откройте файл Level1 и удалите объект Game Managers (он будет предоставляться сценой Startup). Откройте сцену Startup и удалите все, кроме объектов Game Managers, Controller, HUD Canvas и EventSystem. Удалите сценарные компоненты у объекта Controller, а также объекты UI (метку health и объект InventoryPopup), которые являются потомками объекта Canvas.

Теперь у нас совершенно пустой UI, поэтому создайте новый ползунок, как показано на рис. 11.7, и сбросьте флажок Interactable. У объекта Controller теперь тоже отсутствуют компоненты сценария, поэтому создайте новый сценарий `StartupController` и присоедините его к объекту Controller (см. следующий листинг).

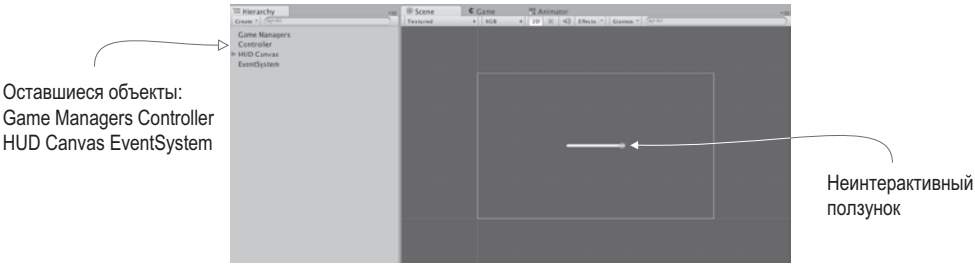


Рис. 11.7. Сцена Startup после удаления всего ненужного

Листинг 11.18. Сценарий StartupController

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

public class StartupController : MonoBehaviour {
  [SerializeField] private Slider progressBar;

  void Awake() {
    Messenger<int, int>.AddListener(StartupEvent.MANAGERS_PROGRESS,
      OnManagersProgress);
    Messenger.AddListener(StartupEvent.MANAGERS_STARTED,
      OnManagersStarted);
  }
  void OnDestroy() {
    Messenger<int, int>.RemoveListener(StartupEvent.MANAGERS_PROGRESS,
      OnManagersProgress);
    Messenger.RemoveListener(StartupEvent.MANAGERS_STARTED,
      OnManagersStarted);
  }

  private void OnManagersProgress(int numReady, int numModules) {
```

```
    float progress = (float)numReady / numModules;
    progressBar.value = progress; ← Обновляем ползунок данными о процессе загрузки.
  }

  private void OnManagersStarted() {
    Managers.Mission.GoToNext(); ← После загрузки диспетчеров загружаем следующую сцену.
  }
}
```

Теперь свяжите объект Slider с ячейкой на панели Inspector. Остался всего один подготовительный шаг. Нужно добавить две сцены в список Build Settings. Процедура создания приложения рассматривается в следующей главе, а пока просто выберите в меню File команду Build Settings для просмотра и корректировки списка сцен. Щелкните на кнопке Add Current, чтобы добавить в список текущую сцену (загрузите обе сцены и проделайте эту операцию для каждой из них).

ПРИМЕЧАНИЕ Добавление в список Build Settings необходимо для загрузки сцен. Без этого Unity не узнает, какие из сцен доступны. В главе 5 нам эта операция не требовалась, так как мы не переходили от одной сцены к другой, а только перезагружали текущую сцену.

Теперь можно загрузить игру, щелкнув на кнопке Play в сцене Startup. Объект Game Managers будет использоваться в обеих сценах.

ВНИМАНИЕ Так как диспетчеры загружаются в сцене Startup, именно отсюда все время нужно запускать игру. Можно запомнить, что перед щелчком на кнопке Play всегда требуется переходить к этой сцене, а можно воспользоваться сценарием http://wiki.unity3d.com/index.php/SceneAutoLoader, который будет автоматически перебрасывать вас в нужное состояние после щелчка на кнопке Play.

Это структурное изменение отвечает за совместное использование диспетчеров различными сценами, но на уровнях пока отсутствуют индикаторы успешного прохождения уровней и гибели персонажа.

11.2.2. Завершение уровня

Для завершения уровня в сцену нужно поместить объект, который в ответ на касание персонажа будет информировать диспетчер MissionManager об успешном достижении цели. Для этого нам потребуется пользовательский интерфейс, реагирующий на сообщение о завершении уровня, поэтому добавьте в сценарий `GameEvent` содержимое следующего листинга.

Листинг 11.19. Код завершения уровня, добавляемый в сценарий GameEvent

```
public static class GameEvent {
  public const string HEALTH_UPDATED = "HEALTH_UPDATED";
  public const string LEVEL_COMPLETE = "LEVEL_COMPLETE";
}
```

Теперь добавьте в сценарий `MissionManager` новый метод для слежения за целями миссий и рассылки новых сообщений о событиях (см. следующий листинг).

Листинг 11.20. Метод слежения за целями в сценарии MissionManager

```
...
public void ReachObjective() {
  // здесь может быть код обработки нескольких целей
  Messenger.Broadcast(GameEvent.LEVEL_COMPLETE);
}
...
```

Отредактируйте сценарий `UIController`, как показано в следующем листинге, чтобы он начал реагировать на данное событие.

Листинг 11.21. Новый подписчик на событие в сценарии UIController

```
...
[SerializeField] private Text levelEnding;
...
void Awake() {
  Messenger.AddListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
  Messenger.AddListener(GameEvent.LEVEL_COMPLETE, OnLevelComplete);
}
void OnDestroy() {
  Messenger.RemoveListener(GameEvent.HEALTH_UPDATED, OnHealthUpdated);
  Messenger.RemoveListener(GameEvent.LEVEL_COMPLETE, OnLevelComplete);
}
...
void Start() {
  OnHealthUpdated();

  levelEnding.gameObject.SetActive(false);
  popup.gameObject.SetActive(false);
}
...
private void OnLevelComplete() {
  StartCoroutine(CompleteLevel());
}
private IEnumerator CompleteLevel() {
  levelEnding.gameObject.SetActive(true);
  levelEnding.text = "Level Complete!";

  yield return new WaitForSeconds(2); ← Отображаем сообщение в течение двух секунд, а потом переходим на следующий уровень.

  Managers.Mission.GoToNext();
}
...
```

Обратите внимание, что в этом листинге есть ссылка на текстовую метку. Откройте сцену Level1 и создайте новый текстовый объект UI. Эта метка будет содержать сообщение о завершении уровня, появляющееся в центре экрана. Присвойте параметру Width этого текста значение 240, параметру Height — значение 60, для выравнивания по горизонтали и вертикали выберите вариант Center, а параметр Font Size сделайте равным 22. В текстовое поле введите слова Level Complete! и свяжите текстовый объект со ссылкой `levelEnding` в сценарии `UIController`.

Теперь осталось создать объект, прикосновение к которому позволит завершить уровень. Пример такого объекта показан на рис. 11.8. Он напоминает доступные для сбора элементы инвентаря: потребуется материал и сценарий, а затем мы превратим этот объект в шаблон экземпляра.

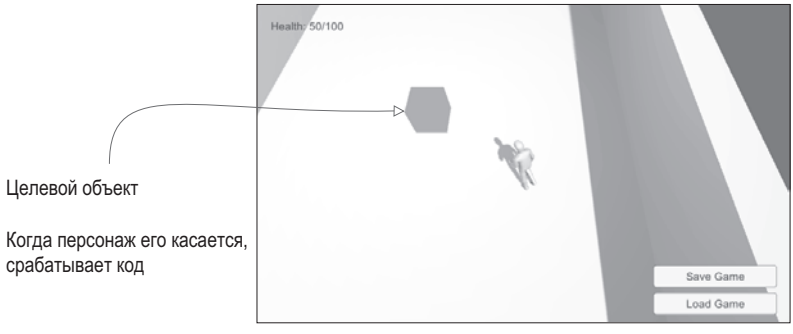


Рис. 11.8. Целевой объект, к которому должен прикоснуться персонаж для завершения уровня

Создайте куб и введите в поля Position значения 18, 1, 0. Установите флажок Is Trigger в свитке Box Collider, отключите возможность отбрасывать и получать тени в свитке Mesh Renderer и поместите объект в слой Ignore Raycast. Создайте новый материал с именем objective; сделайте его ярко-зеленым и выберите вариант раскраски Unlit ▸ Color для получения равномерного яркого представления.

Теперь создайте сценарий `ObjectiveTrigger` (показанный в следующем листинге) и свяжите его с целевым объектом.

Листинг 11.22. Код сценария ObjectiveTrigger для применения к целевым объектам

```
using UnityEngine;
using System.Collections;

public class ObjectiveTrigger : MonoBehaviour {
  void OnTriggerEnter(Collider other) {
    Managers.Mission.ReachObjective(); ← Вызываем новый целевой метод в сценарии MissionManager.
  }
}
```

Перетащите полученный объект со вкладки Hierarchy на вкладку Project, чтобы превратить его в шаблон экземпляра; на следующих уровнях его можно будет вставлять в сцены. Запустите игру и попробуйте добраться до целевого объекта. После этого появится сообщение о завершении уровня.

Теперь нужно создать сообщение, появляющееся в момент смерти персонажа.

11.2.3. Проигрыш уровня

Условием проигрыша является полная утрата здоровья персонажем (из-за успешных атак врагов). Первым делом добавьте еще одну строку в сценарий `GameEvent`:

```
public const string LEVEL_FAILED = "LEVEL_FAILED";
```

Далее отредактируйте сценарий `PlayerManager`, чтобы падение уровня здоровья до нуля сопровождалось рассылкой сообщения, как показано в следующем листинге.

Листинг 11.23. Рассылка сообщения о проигрыше из сценария PlayerManager

```
...
public void Startup(NetworkService service) {
  Debug.Log("Player manager starting...");

  _network = service;

  UpdateData(50, 100); ← Вызываем метод обновления вместо того, чтобы задавать переменные напрямую.

  status = ManagerStatus.Started;
}

public void UpdateData(int health, int maxHealth) {
  this.health = health;
  this.maxHealth = maxHealth;
}

public void ChangeHealth(int value) {
  health += value;
  if (health > maxHealth) {
    health = maxHealth;
  } else if (health < 0) {
    health = 0;
  }

  if (health == 0) {
    Messenger.Broadcast(GameEvent.LEVEL_FAILED);
  }
  Messenger.Broadcast(GameEvent.HEALTH_UPDATED);
}

public void Respawn() { ← Возвращаем игрока в исходное состояние.
  UpdateData(50, 100);
}
...
```

Для возвращения к началу уровня добавим небольшой метод в сценарий `MissionManager` (см. следующий листинг).

Листинг 11.24. Сценарий MissionManager, который может начинать текущий уровень сначала

```
...
public void RestartCurrent() {
  string name = "Level" + curLevel;
  Debug.Log("Loading " + name);
  Application.LoadLevel(name);
}
...
```

После этого добавьте еще одного подписчика на событие в сценарий `UIController`, как показано в следующем листинге.

Листинг 11.25. Реакция на гибель персонажа в сценарии UIController

```
...
Messenger.AddListener(GameEvent.LEVEL_FAILED, OnLevelFailed);
...
Messenger.RemoveListener(GameEvent.LEVEL_FAILED, OnLevelFailed);
...
private void OnLevelFailed() {
  StartCoroutine(FailLevel());
}
private IEnumerator FailLevel() {
  levelEnding.gameObject.SetActive(true);
  levelEnding.text = "Level Failed"; ← Используем ту же самую текстовую метку, но с другим сообщением.

  yield return new WaitForSeconds(2);

  Managers.Player.Respawn();
  Managers.Mission.RestartCurrent(); ← После двухсекундной паузы начинаем текущий уровень сначала.
}
...
```

Запустите игру и позвольте врагам сделать несколько выстрелов; появится сообщение о проигрыше. Прекрасная работа — теперь игрок может завершать уровни и начинать их заново в случае неудачного прохождения! Пришло время научить игру следить за успехами игрока.

11.3. Обработка хода игры

Пока что каждый уровень управляется независимо от других и без связи с игрой в целом. Требуется пара деталей, которые сделают процесс прохождения более завершенным: сохранение достижений игрока и определение факта окончания игры (а не только уровня).

11.3.1. Сохранение и загрузка достижений игрока

Процессы сохранения и загрузки являются важными частями большинства игр. Для этой цели в Unity и Mono добавлена функциональность ввода-вывода. Но перед тем как мы начнем с ней работать, следует добавить в сценарии `MissionManager` и `InventoryManager` метод `UpdateData()`. Он будет функционировать по тому же принципу, что и в сценарии `PlayerManager`, предоставляя внешнему по отношению к диспетчеру коду возможность обновлять данные внутри диспетчера. Отредактированные версии диспетчеров даны в листингах 11.26 и 11.27.

Листинг 11.26. Метод UpdateData() в сценарии MissionManager

```
...
public void Startup(NetworkService service) {
  Debug.Log("Mission manager starting...");

  _network = service;

  UpdateData(0, 1); ← Отредактируйте эту строку, используя новый метод.
  status = ManagerStatus.Started;
```

```
}

public void UpdateData(int curLevel, int maxLevel) {
  this.curLevel = curLevel;
  this.maxLevel = maxLevel;
}
...
```

Листинг 11.27. Метод UpdateData() в сценарии InventoryManager

```
...
public void Startup(NetworkService service) {
  Debug.Log("Inventory manager starting...");

  _network = service;

  UpdateData(new Dictionary<string, int>()); ← Инициализируем пустой список.

  status = ManagerStatus.Started;
}

public void UpdateData(Dictionary<string, int> items) {
  _items = items;
}

public Dictionary<string, int> GetData() { ← Необходима функция чтения для сохранения данных.
  return _items;
}
...
```

Теперь, когда метод `UpdateData()` появился в разных диспетчерах, новый модуль кода позволит нам сохранять данные. Для сохранения нам потребуется процедура, называемая *сериализацией* данных.

ОПРЕДЕЛЕНИЕ Сериализовать (serialize) означает перекодировать пакет данных в форму, допускающую сохранение.

Нашу игру мы сохраним в виде двоичных данных, но имейте в виду, что C# допускает также сохранение в виде текстовых файлов. Например, строки в формате JSON, с которыми вы работали в главе 9, представляли собой данные, сериализованные в виде текста. В предыдущих главах мы пользовались методом `PlayerPrefs`, но сейчас нам нужно сохранить локальный файл (метод `PlayerPrefs` предназначен для сохранения набора значений объемом до одного мегабайта). Создайте сценарий `DataManager` (см. следующий листинг).

ВНИМАНИЕ В интернет-играх отсутствует доступ к файловой системе. Это сделано в целях безопасности, то есть интернет-игры не могут сохранять свое состояние в виде локальных файлов. Сохранение данных в этом случае осуществляется путем их отправки на сервер.

Листинг 11.28. Сценарий для диспетчера данных

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

```
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

public class DataManager : MonoBehaviour, IGameManager {
  public ManagerStatus status {get; private set;}

  private string _filename;

  private NetworkService _network;

  public void Startup(NetworkService service) {
    Debug.Log("Data manager starting...");

    _network = service;

    _filename = Path.Combine(
        Application.persistentDataPath, "game.dat"); ← Генерируем полный путь к файлу game.dat.

    status = ManagerStatus.Started;
  }

  public void SaveGameState() {
    Dictionary<string, object> gamestate = new Dictionary<string,
      object>(); ← Словарь, который будет подвергнут сериализации.
    gamestate.Add("inventory", Managers.Inventory.GetData());
    gamestate.Add("health", Managers.Player.health);
    gamestate.Add("maxHealth", Managers.Player.maxHealth);
    gamestate.Add("curLevel", Managers.Mission.curLevel);
    gamestate.Add("maxLevel", Managers.Mission.maxLevel);

    FileStream stream = File.Create(_filename); ← Создаем файл по указанному адресу.
    BinaryFormatter formatter = new BinaryFormatter();
    formatter.Serialize(stream, gamestate); ← Сериализуем объект Dictionary как
    stream.Close();                           содержимое созданного файла.
  }

  public void LoadGameState() {
    if (!File.Exists(_filename)) { ← Переход к загрузке только при наличии файла.
      Debug.Log("No saved game");
      return;
    }
    Dictionary<string, object> gamestate; ← Словарь для размещения загруженных данных.

    BinaryFormatter formatter = new BinaryFormatter();
    FileStream stream = File.Open(_filename, FileMode.Open);
    gamestate = formatter.Deserialize(stream) as Dictionary<string,
      object>;
    stream.Close();

    Managers.Inventory.UpdateData((Dictionary<string,
      int>)gamestate["inventory"]); ← Обновляем диспетчеры, снабжая их десериализованными данными.
    Managers.Player.UpdateData((int)gamestate["health"],
      (int)gamestate["maxHealth"]);
    Managers.Mission.UpdateData((int)gamestate["curLevel"],
```

```
      (int)gamestate["maxLevel"]);
    Managers.Mission.RestartCurrent();
  }
}
```

В методе `Startup()` с помощью свойства `Application.persistentDataPath`, которое Unity предоставляет как место для сохранения файлов, генерируется путь к файлу. Конкретный путь к файлу на каждой платформе будет своим, но Unity рассматривает его отвлеченно с помощью статической переменной (кстати, в путь включаются данные из полей Company Name и Product Name, входящих в состав настроек Player Settings, поэтому, если нужно, отредактируйте эти переменные). Метод `File.Create()` создает двоичный файл; если вам нужен текстовый файл, воспользуйтесь методом `File.CreateText()`.

ВНИМАНИЕ При конструировании пути для разных платформ используются разные разделители. В C# это обстоятельство учитывается с помощью поля Path.DirectorySeparatorChar.

Откройте сцену Startup и найдите объект Game Managers. Добавьте к этому объекту в качестве компонента сценарий `DataManager`, а затем вставьте новый диспетчер в сценарий `Managers`, как показано в листинге 11.29.

Листинг 11.29. Добавление диспетчера DataManager в сценарий Managers.cs

```
...
[RequireComponent(typeof(DataManager))]
...
public static DataManager Data {get; private set;}
...
void Awake() {
  DontDestroyOnLoad(gameObject);

  Data = GetComponent<DataManager>();
  Player = GetComponent<PlayerManager>();
  Inventory = GetComponent<InventoryManager>();
  Mission = GetComponent<MissionManager>();

  _startSequence = new List<IGameManager>(); ← Диспетчеры запускаются по порядку.
  _startSequence.Add(Player);
  _startSequence.Add(Inventory);
  _startSequence.Add(Mission);
  _startSequence.Add(Data);

  StartCoroutine(StartupManagers());
}
...
```

ВНИМАНИЕ Так как диспетчер DataManager пользуется остальными диспетчерами (с целью их обновления), нужно сделать так, чтобы в загрузочной последовательности остальные диспетчеры фигурировали раньше.

Наконец, чтобы получить возможность пользоваться функциями в диспетчере `DataManager`, добавим кнопки, как показано на рис. 11.9. Создайте две кнопки, сделав их потомками объекта HUD Canvas (а не всплывающего окна Inventory).

Рис. 11.9. Кнопки сохранения и загрузки в нижней правой части экрана

Назовите их (указав имена в настройках присоединенных текстовых объектов) Save Game и Load Game, в окне Anchor Presets выберите вариант bottom-right и поместите их в точки с координатами –100, 65 и –100, 30 соответственно.

Эти кнопки нужно связать с функциями в сценарии `UIController`, поэтому воспользуйтесь методами из следующего листинга.

Листинг 11.30. Методы загрузки и сохранения в сценарии UIController

```
...
public void SaveGame() {
  Managers.Data.SaveGameState();
}

public void LoadGame() {
  Managers.Data.LoadGameState();
}
...
```

Свяжите эти функции с подписчиками на событие OnClick кнопок (добавьте слушание в настройки события OnClick, перетащите объект UIController и выберите функции в меню). Теперь запустите игру, соберите несколько элементов инвентаря, воспользуйтесь пакетом здоровья, чтобы увеличить эту характеристику, и сохраните игру. Заново запустите игру и проверьте свой инвентарь, чтобы удостовериться, что он отсутствует. Щелкните на кнопке Load; у вас появятся уровень здоровья и инвентарь, имеющиеся на момент сохранения игры!

11.3.2. Победа в игре при прохождении всех уровней

Сохранять достижения игрока имеет смысл в играх, состоящих из множества уровней, у нас же пока есть всего один, которым вы пользовались для тестирования. Для правильной обработки набора уровней требуется умение диагностировать завершение не только одного уровня, но и игры в целом. Первым делом добавьте в сценарий `GameEvent` еще одну строку:

```
public const string GAME_COMPLETE = "GAME_COMPLETE";
```

Теперь отредактируйте сценарий `MissionManager`, заставив его рассылать сообщение о прохождении последнего уровня (см. следующий листинг).

Листинг 11.31. Рассылка сообщения о завершении игры сценарием MissionManager

```
...
public void GoToNext() {
  ...
  } else {
    Debug.Log("Last level");
    Messenger.Broadcast(GameEvent.GAME_COMPLETE);
  }
}
```

На это сообщение должен реагировать сценарий `UIController` (см. следующий листинг).

Листинг 11.32. Добавление подписчика события в сценарий UIController

```
...
Messenger.AddListener(GameEvent.GAME_COMPLETE, OnGameComplete);
...
Messenger.RemoveListener(GameEvent.GAME_COMPLETE, OnGameComplete);
...
private void OnGameComplete() {
  levelEnding.gameObject.SetActive(true);
  levelEnding.text = "You Finished the Game!";
}
...
```

Пройдите уровень до конца (для этого персонаж должен коснуться целевого объекта) и посмотрите, что получится. Первым делом должно появиться сообщение Level Complete, которое через пару секунд сменяется сообщением о завершении игры.

Добавление уровней

Теперь можно добавить произвольное количество дополнительных уровней, а диспетчер `MissionManager` будет следить за последним уровнем. Это последнее, что нам осталось сделать в этой главе, ведь нам требовалось продемонстрировать, как выглядит прохождение многоуровневой игры.

Дважды продублируйте сцену Level1 (цифры в именах файлов при этом автоматически увеличатся до Level2 и Level3). Новые уровни следует добавить в список Build Settings, чтобы они могли загружаться в процессе игры. Отредактируйте каждую сцену, чтобы уровни отличались друг от друга; вы можете произвольно менять элементы сцен, сохраняя, однако, следующие объекты: объект player с тегом Player, объект floor, находящийся в слое Ground, целевой объект для выхода с уровня, объекты Controller, HUD Canvas и EventSystem.

СОВЕТ Карты освещенности при загрузке нового уровня система освещения по умолчанию генерирует заново. Но этот механизм работает только в процессе редактирования уровня; при загрузке уровней в игре карты освещенности не генерируются. Как вы делали в главе 9, можно сбросить флажок Continuous Baking в окне настроек (вызываемом командой Window ▸ Lighting) и щелкнуть на кнопке Build для фиксации карты освещенности (напоминаю, что вы не должны трогать появляющуюся при этом папку Scene).

Также нужно внести правки в сценарий `MissionManager`, чтобы он начал загружать новые уровни. Присвойте параметру `maxLevel` новое значение 3, заменив вызов метода `UpdateData(0, 1)` вызовом `UpdateData(0, 3)`.

Теперь запустите игру. Вы начнете с уровня Level1; коснитесь целевого объекта, и вы окажетесь на следующем уровне! Кстати, рекомендую вам выполнить сохранение на более высоком уровне, чтобы еще раз убедиться, что игра сохраняет данные о ваших достижениях.

УПРАЖНЕНИЕ: ВСТАВКА ЗВУКА В ПОЛНУЮ ВЕРСИЮ ИГРЫ

Глава 10 была целиком посвящена созданию звуковых эффектов в Unity. Процесс интеграции звука в проект отдельно не рассматривался, но к настоящему моменту вы уже должны понимать, как он реализован. Я призываю вас попробовать свои силы и самостоятельно добавить аудиофункциональность из предыдущей главы в наш проект. Подсказка: поменяйте клавишу, вызывающую окно с настройками звука, чтобы она не мешала пользоваться всплывающим окном с перечнем инвентаря.

Теперь вы умеете создавать игры с множеством уровней. Осталась нерешенной последняя задача, которую мы и рассмотрим в последней главе нашей книги: как предоставить пользователям доступ к игре.

11.4. Заключение

- Программа Unity позволяет легко менять предназначение ресурсов и кода из проектов в различных игровых жанрах.
- Еще одним замечательным применением метода испускания луча является определение точки, на которой щелкнул игрок.
- В Unity есть простые методы как для загрузки уровней, так и для сохранения определенных объектов при переходе с уровня на уровень.
- Переход с уровня на уровень осуществляется в ответ на различные события в игре.
- Вы можете пользоваться методами ввода-вывода из языка C# для сохранения данных в поле `Application.persistentDataPath`.

12 Развертывание игр на устройствах игроков

- ✓ Создание пакетов прикладных программ для различных платформ
- ✓ Задание параметров сборки, таких как значок или имя приложения
- ✓ Взаимодействие с веб-страницей при использовании веб-игр
- ✓ Разработка подключаемых модулей для приложений на мобильных платформах

До этого момента мы рассматривали исключительно процесс программирования различных игр в Unity, но без внимания остался важный заключительный шаг: доставка этих игр пользователям. Пока игра доступна лишь в редакторе Unity, она представляет интерес разве что для своего разработчика. На последнем этапе Unity выступает во всем своем блеске, позволяя строить приложения для огромного количества игровых платформ. Именно этой теме и посвящена наша последняя глава.

Под словосочетанием «создание для платформы» я подразумеваю генерацию запускаемого на этой платформе прикладного пакета. На каждой платформе (Windows, iOS и т. п.) своя форма пакета, но как только вы сгенерировали исполняемый файл, появляется возможность распространять игру и играть в нее без привязки к Unity. Один проект Unity можно развернуть на разных платформах — его не нужно каждый раз генерировать заново.

Принцип «построй один раз и развертывай где угодно» применим к подавляющему большинству игровых функций, но, к сожалению, не ко всем. По моим оценкам, 95 % написанного в Unity кода (в частности, практически все, что мы делали в этой книге) не имеет привязки к платформе и прекрасно работает везде. Но кое-какие вещи зависят от выбранной платформы, и мы рассмотрим их достаточно подробно.

В целом базовая бесплатная версия Unity позволяет создавать приложения для следующих платформ:

- ❍ Windows PC;
- ❍ Mac OS X;

- Linux;
- Web (как для веб-проигрывателя, так и для программной библиотеки WebGL);
- iOS;
- Android;
- Blackberry 10.

Кроме того, с помощью особых лицензированных модулей Unity позволяет строить приложения для

- XBox 360;
- XBox One;
- PlayStation 3;
- PlayStation 4;
- PS Vita;
- Wii U;
- Windows Phone 8.

Видите, какой длинный список! Положа руку на сердце — он потрясающе длинный, намного длиннее списка поддерживаемых платформ практически любого другого инструмента разработки игр. Подробно мы рассмотрим шесть вариантов из этого списка, и это будут платформы, представляющие интерес для подавляющего большинства пользователей Unity, но не забывайте, что список доступных вариантов намного больше.

Посмотреть все варианты платформ можно в окне Build Settings. Вы пользовались им в предыдущей главе, добавляя загружаемые сцены; для доступа к нему выберите в меню File команду Build Settings. В главе 11 нас интересовал только список в верхней части, теперь же мы обратим внимание и на расположенные внизу кнопки (рис. 12.1). Много места занимает список платформ; активные в настоящий момент платформы отмечены значком Unity. Достаточно выделить платформы в этом списке и щелкнуть на кнопке Switch Platform.

ВНИМАНИЕ В крупных проектах переход на другую платформу часто занимает долгое время; приготовьтесь к ожиданию. Это связано с тем, что Unity оптимальным для каждой платформы способом выполняет повторное сжатие всех ресурсов (таких, как текстуры).

В нижней части этого окна находятся кнопки Player Settings и Build. Щелчок на кнопке Player Settings открывает настройки приложения на панели Inspector, к числу которых относятся имя и значок приложения.

СОВЕТ Кнопка Build And Run отличается от кнопки Build тем, что вдобавок автоматически запускает сгенерированное приложение. Я обычно предпочитаю делать это вручную, соответственно, кнопкой Build And Run практически не пользуюсь.

После щелчка на кнопке Build первым делом открывается окно выбора файла, в котором нужно указать адрес для генерации пакета приложения. Сразу после указания

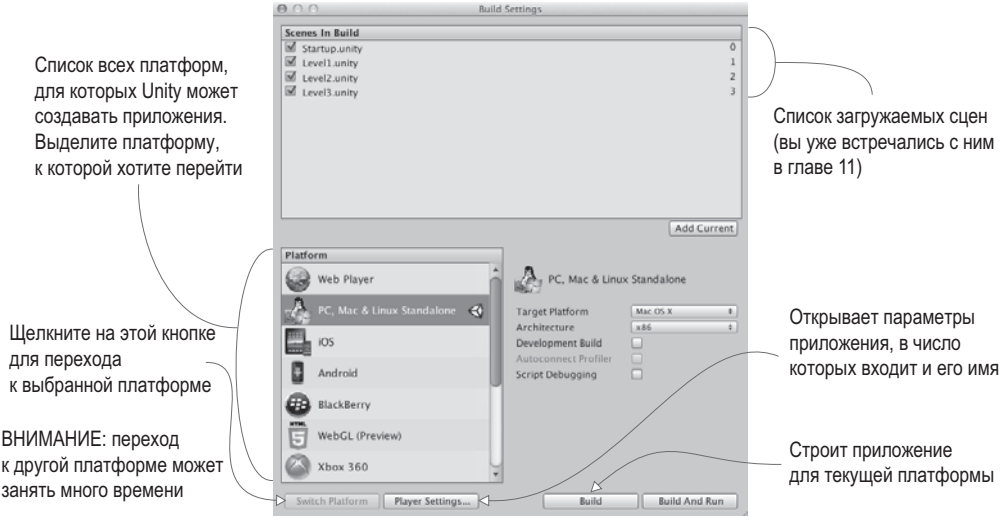


Рис. 12.1. Окно Build Settings

местоположения файла начинается процесс построения, после чего Unity создает исполняемый файл для активной в данный момент платформы; давайте рассмотрим этот процесс для наиболее популярных платформ: настольных компьютеров, Интернета и мобильных устройств.

12.1. Создание приложений для настольных компьютеров: Windows, Mac и Linux

Если вы только приступаете к знакомству с вопросом создания игр в Unity, проще всего начать с развертывания игры на настольных компьютерах под управлением Windows PC, Mac OS X или Linux. Unity работает на настольных компьютерах, соответственно, приложение будет генерироваться для той машины, которой вы уже пользуетесь.

ПРИМЕЧАНИЕ Для упражнений этой главы можно открыть любой проект по вашему вкусу. Я уверяю, что все будет работать; более того, рекомендую для каждого следующего раздела открывать новый проект, чтобы убедиться в способности Unity развертывать любой проект на любой платформе!

12.1.1. Построение приложения

Первым делом выберите в меню File команду Build Settings, чтобы открыть одноименное окно. По умолчанию выбран вариант PC, Mac, and Linux, но если вас интересует другой вариант, укажите его в списке и щелкните на кнопке Switch Platform.

С правой стороны находится раскрывающийся список Target Platform. Он позволяет выбрать между платформами Windows PC, Mac OS X и Linux. В списке слева эти варианты рассматривались как один, но на самом деле это разные платформы, поэтому выберите нужную.

После этого остается щелкнуть на кнопке Build. Откроется окно диалога, в котором нужно будет выбрать местоположение будущего приложения. По умолчанию его предлагается сохранить в Unity-проекте, что не является лучшим вариантом для сборок, поэтому имеет смысл указать более безопасное место. После этого начнется процесс построения, для крупных проектов занимающий изрядное время, но для наших крохотных демонстрационных проектов он должен завершиться достаточно быстро.

СОБСТВЕННЫЙ СЦЕНАРИЙ ПОСТ-СБОРКИ

Базовый процесс построения превосходно подходит для большинства ситуаций, но некоторые предпочитают при построении каждой игры совершать дополнительные действия (например, перемещать справочные файлы в папку, в которой находится приложение). Такие задачи можно легко автоматизировать, поместив их в сценарий, запускаемый после завершения процесса сборки.

Первым делом создайте новую папку на вкладке Project и присвойте ей имя Editor; именно здесь должны находиться все сценарии, влияющие на редактор Unity (это касается и процесса сборки). Создайте в этой папке сценарий с именем TestPostBuild и введите в него следующий код:

```
using UnityEngine;
using UnityEditor;
using UnityEditor.Callbacks;

public static class TestPostBuild {

  [PostProcessBuild]
  public static void OnPostprocessBuild(BuildTarget target, string
        pathToBuiltProject) {
    Debug.Log("build location: " + pathToBuiltProject);
  }
}
```

Директива [PostProcessBuild] заставляет сценарий запускать расположенную непосредственно за ней функцию. Эта функция получит местоположение построенного приложения; после этого вы сможете использовать данный параметр в различных командах для работы с файловой системой из языка C#. В настоящее время путь к файлу выводится на консоль, что позволяет удостовериться в работоспособности сценария.

Приложение появится в указанной вами папке; запустите его двойным щелчком, как любую другую программу. Мои поздравления! Как видите, это было несложно. Процесс построения приложений совершенно тривиален, но допускает различные варианты настройки; посмотрим, как мы можем его доработать.

12.1.2. Настройки проигрывателя: имя и значок приложения

Вернемся в окно Build Settings, но на этот раз щелкнем не на кнопке Build, а на кнопке Player Settings. На панели Inspector появится список настроек, показанный на рис. 12.2; они контролируют различные аспекты готового приложения.

Так как настроек в данном случае много, лучше почитать о них в руководстве пользователя — вот адрес страницы: http://docs.unity3d.com/ru/current/Manual/class-PlayerSettings.html.

Смысл трех верхних параметров очевиден: Company Name (имя компании), Product Name (имя продукта) и Default Icon (значок, предлагаемый по умолчанию). Укажите

значение первых двух. В поле Company Name — название студии-разработчика, а в поле Product Name — название игры. Затем задайте значок игры, перетащив нужное изображение с вкладки Project (при необходимости сначала импортируйте его в проект); когда приложение будет построено, это изображение появится как его значок.

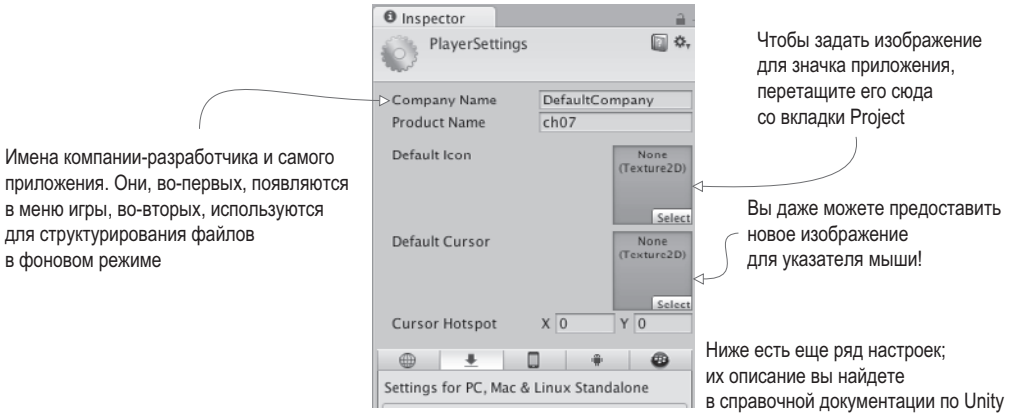
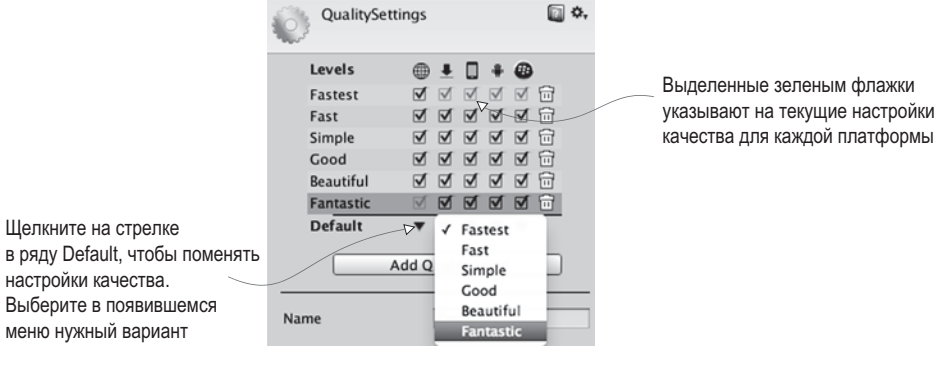


Рис. 12.2. Настройки проигрывателя на панели Inspector

НАСТРОЙКИ КАЧЕСТВА

На генерируемое приложение также влияют настройки проекта, доступ к которым осуществляется через меню Edit. В частности, именно тут настраивается визуальное качество готового приложения. Выберите в меню Edit команду Project Settings, а затем в дополнительном меню — команду Quality.

На панели Inspector появятся элементы управления качеством, наиболее важными из которых являются флажки в расположенной сверху группе. В верхнем ряду находятся значки возможных платформ, а сбоку указаны варианты настроек качества. Установленные флажки показывают доступные для данной платформы настройки, а выделенные зеленым — текущие настройки. В большинстве случаев по умолчанию применяется вариант Fastest (соответствующий минимальному качеству), но если все выглядит плохо, можно выбрать вариант Fantastic; щелчок на стрелке в нижнем ряду под нужным значком платформы открывает меню.



Сетка настроек качества на панели Inspector

Одновременное наличие в интерфейсе флажков и меню Default может показаться избыточным, но на самом деле это не так. Разные платформы зачастую обладают разными функциональными возможностями графических средств, поэтому Unity позволяет индивидуально задавать уровни качества для целевых платформ (например, самое высокое качество для настольных компьютеров и минимальное качество для мобильных устройств).

Значок и имя приложения придают результатам вашего труда ощущение законченности. Другим вариантом настройки поведения приложений является зависящий от платформы код.

12.1.3. Компиляция в зависимости от платформы

По умолчанию весь написанный вами код запускается на всех платформах одним и тем же способом. Но Unity предоставляет нам ряд директив компилятора (известных как *определения платформ*), которые заставляют код работать исключительно на указанной платформе. Полный список определений платформ вы найдете на странице http://docs.unity3d.com/ru/current/Manual/PlatformDependentCompilation.html.

Директивы существуют для всех поддерживаемых Unity платформ, соответственно, на каждой из них вы можете запускать свою версию кода. Обычно большую часть кода помещать внутрь директив не требуется, но отдельные фрагменты иногда имеет смысл запускать с привязкой к конкретной платформе. Некоторые варианты сборки существуют исключительно на одной платформе (например, в главе 11 упоминалось, что доступ к файловой системе в веб-проигрывателях отсутствует), поэтому требуются директивы компилятора, позволяющие обойти это ограничение. Следующий листинг демонстрирует пример написания такого кода.

Листинг 12.1. Сценарий PlatformTest с примером кода, привязанного к платформе

```
using UnityEngine;
using System.Collections;

public class PlatformTest : MonoBehaviour {
  void OnGUI() {
    #if UNITY_EDITOR ← Этот раздел запускается только в редакторе.
      GUI.Label(new Rect(10, 10, 200, 20), "Running in Editor");
    #elif UNITY_STANDALONE ← Только в приложениях для рабочего стола/автономных приложениях.
      GUI.Label(new Rect(10, 10, 200, 20), "Running on Desktop");
    #else
      GUI.Label(new Rect(10, 10, 200, 20), "Running on other platform");
    #endif
  }
}
```

Создайте сценарий `PlatformTest` и скопируйте в него код этого листинга. Свяжите его с произвольным объектом сцены (для тестирования подойдет любой объект), и в верхней левой части экрана появится маленькое сообщение. При воспроизведении игры в редакторе Unity это будет сообщение Running in the Editor (выполняется в редакторе), но если вы сгенерируете приложение и запустите его, появится уже другой текст: Running on Desktop (выполняется на настольном компьютере). В каждом случае запускается свой вариант кода!

Для тестирования мы воспользовались определением платформы, воспринимающим все платформы на базе настольных компьютеров как одну, но, как можно прочитать в документации, существуют отдельные определения платформ для Windows, Mac и Linux. Более того, отдельные определения существуют для всех поддерживаемых в Unity платформ, и для каждой из них вы можете создать свой вариант кода. На этом мы перейдем к рассмотрению следующей важной платформы: Интернета.

12.2. Создание игр для Интернета

Хотя настольные компьютеры являются основной целевой платформой для создаваемых с помощью Unity игр, существуют и другие важные варианты развертывания, например развертывание в Интернете. Оно требуется играм, которые запускаются в веб-браузерх и, соответственно, доступны для игроков в Интернете.

12.2.1. Проигрыватель Unity и HTML5/WebGL

Раньше создаваемые в Unity варианты программ для Интернета воспроизводились с помощью специальных подключаемых модулей браузера. Это было связано с отсутствием встроенных в браузер средств отображения трехмерной графики. Но в последние годы начал развиваться стандарт WebGL. С технической точки зрения он отличается от HTML5, но эти термины связаны друг с другом и часто используются как синонимы.

В Unity 5 в список платформ в окне Build Settings был добавлен вариант WebGL, а в следующих версиях он может превратиться в основной вариант создания приложений для Интернета. Именно это направление развития было выбрано в компании. Оно обусловлено в числе прочего давлением со стороны производителей браузеров, которые в реализации интерактивных веб-приложений, в том числе игр, предпочитают отходить от дополнительных подключаемых модулей в пользу HTML5/WebGL.

Впрочем, независимо от формы итогового приложения, процесс его генерации для веб-проигрывателя и для WebGL практически идентичен. В следующих разделах вы найдете описание процедуры для веб-проигрывателя с упоминаниями, чем именно отличается код, предназначенный для WebGL.

12.2.2. Создание файла Unity и тестовой веб-страницы

Откройте какой-нибудь другой проект (чтобы убедиться, что работать можно с любым проектом) и вызовите окно Build Settings. Выделите строку Web Player и щелкните на кнопке Build. Появится окно выбора файла; введите для данного приложения имя WebTest и при необходимости выберите для него более безопасное место (вне проекта Unity).

На этот раз у вас появятся два файла: сама игра Unity с расширением .unity3d и пустая веб-страница для воспроизведения игры. Откройте эту страницу, и в центре вы увидите встроенную игру.

Никакими особыми характеристиками страница не обладает; это всего лишь пример для тестирования игры. На ней можно настроить код и даже предоставить собственную версию веб-страницы (поверх которой скопирован код Unity). Одним из

наиболее важных вариантов адаптации является добавление возможности взаимодействия Unity с браузером. Давайте посмотрим, каким образом это делается.

12.2.3. Обмен данными с JavaScript в браузере

Созданная в Unity онлайновая игра может обмениваться данными с браузером (точнее, с запущенным в браузере сценарием JavaScript), причем обмен может идти в обоих направлениях — как от Unity к браузеру, так и от браузера к Unity. В первом случае все тривиально: в Unity есть пара специальных команд, напрямую запускающих код в браузере.

В обратной ситуации все несколько сложнее: код JavaScript в браузере идентифицирует объект по имени, после чего Unity передает этому именованному объекту сцены сообщение. То есть в сцене должен присутствовать объект, который будет получать данные от браузера.

Чтобы посмотреть, как все это выглядит на практике, создайте новый сценарий с именем `WebTestObject`. Кроме того, создайте в активной сцене пустой объект с именем Listener (ему следует присвоить именно это имя, потому что оно фигурирует в коде). Свяжите с этим объектом новый сценарий и скопируйте в него код из следующего листинга.

Листинг 12.2. Сценарий WebTestObject для тестирования механизма обмена данными с браузером

```
using UnityEngine;
using System.Collections;

public class WebTestObject : MonoBehaviour {
  private string _message;

  void Start() {
    _message = "No message yet"; ← Исходное значение для сообщения.
  }

  void Update() {
    if (Input.GetMouseButtonDown(0)) { ← Щелчок мыши вызывает функцию в браузере.
      Application.ExternalCall("ShowAlert", "Hello out there!");
    }
  }

  void OnGUI() {
    GUI.Label(new Rect(10, 10, 200, 20), _message); ← Отображаем сообщение в верхнем левом углу экрана.
  }

  public void RespondToBrowser(string message) { ← Функция, вызываемая браузером.
    _message = message;
  }
}
```

Теперь еще раз сгенерируйте приложение для Интернета, чтобы добавить в игру новый код. После этого останется только отредактировать веб-страницу. Нужно вставить пару функций в JavaScript-код и кнопку в HTML-код. Добавьте JavaScript-код

и HTML-тег в следующий листинг; JavaScript-функции вставляются перед закрывающим тегом `<script>`, а кнопка — перед закрывающим тегом `<body>`.

Листинг 12.3. Фрагменты JavaScript- и HTML-кода, поддерживающие обмен данными между браузером и Unity

```
...
function ShowAlert(arg) {
  alert(arg); ← Выводим окно оповещения.
}
function SendToUnity() { ← SendMessage() вызывает функцию внутри Unity.
  u.getUnity().SendMessage("Listener", "RespondToBrowser", "Hello from the
      browser!");
}
-->
</script>
...
<input type="button" value="Send to Unity" onclick="SendToUnity();" /> #C ← Кнопка, вызывающая функцию JavaScript.
</body>
</html>
```

Чтобы посмотреть, как работает этот код, откройте веб-страницу. При пересылке данных от Unity в браузер при щелчке в редакторе Unity сценарий `WebTestObject` вызывает функцию в браузере; сделайте несколько щелчков, и вы увидите, как в браузере появляется окно оповещения. Метод `Application.ExternalCall()` запускает именованную функцию JavaScript. Кроме того, в Unity есть метод `Application.ExternalEval()`, отвечающий за отправку сообщений браузеру; в нашей ситуации вместо вызова конкретной функции запускаются произвольно выбранные фрагменты JavaScript-кода. В большинстве случаев лучше вызывать функции (чтобы JavaScript- и Unity-фрагменты оставались независимыми друг от друга), но иногда имеет смысл запуск произвольно выбранных фрагментов, таких как вот этот код перезагрузки страницы:

```
Application.ExternalEval("location.reload();");
```

Код JavaScript на веб-странице также может посылать сообщения в Unity; достаточно щелкнуть на кнопке, которую мы добавили на страницу, и в Unity появится измененное сообщение. HTML-код кнопки ссылается на функцию JavaScript, которая, в свою очередь, вызывает метод `SendMessage()` в экземпляре Unity, вызывающий именованную функцию у именованного объекта в Unity; первый параметр представляет собой имя объекта, второй — имя метода, третий — строку, которая передается в метод. Листинг 12.3 вызывает метод `RespondToBrowser()` из сценария `WebTestObject`.

ПРИМЕЧАНИЕ WebGL-сборки также могут обмениваться данными с JavaScript-кодом веб-страниц. Код этого взаимодействия практически идентичен показанному. Точнее, в случае отправки сообщения из Unity он совпадает, а вот в обратном направлении — со страницы в Unity — метод SendMessage(), сохранив тот же самый набор параметров, более не требует префикса u.getUnity().

Итак, мы разобрались, как осуществляется взаимодействие с браузером в случае сборок для Интернета; осталось познакомиться с последней платформой (точнее, с набором платформ) — мобильными приложениями.

12.3. Сборки для мобильных устройств: iOS и Android

Мобильные устройства являются еще одной важной целевой платформой. По моим впечатлениям (впрочем, научно не подтвержденным), больше всего коммерческих игр в Unity создается именно в виде приложений для мобильных устройств.

ОПРЕДЕЛЕНИЕ Мобильным называется портативное устройство, которое пользователи носят с собой. Изначально этим термином называли смартфоны, но потом в эту группу попали еще и планшеты. Двумя наиболее распространенными компьютерными платформами для таких устройств являются iOS (от Apple) и Android (от Google).

Настройка процесса сборки для мобильных устройств сложнее, чем для настольных компьютеров и Интернета, поэтому данный раздел относится к необязательным — можете прочитать его с ознакомительными целями, не выполняя упражнений; упражнения могут делать только те читатели, кто приобрел лицензию разработчика для iOS и установил инструменты разработчика для Android.

ВНИМАНИЕ Мобильные устройства настолько часто обновляются, что на момент чтения данной книги процесс сборки может отличаться от описываемого. Высокоуровневые концепции, скорее всего, останутся теми же, но точное описание последовательности выполняемых команд и нажимаемых кнопок вам придется искать в выложенной в Интернете документации. Вот ссылки на страницы с документацией от Apple и Google соответственно: https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/Introduction/Introduction.html, http://developer.android.com/tools/building/index.html.

СЕНСОРНЫЙ ВВОД

Ввод информации в случае мобильных устройств совсем не такой, как при работе с настольным компьютером или в Интернете. Он осуществляется посредством прикосновений к экрану, а не с помощью мыши и клавиатуры. Соответственно, в Unity поддерживается функциональность обработки касаний, в том числе такой код, как Input.touchCount и Input.GetTouch().

Эти команды используются при написании привязанного к платформе кода на мобильных устройствах. Но такой способ обработки ввода неудобен, поэтому существует ряд упрощающих ввод фреймворков. Лично я использую FingerGestures (http://fingergestures.fatalfrog.com/).

Теперь, после всех этих оговорок, можно перейти к объяснению общего процесса сборки для iOS и Android. Еще раз напоминаю, что эти платформы время от времени меняют детали этого процесса.

12.3.1. Настройка инструментов сборки

Мобильные устройства существуют отдельно от компьютера, на котором разрабатывается игра, и именно этот факт несколько осложняет процесс генерации и развертывания приложений. Вам потребуется настроить множество специализированных устройств, и только потом вы получите возможность щелкнуть на кнопке Build.

Настройка инструментов сборки для ios

На высоком уровне процесс развертывания созданной в Unity игры на платформе iOS требует построения Xcode-проекта, а затем его превращения в IPA (iOS App

Package — пакет приложения для iOS). Создать IPA сразу Unity не может, так как все iOS-приложения должны проходить через инструменты сборки от Apple. Это означает, что вам нужно установить Xcode (программную интегрированную среду разработки от Apple), включая iOS SDK.

ВНИМАНИЕ Все это означает, что вы должны работать на компьютере Mac, так как Xcode запускается только в OS X. Разработка игры может выполняться на компьютере с Windows или Mac, а вот сборка iOS-приложения возможна только на машинах Mac.

Скачайте Xcode с сайта Apple со страницы https://developer.apple.com/xcode/downloads/.

ПРИМЕЧАНИЕ Для продажи вашей игры для iOS в App Store вам потребуется членство в программе Apple-разработчиков. Его стоимость составляет 99 долларов в год; подписка осуществляется на странице https://developer.apple.com/programs/.

После установки Xcode вернитесь в Unity и переключитесь на платформу iOS. Вам потребуется скорректировать настройки проигрывателя для iOS-приложения (напоминаю, что для доступа к ним нужно открыть окно Build Settings и щелкнуть на кнопке Player Settings). В настройках проигрывателя вы должны сразу попасть на вкладку iOS, но если это не так, перейдите на нее, щелкнув на значке iPhone. Найдите внизу раздел Other Settings, а в нем — параметр Identification. Нужно скорректировать строку Bundle Identifier, чтобы Apple-устройство смогло правильно идентифицировать приложение.

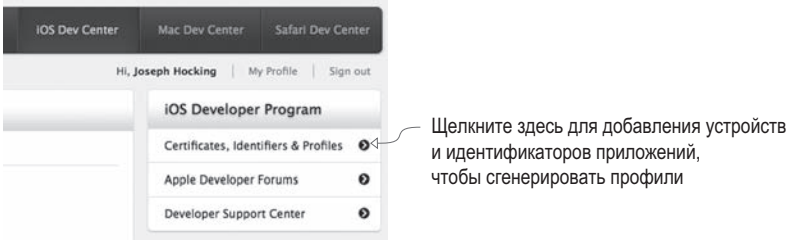
ПРИМЕЧАНИЕ Параметр Bundle Identifier для iOS и Android применяется одним и тем же способом, соответственно, он важен для обеих платформ. Идентификатор в данном случае должен составляться по правилам, применимым для любого другого пакета кода: буквами нижнего регистра в форме com.названиекомпании.названиепродукта.

Другим важным параметром, существующим как для iOS, так и для Android, является Bundle Version (то есть номер версии приложения). Но его форма в большинстве случаев зависит от платформы; к примеру, недавно в iOS был добавлен короткий номер версии, который видим игрокам и отделен от основной версии пакета. Есть еще параметр Scripting Backend, которому раньше по умолчанию присваивался вариант Mono, но новая технология IL2CPP поддерживает обновления платформ, например 64-битные двоичные файлы.

Теперь щелкните на кнопке Build. Выберите место для нового файла сборки, и там появится сгенерированный Xcode-проект. Он допускает прямое редактирование (некоторые простые модификации могут быть частью сценария пост-сборки). В любом случае, откройте этот проект; в папке со сборкой находится множество файлов, вам же нужно дважды щелкнуть на файле с расширением .xcodeproj (он помечен значком с изображением чертежа). Интегрированная среда разработки Xcode загрузит это приложение; большая часть настройки данного проекта уже выполнена в Unity, вам же остается настроить профиль, который будет использоваться в дальнейшем.

ПРОФИЛИ IOS

Из всех аспектов разработки iOS самым необычным и часто меняющимся являются профили (provisioning profiles). Коротко говоря, это файлы, используемые для идентификации и авторизации. Apple строго контролирует, какие приложения могут запускаться на каждом устройстве; отправленные в Apple на утверждение приложения пользуются специальными профилями, позволяющими добавлять их в App Store, в то время как находящиеся на стадии разработки приложения имеют профили, связанные с зарегистрированными устройствами.



Место управления профилями в центре разработчиков iOS

Вам нужно добавить UDID своего устройства iPhone (это связанный с вашим устройством идентификатор) и ID приложения (это параметр Bundle Identifier в Unity) на панель управления, расположенную на сайте Apple для разработчиков iOS. Полностью этот процесс объясняется на странице https://developer.apple.com/devcenter/ios/index.action.

Выберите свое приложение в списке проектов с левой стороны интегрированной среды Xcode-разработки. Появятся несколько относящихся к этому проекту вкладок; на вкладке Build Settings найдите раздел Code Signing, чтобы настроить профили, как показано на рис. 12.3. Заодно убедитесь, что параметр Scheme Destination на верхней панели окна не указывает на симулятор, а имеет значение iOS Device (в противном случае часть настроек окажется недоступной).

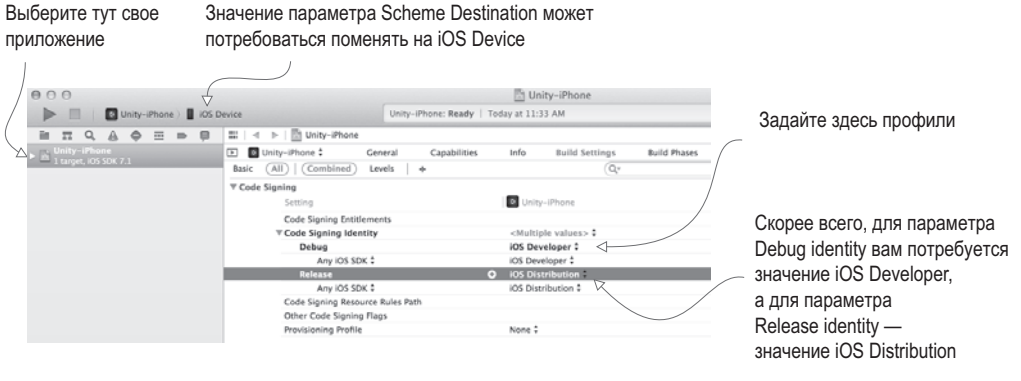


Рис. 12.3. Настройка профилей в интегрированной среде Xcode-разработки

После настройки профилей можно приступать к созданию приложения. Выберите в меню Product команду Run или Archive. Это меню содержит множество команд, в том числе имеющую соблазнительный вид команду Build, но для наших целей требуется

только упомянутая команда Run или Archive. Команда Build генерирует исполняемые файлы, но не собирает их в пакет для iOS, в то время как

- команда Run тестирует приложение на устройстве iPhone, связанном с компьютером при помощи USB-кабеля;
- команда Archive создает прикладной пакет, который можно пересылать на другие зарегистрированные устройства (у Apple это называется «специальной сборкой»).

Команда Archive создает не готовый прикладной пакет, а, скорее, комплект в промежуточном состоянии между необработанными файлами кода и IPA-файлом. Готовый архив появляется в окне Organizer интегрированной среды Xcode-разработки; чтобы получить из такого архива IPA-файл, щелкните на кнопке Distribute, как показано на рис. 12.4. После этого вам нужно будет указать, каким образом следует распространять приложение — через магазин или в виде специальной сборки.

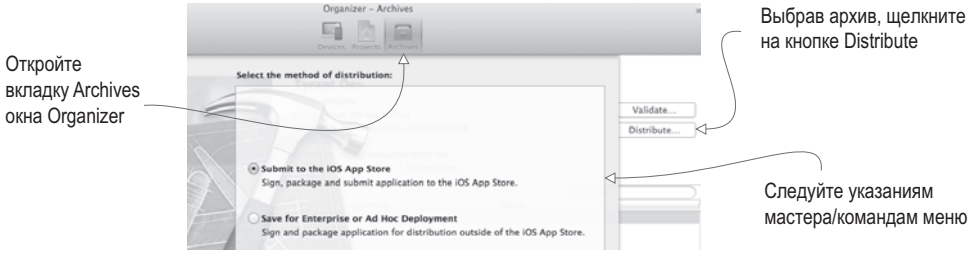


Рис. 12.4. Распространение архивированных приложений iOS через окно Organizer

Выбрав распространение в виде специальной сборки, вы получите IPA-файл, который можно будет отправить тестерам. Можно сделать это напрямую, передав им файл для установки через iTunes, но удобнее воспользоваться сервисом TestFlight (https://developer.apple.com/testflight/).

Настройка инструментов сборки для Android

В отличие от приложений iOS, файлы формата APK (Android Application Packge) Unity может генерировать напрямую. Для этого нужно добавить в Unity путь к Android SDK, включающий в себя нужный компилятор. Скачайте Android SDK с сайта Android и укажите путь к этому файлу в окне Unity Preferences, как показано на рис. 12.5. Скачать SDK можно здесь: http://developer.android.com/sdk/index.html.

После этого нужно задать параметр Bundle Identifier, как вы уже делали для iOS. Эта операция осуществляется в настройках проигрывателя на панели Inspector: укажите идентификатор в виде com.названиекомпании.названиепродукта (как объяснялось выше). Затем щелкните на кнопке Build, чтобы запустить процесс сборки. Как и в случае с другими сборками, вас первым делом попросят указать, где вы хотите сохранить файл. В указанном месте будет создан APK-файл.

Полученный пакет приложения следует установить на устройство. Получить APK-файл на телефон Android можно, скачав его из Интернета или посредством USB-подключения устройства к компьютеру (так называемый *режим sideload*). Конкрет-

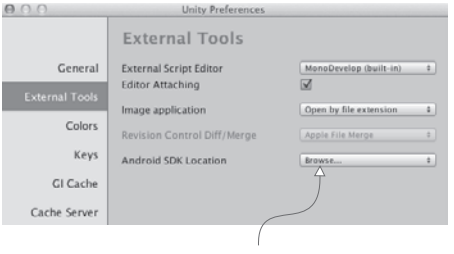


Рис. 12.5. Задание пути к Android SDK в окне Unity Preferences

ная процедура передачи данных зависит от устройства, а после ее завершения приложение можно установить с помощью диспетчера файлов. По какой-то причине в устройства Android диспетчеры файлов не встраиваются, но их можно бесплатно скачать из магазина приложений Play Store.

Найдите в диспетчере файлов свой APK-файл и установите приложение. Как видите, в своей основе процесс сборки для платформы Android намного проще, чем для платформы iOS. К сожалению, с процессами настройки сборки и добавления внешних модулей ситуация ровно обратная. В данном случае все намного сложнее, чем для iOS. Подробности вы узнаете в разделе 12.3.3, а пока давайте поговорим о сжатии текстур.

12.3.2. Сжатие текстур

Ресурсы, в том числе и текстуры, могут сильно увеличивать файл приложения. Решается эта проблема теми или иными вариантами сжатия. Так как мобильные приложения не должны занимать слишком много места, в них применяется сжатие текстур. Существуют различные способы сжатия изображений, каждый со своими достоинствами и недостатками. По этой причине у вас может возникнуть необходимость внесения коррективов в процесс сжатия текстур в Unity.

Управление сжатием текстур является неотъемлемой частью создания приложений для мобильных устройств, хотя эта процедура может применяться и для остальных платформ. Впрочем, в силу большей технической зрелости этих платформ на этот аспект можно не обращать особого внимания, в то время как мобильные устройства к нему крайне чувствительны.

Сжатие текстур в данном случае за вас выполнит Unity; в большинстве инструментов разработки эту процедуру приходится выполнять вручную, в то время как Unity, как правило, импортирует несжатые изображения, постфактум указывая вариант сжатия в настройках импорта (рис. 12.6).

Эти настройки сжатия предоставляются по умолчанию, и в отдельных случаях может потребоваться их корректировка. В частности, для платформы Android сжатие изображений имеет свои особенности. По большей части они обусловлены многообразием устройств Android. Например, так как все устройства iOS пользуются практически одинаковыми видеопроцессорами, в приложениях iOS может использоваться процедура сжатия, оптимизированная для их GPU — ускорителей графики. Для

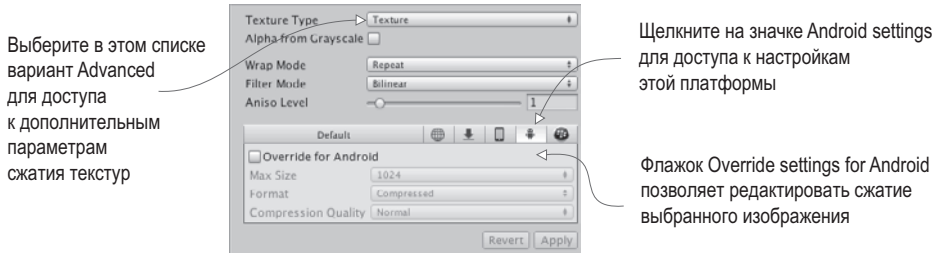


Рис. 12.6. Настройки сжатия текстур на панели Inspector

приложений Android подобное единообразие аппаратного обеспечения недоступно, поэтому для них процедуру сжатия текстур приходится сводить к набору минимально необходимых действий.

А именно, все устройства iOS используют графические ускорители PowerVR; соответственно, в приложениях для iOS может применяться оптимизированный формат сжатия PVR. Эти графические ускорители порой встречаются и у устройств Android, но с такой же частотой встречаются ускорители Adreno от Qualcomm, Mali от ARM и другие варианты. В результате приложения для Android в общем случае пользуются алгоритмом Ericsson (Ericsson Texture Compression, ETC), который поддерживается всеми устройствами Android. К сожалению, этот алгоритм, как и появившийся позже ETC1, и разрабатываемый сейчас ETC2, не поддерживает альфа-канал, соответственно, снабженные им изображения с его помощью сжиматься не могут.

При переходе на другую платформу Unity подвергает изображение повторному сжатию. В случае платформы Android ограничение на работу с каналом прозрачности обходится путем преобразования изображения к 16-битному цвету вместо его сжатия. Размер файла при этом уменьшается, но за счет ухудшения качества изображения. Поэтому периодически приходится вручную возвращать настройки сжатия отдельных изображений в исходное состояние, определяя для каждого изображения в отдельности, где требуется обработка канала прозрачности, а где можно воспользоваться алгоритмом ETC (обеспечив лучшее качество изображения), и выбирая, какие из изображений с альфа-каналом нуждаются в уменьшении размера, а какие можно оставить без сжатия.

Для коррекции способа сжатия текстуры пользуйтесь настройками, показанными на рис. 12.6. Для доступа к ним выберите в раскрывающемся списке Texture Type вариант Advanced и прейдите на вкладку, отмеченную значком Android, чтобы переопределить исходные настройки сжатия.

Эта процедура является важной деталью оптимизации при работе с платформой Android. Тема же следующего раздела одинаково важна как для iOS, так и для Android. Мы поговорим о разработке их собственных подключаемых модулей.

12.3.3. Разработка подключаемых модулей

Инструмент Unity обладает богатейшей встроенной функциональностью, но по большей части это общая для всех платформ функциональность. Использование же

привязанных к конкретной платформе наборов инструментов (таких, как Play Game Services для Android) часто требует для Unity дополнительных модулей.

СОВЕТ Для функциональных особенностей, связанных с платформами iOS и Android, доступно множество готовых модулей. Принцип управления ими аналогичен описываемому в этом разделе, просто вы пользуетесь уже готовым кодом, написанным специально для вас.

Процесс обмена данными с внутренними модулями аналогичен процессу обмена данными с браузером. Со стороны Unity есть специальные команды, вызывающие функции внутри модулей. Модули же, со своей стороны, для отправки сообщений объектам в Unity-сценах пользуются методом `SendMessage()`. Конкретная реализация кода зависит от платформы, но принцип функционирования во всех случаях сохраняется.

ПРИМЕЧАНИЕ Как и исходный процесс сборки, процесс разработки подключаемых модулей для мобильных устройств часто меняется — не со стороны Unity, а со стороны кода аппаратной платформы. Я описываю в этой главе общий принцип, а актуальную информацию, касающуюся деталей реализации, вы можете найти в Интернете.

Кроме того, модули для обеих платформ Unity хранит в одном и том же месте. Создайте на вкладке Project папку Plugins; она относится к папкам, которые Unity обрабатывает особым способом, как, к примеру, папку Editor. В данном случае Unity автоматически ищет в папке Plugins файлы подключаемых модулей. Внутри этой папки создайте еще две — для Android и для iOS; их содержимое Unity будет копировать в процессе сборки.

Подключаемые модули iOS

Подключаемый модуль — это всего лишь некий код для аппаратной платформы, к которому обращается Unity. Поэтому начните с создания сценария `TestPlugin` (скопируйте в него код следующего листинга).

Листинг 12.4. Сценарий TestPlugin, вызывающий из Unity код для iOS

```
using UnityEngine;
using System;
using System.Collections;
using System.Runtime.InteropServices;

public class TestPlugin : MonoBehaviour {
  private static TestPlugin _instance;

  public static void Initialize() {
    if (_instance != null) {
      Debug.Log("TestPlugin instance was found. Already initialized");
      return;
    }
    Debug.Log("TestPlugin instance not found. Initializing...");

    GameObject owner = new GameObject("TestPlugin_instance");
```

Объект создается в этой статической функции, поэтому в редакторе его создавать не нужно.

```
    _instance = owner.AddComponent<TestPlugin>();
    DontDestroyOnLoad(_instance);
  }

  #region iOS ← Тег, определяющий раздел кода; сам по себе он ничего не делает.
  [DllImport("__Internal")]
  private static extern float _TestNumber();
                                              Ссылка на функцию в коде iOS.
  [DllImport("__Internal")]
  private static extern string _TestString(string test);
  #endregion iOS

  public static float TestNumber() {
    float val = 0f;

    if (Application.platform == RuntimePlatform.IPhonePlayer)
    val = _TestNumber(); ← Вызывается в случае платформы IPhonePlayer.
    return val;
  }

  public static string TestString(string test) {
    string val = "";
    if (Application.platform == RuntimePlatform.IPhonePlayer)
    val = _TestString(test);
    return val;
  }
}
```

Первым делом обратите внимание, что статическая функция `Initialize()` создает в сцене постоянный объект, избавляя вас от необходимости делать это вручную в редакторе Unity. Код, создающий объекты с нуля, вам раньше не встречался, так как в большинстве случаев намного проще воспользоваться для этой цели шаблоном экземпляра, но сейчас аккуратнее будет получить нужный объект программно (это даст вам возможность пользоваться сценарием модуля без редактирования сцены).

Именно здесь происходит основное действие, в том числе использование атрибута `DLLImport` и статических внешних команд. Именно они связывают Unity с функциями написанного вами кода для устройств. Затем эти функции вызываются в методах сценария (с условной инструкцией, проверяющей, что код запущен на платформе iPhone/iOS).

Теперь нужно протестировать функции модуля. Создайте сценарий `MobileTestObject`, а также пустой объект сцены, с которым нужно будет связать этот сценарий. Код сценария скопируйте из следующего листинга.

Листинг 12.5. Использование подключаемого модуля в сценарии MobileTestObject

```
using UnityEngine;
using System.Collections;

public class MobileTestObject : MonoBehaviour {
  private string _message;

  void Awake() {
    TestPlugin.Initialize(); ← Инициализация модуля в начале кода.
```

```
  }

  // Используем это для инициализации
  void Start() {
    _message = "START: " + TestPlugin.TestString("ThIs Is A tEsT");
  }

  // Функция Update вызывается в каждом кадре
  void Update() {

    // Проверяем, коснулся ли пользователь экрана
    if (Input.touchCount==0){return;}

    Touch touch = Input.GetTouch(0); ← Ответ на ввод данных методом касания.
    if (touch.phase == TouchPhase.Began) {
      _message = "TOUCH: " + TestPlugin.TestNumber();
    }
  }

  void OnGUI() {
    GUI.Label(new Rect(10, 10, 200, 20), _message); ← Отображение сообщения в углу экрана.
  }
}
```

Приведенный в этом листинге сценарий инициализирует представляющий наш модуль объект и в ответ на ввод данных прикосновением к экрану вызывает методы этого модуля. После запуска этого сценария на устройстве вы увидите, как в ответ на прикосновение к экрану меняется тестовое сообщение в углу.

Теперь осталось написать код для устройства, на который ссылается сценарий `TestPlugin`. Для устройств iOS он пишется на языке Objective C и/или C, поэтому нам потребуется как файл заголовка с расширением .h, так и файл реализации с расширением .mm. Как уже упоминалось, такие файлы должны находиться в папке Plugins/iOS/ на вкладке Project. Создайте в этой папке сценарии `TestPlugin.h` и `TestPlugin.mm`; в файл с расширением .h скопируйте код следующего листинга.

Листинг 12.6. Заголовок TestPlugin.h для iOS-кода

```
#import <Foundation/Foundation.h>

@interface TestObject : NSObject {
  NSString* status;
}

@end
```

Объяснение таких сложных вопросов, как программирование для iOS, выходит за рамки темы данной книги, поэтому, чтобы понять смысл данного кода, обратитесь к справочной документации. Код следующего листинга нужно ввести в файл с расширением .mm.

Листинг 12.7. Реализация TestPlugin.mm

```
#import "TestPlugin.h"

@implementation TestObject
```

```
@end

NSString* CreateNSString (const char* string)
{
  if (string)
  return [NSString stringWithUTF8String: string];
  else
  return [NSString stringWithUTF8String: ""];
}

char* MakeStringCopy (const char* string)
{
  if (string == NULL)
  return NULL;

  char* res = (char*)malloc(strlen(string) + 1);
  strcpy(res, string);
  return res;
}

extern "C" {
  const char* _TestString(const char* string) {
    NSString* oldString = CreateNSString(string);
    NSString* newString = [oldString uppercaseString];
    return MakeStringCopy([newString UTF8String]);
  }

  float _TestNumber() {
    return (arc4random() % 100)/100.0f;
  }
}
```

И снова подробное объяснение кода выходит за рамки темы нашей книги. Обратите внимание, сколько функций используется для преобразования Unity-строк в форму, которую понимает данный код.

СОВЕТ В рассмотренном примере взаимодействие осуществлялось только в одном направлении: из Unity к модулю. Но код модуля также может отправлять данные в Unity методом UnitySendMessage(). Сообщения отправляются именованному объекту в сцене; в процессе инициализации модуля создается предназначенный именно для этого объект TestPlugin_instance.

Теперь, когда у нас есть код для аппаратной платформы, можно сгенерировать приложение для iOS и протестировать его работу на устройстве. Потрясающе! Но мы пока научились создавать модули только для iOS, давайте посмотрим, как этот процесс выглядит для платформы Android.

Модули для Android

Создание модуля для Android со стороны Unity представляет собой практически аналогичный процесс. Нам даже не нужно вносить правки в сценарий `MobileTestObject`. Дополнения, показанные в следующем листинге, нужно вставить только в сценарий `TestPlugin`.

Листинг 12.8. Редактирование сценария TestPlugin под платформу Android

```
...
  #region iOS
  [DllImport("__Internal")]
  private static extern float _TestNumber();

  [DllImport("__Internal")]
  private static extern string _TestString(string test);
  #endregion iOS

#if UNITY_ANDROID
  private static Exception _pluginError;
  private static AndroidJavaClass _pluginClass;
  private static AndroidJavaClass GetPluginClass() {
    if (_pluginClass == null && _pluginError == null) {
      AndroidJNI.AttachCurrentThread();
      try {
        _pluginClass = new
        AndroidJavaClass("com.companyname.testplugin.TestPlugin");
      } catch (Exception e) {
        _pluginError = e;
      }
    }
    return _pluginClass;
  }
  private static AndroidJavaObject _unityActivity;
  private static AndroidJavaObject GetUnityActivity() {
    if (_unityActivity == null) {
      AndroidJavaClass unityPlayer = new
    AndroidJavaClass("com.unity3d.player.UnityPlayer");
      _unityActivity =
        unityPlayer.GetStatic<AndroidJavaObject>("currentActivity");
    }
    return _unityActivity;
  }
  #endif

  public static float TestNumber() {
    float val = 0f;
    if (Application.platform == RuntimePlatform.IPhonePlayer)
       val = _TestNumber();
    #if UNITY_ANDROID
    if (!Application.isEditor && _pluginError == null)
       val = GetPluginClass().CallStatic<int>("getNumber");
    #endif
    return val;
  }

  public static string TestString(string test) {
    string val = "";
    if (Application.platform == RuntimePlatform.IPhonePlayer)
    val = _TestString(test);
    #if UNITY_ANDROID
    if (!Application.isEditor && _pluginError == null)
```

Обеспечиваемая Unity функциональность AndroidJNI.

Имя запрограммированного нами класса; измените его, если нужно.

Unity создает экран для приложения Android.

Обращение к функциям в модуле .jar.

```
    val = GetPluginClass().CallStatic<string>("getString", test);
    #endif
    return val;
  }
}
```

Думаю, вы заметили, что большинство дополнений появилось внутри определений платформы `UNITY_ANDROID`; как объяснялось в начале этой главы, эти директивы компилятора приводят к тому, что код применяется только при работе на определенной платформе, во всех остальных случаях он просто игнорируется. Если код для iOS на остальных платформах не производил никаких действий (он ничего не делал, но и не вызывал ошибок), код модулей для Android будет компилироваться, только если инструмент Unity настроен на работу с платформой Android.

В частности, обратите внимание на обращения к объекту `AndroidJNI`. Эта система в Unity обеспечивает связь с кодом устройства для Android. Другим, возможно, не совсем понятным элементом будет класс `Activity`; в приложениях для Android он соответствует экрану приложения. Экраном в нашем случае будет служить Unity, значит, коду модуля нужен доступ к этому экрану, чтобы в случае необходимости обойти его.

Наконец, вам требуется код для Android. Если код для iOS пишется на таких языках, как Objective C и C, приложения для Android программируются на языке Java. Но для модуля мы не можем взять и написать код на Java; модуль должен находиться в JAR-архиве. К сожалению, в данном случае подробное рассмотрение происходящего выходит за рамки книги. Впрочем, для примера вы можете посмотреть приведенный далее листинг, демонстрирующий файл сборки Ant (укажите собственные варианты пути к файлам; особое внимание обратите на Unity-файл `classes.jar`, применяемый при сборке модулей для Android), а также листинг 12.10 с кодом Java для используемого модуля.

Листинг 12.9. Сценарий build.xml, генерирующий JAR-архив из Java-кода

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="TestPluginJava">
  <!-- Измените это в соответствии с вашей конфигурацией -->
  <property name="sdk.dir"
    value="LOCATION OF ANDROID SDK"/>
  <property name="target" value="android-18"/>
  <property name="unity.androidplayer.jarfile"
    value="/Applications/Unity/Unity.app/Contents/PlaybackEngines/
   AndroidPlayer/development/bin/classes.jar"/>
  <!-- Исходная папка -->
  <property name="source.dir"
value="LOCATION OF THIS PROJECT/Assets/Plugins/ Android/TestPlugin" />
  <!—Выходная папка для файлов .class -->
  <property name="output.dir"
value="LOCATION OF THIS PROJECT/Assets/Plugins/ Android/TestPlugin/classes"/>
  <!-- Имя создаваемого архива jar. Обратите внимание, что оно
   должно совпадать с именем класса и с именем,
   указанным в файле AndroidManifest.xml-->
  <property name="output.jarfile" value="../TestPlugin.jar"/>
    <!-- Создает выходные папки, если они пока отсутствуют. -->
```

```
  <target name="-dirs" depends="message">
    <echo>Creating output directory: ${output.dir} </echo>
    <mkdir dir="${output.dir}" />
  </target>
<!-- Компилирует файлы .java этого проекта в файлы .class. -->
  <target name="compile" depends="-dirs"
   description="Compiles project's .java files into .class files">
    <javac encoding="ascii" target="1.6" debug="true"
      destdir="${output.dir}" verbose="${verbose}"
      includeantruntime="false">
        <src path="${source.dir}" />
        <classpath>
          <pathelement
            location="${sdk.dir}\platforms\${target}\android.jar"/>
          <pathelement location="${unity.androidplayer.jarfile}"/>
        </classpath>
    </javac>
  </target>
  <target name="build-jar" depends="compile">
    <zip zipfile="${output.jarfile}" basedir="${output.dir}" />
  </target>
  <target name="clean-post-jar">
    <echo>Removing post-build-jar-clean</echo>
    <delete dir="${output.dir}"/>
  </target>
  <target name="clean"
    description="Removes output files created by other targets.">
      <delete dir="${output.dir}" verbose="${verbose}" />
  </target>
  <target name="message">
    <echo>Android Ant Build for Unity Android Plugin</echo>
      <echo> message: Displays this message.</echo>
      <echo> clean: Removes output files created by other targets.
        </echo>
      <echo> compile: Compiles .java files into .class files.</echo>
      <echo> build-jar: Compiles .class files into .jar file.</echo>
  </target>
</project>
```

Листинг 12.10. Файл TestPlugin.java, компилируемый в JAR-архив

```
package com.companyname.testplugin;

public class TestPlugin {
  private static int number = 0;

  public static int getNumber() {
    number++;
    return number;
  }

  public static String getString(String message) {
    return message.toLowerCase();
  }
}
```

МАНИФЕСТ ANDROID И ПАПКА RESOURCES

Для нашего простого тестового модуля это было не нужно, но модулям для Android часто требуется редактировать файл манифеста. Все приложения для Android контролируются основным конфигурационным файлом AndroidManifest.xml; если вы не предоставляете этого файла, Unity создает его базовую версию, но лучше сделать это самостоятельно, поместив файл манифеста в папку Plugins/Android/ вместе с содержащим наш модуль JAR-архивом.

При сборке приложения для Android система Unity помещает сгенерированный файл манифеста в папку Temp по адресу StagingArea/AndroidManifest.xml. Скопируйте его, чтобы отредактировать вручную (пример такого файла вы найдете и в сопроводительных фрагментах кода).

Аналогично существует папка res, в которой сохраняются такие ресурсы, как, к примеру, собственные нестандартные значки; она создается внутри папки plugins.

Сгенерированный сценарием сборки JAR-файл сохраняется в папке Plugins/Android (для ясности люди часто копируют сюда весь Java-проект, но с технической точки зрения значение имеет только JAR-архив). Теперь выполните сборку игры, и при любом касании экрана сообщение будет меняться. Аналогично iOS-модулям, модули для Android могут посылать данные объекту сцены методом `UnityPlayer.UnitySendMessage()` (этому Java-коду потребуется импортировать из Unity библиотеку/JAR-архив для проигрывателя Android).

Я обошел вниманием большую часть процесса разработки JAR-архивов для Android, но это связано как с крайней трудоемкостью, так и с частой изменчивостью этого процесса. Если вы хотите самостоятельно разрабатывать модули своих игр для Android, читайте документацию на сайте Android-разработчиков.

Поздравляю, вы добрались до конца!

Мои поздравления, вы освоили все этапы развертывания Unity-игр на мобильных устройствах. Базовый процесс сборки для всех платформ очень прост (и осуществляется с помощью единственной кнопки), трудности представляет настройка приложений под различные платформы. Но теперь вы готовы к самостоятельному плаванию и к созданию собственных игр!

12.4. Заключение

- Unity позволяет создавать исполняемые приложения для огромного количества платформ, включая настольные компьютеры, мобильные устройства и веб-сайты.
- Существует множество параметров сборок, в том числе таких, как значок приложения и его название.
- Веб-игры могут обмениваться данными с веб-страницей, в которую они встроены, позволяя создавать интересные варианты приложений.
- Для расширения своей функциональности Unity поддерживает нестандартные подключаемые модули.

Приложение А. Перемещение по сцене и клавиатурные комбинации

Хотя управление Unity осуществляется посредством мыши и клавиатуры, точный порядок действий для новичков *не очевиден*. Мышь и клавиатура применяются для перемещения по сцене и осмотра трехмерных объектов. Есть в Unity и клавиатурные комбинации, ускоряющие выполнение наиболее распространенных операций.

В этом приложении я расскажу об элементах управления вводом, кроме того, вы можете воспользоваться информацией со следующих страниц:

- http://docs.unity3d.com/ru/current/Manual/SceneViewNavigation.html;
- http://docs.unity3d.com/ru/current/Manual/UnityHotkeys.html.

А.1. Навигация с помощью мыши

Навигация по сцене сводится к трем основным маневрам: перемещение, облет и масштабирование. Все они выполняются путем нажатия кнопки мыши и перетаскивания с одновременным удерживанием комбинации клавиш Alt (или Option на компьютерах Mac) и Ctrl. Конкретный порядок действий зависит от того, с какой мышью — однокнопочной, двухкнопочной или трехкнопочной — вы работаете; все эти варианты перечислены в табл. А.1.

Таблица А.1. Элементы управления навигацией для мышей различных типов

Действие	Трехкнопочная мышь	Двухкнопочная мышь	Однокнопочная мышь
Перемещение	Нажатие средней кнопки мыши/перетаскивание	Удерживание клавиш Alt и Command плюс нажатие левой кнопки мыши/перетаскивание	При нажатых клавишах Alt+Command+нажатие кнопки мыши/перетаскивание
Облет	Удерживание клавиши Alt плюс нажатие левой кнопки мыши/перетаскивание	Удерживание клавиши Alt плюс нажатие левой кнопки мыши/перетаскивание	Удерживание клавиши Alt плюс нажатие кнопки мыши/перетаскивание
Масштабирование	Удерживание клавиши Alt плюс нажатие правой кнопки мыши/перетаскивание	Удерживание клавиши Alt плюс нажатие правой кнопки мыши/перетаскивание	Удерживание клавиш Alt и Ctrl плюс нажатие кнопки мыши/перетаскивание

ПРИМЕЧАНИЕ Хотя Unity можно управлять с помощью одно- и двухкнопочной мыши, я крайне рекомендую вам приобрести трехкнопочную мышь (которая к тому же прекрасно работает с Mac OS X).

Кроме навигационных маневров, выполняемых с помощью мыши, есть и варианты управления с клавиатуры. При нажатой правой кнопке мыши клавиши WASD начинают играть роль клавиш со стрелками, перемещая вас по сцене в манере, общей для большинства игр от первого лица. Выполнение операций при нажатой клавише Shift ускоряет эти операции, заставляя персонажа перемещаться быстрее. Но самой важной является клавиша F, нажатие которой при выделенном объекте в сцене вызывает автоматическое панорамирование и масштабирование, позволяющее сфокусироваться на этом объекте. Если в процессе перемещений по сцене вы утратили представление о том, где находитесь, достаточно выделить какой-либо объект на вкладке Hierarchy и нажать клавишу F.

A.2. Распространенные клавиатурные комбинации

В Unity есть ряд клавиатурных комбинаций для быстрого доступа к важным функциям. Самыми важными являются клавиши W, E, R и T, активирующие инструменты преобразования Translate, Rotate и Scale (если вы не помните, какую функцию они выполняют, перечитайте главу 1), а также инструмент 2D Rect. Эти клавиши расположены рядом, поэтому в процессе работы на них обычно держат левую руку, в то время как правая манипулирует мышью.

Дополнительные клавиатурные комбинации, которые могут пригодиться вам при работе с Unity, перечислены в табл. А.2.

Таблица А.2. Полезные клавиатурные комбинации

Клавиатурная комбинация	Функция
W	Translate (перемещение выделенного объекта)
E	Rotate (поворот выделенного объекта)
R	Scale (изменение размера выделенного объекта)
T	Rect (манипулирование двухмерными объектами)
F	Фокусировка на выделенном объекте
V	Привязка к вершинам
Ctrl/Command+Shift+N	Создание нового объекта GameObject
Ctrl/Command+P	Воспроизведение игры
Ctrl/Command+R	Обновление проекта
Ctrl/ Command+1	Отображение вкладки Scene в активном окне
Ctrl/Command+2	Активация окна Game
Ctrl/Command+3	Переход на панель Inspector
Ctrl/Command+4	Переход на вкладку Hierarchy
Ctrl/Command+5	Переход на вкладку Project
Ctrl/Command+6	Переход на вкладку Animation

В Unity определены и другие клавиатурные комбинации, но они соответствуют достаточно сложным операциям.

Приложение Б. Внешние инструменты, используемые вместе с Unity

В процессе разработки игр в Unity для решения некоторых задач приходится пользоваться внешними инструментами. В главе 1 мы уже обсуждали один из них; инструмент MonoDevelop с технической точки зрения представляет собой отдельное приложение, хотя и входит в один пакет с Unity. Аналогичным образом разработчики прибегают к многочисленным дополнительным инструментам для решения задач, не реализуемых средствами Unity.

Это не означает, что в Unity отсутствует необходимая функциональность. Просто процесс разработки игр столь сложен и многогранен, что любой хорошо спроектированный инструмент с четким предназначением по определению является узкоспециализированным. В нашем случае Unity является связующим элементом и движком для всех компонентов игры, обеспечивая их совместную работу, а создание отдельных элементов может выполняться внешними инструментами. Давайте рассмотрим несколько категорий программного обеспечения, которое может вам пригодиться.

Б.1. Инструменты программирования

Вы уже знакомы с приложением MonoDevelop, которое представляет собой наиболее интересный инструмент программирования для Unity. Но есть и другие программы, которые следует иметь в виду.

Б.1.1. Visual Studio

Как упоминалось в главе 1, Unity поставляется с приложением MonoDevelop, причем эта интегрированная среда разработки действует как на платформе Windows, так и на Mac. Однако при работе в операционных системах семейства Windows вы можете воспользоваться также средой разработки Visual Studio. Недавно корпорация Microsoft приобрела компанию SyntaxTree, занимавшуюся совершенствованием механизма интеграции Visual Studio: http://unityvs.com.

Б.1.2. Xcode

Инструмент Xcode представляет собой среду программирования от Apple (в частности, это IDE, но туда входят и SDK для платформ Apple). Несмотря на то что большая часть работы выполняется в Unity, при развертывании игр на платформе iOS

вам потребуется Xcode для отладки и профилирования ваших приложений: https://developer.apple.com/xcode/.

Б.1.3. Android SDK

Аналогично тому, как для развертывания в iOS вам необходимо приложение Xcode, для развертывания игр на платформе Android вам нужно будет скачать Android SDK. Но в отличие от сборки игр для iOS вам не потребуется прибегать к внешним по отношению к Unity инструментам разработки — достаточно указать в Unity настройки, относящиеся к Android SDK: http://developer.android.com/sdk/index.html.

Б.1.4. SVN, Git или Mercurial

Любой крупный проект по разработке программного обеспечения включает в себя множество версий файлов кода, поэтому программисты разработали особый класс программного обеспечения, который называется VCS (Version Control System — система контроля версий). Три наиболее популярные системы — это Subversion (также известна как SVN), Git и Mercurial. Если вы еще не пользуетесь системой контроля версий, рекомендую установить себе какую-нибудь из них. Папку проекта Unity заполняет временными файлами и рабочими настройками, но для системы управления версиями интерес представляют только папки Assets (убедитесь, что ваша система контроля версий воспринимает генерируемые Unity файлы метаданных) и Project Settings: http://subversion.apache.org/, http://git-scm.com/, http://mercurial.selenic.com/wiki/Mercurial.

Б.2. Приложения для работы с трехмерной графикой

Хотя Unity в состоянии обрабатывать двухмерную графику (именно этой теме были посвящены главы 5 и 6), изначально Unity представлял собой движок для трехмерных игр и до сих пор обладает мощным функционалом для работы с 3D-графикой. Многие специализирующиеся на трехмерной графике художники пользуются по крайней мере одним из перечисленных в этом разделе инструментов.

Б.2.1. Maya

Программа Maya представляет собой редактор для трехмерной графики и анимации, уходящий корнями в киноиндустрию. Ее функционал позволяет решить практически любую возникающую перед художником задачу, начиная созданием прекрасной анимации и заканчивая эффективными, готовыми к использованию в играх моделями. Смоделированная в Maya трехмерная анимация (например, двигающиеся персонажи) может быть экспортирована в Unity: www.autodesk.com/products/autodesk-maya/overview.

Б.2.2. 3ds Max

Другой широко распространенный редактор для работы с трехмерной графикой и анимацией — 3ds Max — представляет практически идентичную Maya функциональность. Совпадает и основная схема работы. Приложение 3ds Max работает

только в операционных системах семейства Windows (в то время как остальные инструменты, в том числе Maya, являются кросс-платформенными), но часто применяется в игровой индустрии: www.autodesk.com/products/autodesk-3ds-max/overview.

Б.2.3. Blender

Хотя и не столь распространенное, как 3ds Max или Maya, приложение Blender сравнимо с этими двумя редакторами. Ведь оно тоже позволяет решать практически все возникающие задачи 3D-моделирования и при этом является приложением с открытым исходным кодом. Поэтому вы можете бесплатно скачать версию Blender для любой платформы. Так как это бесплатное приложение, предполагается, что в процессе чтения данной книги вы будете использовать именно его: www.blender.org.

Б.3. Редакторы двухмерной графики

Двухмерная графика является важным компонентом всех игр вне зависимости от того, отображается ли она напрямую в двухмерных играх или в виде текстур на поверхности трехмерных моделей. В процессе разработки игр применяется ряд инструментов для работы с двухмерной графикой, список которых вы найдете в этом разделе.

Б.3.1. Photoshop

Графический редактор Photoshop является самым распространенным из упоминаемых здесь приложений. Встроенные в него инструменты позволяют редактировать существующие изображения, накладывать фильтры и даже рисовать изображения с нуля. В Photoshop поддерживается множество форматов файлов, в том числе все форматы, используемые в Unity: www.photoshop.com.

Б.3.2. GIMP

Эта аббревиатура происходит от GNU Image Manipulation Program и является названием наиболее известного редактора двухмерной графики с открытым исходным кодом. Программа GIMP в смысле функциональности и удобства применения уступает Photoshop, но это все равно полезный и к тому же бесплатный графический редактор: www.gimp.org.

Б.3.3. TexturePacker

В то время как все перечисленные инструменты применяются не только в разработке игр, приложение TexturePacker используется только в этой области. Но оно отменно справляется с задачами, для которых было создано: со сборкой листов спрайтов двухмерных игр. Если вы разрабатываете двухмерную игру, рекомендую установить себе TexturePacker: www.codeandweb.com/texturepacker.

Б.4. Звуковое программное обеспечение

Существует невообразимое количество программ для работы со звуком, включая как звуковые редакторы (которые работают с исходными звуковыми файлами), так

и секвенсеры (создающие музыку из последовательности нот). Чтобы дать вам представление о доступных возможностях, мы рассмотрим два основных инструмента редактирования звука (из не вошедших в список я хотел бы отметить Logic, Ableton и Reason).

Б.4.1. Pro Tools

Это приложение для работы со звуком может похвастаться множеством полезных функций и рассматривается многочисленными производителями музыки и звукооператорами как индустриальный стандарт. Оно часто используется для решения различных профессиональных задач, в том числе при разработке игр: www.avid.com/US/products/family/Pro-Tools.

Б.4.2. Audacity

Не относящаяся к числу профессиональных инструментов для работы со звуком, программа Audacity представляет собой отличный редактор для решения не слишком сложных задач, например подготовки коротких аудиофайлов со звуковыми эффектами для игры. Эту программу часто выбирают те, кто предпочитает программы с открытым исходным кодом: http://audacity.sourceforge.net/.

Приложение В. Моделирование скамейки в программе Blender

Когда мы разрабатывали уровни в главах 2 и 4, то располагали там стены и пол. А как быть с более детализированными объектами? Что делать, если вы захотите обставить комнаты интересной мебелью? Для решения этой задачи вам потребуется внешний редактор трехмерной графики. Напомню определение, данное во введении к главе 4: 3D-модели в играх представляют собой сеточные объекты (то есть трехмерные фигуры). В этом приложении на примере простой скамейки, показанной на рис. В.1, я опишу принцип создания таких объектов.

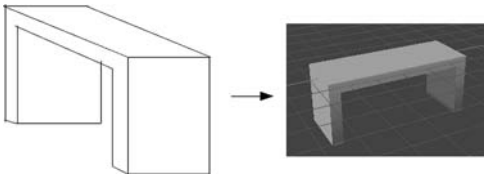


Рис. В.1. Схема простой скамейки, которую мы собираемся смоделировать

В приложении Б перечислен ряд инструментов для работы с трехмерной графикой. Для этого упражнения мы воспользуемся программой Blender, так как это доступная всем читателям программа с открытым исходным кодом. Вы создадите сеточный объект и экспортируете его в Unity в виде графического ресурса.

СОВЕТ Моделирование — это большая сложная тема, а мы затронем только необходимые для создания скамейки функции. Если после чтения этого приложения вы захотите более подробно познакомиться с процессом моделирования объектов, воспользуйтесь соответствующей литературой и онлайновыми справочными материалами (для начала рекомендую сайт www.blender.org).

ВНИМАНИЕ У меня установлена версия Blender 2.67, соответственно, именно к ней относятся все объяснения и снимки экрана. Версии программы Blender выходят часто, поэтому у вас расположения кнопок или имена команд могут слегка отличаться от описанного.

В.1. Создание сеточной геометрии

Запустите Blender; начальный экран с кубом посреди сцены показан на рис. В.2. Управление камерой осуществляется с помощью средней кнопки мыши: перетаскивание при нажатой средней кнопке ведет к повороту камеры, при нажатой

клавише Shift — к панорамированию сцены, а при нажатой клавише Ctrl — к масштабированию.

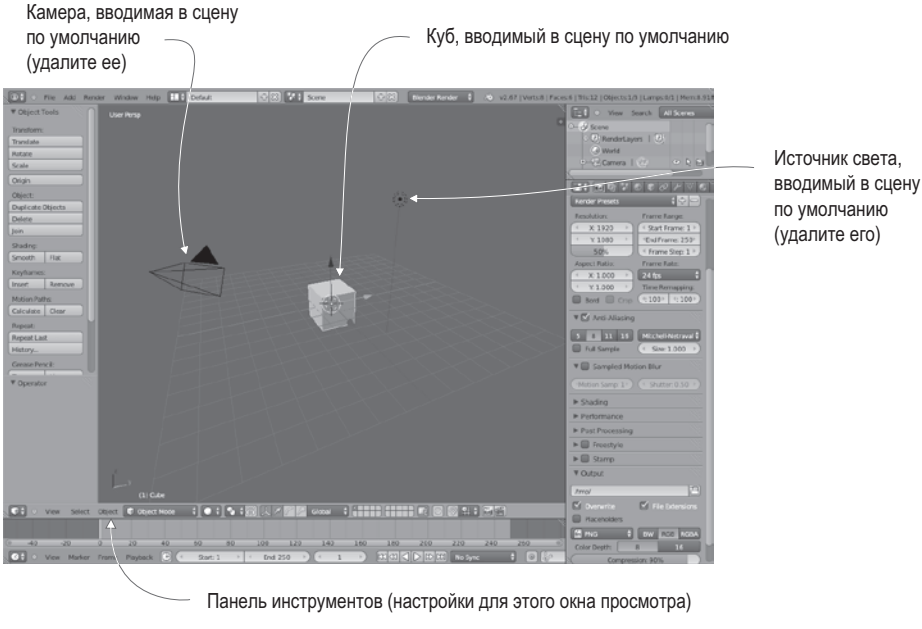


Рис. В.2. Начальный экран в программе Blender

Работа с программой Blender начинается в режиме Object mode, что, как несложно догадаться, означает манипуляцию объектами в целом путем перемещения их по сцене. Для перехода к редактированию формы объекта нужно перейти в режим Edit mode — для этого используется меню, показанное на рис. В.3 (нужный вам пункт меню появляется только при наличии в сцене выделенного объекта, впрочем, сразу после загрузки приложения Blender объект выделятся автоматически). Кроме того, при первом переключении в режим редактирования вы попадаете в режим выделения вершин. Показанные на рис. В.4 кнопки позволяют переключаться между режимами выделения вершин (Vertex), ребер (Edge) и граней (Face). Именно таким способом осуществляется выделение различных *элементов сетки*.

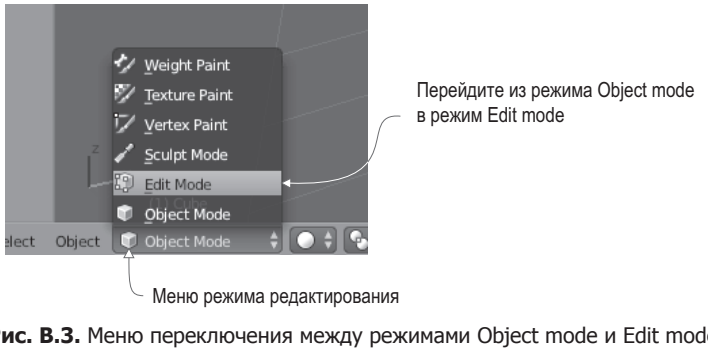


Рис. В.3. Меню переключения между режимами Object mode и Edit mode

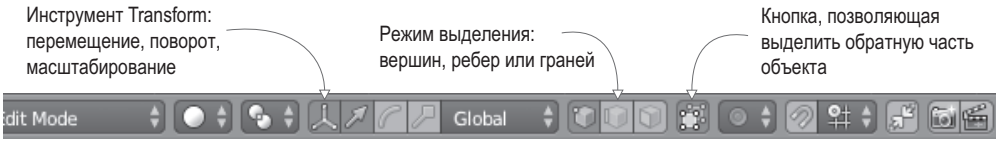


Рис. B.4. Элементы управления в нижней части окна просмотра

ОПРЕДЕЛЕНИЕ Элементами сетки (mesh elements) называются определяющие ее геометрию вершины, ребра и грани, то есть отдельные угловые точки, соединяющие эти точки линии и образованные соединенными линиями фигуры.

КЛАВИАТУРНЫЕ КОМБИНАЦИИ И РАБОТА С МЫШЬЮ В BLENDER

Рисунок B.4 демонстрирует нам также различные инструменты преобразования. Как и в Unity, преобразования сводятся к перемещениям, поворотам и масштабированию. Первая кнопка включает и отключает габаритный контейнер преобразования (Transform Gizmo), обозначаемый в сцене стрелками; я рекомендую оставить его включенным, потому что в противном случае вы сможете работать с инструментами преобразования только с помощью клавиатурных комбинаций. При этом данные комбинации в Blender зачастую крайне неочевидны, как и приемы работы с мышью.

Если, к примеру, управление камерой при помощи средней кнопки мыши интуитивно понятно, то за выделение элементов сцены отвечает правая кнопка мыши (в то время, как в большинстве приложений эта операция выполняется левой кнопкой). Что еще более странно, для выделения куба вам потребуется нажать клавишу B, а затем щелкнуть на объекте левой кнопкой мыши и перетащить указатель. Чтобы добавить элемент в выделенный набор (а не заменять один выделенный элемент другим), удерживайте во время щелчка на элементе клавишу Shift, а для снятия выделения достаточно нажать клавишу A.

Итак, теперь вы знакомы с основными элементами управления в приложении Blender, давайте посмотрим на функции редактирования модели. Для начала превратим куб в длинную тонкую доску. Для этого выделите все вершины модели (в том числе и вершины задней грани) и активируйте инструмент Scale. Для уменьшения вертикальных размеров перетащите мышью синюю стрелку, которая соответствует оси *Z*, а затем таким же способом растяните объект вдоль оси *Y*, чтобы придать ему форму, показанную на рис. B.5.

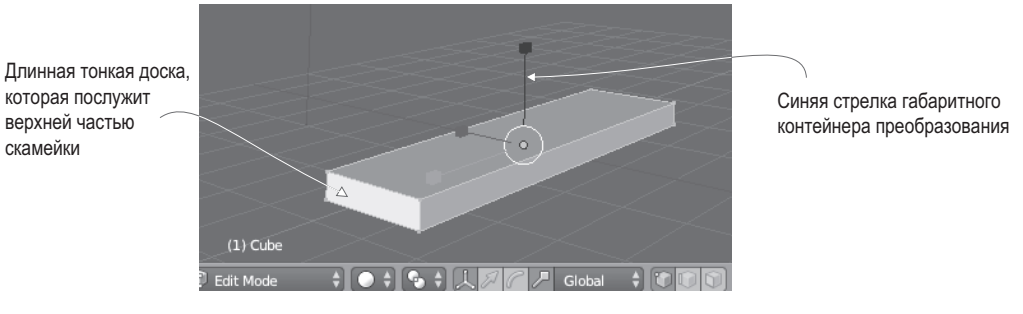


Рис. B.5. Сетка, путем масштабирования преобразованная в длинную тонкую доску

Переключитесь в режим выделения граней (используйте для этого кнопку, показанную на рис. В.4) и выделите две самые маленькие грани нашей доски. Далее в меню Mesh, расположенном в нижней части окна, выберите команду Extrude Individual, как показано на рис. В.6. В результате перемещение указателя мыши будет сопровождаться появлением на концах доски дополнительных секций; слегка вытяните их и щелкните левой кнопкой мыши для завершения процесса. Эти дополнительные секции увеличивают скамейку на ширину ее ножек, предоставляя вам дополнительную геометрию.

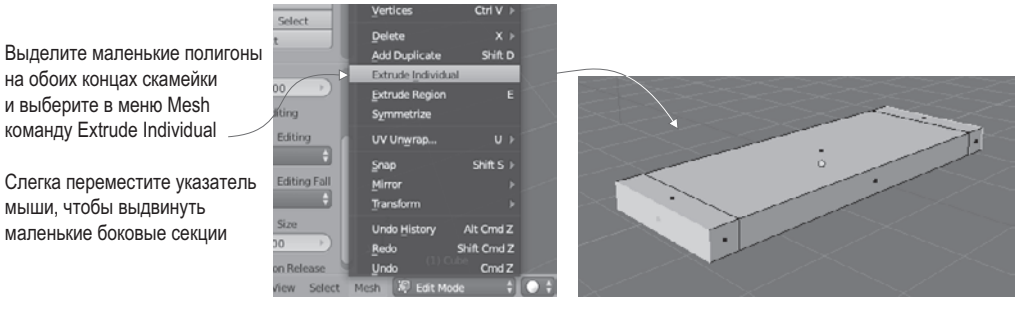


Рис. В.6. Создание дополнительных фрагментов объекта

ОПРЕДЕЛЕНИЕ Процедура выдавливания (extrude) создает новую геометрию с сечением в форме выделенных граней. Существует две команды, определяющие действия в случае набора выделенных элементов: команда Extrude Individual выдавливает каждый элемент как отдельный фрагмент, в то время как команда Extrude Region выдавливает все элементы как одно целое.

Теперь посмотрите на нижнюю часть нашей доски и выделите два узких элемента на ее концах. Еще раз воспользуйтесь командой Extrude Individual, чтобы сформировать ножки скамейки, как показано на рис. В.7.

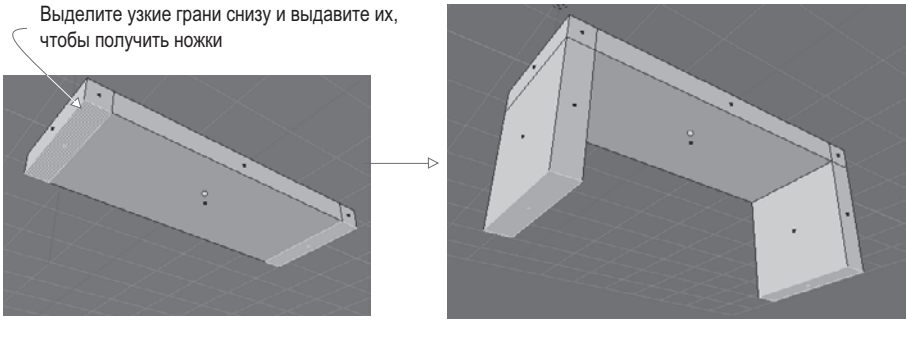


Рис. В.7. Создание ножек скамейки

Фигура готова! Но перед экспортом модели в Unity нужно добавить к ней материал.

B.2. Назначение материала

На поверхности трехмерных моделей можно отображать двухмерные изображения, называемые *текстурами*. В случае больших плоских поверхностей процесс назначения материала очевиден; достаточно растянуть его по поверхности. А что делать с моделями более сложной формы? Здесь нам на помощь приходит такое понятие, как *текстурные координаты*.

Эти координаты определяют положение различных точек текстуры относительно сетки. Фактически, с их помощью сеточные элементы проецируются на области текстуры. Представьте себе оберточную бумагу, как показано на рис. B.8; трехмерная модель — это заворачиваемый в бумагу ящик, текстура — бумага, а координаты показывают, какая сторона бумаги придется на конкретную грань ящика. Они задают на двухмерной картинке точки и фигуры; эти фигуры соответствуют полигонам сетки, благодаря чему части изображения появляются на фрагментах сетки.

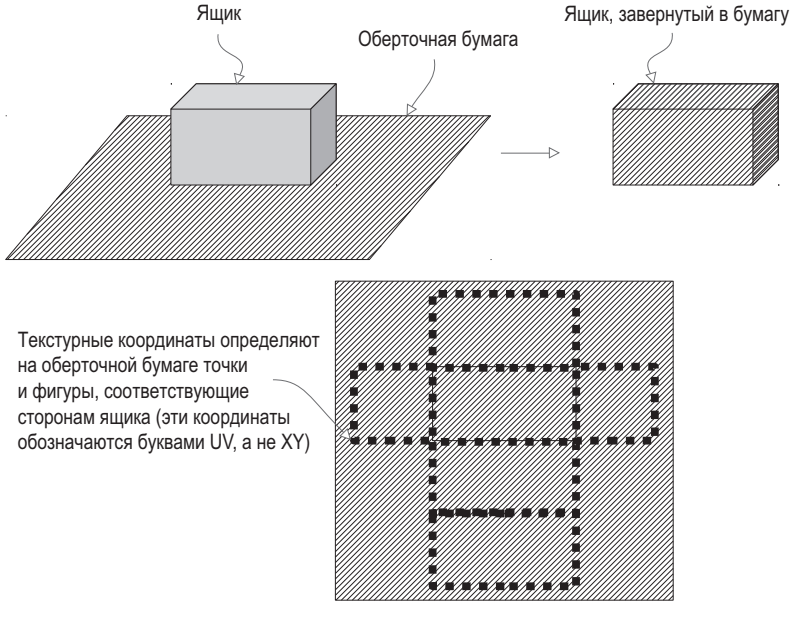


Рис. B.8. Оберточная бумага хорошо иллюстрирует принцип работы текстурных координат

СОВЕТ Еще текстурные координаты называют UV-координатами. Это название появилось из-за того, что текстурные координаты обозначаются буквами U и V, в то время как для обозначения координат трехмерной модели используются буквы X, Y и Z.

Процесс совмещения фрагментов одной сущности с фрагментами другой называется *проецированием* (mapping) — соответственно, термин *текстурное проецирование* (texture mapping) обозначает процесс создания текстурных координат. Из аналогии с оберточной бумагой родилось еще одно название этого процесса — *распаковывание* (unwrapping). Есть также термины, образованные смешением обоих понятий,

например *UV-распаковывание* (UV unwrapping); вам надо постараться не запутаться в многочисленных, по существу, синонимичных понятиях, связанных с текстурным проецированием.

Сам по себе процесс текстурного проецирования достаточно сложен, но, к счастью, приложение Blender оснащено инструментами, превращающими его в рутинную процедуру. Первым делом нужно определить швы; если снова представить процесс упаковывания ящика в бумагу (точнее, лучше представить обратный процесс — распаковывание), становится понятно, что далеко не каждая часть трехмерной фигуры при превращении ее в плоскость остается бесшовной. И там, где фрагменты этой фигуры расходятся в стороны, появляются швы. Приложение Blender позволяет выделять ребра, объявляя их швами.

Переключитесь в режим выделения ребер (см. кнопки на рис. В.4) и выделите ребра с внешней стороны нижней части скамейки. Далее выберите в меню Mesh команду Edges и затем — Mark Seam, как показано на рис. В.9. После этого Blender отделит нижнюю часть скамейки с целью проецирования текстур. Проделайте ту же операцию для сторон скамейки, но полностью их не разделяйте. Вместо этого превратите в швы только ребра, идущие вверх по ножкам; в этом случае стороны сохранят соединение со скамейкой, но раскроются, как крылья.

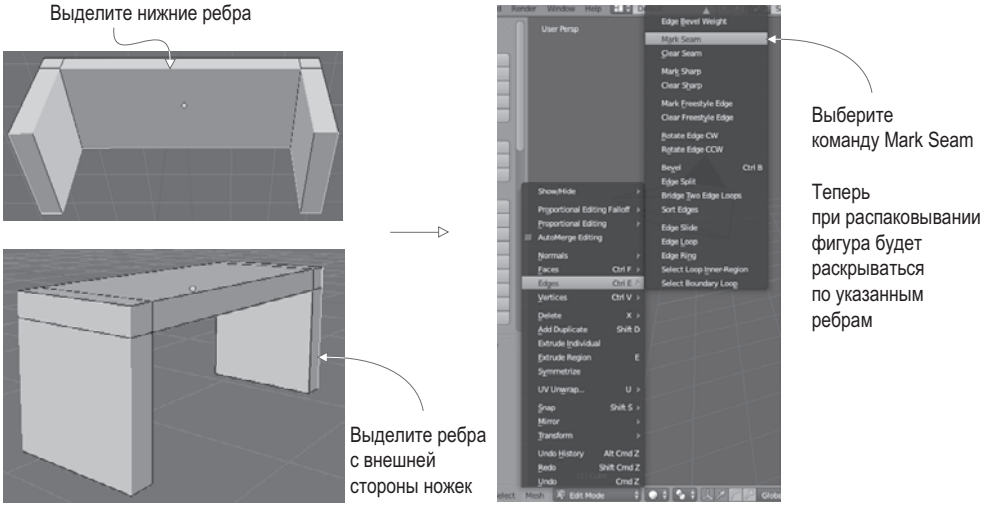


Рис. В.9. Ребра вдоль нижней части скамейки и вдоль ножек превращаются в швы

Пометив все швы, воспользуйтесь командой Texture Unwrap. Для начала выделите сетку целиком (не забудьте невидимую для вас обратную сторону объекта). Затем выберите в меню Mesh команду UV Unwrap ▸ Unwrap, чтобы сгенерировать текстурные координаты. Но пока вы их не увидите, так как по умолчанию Blender демонстрирует трехмерное отображение сцены. Для просмотра текстурных координат нужно переключиться в UV-редактор, воспользовавшись крайним слева меню панели инструментов, которое называется Viewports (вам нужно не слово View, а маленький значок, показанный на рис. В.10).

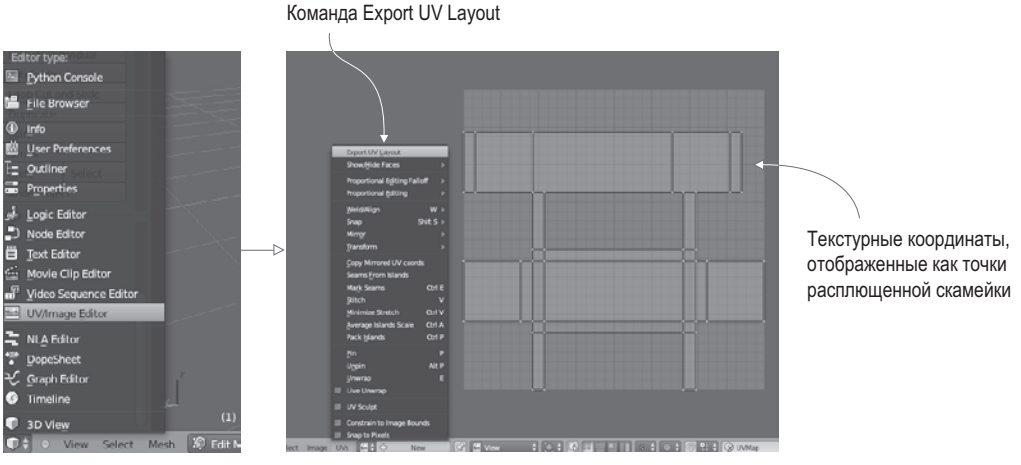


Рис. B.10. Чтобы увидеть текстурные координаты, смените 3D View на UV Editor

После этого вы увидите полигоны скамейки, разложенные на плоскости, отделенные друг от друга и раскрытые по указанным вами швам. Для раскраски текстуры эти UV-координаты должны быть видимы в программе редактирования изображений. Снова обратимся к рис. B.10: выберите в меню UVs команду Export UV Layout и сохраните изображение под именем bench.png (именно это имя будет использоваться для импорта в Unity).

Откройте это изображение в графическом редакторе и раскрасьте части текстуры в разные цвета. Благодаря UV-координатам эти цвета окажутся на соответствующих гранях. Например, на рис. B.11 темно-синим цветом окрашена нижняя часть скамейки, а красным — ее боковые стороны. Теперь изображение можно вернуть в программу Blender и использовать в качестве текстуры для нашей модели — для этого выберите в меню Image команду Open Image.

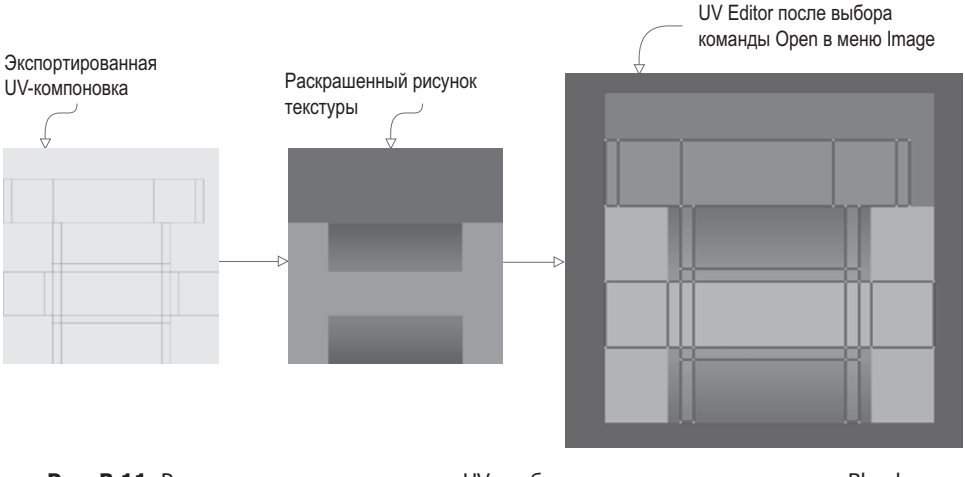


Рис. B.11. Раскрасьте экспортированное UV-изображение и верните текстуру в Blender

Теперь можно вернуться к трехмерному представлению (с помощью того же самого меню, которым мы пользовались для переключения в редактор UV-координат). Текстура на модели все еще не видна, но это легко исправить. Достаточно удалить присутствующий в сцене по умолчанию источник света и включить отображение текстур в окне проекции, как показано на рис. В.12.

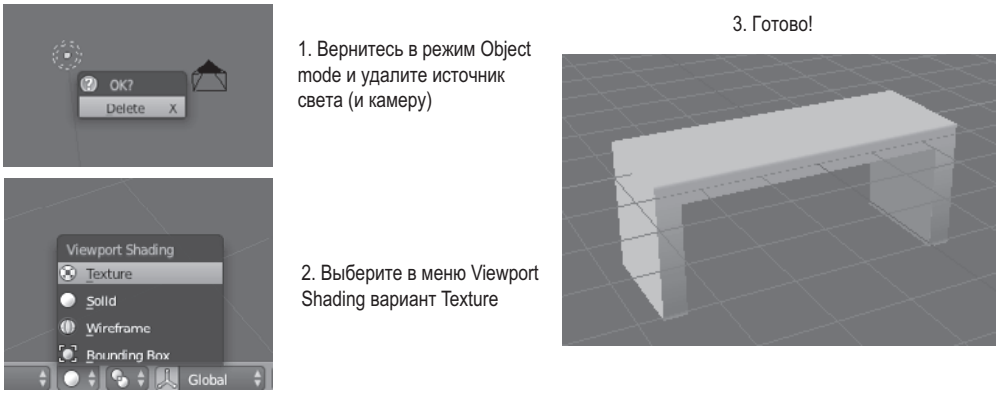


Рис. В.12. Отображение текстуры на поверхности модели

Чтобы удалить источник света, первым делом вернитесь в режим работы с объектами и выделите его (используйте меню, при помощи которого вы переходили в режим редактирования). Нажмите клавишу X для удаления выделенного объекта. Наконец, выберите в меню Viewport Shading команду Texture. В окне проекции появится готовая скамейка с назначенной ей текстурой!

Сохраните модель. Blender по умолчанию сохраняет файлы с расширением .blend, то есть в собственном формате. Воспользуйтесь именно этим форматом, чтобы корректно сохранить все функциональные особенности модели Blender. Позднее вам потребуется экспортировать модель в файл другого формата, предназначенный для импорта в Unity. Обратите внимание, что изображение текстуры вместе с моделью не сохраняется; сохраняется ссылка на него, но вам по-прежнему требуется сам графический файл.