Алан Торн

Искусство создания сценариев в Unity

Alan Thorn

Mastering Unity Scripting

Learn advanced C# tips and techniques to make professional-grade games with Unity

[PACKT] PUBLISHING
BIRMINGHAM – MUMBAI

Алан Торн

Искусство создания сценариев в Unity

Практические советы и приемы создания игр профессионального уровня на C# в Unity

ДМК ПРЕСС
издательство
Москва, 2016

УДК 004.4'2Unity3D
ББК 32.972
Т60

Торн А.
Т60 Искусство создания сценариев в Unity / пер. с англ. Р. Н. Рагимова. – М.: ДМК Пресс, 2016. – 360 с.: ил.

ISBN 978-5-97060-381-9

Это простое и доступное руководство, в котором вы найдете полезные советы и современные приемы программирования игр на C# в Unity. Десять исчерпывающих глав книги содержат практические и наглядные примеры творческого подхода к программированию на C# и созданию коммерчески успешных игр профессионального уровня.

Вы научитесь наделять игровых персонажей впечатляющим искусственным интеллектом, настраивать камеры для создания эффектов постобработки и управлять сценой, опираясь на понимание компонентной архитектуры. Кроме того, вы познакомитесь с классами .NET, позволяющими повысить надежность программ, увидите, как обрабатывать наборы данных, такие как файлы CSV, и как создавать сложные запросы к данным. Прочтя эту книгу до конца, вы станете сильным разработчиком Unity, вооруженным множеством инструментов и приемов быстрой и эффективной разработки коммерческих игр.

Издание предназначено для студентов, преподавателей и специалистов, знакомым с Unity, а также с основами программирования. Неважно, как давно вы знакомы с Unity, в этой книге вы найдете важную и полезную информацию, которая поможет вам эффективно наладить процесс создания игр.

УДК 004.4'2Unity3D
ББК 32.972

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-78439-065-5 (анг.)
ISBN 978-5-97060-381-9 (рус.)

Содержание

Об авторе **10**
О технических рецензентах **11**
Предисловие **13**

Глава 1. Основы C# в Unity **19**

Почему C#? 20
Создание файлов сценариев 21
Подключение сценариев 24
Переменные 26
Условные операторы 28
Оператор if 28
Оператор switch 31
Массивы 34
Циклы 37
Цикл foreach 38
Цикл for 39
Цикл while 40
Бесконечные циклы 42
Функции 42
События 45
Классы и объектно-ориентированное программирование 46
Классы и наследование 49
Классы и полиморфизм 51
Свойства в C# 55
Комментарии 57
Видимость переменных 60
Оператор ? 62
Методы SendMessage и BroadcastMessage 62
Итоги 65

Глава 2. Отладка **66**

Ошибки компиляции и консоль 67
Отладка с помощью Debug.log – определяемые программистом сообщения 70
Переопределение метода ToString 73

Визуальная отладка 76
Регистрация ошибок 80
Отладка с помощью редактора 85
Профилирование 87
Отладка с помощью MonoDevelop – начало 92
Отладка с помощью MonoDevelop – окно Watch 97
Отладка с помощью MonoDevelop – продолжение и пошаговый режим 101
Отладка с помощью MonoDevelop – стек вызовов 103
Отладка с помощью MonoDevelop – окно Immediate 105
Отладка с помощью MonoDevelop – точки останова с условием 107
Отладка с помощью MonoDevelop – точки трассировки 108
Итоги 111

Глава 3. Синглтоны, статические члены, игровые объекты и миры 112

Игровые объекты 112
Взаимодействия компонентов 114
Функция GetComponent 116
Получение нескольких компонентов 117
Компоненты и сообщения 118
Игровые объекты и игровой мир 120
Поиск игровых объектов 120
Сравнение объектов 122
Получение ближайшего объекта 123
Поиск любого объекта определенного типа 124
Проверка препятствий между игровыми объектами 124
Доступ к иерархии объектов 126
Игровой мир, время и обновление 128
Правило № 1 – важность событий обновления кадров 130
Правило № 2 – движение должно основываться на времени 130
Неуничтожаемые объекты 132
Синглтоны и статические переменные 134
Итоги 138

Глава 4. Событийное программирование 139

События 140
Управление событиями 144
Основы управления событиями с помощью интерфейсов 145
Создание класса EventManager 148

Директивы #region и #endregion для свертывания кода в MonoDevelop ... 153
Использование EventManager ... 154
Альтернативный способ, основанный на делегировании ... 155
События класса MonoBehaviour ... 159
События мыши и сенсорной панели ... 160
Фокус приложения и пауза ... 164
Итоги ... 167

Глава 5. Камеры и отображение сцены ... 168

Визуальное представление камеры ... 168
Быть на виду ... 171
Определение видимости объекта ... 172
Подробнее о видимости ... 174
Проверка поля зрения – отображаемые компоненты ... 174
Проверка поля зрения – точки ... 176
Проверка поля зрения – заслонение ... 176
Видимость для камеры – впереди или позади ... 178
Ортографические камеры ... 179
Вывод изображения с камеры и постобработка ... 183
Дрожание камеры ... 189
Камеры и анимация ... 192
Сопровождающие камеры ... 193
Управление движением камеры ... 195
Траектория камеры – iTween ... 197
Итоги ... 201

Глава 6. Работа с фреймворком Mono ... 202

Списки и коллекции ... 203
Класс List ... 204
Класс Dictionary ... 207
Класс Stack ... 208
Интерфейсы IEnumerable и IEnumerator ... 210
Перебор врагов с помощью интерфейса IEnumerator ... 211
Строки и регулярные выражения ... 216
Null, пустые строки и пробелы ... 216
Сравнение строк ... 217
Форматирование строк ... 219
Цикл по символам строке ... 219
Создание строк ... 220

Поиск в строках ... 220
Регулярные выражения ... 220
Произвольное количество аргументов ... 222
Язык интегрированных запросов ... 223
Linq и регулярные выражения ... 226
Работа с текстовыми ресурсами ... 227
Текстовые ресурсы – статическая загрузка ... 227
Текстовые ресурсы – загрузка из локальных файлов ... 228
Текстовые ресурсы – загрузка из INI-файлов ... 230
Текстовые ресурсы – загрузка из CSV-файлов ... 231
Текстовые ресурсы – загрузка из Интернета ... 232
Итоги ... 232

Глава 7. Искусственный интеллект ... 233

Искусственный интеллект в играх ... 234
Начало проекта ... 235
Внедрение навигационного меша ... 237
Создание агента искусственного интеллекта ... 242
Конечные автоматы в Mecanim ... 244
Конечный автомат состояний в C# – начало ... 251
Создание состояния Idle ... 252
Создание состояния Patrol ... 256
Создание состояния Chase ... 260
Создание состояния Attack ... 262
Создание состояния бегства SeekHealth ... 263
Итоги ... 266

Глава 8. Настройка редактора Unity ... 268

Пакетное переименование ... 268
Атрибуты C# и рефлексия ... 274
Смешивание цветов ... 278
Отображение свойств ... 283
Локализация ... 289
Итоги ... 296

Глава 9. Работа с текстурами, моделями и двухмерными изображениями ... 298

Скайбокс ... 299
Процедурные меши ... 305

Анимация UV-координат – прокручивание текстур 311
Рисование на текстуре 313
Шаг 1 – создание шейдера смешивания текстур..... 315
Шаг 2 – создание сценария рисования текстуры..... 319
Шаг 3 – настройка текстуры рисования 326
Итоги 328

Глава 10. Управление исходными текстами и другие подсказки 331

Git – управление исходными текстами..... 331
Шаг № 1 – загрузка 333
Шаг № 2 – добавление проекта в репозиторий 334
Шаг № 3 – настройка Unity для управления исходными текстами 336
Шаг № 4 – создание репозитория 337
Шаг № 5 – игнорируемые файлы..... 338
Шаг № 6 – первая фиксация изменений 339
Шаг № 7 – изменение файлов 341
Шаг № 8 – получение файлов из хранилища 343
Шаг № 9 – просмотр репозитория 345
Папка ресурсов и внешние файлы 347
Пакеты ресурсов и внешние файлы 349
Хранимые данные и сохранение игры 352
Итоги 356

Предметный указатель 357

Об авторе

Алан Торн (Alan Thorn), разработчик игр, независимый программист и писатель, с более чем 13-летним опытом работы, живущий в Лондоне. В 2010 году основал компанию Wax Lyrical Games и является создателем игры «Baron Wittard: Nemesis of Ragnarok», удостоенной многочисленных наград. Автор 10 видеокурсов и 11 книг по разработке игр, в том числе «Unity 4 Fundamentals: Get Started at Making Games with Unity» (Focal Press), «UDK Game Development» и «Pro Unity Game Development with C#» (Apress). Кроме того, как приглашенный лектор читает курс «Game Design & Development Masters Program» в Национальной школе кино и телевидения.

Участвовал как независимый разработчик в более чем 500 проектах по созданию игр, симуляторов, игровых киосков, «серьезных» игр, программ дополненной реальности для игровых студий, музеев и тематических парков по всему миру. В настоящее время работает над приключенческой игрой «Mega Bad Code» для настольных компьютеров и мобильных устройств. Алан обожает графику. Увлекается философией, йогой и пешими загородными прогулками. Его адрес электронной почты: directx_user_interfaces@hotmail.com.

О технических рецензентах

Дилан Агис (Dylan Agis), программист и дизайнер игр, в настоящее время участвует в нескольких сторонних проектах, как независимый разработчик, и одновременно развивает несколько собственных. Имеет большой опыт работы на C++ и C#, а также в Unity, и любит решать проблемы.

Я хотел бы поблагодарить издательство Packt Publishing за предоставленную возможность ознакомиться с книгой и автора за интересное чтение.

Джон П. Доран (John P. Doran), дизайнер игр, созданием которых занимается более 10 лет. Участвовал в разработке разнообразных игр, и в одиночку, и в командах, численностью до 70 человек, в рамках в учебных, и профессиональных проектов.

Одно время работал в компании LucasArts над созданием игры «Star Wars: 1313» как дизайнер-стажер, где был единственным начинающим дизайнером в команде опытных специалистов. Также был ведущим преподавателем на курсах DigiPen®-Ubisoft® Campus Game Programming Program, где обучал студентов-выпускников программированию игр по интенсивной программе.

В настоящее время Джон занимает пост технического дизайнера в отделе исследований и разработки, в институте DigiPen. Кроме того, он преподает и консультирует студентов по нескольким предметам, читает лекции по разработке игр, в том числе на C++, в Unreal, Flash, Unity и др.

Был техническим рецензентом девяти книг по разработке игр и является автором книг «Unity Game Development Blueprints», «Getting Started with UDK», «UDK Game Development [Video]» и «Mastering UDK Game Development HOTSHOT». Все они вышли в издательстве Packt Publishing. Также является соавтором книги «UDK iOS Game Development Beginner's Guide» (Packt Publishing).

Алессандро Моки (Alessandro Mochi) играет в видеоигры со времен появления «Amstrad» и «NES» на всех возможных устройствах: компьютере, консоли и мобильном телефоне. Большие и маленькие видеоигры – его любовь и страсть. Ролевые игры (RPG), стратегии, динамические игры-платфомеры... ничто не ускользнуло от него.

Профессионально занимаясь программированием, имея диплом с отличием в области управления проектами, свободно владея испанским, итальянским и английским языками, он получил глубокое знание многих программ. Всегда готов встретить новые вызовы.

В настоящее время внештатный дизайнер и программист, помогает молодым разработчикам воплотить идеи в реальность. Хотя он часто путешествует по всему миру, его по-прежнему легко найти через его портфолио на www.amochi-portfolio.com.

Райан Уоткинс (Ryan Watkins) любит веселиться. Его можно найти на LinkedIn: www.linkedin.com/in/ryanswatkins.

Предисловие

Книга «Искусство создания сценариев в Unity» – это сжатое и специализированное исследование некоторых продвинутых, нетрадиционных и эффективных методов разработки игровых сценариев на C# в Unity. Это делает книгу очень актуальной, потому что, несмотря на большое число книг «для начинающих» и учебных пособий по Unity, очень немногие из них описывают приемы профессиональной разработки в ясной и структурированной форме. Автор книги предполагает, что вы уже знакомы с основами Unity, такими как импорт ресурсов, проектирование уровней, карты освещения и основы разработки сценариев на C# или JavaScript. Книга сразу начинается с рассмотрения примеров творческого использования сценариев для решения сложных задач, таких как отладка, искусственный интеллект, нестандартное отображение, расширение редактора, анимация и движение, и многое другое. Главная цель заключается не в демонстрации абстрактных принципов и теоретических основ, а в том, чтобы на реальных примерах показать, как применить теорию на практике, что поможет вам в полную силу задействовать свои знания в области программирования для создания качественных игр, которые не просто работают, но работают оптимально. Чтобы получить максимальную отдачу от этой книги, читайте ее главы по порядку, от начала до конца, и используйте навыки обобщенного и абстрактного мышления. То есть, рассматривайте каждую главу, как конкретный пример и демонстрацию более общих принципов, сохраняющихся во времени и пространстве; их можно выделить из конкретного контекста в данной книге и повторно использовать в иных ситуациях, где бы они не потребовались. Проще говоря, рассматривайте приведенные здесь сведения вне связи с конкретными примерами и выбранной мной тематикой, а как весьма актуальные знания для ваших собственных проектов. Итак, давайте начнем.

О чем рассказывается в этой книге

Глава 1 «*Основы C# в Unity*» кратко напоминает основы написания сценариев для Unity. Она не является полным и исчерпывающим руководством по основам. Скорее, это курс повторения для тех, кто ранее уже изучал основы, но, возможно, не писал сценарии некоторое время и был бы рад освежить память перед тем, как начать работу в следующих главах. Если вы знакомы с основами сценариев (такими как классы, наследование, свойства и полиморфизм), можете пропустить эту главу.

Глава 2 «*Отладка*» глубоко исследует процесс отладки. Надежность и эффективность кода нередко зависит от возможности успешно находить и исправлять ошибки при их появлении. Это делает отладку очень важным умением. В этой главе мы не только рассмотрим основы, но и опишем отладку в интерфейсе MonoDevelop, а также установим полезную систему регистрации ошибок.

Глава 3 «*Синглтоны, статические члены, игровые объекты и миры*» исследует широкий спектр возможностей для доступа, изменения и управления игровыми объектами. В частности, мы познакомимся с шаблоном проектирования «Одиночка» (синглтон) для создания глобальных объектов, а также со многими приемами поиска, перечисления, сортировки и размещения объектов. Сценарии в Unity манипулируют объектами в едином игровом мире (пространстве координат), обеспечивая реалистичность игр.

Глава 4 «*Событийное программирование*» рассматривает событийное программирование как важный подход к перестройке архитектуры игры с целью оптимизации. Переложив тяжелую нагрузку с часто возникающих событий и событий обновления на комплекс других событий, можно высвободить много ценного времени для решения других задач.

Глава 5 «*Камеры и отображение сцены*» глубоко исследует работу камер, подробно описывает их архитектуру и настройку отображения сцены. Мы изучим проверку попадания в область видимости, исследуем вопросы отбраковки, познакомимся с такими понятиями, как прямая видимость, ортогональная проекция, глубина, слои, эффекты постобработки и прочее.

Глава 6 «*Работа с фреймворком Mono*» исследует обширную библиотеку Mono и некоторые из ее наиболее практичных классов, от словарей, списков и стеков до таких функций и понятий, таких как строки, регулярные выражения и фреймворк запросов Linq. К концу

этой главы вы овладеете приемами быстрой и эффективной обработки больших объемов данных.

Глава 7 «*Искусственный интеллект*» содержит пример практического применения почти всего описанного ранее в одном проекте для создания искусственного интеллекта, а точнее – умного врага, способного ходить, преследовать, патрулировать, нападать, убегать и искать аптечки для восстановления здоровья. В процессе создания этого персонажа мы рассмотрим вопросы прямой видимости, обнаружения и прокладки маршрутов.

Глава 8 «*Настройка редактора Unity*» описывает редактор Unity, функциональность которого способна удовлетворить самые разные потребности, но иногда бывает нужно нечто большее. В этой главе рассказывается, как создавать классы для настройки самого редактора, чтобы сделать работу в нем удобнее и эффективнее. Мы создадим собственный инспектор свойств и полнофункциональную систему локализации для разработки многоязычных игр.

Глава 9 «*Работа с текстурами, моделями и двухмерной графикой*» содержит описание функций для работы с двухмерными графическими элементами, такими как спрайты, текстуры и элементы пользовательского интерфейса. Двухмерные элементы играют важную роль даже в трехмерных играх, и здесь мы рассмотрим ряд задач работы с двухмерной графикой и эффективные способы их решения.

Глава 10 «*Управление исходными текстами и другие подсказки*» завершает книгу. Она содержит множество советов и рекомендаций, которые не вписываются в какую-либо конкретную категорию, но в целом очень важны. Мы рассмотрим полезные навыки программирования, советы по поддержанию чистоты кода, сериализацию данных, применение систем управления версиями исходных кодов и многое другое.

Что потребуется для работы с книгой

Эта книга, как следует из ее названия, описывает работу с платформой Unity, а это значит, что понадобится только копия Unity. Unity поставляется со всем необходимым для работы с книгой, в том числе со встроенным редактором кода. Дистрибутив Unity можно загрузить с сайта http://unity3d.com/. Приложение Unity поддерживает две основные лицензии, бесплатную и профессиональную. Бесплатная лицензия ограничивает доступ к некоторым функциям, но сохраняет доступность обширного набора основных функций. В целом боль-

шинство глав и примеров в этой книге соответствуют бесплатной версии, то есть, для опробования примеров можно пользоваться бесплатной версией. Тем не менее, некоторые главы и примеры требуют наличия профессиональной версии.

Кому адресована эта книга

Эта книга адресована студентам, преподавателям и специалистам, знакомым с основами Unity и с приемами создания сценариев. Неважно, как давно вы знакомы с Unity, эта книга найдет, что предложить вам, чтобы помочь усовершенствовать приемы разработки игр.

Соглашения

В этой книге используется несколько разных стилей оформления текста для выделения разных видов информации. Ниже приведены примеры этих стилей с объяснением их назначения.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, адреса страниц в Интернете, пользовательский ввод и ссылки в Twitter будут выглядеть так: «После создания новый файл сценария будет сохранен в папке `Project` с расширением `.cs`».

Блоки программного кода оформляются так:

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyNewScript : MonoBehaviour
05 {
```

Когда нам потребуется привлечь ваше внимание к определенному фрагменту в блоке программного кода, мы будем выделять его жирным шрифтом:

```
// Объект следует скрыть, если его координата Y
// получила значение выше 100
bool ShouldHideObject = (transform.position.y > 100) ? true : false;

// Изменить видимость объекта
gameObject.SetActive(!ShouldHideObject);
```

Новые термины и **важные определения** будут выделяться в обычном тексте жирным. Текст, отображаемый на экране, например в меню или в диалогах, будет оформляться так: «Выберите в меню приложения пункт **Assets ⇒ Create ⇒ C# Script**».


Так будут оформляться предупреждения и важные примечания.


Так будут оформляться советы и рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или может быть не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Загрузка исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Загрузка цветных иллюстраций к книге

Вы также можете скачать файл в формате PDF с цветными иллюстрациями и диаграммами к этой книге. Цветные изображения помогут вам лучше понять содержание книги. Загрузить этот файл можно по адресу https://www.packtpub.com/sites/default/files/downloads/0655OT_ColoredImages.pdf.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства ДМК Пресс и Packt очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, и помогающую нам предоставлять вам качественные материалы.

Вопросы

Вы можете присылать любые вопросы, касающиеся данной книги, по адресу dmkpress@gmail.com или questions@packtpub.com. Мы постараемся разрешить возникшие проблемы.

Глава 1

Основы C# в Unity

Эта книга посвящена освоению приемов создания сценариев для Unity, в частности игровых сценариев на языке C#. Перед тем как двигаться дальше, необходимо дать определение понятия освоения приемов создания сценариев. Под освоением подразумевается, что эта книга поможет вам совершить переход от теоретических знаний к более свободному, практическому и продвинутому овладению навыками разработки сценариев. Здесь ключевым является слово «свободное». С самого начала изучения любого языка программирования, в центре внимания неизменно оказывается его синтаксис, правила и законы, то есть формальная часть языка, включающая такие понятия, как переменные, циклы и функции. Однако, по мере накопления опыта, внимание программиста смещается от самого языка к творческим способам его применения для решения насущных задач; от задач, ориентированных на сам язык, к вопросам контекстно-зависимого применения. Следовательно, большая часть этой книги будет посвящена вовсе не формальному синтаксису языка C#.

В следующих главах я буду считать, что вы уже знакомы с основами языка C#. Поэтому далее речь пойдет о конкретных применениях и реальных примерах использования C#. Однако сначала в этой главе основное внимание будет уделено именно основам C#. И это не случайно. Эта глава кратко охватит все основные понятия C#, необходимые для продуктивной работы с последующими главами. Я настоятельно рекомендую прочесть ее от начала до конца, независимо от вашего опыта. Она адресована, прежде всего, читателям, имеющим поверхностное знакомство с C#, но стремящимся углубить свои знания. Однако, она также может помочь опытным разработчикам закрепить имеющиеся знания и, возможно, приобрести новые, свежие идеи. В этой главе я кратко опишу основы C# с нуля, шаг за шагом. Я буду излагать так, как будто вы уже знакомы с основами программирования, может быть на другом языке, но никогда не сталкивались с C#. Итак, начнем.

Почему C#?

Когда дело доходит до сценариев для Unity, перед началом работы над новой игрой всегда возникает вопрос, какой язык выбрать, потому что Unity предлагает выбор. Официально на выбор предлагается три варианта: Boo, C# и JavaScript. В настоящее время не утихают дебаты о том, как правильнее называть JavaScript – «JavaScript» или «UnityScript», – из-за ряда специфичных изменений, внесенных в язык для Unity. Но не это должно нас сейчас волновать. Вопрос в том, какой язык выбрать для проекта. Кроме того, может показаться, что у нас есть еще один вариант – можно выбрать оба языка и писать одни файлы сценария на одном языке, а другие – на другом, фактически смешав языки. Технически это возможно. Unity не запрещает так поступить. Тем не менее это плохо, потому что подобная практика, как правило, приводит к путанице, а также к конфликтам при компиляции, это все равно, что пытаться рассчитать расстояние в милях и километрах одновременно.

Рекомендуемый подход состоит в том, чтобы выбрать один язык и использовать его повсюду в проекте в качестве главного языка. Это упростит работу, но это также означает, что придется выбрать один язык, а от других отказаться. В этой книге выбран язык C#. Почему? Во-первых, не потому, что язык C# лучше других. На мой взгляд, нет абсолютно «лучшего» или абсолютно «худшего» языка программирования. Каждый язык имеет свои достоинства и недостатки, и все языки одинаково хорошо подходят для создания игр в Unity. Основная причина в том, что C# является, пожалуй, наиболее широко используемым и поддерживаемым языком в Unity, и позволяет большинству разработчиков применить уже имеющиеся знания. Большинство учебников по Unity ориентированы на C#, потому что он часто применяется для разработки приложений в других областях. Язык C# исторически привязан к платформе .NET, которая используется в Unity (под именем Mono), к тому же C# напоминает C++, который очень популярен у разработчиков игр. Кроме того, изучив язык C#, вы обнаружите, что ваши знания и умения работать с Unity востребованы в современной игровой индустрии. Таким образом, я выбрал C#, чтобы обеспечить этой книге более широкую аудиторию и позволить вам дополнительно использовать обширный набор уже существующих учебников и литературы. Этот выбор позволит найти применение знаний, полученных при чтении данной книги.

Создание файлов сценариев

Чтобы определить логику игры или поведение ее персонажей, потребуется написать сценарии. Разработка сценариев в Unity начинается с создания нового файла сценария – обычного текстового файла, добавляемого в проект. Этот файл содержит программные инструкции, каждая из которых является командой для Unity. Как уже упоминалось, программный код может быть написан на одном из языков: C#, JavaScript или Boo. В этой книге будет применяться язык C#. В Unity есть несколько способов создания файлов сценария.

Один из них состоит в выборе пункта **Assets** ⇒ **Create** ⇒ **C# Script** (Ресурсы ⇒ Создать ⇒ Сценарий C#) в меню приложения, как показано на рис. 1.1.

Другой способ – щелкнуть правой кнопкой мыши на пустом пространстве в любом месте панели **Project** (Проект) и выбрать в контекстном меню пункт **Create** ⇒ **C# Script** (Создать ⇒ Сценарий C#),

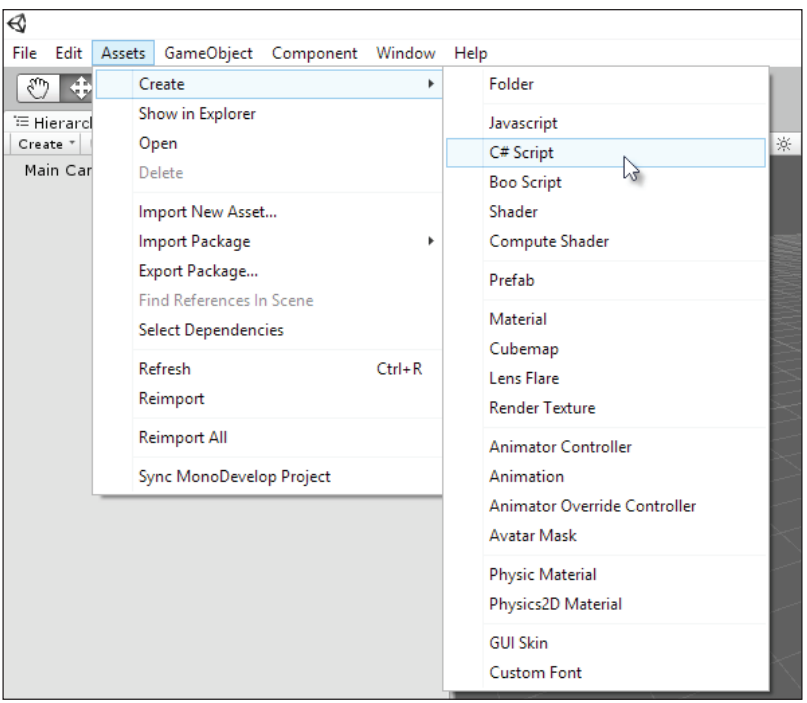


Рис. 1.1. Создание файла сценария с помощью меню приложения

как показано на рис. 1.2. При этом файл сценария будет создан в открытой в данный момент папке.

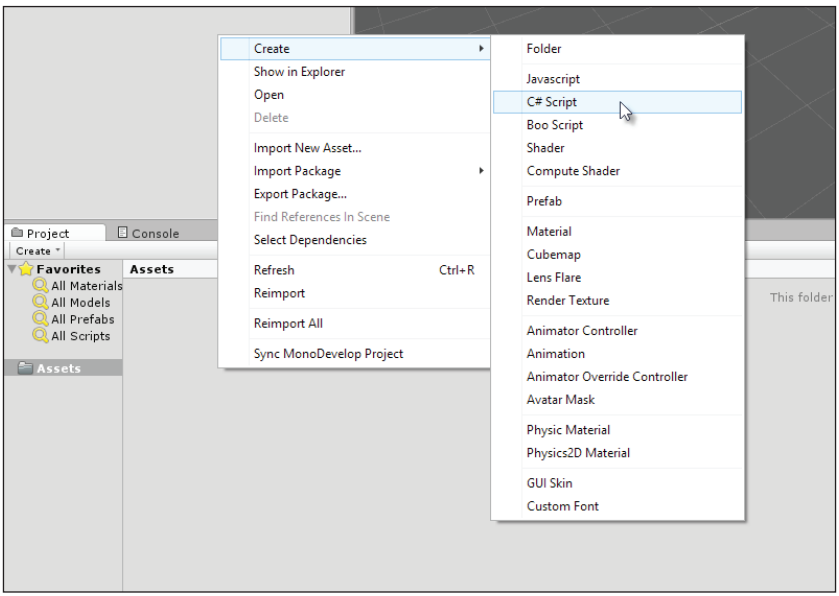


Рис. 1.2. Создание файла сценария с помощью контекстного меню панели **Project**

После этого в папке `Project` будет создан новый файл с расширением `.cs` (сокращенно от C Sharp). Имя файла особенно важно, и его изменение будет иметь серьезные последствия, потому что Unity использует имена файлов для определения имен классов C# в этих файлах. Классы будут рассмотрены более подробно далее в этой главе. Проще говоря, выбирайте для своих файлов уникальные и значимые имена.

Под уникальным именем имеется в виду, что ни какой другой файл в проекте не должен иметь то же имя, независимо от того, в какой папке он находится. Все файлы сценария должны иметь уникальное имя в рамках проекта. Имя должно быть осмысленным и явно выражать назначение сценария. Кроме того, существуют правила, определяющие допустимость имен файлов, а также имен классов в C#. Формальное определение этих правил можно найти по адресу http://msdn.microsoft.com/en-us/library/aa664670%28VS.71%29.aspx. Проще

говоря, имя файла может начинаться только с буквы или символа подчеркивания (цифры для первого символа не подходят), и имя не должно включать пробелов, их рекомендуется заменять символами подчеркивания (_), как показано на рис. 1.3.

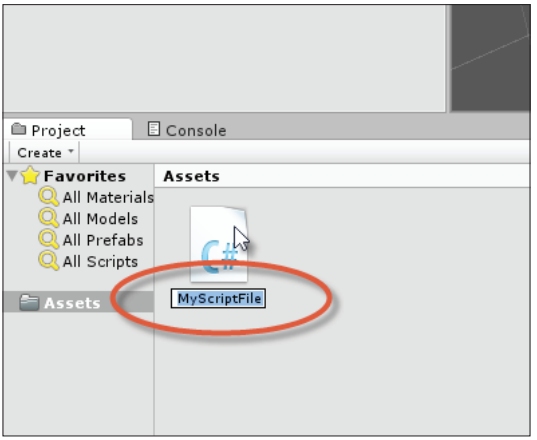


Рис. 1.3. Имена файлов должны быть уникальными и соответствовать принятым в C# соглашениям об именах классов

Файлы сценариев для Unity можно открывать и просматривать в любом текстовом редакторе или интегрированной среде разработки (IDE), в том числе в Visual Studio или Notepad ++, но в состав Unity входит бесплатный редактор исходного кода **MonoDevelop**. Эта программа является частью основного пакета Unity и входит в установочный дистрибутив, но не может быть загружена отдельно. Если дважды щелкнуть на файле сценария в панели **Project** (Проект), файл автоматически откроется в редакторе MonoDevelop. Если потом вы решите переименовать файл сценария, вам также придется переименовать класс C# в файле, чтобы его имя в точности соответствовало новому имени файла, как показано на рис. 1.4. В противном случае будут возникать ошибки во время компиляции и проблемы при подключении файла сценария к объектам.

Компиляция кода. Чтобы скомпилировать код в Unity, достаточно сохранить файл сценария в MonoDevelop, выбрав пункт меню **File** ⇒ **Save** (Файл ⇒ Сохранить) (или нажав **Ctrl**+**S** на клавиатуре), и вернуться в главное окно редактора Unity. При повторном получении фокуса ввода, Unity автоматиче-

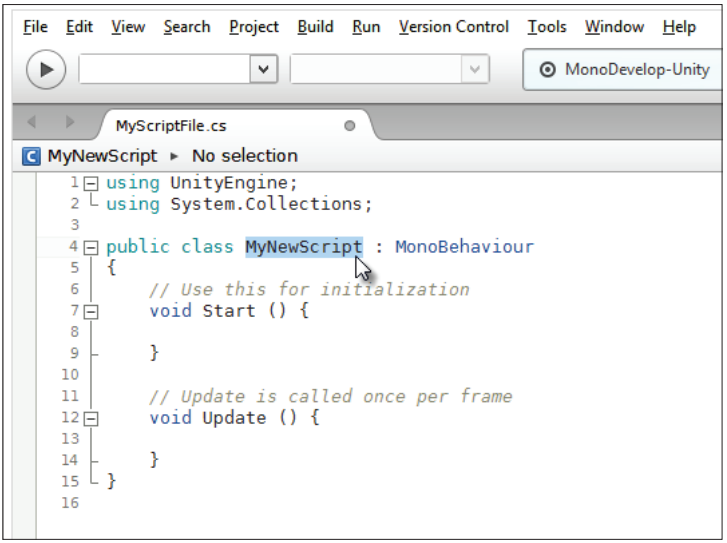


Рис. 1.4. Переименование класса для соответствия имени файла

ски обнаружит изменения в файлах и скомпилирует их. Если при компиляции возникнут ошибки, игру невозможно будет запустить и в окне консоли появится сообщение об ошибках. Если компиляция прошла успешно, игру можно запустить простым щелчком на кнопке **Play** (Играть) в панели инструментов редактора, после чего игра будет запущена в тестовом режиме. Имейте в виду, что если забыть сохранить файл после внесения изменений, Unity будет использовать старую версию кода, скомпилированную прежде. По этой причине, а также в целях резервного копирования важно регулярно сохранять файлы, так что не забывайте нажимать **Ctrl+S**.

Подключение сценариев

Каждый файл сценария для Unity определяет один главный класс, который, подобно шаблону, можно использовать для создания экземпляров. Он представляет собой совокупность связанных между собой переменных, функций и событий (с которыми мы скоро познакомимся). Формально файл сценария подобен любым другим видам ресурсов в Unity, таким как меши (mesh) или аудиофайлы. Он ждет своей очереди в папке `Project` и ничего не делает, пока не будет добавлен в определенную сцену (точнее, подключен к объекту в качестве компонента), где он оживет во время выполнения сцены. Сценарии,

имеющие логическую, или математическую природу, не добавляются в сцену как материальные, независимые объекты, подобно мешам. Вы не увидите и не услышите их непосредственно, потому что они не имеют никакого видимого или слышимого воплощения. Вместо этого они подключаются к объектам игры в виде компонентов, определяющих их поведение. Процесс вовлечения сценария в работу в виде отдельного компонента конкретного объекта называют созданием его экземпляра. Из одного файла сценария можно создать множество экземпляров для нескольких объектов, если их поведение должно быть похожим, это позволяет избежать создания отдельного файла для каждого объекта, например когда несколько вражеских персонажей должны иметь одинаковый искусственный интеллект. В идеале назначение сценария состоит в том, чтобы определить некую абстрактную формулу или модель поведения объекта, которая может быть успешно применена ко многим подобным объектам во всевозможных обстоятельствах. Чтобы подключить файл сценария к объекту, перетащите его из панели **Project** (Проект) на нужный объект в сцене. В результате будет создан экземпляр главного класса в сценарии и прикреплен как компонент, а его общедоступные переменные станут видны в инспекторе при выборе объекта, как показано на рис. 1.5.

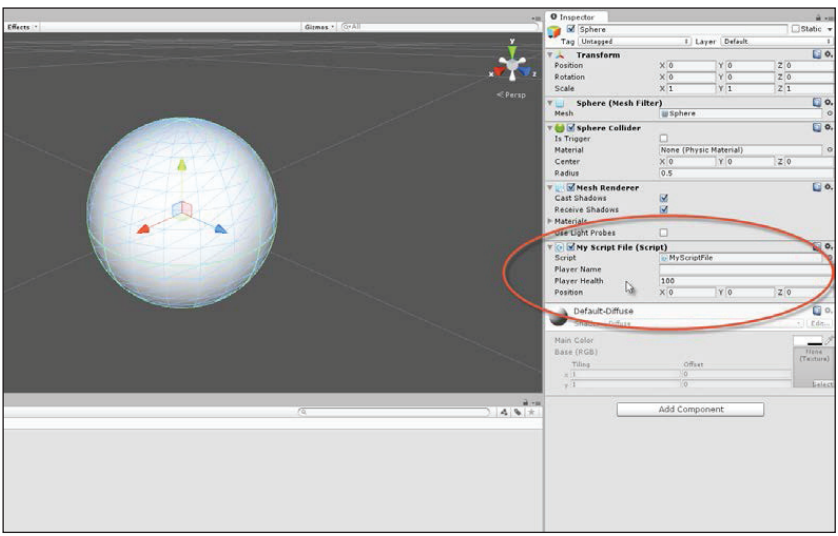


Рис. 1.5. Подключение сценария к объекту игры

Подробнее переменные будут описаны в следующем разделе.

Более подробную информацию о создании и использовании сценариев в Unity можно найти по адресу http://docs.unity3d.com/412/Documentation/Manual/Scripting.html.

Переменные

Самым важным, пожалуй, понятием в программировании вообще и в языке C# в частности является переменная. Переменные часто соответствуют буквам, используемым в алгебре для записи числовых величин, например *X*, *Y* и *Z* или *a*, *b* и *c*. Если потребуется хранить некоторую информацию, такую как имя игрока, счет, положение, ориентацию, количество боеприпасов, здоровье или любые другие сведения (которые можно выразить существительными), переменные помогут вам в этом. Переменная представляет собой единичный элемент информации. То есть, для хранения нескольких элементов информации потребуется несколько переменных, по одной переменной для каждого элемента. Кроме того, каждый элемент будет иметь определенный тип, или вид. Например, имя игрока определяется последовательностью букв, таких как «Джон», «Том» или «Давид». Здоровье игрока, напротив, определяется в числовом виде, например 100 процентов (1) или 50 процентов (0,5), в зависимости от того, какие повреждения получил игрок. Итак, каждая переменная обязательно имеет тип данных. В C# переменные создаются с помощью специального синтаксиса. Взгляните на пример файла сценария в листинге 1.1, содержащего класс с именем `MyNewScript`, в котором объявлены три переменные разных типов с областью видимости класса. Слово «объявить» означает, что мы, как программисты, сообщаем компилятору C# о необходимости создать переменные:

Листинг 1.1. Типы переменных

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyNewScript : MonoBehaviour
05 {
06   public string PlayerName = "";
07   public int PlayerHealth = 100;
08   public Vector3 Position = Vector3.zero;
09
10   // Этот метод выполняет инициализацию
11   void Start () {
12
```

```
13    }
14
15    // Вызывается при отображении каждого кадра
16    void Update () {
17
18    }
19 }
```

Типы данных переменных. Каждая переменная имеет определенный тип данных. Наиболее употребительными являются: `int`, `float`, `bool`, `string` и `Vector3`. Ниже приводится несколько примеров переменных этих типов:

- `int` (целое число) = –3, –2, –1, 0, 1, 2, 3...
- `float` (вещественное, или десятичное число) = –3.0, –2.5, 0.0, 1.7, 3.9...

Обратите внимание на строки 06–08 в листинге 1.1, где каждой переменной присваивается начальное значение и явно указан ее тип данных как `int` (целое число), `string` (строка) и `Vector3`, который представляет координаты точки в трехмерном пространстве (этот тип может также представлять направления, как будет показано ниже). Это не полный список всех возможных типов данных, а только самых распространенных из них. Перечень используемых типов будет зависеть от вашего проекта (а кроме того, вы сможете создавать свои собственные типы!). В этой книге мы будем работать с наиболее распространенными типами данных и вы увидите массу примеров их использования. Наконец, каждая строка объявления переменной начинается с ключевого слова `public`, определяющего степень доступности переменной. Обычно переменные объявляются как общедоступные (`public`) или закрытые (`private`) (существует еще защищенные (`protected`) переменные, но он здесь не рассматривается). Значения общедоступных `public` переменных можно изменять в инспекторе объектов (как мы скоро это увидим, можете также взглянуть на рис. 1.5) или обращаться к ним из других классов.

Переменные названы так потому, что их значения могут меняться в разные моменты времени. Конечно, они меняются не произвольно или непредсказуемо, а только когда мы явно изменяем их: либо путем присваивания нового значения в коде, либо из инспектора объектов, либо вызывая методы и функции. Они могут изменяться непосредственно или косвенно. Переменным можно присваивать значения непосредственно, например:

```
PlayerName = "NewName";
```

или косвенно, с помощью выражений, чье окончательное значение должно быть вычислено до его присваивания переменной:

```
// Переменная получит значение 50, потому что: 100 x 0.5 = 50
PlayerHealth = 100 * 0.5;
```

Область видимости переменных. Каждая переменная объявляется с неявной областью видимости. Область видимости определяет время жизни переменной, то есть области в файле, где на переменную можно сослаться и получить к ней доступ. Область видимости определяется местом объявления переменной. Для переменных, объявленных в листинге 1.1, областью видимости является класс, потому что они объявлены в начале класса и вне функций. Это значит, что они доступны во всем классе, а также (будучи общедоступными (`public`)) из других классов. Переменные могут также объявляться внутри функций. Такие переменные называют локальными, потому что область их видимости ограничена функцией, то есть локальная переменная недоступна за пределами функции, в которой она была объявлена. Классы и функции будут рассмотрены позже в этой же главе.
Более подробную информацию о переменных и их использовании в C# можно найти по адресу http://msdn.microsoft.com/en-us/library/aa691160%28v=vs.71%29.aspx.

Условные операторы

Значения переменных могут изменяться во множестве разных ситуаций: игрок поменял свою позицию, враги были уничтожены, произошла смена уровня и т. д. Следовательно, необходимо часто проверять переменные, чтобы обеспечить выполнение в сценарии разных действий, в зависимости от их текущих значений. Например, если значение переменной `PlayerHealth`, определяющее здоровье игрока, достигнет 0 процентов, сценарий должен выполнить фрагмент кода, отмечающий смерть игрока, а если значение переменной `PlayerHealth` стало равно 20 процентам, может быть желательно вывести предупреждение. В этом конкретном примере значение переменной `PlayerHealth` направит сценарий в указанном направлении. Язык C# предлагает два основных условных оператора для такого ветвления программного кода. Это операторы `if` и `Switch`. Оба они очень полезны.

Оператор if

Оператор `if` имеет несколько разных форм. Основная форма проверяет условие и выполняет следующий за ней блок кода, если и только если условие истинно, то есть его значение равно `true`. Рассмотрим следующий пример в листинге 1.2.

Листинг 1.2. Оператор if

```
01 using UnityEngine;
02 using System.Collections;
03
```

```
04 public class MyScriptFile : MonoBehaviour
05 {
06   public string PlayerName = "";
07   public int PlayerHealth = 100;
08   public Vector3 Position = Vector3.zero;
09
10   // Этот метод выполняет инициализацию
11   void Start () {
12   }
13
14   // Вызывается при отображении каждого кадра
15   void Update ()
16   {
17     // Проверить здоровье игрока - скобки {} необязательны
18     // для однострочного оператора if
19     if(PlayerHealth == 100)
20     {
22       Debug.log ("Player has full health");
23     }
24   }
25 }
```

Этот сценарий можно запустить так же, как любой другой сценарий – щелчком на кнопке **Play** (Играть) в панели инструментов – и он будет выполняться, пока экземпляр класса сценария остается связанным с объектом в активной сцене. Оператор `if` в строке 19 непрерывно проверяет текущее значение переменной `PlayerHealth` класса. Если она в точности равна (`==`) `100`, будет выполнен код внутри скобок `{}` (строки 20–22). Это объясняется тем, что результаты всех проверок приводятся к значению логического типа: либо `true`, либо `false`; на самом деле условный оператор проверяет равенство условия (`PlayerHealth == 100`) значению `true`. Теоретически код в фигурных скобках может содержать несколько строк и выражений. Но здесь он содержит единственную строку 21 – вызов функции `Debug.log`, которая выводит в консоль строку «Player has full health», как показано на рис. 1.6. Конечно, оператор `if` может направить выполнение кода и в другом направлении, то есть, если значение переменной `PlayerHealth` не равно 100 (возможно, оно равно 99 или 101), сообщение не будет выведено. Появление сообщения зависит от равенства условного выражения в предыдущем операторе `if` значению `true`.

Дополнительную информацию об операторах `if`, `if-else` и их использовании в C# можно найти в по адресу http://msdn.microsoft.com/ru-ru/library/5011f09h.aspx.

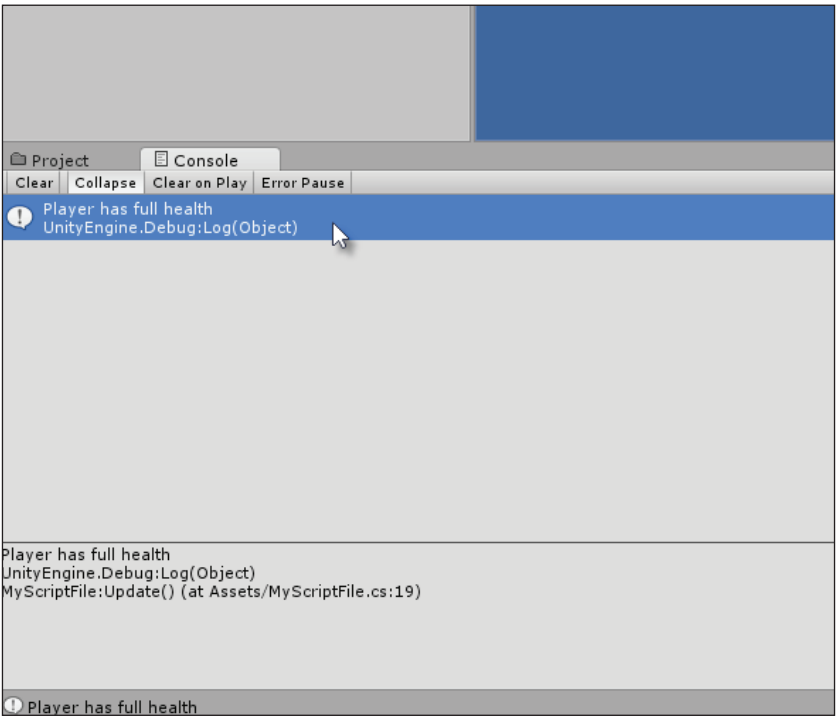


Рис. 1.6. Консоль Unity удобно использовать для вывода отладочных сообщений

Консоль Unity. Как показано на рис. 1.6, консоль Unity является инструментом отладки. Это то место, куда функция `Debug.log` (функция вывода в консоль) выводит сообщения. Консоль удобно использовать для диагностики проблем во время выполнения или компиляции. Если во время компиляции или выполнения будут выведены сообщения об ошибках, их можно найти в списке на вкладке **Console** (Консоль). По умолчанию эта вкладка видна в редакторе Unity, но если это не так, ее можно сделать видимой, выбрав пункт **Window** ⇒ **Console** (Окно ⇒ Консоль) в меню приложения Unity. Более подробную информацию о функции `Debug.log` можно найти по адресу http://docs.unity3d.com/ScriptReference/Debug.log.html.

Кроме проверки равенства (==), как показано в листинге 1.2, можно проверять и другие условия. Например, с помощью операторов > и < можно проверить, является ли переменная больше или меньше заданного значения, соответственно. С помощью оператора != можно проверить неравенство переменной заданному значению. Кроме того, можно даже объединить несколько проверок с помощью операторов

&& (И) и || (ИЛИ). Например, взгляните на следующий оператор if. Он выполняет блок кода между скобками {}, только если значение переменной PlayerHealth находится в интервале между 0 и 100, и не равно 50:

```
if(PlayerHealth >= 0 && PlayerHealth <= 100 && PlayerHealth !=50)
{
  Debug.log ("Player has full health");
}
```


Оператор if-else. Одной из разновидностей оператора if является оператор if-else. Оператор if выполняет блок кода, если условие истинно. Оператор if-else является его расширением. Он выполнит блок кода X, если условие истинно, и блок кода Y, если условие ложно:

```
if(MyCondition)
{
  // X - этот блок выполняется, если условие MyCondition истинно
}
else
{
  // Y - этот блок выполняется, если условие MyCondition ложно
}
```

Оператор switch

Как мы видели, оператор if проверяет истинность условия и на основании результатов проверки принимает решение, выполнять ли следующий за ним блок кода. Оператор switch, напротив, позволяет проверить несколько условий сразу и продолжает выполнение программы в одном из нескольких возможных направлений, а не только в одном или другом, как в случае с оператором if. Например, если персонаж врага может находиться в одном из нескольких состояний (погоня (CHASE), бегство (FLEE), бой (FIGHT), засада (HIDE) и т. д.), вам понадобится несколько веток кода, чтобы обработать каждое состояние. Ключевое слово break используется для выхода из обработки некоторого состояния и перехода в конец оператора switch. В листинге 1.3 показано, как управлять действиями врага с помощью перечисления.

Листинг 1.3. Оператор switch

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06   // Определение возможных состояний врага в виде перечисления
07   public enum EnemyState {CHASE, FLEE, FIGHT, HIDE};
```

```
08
09   // Текущее состояние врага
10   public EnemyState ActiveState = EnemyState.CHASE;
11
12   // Этот метод выполняет инициализацию
13   void Start () {
14   }
15
16   // Вызывается при отображении каждого кадра
17   void Update ()
18   {
19     // Проверить переменную ActiveState
20     switch(ActiveState)
21     {
22       case EnemyState.FIGHT:
23       {
24         // Релизовать бой
25         Debug.log ("Entered fight state");
26       }
27       break;
28
29
30       case EnemyState.FLEE:
31       case EnemyState.HIDE:
32       {
33         // Бегство и засада реализуются одинаково
34         Debug.log ("Entered flee or hide state");
35       }
36       break;
37
38       default:
39       {
40         // Случай по умолчанию, когда никакой другой случай
41         // не подходит. Сейчас обрабатывает состояние погони.
42         Debug.log ("Entered chase state");
43       }
44       break;
45     }
46   }
47 }
```

Перечисления. Строка 07 в листинге 1.3 объявляет перечисление (`enum`) с именем `EnemyState`. Перечисление – это специальная структура, хранящая диапазон допустимых значений для одной или более переменных. Само по себе перечисление не является переменной как таковой, это лишь способ определения набора значений, которые переменная может иметь. В листинге 1.3 переменная `ActiveState` объявлена в строке 10, как переменная типа перечисления `EnemyState`. Ее значением может быть любое из значений перечисления `ActiveState`. Перечисления дают отличную возможность контролировать переменные, ограничивая их значения определенным набором или списком вариантов.

Еще одно большое преимущество перечислений в том, что переменные-перечисления отображаются в инспекторе объектов в виде раскрывающегося списка доступных вариантов значений, как показано на рис. 1.7.

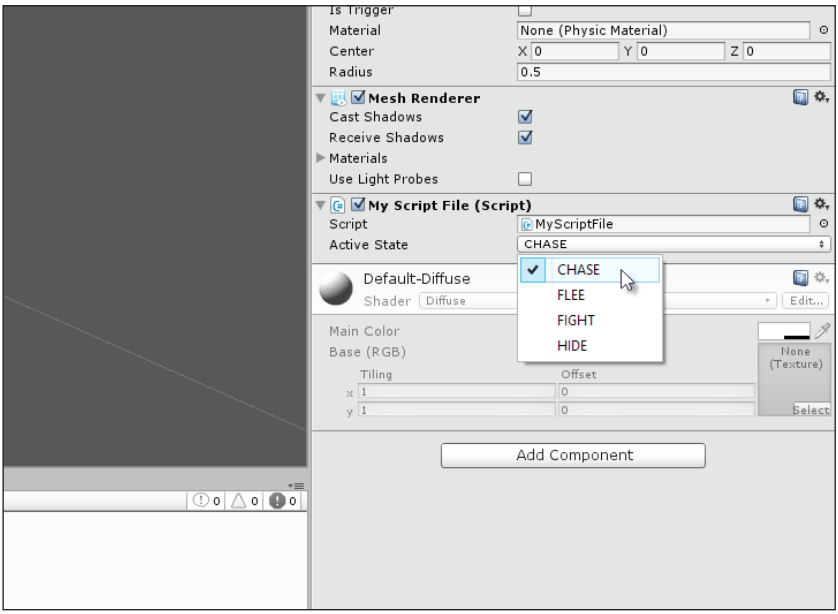


Рис. 1.7. Перечисление в виде раскрывающегося списка вариантов для переменной в инспекторе объектов

Дополнительную информацию о перечислениях и их использовании в C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/sbbt4032.aspx.

Ниже приводится несколько комментариев к листингу 1.3:

- **Строка 20**: начало оператора `switch`. В скобки `()` заключена переменная, чье значение или состояние должно проверяться. В данном случае это `ActiveState`.
- **Строка 22**: первый оператор `case` внутри оператора `switch`. Следующий за ним блок кода (строки 24 и 25) будет выполнен, если значение переменной `ActiveState` равно `EnemyState.Fight`. В противном случае этот блок кода будет пропущен.
- **Стоки 30 и 31**: здесь два оператора `case` следуют друг за другом. Блок кода в строках 33 и 34 будет выполнен, если и толь-

ко если значение переменной `ActiveState` равно `EnemyState.Flee` или `EnemyState.Hide`.

- Строка 38: оператор `default`, который является необязательным в операторе `switch`. Когда этот оператор присутствует, он получает управление, если не подошел ни один оператор `case`. В данном случае это произойдет, если переменная `ActiveState` будет иметь значение `EnemyState.Chase`.
- Строки 27, 36 и 44: оператор `break` должен находиться в конце каждого оператора `case`. Он выполняет выход за пределы оператора `switch`, внутри которого находится, выполнение программы продолжится со строки, следующей за оператором `switch`, в данном случае это строка 45.

Массивы

Списки и последовательности присутствуют в играх повсюду. По этой причине часто бывает нужно хранить наборы данных одного вида: все враги в уровне, все вооружения, которые были собраны, все бонусы, которые могли бы быть собраны, все заклинания и пункты инвентарной ведомости и т. д. Одной из разновидностей списков является массив. Каждый элемент в массиве, по сути, является единицей информации, которая может изменяться во время игры, и для хранения каждого элемента массива потребовалась бы отдельная переменная. Однако гораздо удобнее оперировать набором однотипных переменных (все враги, все оружие и т. д.), собранных в единую, линейную и перечисляемую структуру. Это и есть массив. В C# есть два вида массивов: статические и динамические. Статические массивы могут хранить не более фиксированного числа элементов, заданного заранее, и это число остается неизменным в течение всего времени работы программы, даже если потребуется хранить меньше элементов, чем ранее предполагалось. Это означает, что некоторые слоты, или элементы могут занимать память без пользы. Динамические массивы могут расти и уменьшаться для размещения необходимого числа элементов. Со статическими массивами работать легче и они обрабатываются быстрее, но динамические массивы позволяют избегать потерь памяти. В этой главе рассматриваются только статические массивы, динамические массивы будут рассмотрены позднее. Пример статического массива приводится в листинге 1.4.

Листинг 1.4. Статический массив

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06   // Массив игровых объектов в сцене
07   public GameObject[] MyObjects;
08
09   // Этот метод выполняет инициализацию
10   void Start ()
11   {
12   }
13
14   // Вызывается при отображении каждого кадра
15   void Update ()
16   {
17   }
18 }
```

В строке 07 (листинг 1.4) объявляется пустой массив `MyObjects` с типом `GameObjects`. Для создания массивов используется синтаксис, с квадратными скобками `[]`, следующими за типом `GameObject`, это означает, что объявляется список объектов `GameObjects`, а не один объект `GameObject`. В данном примере объявлен массив, который будет списком всех объектов в сцене. Изначально он пуст, но с помощью инспектора объектов в редакторе Unity можно заполнить этот массив вручную, установив его емкость и добавив в него все необходимые объекты. Чтобы сделать это, выберите объект, к которому прикреплен сценарий, и в разделе **My Objects** (Мои объекты) введите в поле **Size** (Размер) значение, соответствующее емкости массива. Это значение должно быть равно количеству объектов, которые вы хотите в него поместить. Затем перетащите мышью объекты по одному из панели с иерархией сцены в слоты массива, чтобы заполнить список, как показано на рис. 1.8.

Массив можно также заполнить программно, используя функцию `Start` вместо инспектора объектов. Заполнение массива в функции `Start` гарантирует заполнение массива в момент запуска уровня. И тот, и другой методы работают одинаково хорошо. Второй метод демонстрируется в листинге 1.5.

Листинг 1.5. Заполнение массива

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
```

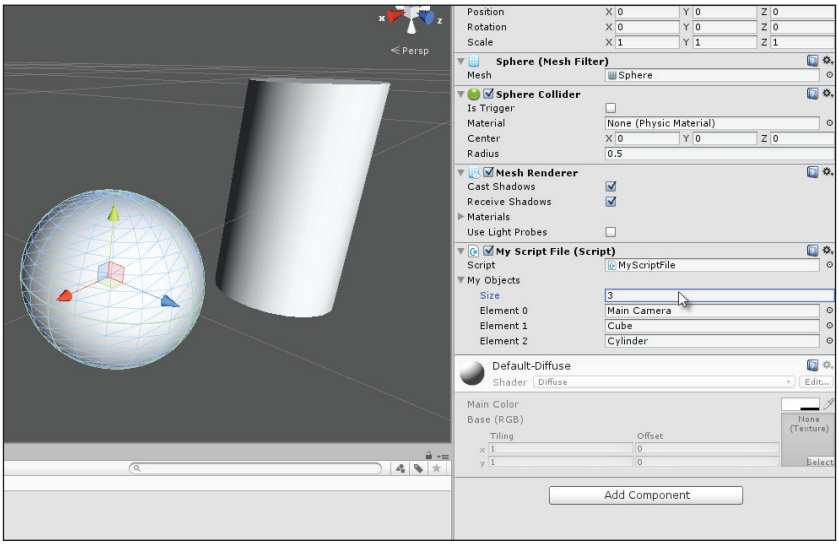


Рис. 1.8. Заполнение массива из инспектора объектов Unity

```
05 {
06   // Массив игровых объектов в сцене
07   public GameObject[] MyObjects;
08
09   // Этот метод выполняет инициализацию
10   void Start ()
11   {
12     // Сконструировать массив программно
13     MyObjects = new GameObject[3];
14     // Сцена должна иметь камеру, обозначенную как MainCamera
15     MyObjects[0] = Camera.main.gameObject;
16     // Использовать функцию GameObject.Find для
17     // поиска объектов в сцене по именам
18     MyObjects[1] = GameObject.Find("Cube");
19     MyObjects[2] = GameObject.Find("Cylinder");
20   }
21
22   // Вызывается при отображении каждого кадра
23   void Update ()
24   {
25   }
26 }
```

Ниже приводится несколько комментариев к листингу 1.5:

- **Строка 10**: функция `Start` вызывается в момент запуска уровня. Функции будут рассматриваться в этой главе ниже.

- **Строка 13**: ключевое слово `new` используется для создания нового массива с тремя элементами. Это означает, что массив не сможет хранить более трех элементов. По умолчанию все элементы получают начальное значение `null` (то есть, ничего). Они пусты.
- **Строка 15**: здесь в первый элемент массива записывается ссылка на объект основной камеры в сцене. Следует отметить два важных момента. Во-первых, к элементам массива можно обращаться по индексам, с помощью оператора `[]`. То есть, к первому элементу массива `MyObjects` можно обратиться с помощью выражения `MyObjects[0]`. Во-вторых, в C# нумерация элементов массива начинается с нуля. Это значит, что первый элемент занимает позицию `0`, следующий – позицию `1`, следующий – позицию `2` и т. д. Для массива `MyObjects` с тремя элементами доступ к элементам можно получить с помощью выражений `MyObjects[0]`, `MyObjects[1]` и `MyObjects[2]`. Обратите внимание, что последний элемент имеет индекс `2`, а не `3`.
- **Строки 18** и **19**: в элементы `1` и `2` массива `MyObjects` записываются ссылки на объекты, полученные с помощью функции `GameObject.Find`. Эта функция производит поиск игрового объекта в активной сцене по его имени (с учетом регистра), затем полученная ссылка помещается в указанный элемент массива `MyObjects`. Если объект с заданным именем не будет найден, в элемент массива будет записано значение `null`.

Более подробную информацию о массивах и их использовании в C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/9b9dty7d.aspx.

Циклы

Циклы являются одним из самых мощных инструментов в программировании. Представьте игру, где на уровень может быть сброшена атомная бомба. Когда это случится, сценарий должен уничтожить почти все, что находится в сцене. Сделать это можно программно, удаляя объекты по одному, написав для каждого отдельную строку кода. Для небольшой сцены из нескольких объектов код удаления займет всего несколько строк, что не проблематично. Но для больших сцен с сотнями объектов придется написать много строк кода, и этот код придется изменять при изменении содержимого сцены. Это будет утомительно. Циклы помогут сжать код до нескольких строк, независимо от сложности сцены и количества объектов. Они позволяют

многократно выполнять одни и те же операции над множеством объектов. В C# имеется несколько видов циклов. Давайте теперь рассмотрим несколько примеров.

Цикл foreach

Самым простым, пожалуй, циклом в C# является цикл `foreach`. С помощью `foreach` можно перебрать все элементы массива по одному, от первого до последнего, и выполнить с каждым из них необходимые операции. Рассмотрим пример в листинге 1.6, удаляющий все объекты `GameObjects` из массива `GameObject`.

Листинг 1.6. Цикл foreach

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06   // Массив игровых объектов в сцене
07   public GameObject[] MyObjects;
08
09   // Этот метод выполняет инициализацию
10   void Start ()
11   {
12     // Выполнить операцию со всеми объектами в массиве
13     foreach(GameObject Obj in MyObjects)
14     {
15       // Уничтожить объект
16       Destroy (Obj);
17     }
18   }
19
20   // Вызывается при отображении каждого кадра
21   void Update ()
22   {
23   }
24 }
```

Загрузка примеров кода. Файлы с примерами можно загрузить с сайта www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Цикл `foreach` выполняет блок кода в строках 14–17, заключенный в фигурные скобки `{}` по одному разу для каждого элемента массива `MyObjects`. Каждый проход, или повтор цикла называется итерацией. Время выполнения цикла зависит от размера массива – чем больше массив, тем больше требуется итераций и больше времени на их выполнение. Цикл также содержит локальную переменную `obj`. Она

объявлена внутри оператора foreach в строке 13. Эта переменная содержит выбранный, или активный элемент массива в каждой итерации цикла, то есть, при выполнении первой итерации цикла переменная obj будет содержать первый элемент, при выполнении второй итерации – второй элемент и т. д.

Более подробную информацию о циклах foreach и их использовании в C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/ttw7t8t6.aspx.

Цикл for

Цикл foreach удобен, когда нужно последовательно перебрать элементы одного массива от начала до конца. Но иногда может понадобиться более полный контроль над итерациями. Например, может потребоваться выполнить цикл в обратном направлении, от конца к началу; обработать сразу два массива одинаковой длины или обработать только определенные элементы массива. Этих целей можно достичь с помощью цикла for, например:

```
// Обработать элементы массива в обратном направлении
for(int i = MyObjects.Length-1; i >= 0; i--)
{
  // Уничтожить объект
  DestroyMyObjects[i]);
}
```

Ниже приводится несколько комментариев к предыдущему фрагменту:

- Здесь цикл for выполняет обход элементов массива MyObjects в обратном направлении, от конца к началу, удаляя каждый GameObject в сцене. При этом он использует локальную переменную i. Ее иногда называют итератором, так как она управляет ходом выполнением цикла.
- Оператор цикла for состоит из трех основных разделов, отделенных друг от друга точкой с запятой:
  - i: инициализация переменной значением MyObjects.Length - 1 (индекс последнего элемента массива). Напомню, что индексация массивов начинается с нуля, поэтому индекс последнего элемента массива всегда равен длине массива минус единицу. Раздел инициализации гарантирует, что цикл начнется с конца массива;
  - i >= 0: выражение, определяющее условие продолжения цикла. Переменная i действует как обратный счетчик, уменьшаясь по мере обхода массива. Цикл будет продолжаться, пока

значение i больше или равно 0, так как 0 является индексом первого элемента массива;

- i--: это выражение выполняется в конце каждой итерации и определяет изменение переменной i. Здесь i будет уменьшаться на единицу в конце каждой итерации, то есть 1 будет вычитаться из i при каждом проходе цикла. Оператор ++, напротив, будет добавлять 1.

❍ Для обращения к элементам массива в теле цикла используется выражение MyObjects[i].

Более подробную информацию о циклах for и их использовании в C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/ch45axte.aspx.

Цикл while

Циклы for и foreach хорошо подходят для обхода массивов с выполнением определенных действий в каждой итерации. Цикл while, напротив, повторяет определенные действия, пока заданное условие остается истинным. Например, если потребуется, чтобы игрок продолжал получать повреждения, пока стоит на горячей лаве, или транспортное средство двигалось, пока не сломается, в таких ситуациях пригодится цикл while. Его применение демонстрирует листинг 1.7.

Листинг 1.7. Цикл while

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06   // Этот метод выполняет инициализацию
07   void Start ()
08   {
09     // Счетчик числа сообщений для вывода
10     int NumberOfMessages = 0;
11
12     // Продолжать, пока в консоль не будет выведено 5 сообщений
13     while(NumberOfMessages < 5)
14     {
15       // Вывести сообщение
16       Debug.log("This is Message: " + NumberOfMessages.ToString());
17
18       // Увличить счетчик
19       ++NumberOfMessages;
20     }
21   }
22
```

```
23    // Вызывается при отображении каждого кадра
24    void Update ()
25    {
26    }
27 }
```

Функция ToString. Многие классы и объекты в Unity имеют функцию `ToString` (строка 16 в листинге 1.7). Эта функция преобразует объект, например `integer` (целое число), в удобочитаемый текст, который можно вывести в окно **Console** (Консоль) или **Debugging** (Отладка). Она может пригодиться для вывода объектов и данных в консоль во время отладки. Обратите внимание, что вывод числовых объектов в виде строк требует неявного преобразования.

Ниже приводится несколько комментариев к листингу 1.7:

- **Строка 13**: начало цикла `while` с условием, которое обеспечивает повторение цикла, пока значение переменной `NumberOfMessages` остется меньше `5`.
- Блок кода в строках 15–19 повторяется как тело цикла `while`.
- **Строка 19**: увеличивает значение переменной `NumberOfMessages` в каждой итерации.

Результатом выполнения сценария в листинге 1.7 будет вывод пяти текстовых сообщений в консоль, как показано на рис. 1.9.

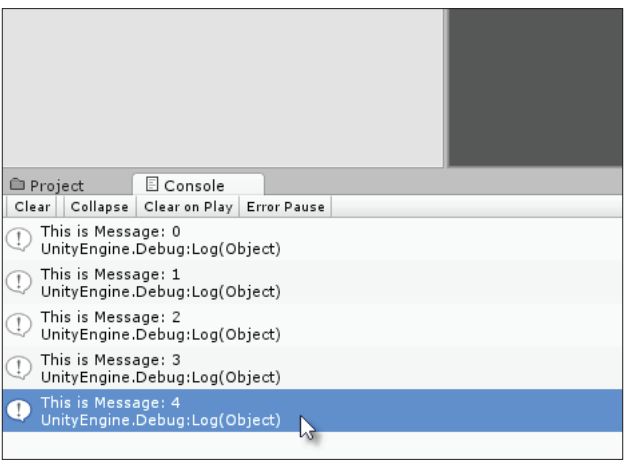


Рис. 1.9. Вывод сообщений в консоль в цикле while

Более подробную информацию о циклах `while` и их использовании в C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/2aeyhxcd.aspx.

Бесконечные циклы

Одной из опасностей при использовании циклов является возможность случайного создания бесконечного цикла, то есть цикла, который не может закончиться. Особенно это характерно для циклов `while`. Когда игра входит в бесконечный цикл, она обычно зависает и ее приходится прерывать принудительно. Иногда даже вход в бесконечный цикл может привести к краху системы! Часто Unity успевает вовремя обнаружить проблему и прервать сценарий, но не полагайтесь на это. Например, если из листинга 1.7 убрать строку 19, получится бесконечный цикл, поскольку значение переменной `NumberOfMessages` никогда не достигнет величины, удовлетворяющей условию выхода из цикла. Основной посыл данного раздела можно выразить словами: «Будьте осторожны, используя циклы, чтобы избежать зацикливания». Ниже приведен еще один классический пример бесконечного цикла, который, безусловно, может стать источником проблем в вашей игре, поэтому постарайтесь избежать их:

```
// Бесконечный цикл
while(true)
{
}
```

Однако, хотите верьте, хотите нет, но иногда правильно подготовленные бесконечные циклы просто необходимы! Если вам понадобится платформа, постоянно движущаяся вверх и вниз, непрерывно крутящийся волшебный шар или повторяющийся цикл смены дня и ночи, реализовать их вам поможет правильно организованный бесконечный цикл. Далее в этой книге мы рассмотрим примеры, где с успехом можно использовать бесконечные циклы. Циклы – мощный и удобный инструмент, но при неправильном применении они могут стать источником аварий, зависаний и проблем с производительностью, поэтому будьте аккуратнее. В этой книге мы еще рассмотрим примеры успешного применения циклов на практике.

Функции

В этой главе мы уже использовали функции, такие как `Start` и `Update`. Однако только сейчас пришло время рассмотреть их более формально и подробно. В сущности, функция представляет собой множество операторов, объединенных в единый смысловой блок, которому присваивается имя и который можно выполнять по требованию. При реализации логики игры бывают ситуации, когда нужно выполнять

с объектами некоторые повторяющиеся действия, такие как выстрелы из оружия, прыжки в воздух, уничтожение врагов, обновление счета или воспроизведение звука. Можно просто копировать и вставлять соответствующий фрагмент кода везде, где он понадобился, но это очень нехорошая привычка. Гораздо проще оформить повторяемый код в виде функции и вызывать ее в нужные моменты по имени, как показано в листинге 1.8:

Листинг 1.8. Функции

```
using UnityEngine;
using System.Collections;

public class MyScriptFile : MonoBehaviour
{
  // Закрытая переменная для подсчета очков
  // Доступна только в данном классе
  private int Score = 0;

  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Вызвать функцию обновления счета
    UpdateScore(5, false);   // Прибавить пять очков
    UpdateScore (10, false); // Прибавить десять очков
    // Прибавить пятнадцать очков и сохранить результат
    int CurrentScore = UpdateScore (15, false);

    // Удвоить число очков
    UpdateScore(CurrentScore);
  }

  // Вызывается при отображении каждого кадра
  void Update ()
  {
  }

  // Обновляет счет
  public int UpdateScore(int AmountToAdd, bool PrintToConsole=true)
  {
    // Прибавить число очков к счету
    Score += AmountToAdd;

    // Вывести в консоль?
    if(PrintToConsole){Debug.log ("Score is: "+Score.ToString());}

    // Завершить функцию и вернуть текущий счет
    return Score;
  }
}
```

Ниже приводится несколько комментариев к листингу 1.8:

- **Строка 08**: объявление локальной целочисленной переменной `Score` для хранения счета. Эта переменная будет далее в функции `UpdateScore`.
- **Строки 11, 24 и 29**: класс `MyScriptFile` содержит три функции (чаще их называют методами, или функциями-членами): `Start`, `Update` и `UpdateScore`. Функции `Start` и `Update` являются специальными функциями и предоставляются для нужд Unity, с этими функциями мы познакомимся чуть ниже. Функция `UpdateScore` – это обычная функция класса `MyScriptFile`.
- **Строка 29**: функция `UpdateScore` – это цельный блок кода между строками 30 и 39. Она должна вызываться при каждом изменении счета в игре. Когда происходит вызов этой функции, выполняется блок кода (строки 30–39). Таким образом, функция обеспечивает повторное использование кода.
- **Строки 14–20**: функция `UpdateScore` несколько раз вызывается из функции `Start`. При каждом таком вызове, функция `Start` приостанавливает выполнение до завершения вызванной функции `UpdateScore`, а затем продолжает со следующей строки.
- **Строка 29**: функция `UpdateScore` принимает два параметра, или аргумента. Это целочисленный параметр `AmountToAdd` и логический параметр `PrintToConsole`. Параметры действуют как устройства ввода, которые можно подключать к функциям, чтобы оказывать влияние на их работу. Значение параметра `AmountToAdd` определяет число очков, прибавляемых к текущему значению переменной `Score`, а значение параметра `PrintToConsole` определяет необходимость вывода значения переменной `Score` в консоль. Количество аргументов функции теоретически не ограничено, более того, функция может вообще не иметь аргументов, как функции `Start` и `Update`.
- **Строки 32–35**: собственно, здесь изменяется счет и выводится в окно консоли, если необходимо. Обратите внимание, что параметр `PrintToConsole` получает значение по умолчанию `true`, указанное в объявлении функции, в строке 29. Это делает аргумент необязательным при вызове функции. В строках 14, 15, и 17 значение по умолчанию переопределяется и в параметре `PrintToConsole` явно передается значение `false`. В строке 20, напротив, второй аргумент опущен и поэтому в параметре `PrintToConsole` передается значение по умолчанию `true`.

- **Строки 29 и 38**: функция `UpdateScore` возвращает значение, тип которого указан в строке 29 перед именем функции. В данном случае `UpdateScore` возвращает значение типа `int`. То есть, по завершении функция вернет целое число – текущий счет. Выход с возвратом значения осуществляет оператор `return` в строке 38. Функции не обязаны возвращать значения, если в этом нет необходимости. Если не требуется возвращать значение, тип возвращаемого значения определяется как `void`, что и сделано в объявлениях функций `Start` и `Update`.

Более подробную информацию о функциях и их использовании в C# можно найти по адресу http://csharp.net-tutorials.com/basics/functions/.

События

События, по сути, являются функциями, используемыми особым способом. Уже знакомые нам функции `Start` и `Update` – не что иное, как специальные события Unity. События – это функции, предназначенные для уведомления объекта о значительных происшествиях: запуске уровня, начале нового кадра, уничтожении врага, прыжке игрока и др. Функции событий вызываются в критические моменты, давая объекту шанс адекватно отреагировать, если это необходимо. Функция `Start` вызывается автоматически при создании объекта, как правило, это происходит при запуске уровня. Функция `Update` также вызывается автоматически в начале каждого кадра. Функция `Start` дает возможность выполнить некоторые действия при запуске уровня, а функция `Update` вызывается для каждого кадра, несколько раз в секунду. Поэтому функция `Update` особенно полезна для создания анимационных эффектов в играх. В листинге 1.9 функция `Update` обеспечивает вращение объекта с течением времени.

Листинг 1.9. Простой анимационный эффект – вращение объекта

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyScriptFile : MonoBehaviour
05 {
06   // Этот метод выполняет инициализацию
07   void Start ()
08   {
09   }
10
11   // Вызывается при отображении каждого кадра
```

```
12   void Update ()
13   {
14     // Повернуть объект на 2 градуса вокруг оси Y
15     transform.Rotate(new Vector3(0.0f, 2.0f, 0.0f));
16   }
17 }
```

Оператор в строке 15, в листинге 1.9, вызывается один раз в начале каждого кадра и обеспечивает непрерывное вращение объекта со скоростью 2 градуса за кадр вокруг оси *y*. Производимый эффект зависит от частоты кадров, а это значит, что объект будет вращаться быстрее на компьютерах, способных обеспечить большую частоту кадров, так как функция `Update` будет вызываться чаще. Существуют методы, позволяющие добиться независимости от частоты кадров и гарантирующие одинаковую работу игр на всех компьютерах, независимо от частоты кадров. Мы рассмотрим их в следующей главе. Вы можете узнать частоту кадров для своей игры прямо во вкладке **Game** (Игра), в окне редактора Unity. Выберите вкладку **Game** (Игра) и щелкните на кнопке **Stats** (Статистики) в правом верхнем углу панели инструментов. При этом откроется панель статистики **Stats**, с общей статистической информацией о производительности игры. Эта панель отображает **частоту кадров в секунду** (**Frames Per Second**, **FPS**) игры, которая определяет и как часто вызывается функция `Update`, и общую производительность игры в данной системе. В общем случае, если частота кадров в секунду ниже 15, это указывает на значительные проблемы с производительностью. Нужно стремиться к частоте кадров 30 или выше. На рис. 1.10 показано, как вызвать панель **Stats** (Статистики).

Число типов событий слишком велико, чтобы перечислить их здесь все. Тем не менее, некоторые общие события Unity, такие как `Start` и `Update`, можно найти в классе `MonoBehaviour`. Более подробную информацию о классе `MonoBehaviour` можно найти по адресу http://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

Классы и объектно-ориентированное программирование

Класс представляет собой смешение многих взаимосвязанных переменных и функций, собранных в отдельный модуль или «сущность». Иными словами, представьте игру (например, ролевую игру в стиле фэнтези), наполненную множеством независимых сущностей, таких как волшебники, орки, деревья, дома, игрок, рыцари, предметы, оружие, заклинания, дверные проемы, мосты, силовые поля, порталы, ох-

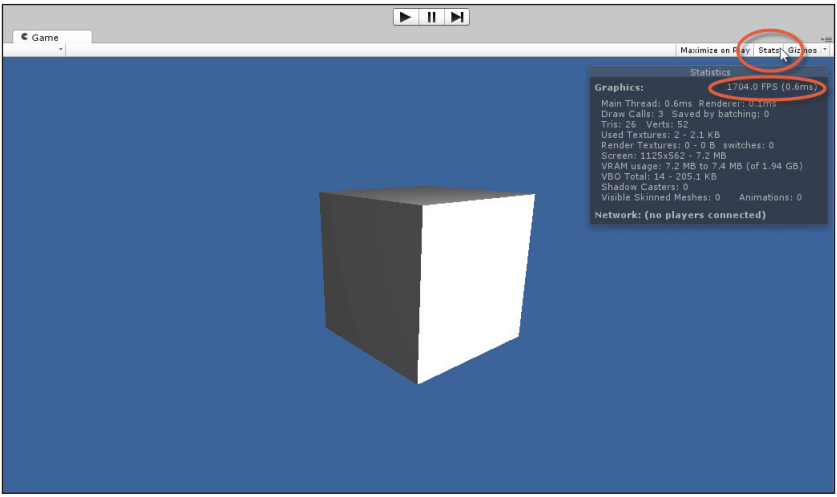


Рис. 1.10. Доступ к панели **Stats** во вкладке **Game** для просмотра частоты кадров в секунду

ранники и т. д. Многие из этих объектов соответствуют объектам в реальном мире. Но самое важное, что каждая из этих сущностей является независимым объектом, волшебник является отдельной сущностью, отличной от силового поля; охранник является отдельной сущностью, отличной от дерева. Каждую из сущностей можно рассматривать как объект определенного типа. Если сосредоточить внимание на одном конкретном объекте, например вражеском орке, можно определить свойства и поведение этого объекта. Орк будет иметь определенную позицию и размеры, и повернут в определенном направлении – каждой из этих характеристик будет соответствовать своя переменная.

Орк может также владеть несколькими видами нападения, например в ближнем бою с топором и дальнем бою с арбалетом. Эти нападения реализуются с помощью функций. Таким образом, наборы переменных и функций соединяются вместе важными связями. Этот процесс объединения называется инкапсуляцией. В этом примере свойства и поведение орка инкапсулированы в класс. Класс в данном случае представляет шаблон общего, абстрактного орка (концепция орка). Объекты – напротив, частные, конкретные экземпляры класса `Orc`. Файл сценария в Unity определяет класс. Чтобы создать экземпляр этого класса в уровне, нужно добавить его в `GameObject`. Как мы видели, классы подключаются к игровым объектам в виде компонен-

тов. Компоненты – это объекты, а множество компонентов образует GameObject. Следующий листинг 1.10 содержит набросок класса Orc.

Листинг 1.10. Класс Orc

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Orc : MonoBehaviour
05 {
06   // Ссылка на трансофрмацию объекта орка
07   // (позиция, направление, масштаб)
08   private Transform ThisTransform = null;
09
10   // Перечисление с набором состояний орка
11   public enum OrcStates {NEUTRAL, ATTACK_MELEE, ATTACK_RANGE};
12
13   // Текущее состояние орка
14   public OrcStates CurrentState = OrcStates.NEUTRAL;
15
16   // Скорость перемещения в метрах в секунду
17   public float OrcSpeed = 10.0f;
18
19   // Дружественность к игроку
20   public bool isFriendly = false;
21
22   //-----------------------------------------------
23   // Этот метод выполняет инициализацию
24   void Start ()
25   {
26     // Получить трансформацию орка
27     ThisTransform = transform;
28   }
29   //-----------------------------------------------
30   // Вызывается при отображении каждого кадра
31   void Update ()
32   {
33   }
34   //-----------------------------------------------
35   // Действия орка в разных состояниях
36   public void AttackMelee()
37   {
38     // Реализация приемов ближнего боя
39   }
40   //-----------------------------------------------
41   public void AttackRange()
42   {
43     // Реализация приемов дальнего боя
44   }
45   //-----------------------------------------------
46 }
```

Ниже приводится несколько комментариев к листингу 1.10:

- **Строка 04**: здесь с помощью ключевого слова `class` определяется класс `Orc`. Этот класс является производным от `MonoBehaviour`. В следующем разделе этой главы мы подробнее рассмотрим наследование и производные классы.
- **Строки 10–20**: в класс `Orc` добавлено несколько переменных и перечисление. Переменные имеют разные типы, но они связаны с общей концепцией орка.
- **Строки 36–46**: орк имеет два метода: `AttackMelee` и `AttackRange`.

Более подробную информацию о классах и их использовании в C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/x9afc042.aspx.

Классы и наследование

Представьте, что вы создали класс `Orc` для игровых объектов и потом решили создать еще два модернизированных типа орков: орк-военачальник, с улучшенными броней и оружием и орк-маг, который, как следует из названия, использует заклинания. Оба могут делать все, что обычный орк, и кое-что еще. Для этого можно создать три отдельных класса `Orc`, `OrcWarlord` и `OrcMage` путем копирования общего кода.

Проблема в том, что орк-военачальник и орк-маг имеют много общего с обычным орком и, следовательно, придется повторить массу общего кода, что само по себе является ненужным расточительством. Кроме того, если обнаружится ошибка в общем коде в одном из классов, вам придется скопировать исправления в другие классы. Это и утомительно, и технически не оправдано, так как вы зря потратите время и можете допустить ошибку, создав ненужные сложности. Решить эту проблему вам поможет объектно-ориентированная концепция наследования. Наследование позволяет создать совершенно новый класс, неявно включающий функциональность другого класса, то есть, создать новый класс, который расширяет существующий, не затрагивая при этом оригинала. При наследовании между двумя классами образуются определенные взаимоотношения. Оригинальный класс (например, класс `Orc`) называется суперклассом, классом-предком, или родительским классом. Новый класс (например, `OrcWarlord` или `OrcMage`), расширяющий родительский класс, называется дочерним или производным классом.

Более подробную информацию о наследовании в C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/ms173149%28v=vs.80%29.aspx.

По умолчанию каждый новый файл сценария Unity создает новый класс, производный от класса MonoBehaviour. Это значит, что каждый новый сценарий содержит весь функционал класса MonoBehaviour и потенциально расширяет его с помощью дополнительного кода. Чтобы доказать это, обратимся к листингу 1.11.

Листинг 1.11. Наследование

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class NewScript : MonoBehaviour
05 {
06   //-------------------------------------------------
07   // Этот метод выполняет инициализацию
08   void Start ()
09   {
10     name = "NewObject";
11   }
12   //-------------------------------------------------
13   // Вызывается при отображении каждого кадра
14   void Update ()
15   {
16   }
17 }
```

Ниже приводится несколько комментариев к листингу 1.11:

- **Строка 04**: класс NewScript порожден от класса MonoBehaviour. Впрочем, имя MonoBehaviour можно заменить на любое действительное имя класса, свойства и методы которого должен унаследовать ваш класс.
- **Строка 10**: здесь переменной name внутри события Start присваивается строка. Обратите внимание, что переменная name не объявлена явно в исходном файле NewScript. Если бы NewScript был совершенно новым классом, не имеющим предка, компилятор обнаружил бы ошибку в строке 10. Но, из-за того, что класс NewScript порожден от класса MonoBehaviour, он автоматически наследует все его переменные, что позволяет обращаться к ним в классе NewScript.

Когда применять наследование. Используйте наследование, только если это действительно необходимо; в противном случае вы сделаете ваши классы большими, тяжелыми и запутанными. Если новый класс должен иметь много общих функций с другим классом, имеет смысл установить связь между ними и использовать наследование. Еще одно применение наследования, как мы увидим далее, заключается в переопределении функций.

Классы и полиморфизм

Иллюстрацию полиморфизма в C# начнем с рассмотрения примера в листинге 1.12. Этот пример не является прямой демонстрацией полиморфизма, а представляет исходную ситуацию, где может пригодиться полиморфизм. Он содержит базовый класс персонажа, **не являющегося игроком (non-player character**, сокращенно **NPC)** в типичной ролевой игре (RPG). Класс намеренно сделан неполным и содержит только самые необходимые переменные, намечающие основу для создания персонажа. Самое главное здесь, что класс имеет функцию `SayGreeting`, которая должна вызваться, когда игрок начинает разговор с персонажем. Эта функция выводит в консоль обычное приветствие, как показано ниже:

Листинг 1.12

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyCharacter
05 {
06   public string CharName = "";
07   public int Health = 100;
08   public int Strength = 100;
09   public float Speed = 10.0f;
10   public bool isAwake = true;
11
12   // Приветствовать игрока при вступлении в диалог
13   public virtual void SayGreeting()
14   {
15     Debug.log ("Hello, my friend");
16   }
17 }
```

Если попытаться представить, как он будет работать в игре, сразу же возникает первая проблема, связанная с разнообразием применений и реалистичности поведения. В частности, все персонажи, созданные как экземпляры класса `MyCharacter`, будут отвечать одним и тем же приветствием при вызове функции `SayGreeting`: мужчины, женщины, орки и все остальные. Все они скажут одно и то же, а именно: «Hello, my friend» («Привет, мой друг»). Это неправдоподобно и нежелательно. Проще всего было бы добавить в класс общедоступную переменную, позволив тем самым определить желаемый текст сообщения при настройке объекта. Но, чтобы показать полиморфизм в действии, попробуем другое решение. Можно было бы создать не-

сколько дополнительных классов, наследующих MyCharacter, по одному для каждого типа персонажей, и каждый класс мог бы выводить уникальное приветствие при вызове функции SayGreeting. В случае с классом MyCharacter это вполне возможно, потому что функция SayGreeting объявлена с ключевым словом virtual (строка 13). Благодаря этому производные классы смогут переопределять поведение функции SayGreeting из класса MyCharacter. Это означает, что функции SayGreeting в производных классах будут подменять исходную функцию в базовом классе. Такое решение приводится в листинге 1.13.

Листинг 1.13. Переопределение функции

```
01 using UnityEngine;
02 using System.Collections;
03 //-----------------------------------------
04 public class MyCharacter
05 {
06   public string CharName = "";
07   public int Health = 100;
08   public int Strength = 100;
09   public float Speed = 10.0f;
10   public bool isAwake = true;
11
12   // Приветствовать игрока при вступлении в диалог
13   public virtual void SayGreeting()
14   {
15     Debug.log ("Hello, my friend");
16   }
17 }
18 //-----------------------------------------
19 public class ManCharacter: MyCharacter
20 {
21   public override void SayGreeting()
22   {
23     Debug.log ("Hello, I'm a man");
24   }
25 }
26 //-----------------------------------------
27 public class WomanCharacter: MyCharacter
28 {
29   public override void SayGreeting()
30   {
31     Debug.log ("Hello, I'm a woman");
32   }
33 }
34 //-----------------------------------------
35 public class OrcCharacter: MyCharacter
36 {
```

```
37   public override void SayGreeting()
38   {
39     Debug.log ("Hello, I'm an Orc");
40   }
41 }
42 //----------------------------------------
```

Здесь сделаны некоторые улучшения: созданы разные классы для каждого типа персонажей, а именно `ManCharacter`, `WomanCharacter` и `OrcCharacter`. Все они выводят разные приветствия в своих функциях `SayGreeting`. Кроме того, каждый персонаж наследует общее поведение от базового класса `MyCharacter`. Однако возникает техническая проблема, связанная со специфичностью классов. Представьте таверну, внутри которой расположились несколько персонажей разных типов, попивающих грог из кружек. Когда игрок входит в таверну, все персонажи должны вывести свое уникальное приветствие. Для этого было бы здорово иметь массив всех персонажей и просто вызывать функции `SayGreeting` элементов массива в цикле, чтобы каждый персонаж вывел свое приветствие. Но, похоже, сейчас мы не можем сделать этого. Это связано с тем, что все элементы в одном массиве должны иметь один и тот же тип данных, например `MyCharacter[]` или `OrcCharacter[]`. Мы не можем смешивать типы данных в одном массиве. Можно было бы объявить несколько массивов, по одному для каждого типа персонажей, но это не лучшее решение, и оно не позволит легко добавлять новые типы персонажей, после того как код обработки массивов будет написан. Чтобы решить эту проблему, нужно другое решение. В этом нам поможет полиморфизм. Посмотрите на следующий пример в листинге 1.14, где определяется новый класс `Tavern` в отдельном файле сценария.

Листинг 1.14. Класс Tavern

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class Tavern : MonoBehaviour
05 {
06   // Массив персонажей в таверне
07   public MyCharacter[] Characters = null;
08   //-------------------------------------------------------
09   // Этот метод выполняет инициализацию
10   void Start () {
11
12     // Массив может содержать до 5 персонажей
13     Characters = new MyCharacter[5];
```

```
14
15      // Добавить персонажей разных типов в массив MyCharacter
16      Characters[0] = new ManCharacter();
17      Characters[1] = new WomanCharacter();
18      Characters[2] = new OrcCharacter();
19      Characters[3] = new ManCharacter();
20      Characters[4] = new WomanCharacter();
21
22      // Теперь игрок входит в таверну
23      EnterTavern();
24    }
25    //-----------------------------------------------------
26    // Вызывается, когда игрок входит в таверну
27    public void EnterTavern()
28   {
29      // Все приветствуют игрока
30      foreach(MyCharacter C in Characters)
31      {
32        // вызов SayGreeting в дочернем классе
33        // Дочерний класс доступен через базовый класс
34        C.SayGreeting();
35      }
36    }
37    //-----------------------------------------------------
38  }
```

Ниже приводится несколько комментариев к листингу 1.14:

- **Строка 07**: для хранения ссылок на всех персонажей в таверне, независимо от их типов, объявляется массив (`Characters`) типа `MyCharacter[]`.
- **Строки 16–20**: массив `Characters` заполняется несколькими персонажами разных типов. Это возможно, потому что персонажи разных типов происходят от одного и того же базового класса.
- **Строка 27**: функция `EnterTavern` вызывается при запуске уровня.
- **Строка 34**: цикл `foreach` перебирает персонажей в массиве и для каждого вызывает функцию `SayGreeting`. Результат работы примера показан на рис. 1.11. Вместо общего сообщения, определенного в базовом классе, выводятся уникальные сообщения для каждого персонажа. Полиморфизм позволяет заменить метод базового класса в производных классах.

Более подробную информацию о полиморфизме в C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/ms173152.aspx.

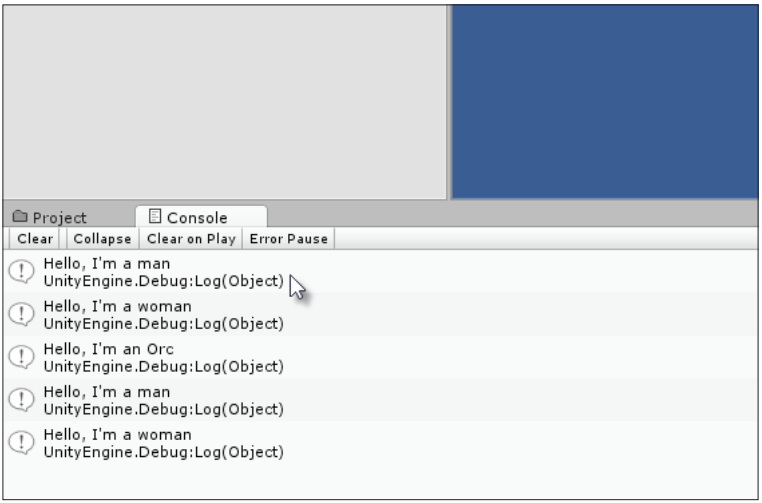


Рис. 1.11. Полиморфизм обеспечивает обратную связь для типов, имеющих общую родословную

Свойства в C#

При присваивании значений переменным класса, например, `MyClass.x = 10;`, есть пара важных аспектов, которым следует уделить внимание. Во-первых, как правило, необходимо проверять присваиваемое значение на соответствие заданным требованиям. Типичными случаями являются ограничение целого значения минимальным и максимальным пределами или ограничение ввода для строковой переменной определенным набором строк. Во-вторых, может потребоваться определить момент изменения значения переменной для инициализации других зависимых функций. Свойства C# позволяют обеспечить обе эти возможности. В листинге 1.15 реализованы ограничение значений целой переменной диапазоном от 1 до 10 и вывод сообщения при его изменении.

Листинг 1.15. Свойства

```
01 using UnityEngine;
02 using System.Collections;
03 //-------------------------------------------------------
04 // Пример класса – пожет подключаться к игровому объекту
05 public class Database : MonoBehaviour
06 {
```

```
07   //-------------------------------------------------------
08   // Общедоступное свойство для закрытой переменной iMyNumber
09   // Это общедоступное свойство для закрытой переменной iMyNumber
10   public int MyNumber
11   {
12     // Вызывается при попытке извлечь значение
13     get
14     {
15       return iMyNumber; // Вернет значение iMyNumber
16     }
17
18     // Вызывается при попытке присвоить значение
19     set
20     {
21       // Если значение в диапазоне 1-10, привоить его
22       if(value >= 1 && value <= 10)
23       {
24         // Изменить закрытую переменную
25         iMyNumber = value;
26
27         // Вызвать событие
28         NumberChanged();
29       }
30     }
31   }
32   //-------------------------------------------------------
33   // Внутренняя переменная, число в диапазоне 1-10
34   private int iMyNumber = 0;
35   //-------------------------------------------------------
36   // Этот метод выполняет инициализацию
37   void Start ()
38   {
39     // Присвоить число свойству MyNumber
40     MyNumber = 11; // Ничего не произойдет, потому что число > 10
41
42     // Присвоить число свойству MyNumber
43     MyNumber = 7; // Изменится, так как число в диапазоне 1-10
44   }
45   //-------------------------------------------------------
46   // Событие, вызываемое при изменении iMyNumber
47   void NumberChanged()
48   {
49     Debug.log("Variable iMyNumber changed to: "+iMyNumber.ToString());
50   }
51   //-------------------------------------------------------
52 }
53 //---------------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 1.15:

- **Строка 10**: объявление целочисленного общедоступного свойства. Данное свойство не является независимой переменной,

это лишь обертка, или интерфейс доступа к локальной переменной `iMyNumber`, объявленной в строке 34.

- **Строка 13**: при попытке чтения из свойства `MyNumber` будет вызвана внутренняя функция `get`.
- **Строка 14**: при попытке присвоить значение свойству `MyNumber` будет вызвана внутренняя функция `set`.
- **Строка 25**: функция `set` в качестве неявного аргумента принимает присваиваемое значение.
- **Строка 28**: при изменении значения переменной `iMyNumber` будет вызвано событие `NumberChanged`.

Свойства и Unity. Свойства особенно удобны для проверки значений, присваиваемых переменным. Основной проблемой свойств в Unity является их невидимость в инспекторе объектов. То есть, свойства C# не отображаются в инспекторе объектов. Вы не сможете получать или присваивать им значения в редакторе. Однако, сообществом Unity создано множество сценариев и решений, изменяющих такое поведение по умолчанию и делающих свойства C# видимыми в редакторе. Эти сценарии и решения можно найти по адресу http://wiki.unity3d.com/index.php?title=Expose_properties_in_inspector.

Более подробную информацию о свойствах в C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/x9fsa0sw.aspx.

Комментарии

Комментарии – это сообщения в коде, поясняющие, что этот код делает. В C# однострочные комментарии начинаются с пары символов `//`, а многострочные –начинаются с символов `/*` и заканчиваются символами `*/`. Комментарии используются во всех примерах в этой книге. Комментарии важны, и я рекомендую взять в привычку писать их, если вы еще не имеете такой привычки. Они пригодятся не только другим разработчикам вашей команды (если вы работаете не одни), но и вам тоже! Они напомнят, что делает код, когда вы вернетесь к нему спустя несколько недель или месяцев, и даже помогут достичь ясности в понимании своего кода, который вы пишете прямо сейчас. Конечно, все эти преимущества зависят от того, пишете вы краткие и содержательные комментарии или длинные и пространные эссе. Кроме того, MonoDevelop дает возможность писать комментарии, основанные на XML, описывающие функции и аргументы, и интегрирующиеся с подсказками, которые может выводить редактор. Эти комментарии помогут ускорить рабочий процесс, особенно когда вы работаете в команде. Давайте посмотрим, как их использовать. Начнем с произвольной функции, представленной на рис. 1.12.

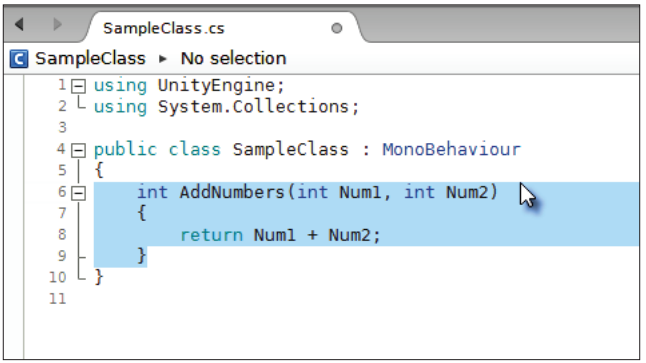


Рис. 1.12. Ввод кода функции `AddNumbers` в редакторе MonoDevelop (подготовка к комментированию кода)

Затем вставим три символа косой черты (///) над строкой с заголовком функции, как показано на рис. 1.13.

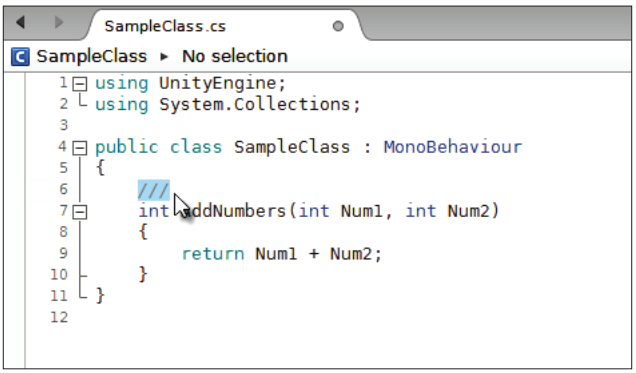


Рис. 1.13. Добавление /// над заголовком функции для создания XML-комментария

После этого MonoDevelop автоматически вставит готовый шаблон XML-комментария, который вы можете заполнить соответствующими сведениями. Этот шаблон содержит сводный раздел для общего описания и записи для каждого параметра функции, как показано на рис. 1.14.

Заполните шаблон описанием функции. Удостоверьтесь, что каждый параметр получил соответствующий комментарий, как показано на рис. 1.15.

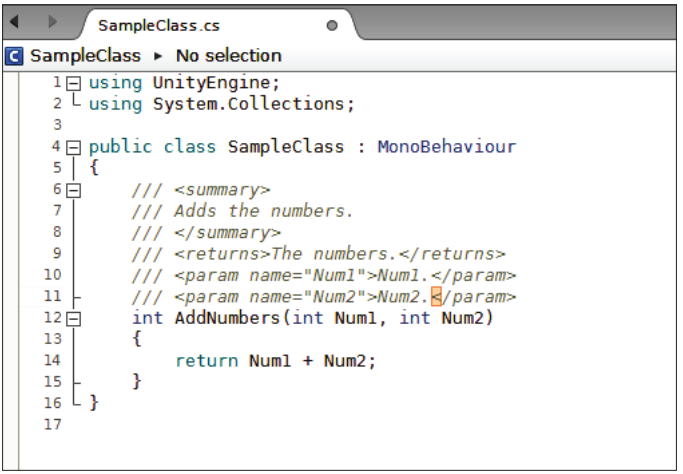


Рис. 1.14. Добавление /// над заголовком функции привело к созданию XML-комментария

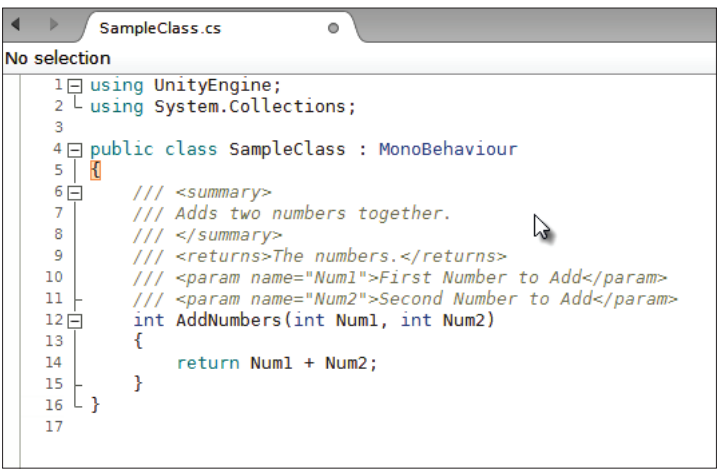


Рис. 1.15. Описание функций с помощью XML-комментариев

Если теперь в редакторе попытаться написать код вызова функции `AddNumbers`, на экране появится всплывающее окно с общим описанием функции и контекстной подсказкой для каждого параметра, как показано на рис. 1.16.

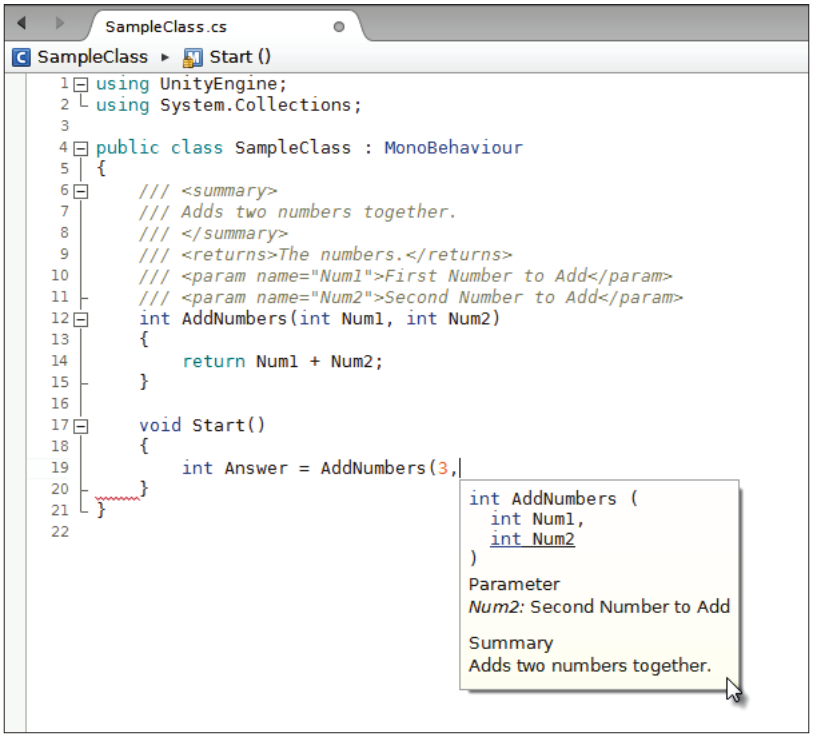


Рис. 1.16. Подсказка с описанием функции при попытке написать ее вызов

Видимость переменных

Одной из замечательных особенностей Unity является функция отображения общедоступных переменных в инспекторе объектов редактора Unity, позволяющая просматривать и редактировать значения переменных даже во время выполнения. Это особенно удобно для отладки. Однако, по умолчанию инспектор объектов не отображает значения локальных переменных. Они, как правило, скрыты от просмотра. Это не всегда удобно, потому что часто бывает желательно иметь возможность изменить или хотя бы проверить локальную переменную в инспекторе объектов, не меняя область их видимости. Есть два простых решения этой проблемы.

Первое позволяет просматривать все общедоступные и закрытые переменные класса. Инспектор объектов можно перевести в режим

отладки. Для этого щелкните на значке контекстного меню в правом верхнем углу окна инспектора и выберите пункт **Debug** (Отладка), как показано на рис. 1.17. В результате будут отображены все общедоступные и закрытые переменные класса.

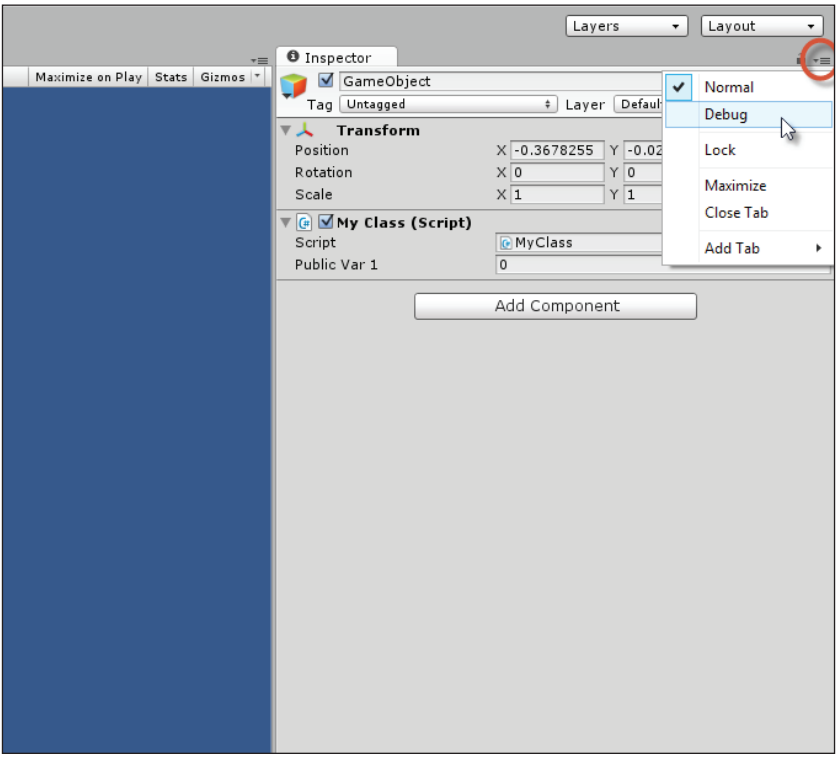


Рис. 1.17. Включение режима отладки в инспекторе объектов

Второе позволяет увидеть конкретные закрытые переменные, явно отмеченные для отображения в инспекторе объектов. Они будут видны в обоих режимах – **Normal** (Обычный) и **Debug** (Отладка). Для этого объявите закрытую переменную с атрибутом `[SerializeField]`. Атрибуты C# будут рассматриваться далее в этой книге. Ниже приводится пример объявления закрытой переменной с атрибутом `[SerializeField]`:

```
01 using UnityEngine;
02 using System.Collections;
```

```
03
04 public class MyClass : MonoBehaviour
05 {
06   // Всегда будет отображаться в инспекторе объектов
07   public int PublicVar1;
08
09   // Всегда будет отображаться в инспекторе объектов
10   [SerializeField]
11   private int PrivateVar1;
12
13   // Будет отображаться только в режиме отладки
14   private int PrivateVar2;
15
16   // Будет отображаться только в режиме отладки
17   private int PrivateVar3;
18 }
```

Можно также использовать атрибут `[HideInInspector]` для отключения отображения общедоступных переменных в инспекторе.

Оператор ?

Оператор `if-else` так часто используется в C#, что для него была создана специальная сокращенная форма записи, более простая и не занимающая нескольких строк кода, как полная форма записи оператора `if-else`. Эта сокращенная форма называется оператором `?`. Ниже приводится базовый синтаксис оператора:

```
// Если условие истинно, выполняется выражение_1, иначе выражение_2
(условие) ? выражение_1 : выражение_2;
```

Рассмотрим применение оператора `?` на практическим примере:

```
// Объект следует скрыть, если его координата Y
// получила значение выше 100
bool ShouldHideObject = (transform.position.y > 100) ? true : false;

// Изменить видимость объекта
gameObject.SetActive(!ShouldHideObject);
```

Оператор `?` удобно использовать для коротких инструкций, но при использовании с длинными и сложными операторами, он может сделать код трудно читаемым.

Методы SendMessage и BroadcastMessage

Класс `MonoBehaviour`, входящий в состав в Unity API и используемый в качестве базового для большинства сценариев, предоставляет методы `SendMessage` и `BroadcastMessage`. С их помощью можно вызвать по

именам функции в любых компонентах, прикрепленных к объекту. Для вызова метода класса обычно нужна локальная ссылка на этот класс, чтобы получить к нему доступ и вызывать его функции, что верно и переменных. Методы `SendMessage` и `BroadcastMessage`, напротив, позволяют вызывать функции по их именам, которые передаются в виде строковых значений. Это очень удобно и делает код намного проще и короче, как показано ниже, в листинге 1.16.

Листинг 1.16. Вызов функций по именам в виде строк

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class MyClass : MonoBehaviour
05 {
06   void start()
07   {
08     // Вызвать функцию MyFunction во ВСЕХ компонентах/сценариях,
09     // подключенных к объекту (и имеющих эту функцию)
10     SendMessage("MyFunction", SendMessageOptions.DontRequireReceiver);
11   }
12
13   // Вызывается с помощью метода SendMessage
14   void MyFunction()
15   {
16     Debug.log ("hello");
17   }
18 }
```

Ниже приводится несколько комментариев к листингу 1.16:

- **Строка 10**: вызывается метод `SendMessage` для вызова функции `MyFunction`. В данном случае будет вызвана функция `MyFunction` не только в данном классе, но и во всех других компонентах, подключенных к `GameObject`, если они имеют функцию `MyFunction`, включая компонент `Transform` и другие.
- **Строка 10**: аргумент `SendMessageOptions.DontRequireReceiver` определяет, что должно произойти, если компонент не имеет функции `MyFunction`. Он указывает, что такой компонент следует просто игнорировать и должен быть выполнен переход к следующему компоненту.

Термины «функция» и «метод» означают одно и то же, если функция принадлежит классу. Функцию, принадлежащую классу, называют методом класса.

Мы видели, что метод `SendMessage` вызывает заданную функцию во всех компонентах, подключенных к объекту `GameObject`. Метод

BroadcastMessage делает то же, что и метод SendMessage, но в более широком масштабе – он вызывает указанную функцию во всех компонентах объекта GameObject, а затем повторяет этот процесс рекурсивно для всех его дочерних объектов в иерархии сцены, двигаясь вниз по иерархии.

Более подробную информацию о методах SendMessage и BroadcastMessage можно найти по адресам: http://docs.unity3d.com/ScriptReference/GameObject.SendMessage.html и http://docs.unity3d.com/ScriptReference/Component.BroadcastMessage.html.

Рефлексия. Методы SendMessage и BroadcastMessage обеспечивают эффективный способ взаимодействия объектов и компонентов. То есть, они являются отличным способом организации общения компонентов друг с другом, если потребуется синхронизировать их действия или воспользоваться их функциями. Однако методы SendMessage и BroadcastMessage основаны на механизме C# с названием «рефлексия» (reflection). При вызове функции по ее имени в виде строки, приложению требуется осмотреть себя во время выполнения, чтобы найти точку входа в искомую функцию. Этот процесс требует дополнительных вычислительных затрат, по сравнению с вызовом функции в обычном режиме. По этой причине старайтесь не пользоваться методами SendMessage и BroadcastMessage, особенно в событиях Update или в других функциях, частота вызова которых напрямую связана с частотой кадров, так как отрицательное влияние на производительность может быть значительным. Это не значит, что вы никогда не должны использовать их. Существуют ситуации, когда их редкое, нечастое использование удобно и практически не оказывает заметного влияния на производительность. В последующих главах этой книги будут представлены альтернативные и более быстрые приемы с использованием делегатов и интерфейсов.

Если вы интересно узнать больше о языке C# и его использовании, прежде чем продолжить чтение этой книги, я рекомендую следующие источники:

- Терри Нортон (Terry Norton). «Learning C# by Developing Games with Unity 3D Beginner's Guide». Packt Publishing;
- Алан Торн (Alan Thorn). «Intro to C# Programming and Scripting for Games in Unity» (видеокурс по адресу https://www.udemy.com/3dmotive-intro-to-c-programming-and-scripting-for-games-in-unity/);
- Алан Торн (Alan Thorn). «Pro Unity Game Development with C#». Apress.

И еще несколько ресурсов в Интернете:

- http://msdn.microsoft.com/ru-rulibrary/aa288436%28v=vs.71%29.aspx;
- http://www.csharp-station.com/tutorial.aspx;
- http://docs.unity3d.com/ScriptReference/.

Итоги

В этой главе был представлен общий обзор C#, в котором были рассмотрены наиболее распространенные функции языка, широко используемые в разработке игр для Unity. В следующих главах мы вновь вернемся к некоторым из рассмотренных здесь проблем и применим для их решения более продвинутые методы, но все затронутые здесь вопросы будут иметь решающее значение для понимания и написания кода в следующих главах.

Глава 2

Отладка

Отладка – это процесс поиска, идентификации и исправления «жучков» (ошибок или опечаток) в коде. Чтобы эффективно писать сценарии, нужно знать основные приемы отладки и инструменты, поддерживаемые в Unity. Прежде чем начать их рассмотрение, остановимся на общих ограничениях отладки, то есть на том, чего она не может предложить. Отладка – не волшебный эликсир от всех болезней, она не в состоянии избавить код от всех ошибок и гарантировать безошибочную работу приложения. Специалист в области информационных технологий Эдсгер Вибе Дейкстра (Edsger Wybe Dijkstra) сказал: «Тестирование может быть использовано для демонстрации наличия ошибок, но никогда для их отсутствия». То есть, во время тестирования можно столкнуться с одной или несколькими ошибками. Эти ошибки будут выявлены, идентифицированы и исправлены при отладке. И все же, тесты, даже очень обширные и детальные, никогда не охватят всех возможных ситуаций на всех аппаратных платформах при всевозможных условиях, так как число таких комбинаций практически бесконечно. Поэтому, никто не может быть абсолютно уверен, что нашел и исправил все возможные ошибки. Даже в момент выпуска игры она все еще может содержать «жучки», которые не смогли выявить при тестировании. Конечно, в программе может не остаться никаких ошибок, но знать наверняка вы этого не можете. Поэтому отладка не может гарантировать безошибочную работу приложения. Ее цель скромнее. Это систематическое тестирование игры для большинства типичных ситуаций, чтобы найти и исправить столько ошибок, сколько возможно обнаружить, или, по крайней мере, выявить критические ошибки, насколько позволят время, отпущенное на разработку, и бюджет. В любом случае, отладка является важной частью процесса разработки сценариев, потому что без нее вы вообще не будете иметь возможности выявлять и исправлять ошибки. Существует много методов отладки, простых и сложных. В этой главе мы охватим достаточно широкий их диапазон.

Ошибки компиляции и консоль

Под отладкой обычно понимается выявление ошибок, обнаруживающих себя во время выполнения кода, то есть, это поиск и исправление ошибок, возникающих во время выполнения игры. При таком понимании отладки предполагается, что программный код уже скомпилирован и запущен. Это значит, вы правильно записали все операторы, код скомпилировался и вы хотите найти ошибки времени выполнения, которые являются результатом просчетов в логике работы программы. То есть, в центре внимания находится не синтаксис, а логика, и, в общем, так и должно быть. Однако, в этом разделе я очень кратко остановлюсь на компиляции кода, точнее на написании допустимого кода, а также выявлении и исправлении ошибок компиляции, которые проявляются в виде сообщений в консоли. Для этого важно знать основные приемы работы с окном консоли и иметь прочную основу для более глубокого понимания отладки. Рассмотрим пример в листинге 2.1.

Листинг 2.1. Файл сценария ErrorScript.cs с ошибками

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class ErrorScript : MonoBehaviour
05 {
06   int MyNumber = 5;
07
08   // Этот метод выполняет инициализацию
09   void Start () {
10
11     mynumber = 7;
12   }
13
14   // Вызывается при отображении каждого кадра
15   void Update () {
16     mynumber = 10;
17   }
18 }
```

Чтобы скомпилировать пример из листинга 2.1, просто сохраните файл сценария в MonoDevelop (**Ctrl**+**S**), а затем переключитесь из окна редактора Unity и обратно. В момент повторного получения фокуса ввода окном редактора, автоматически запустится компиляция. Если этого не произошло, щелкните правой кнопкой мыши на файле сценария в панели **Project** (Проект) и выберите в контекстном меню

пункт **Reimport** (Импортировать повторно). При компиляции примера из листинга 2.1 будут найдены две ошибки, и соответствующие сообщения появятся в окне консоли. Если окно консоли еще не открыто, откройте его, выбрав в меню приложения пункт **Window** ⇒ **Console** (Окно ⇒ Консоль). Окно консоли играет очень важную роль и постоянно должно присутствовать в интерфейсе. Через это окно Unity общается с вами как с разработчиком. Если в вашем коде имеются ошибки компиляции, Unity сообщит о них через консоль.

Пример в листинге 2.1 сгенерирует две ошибки компиляции, как это показано на рис. 2.1. Они возникли в строках 11 и 16 из-за обращения к несуществующей переменной `mynumber`, хотя в сценарии присутствует переменная `MyNumber` (чувствительность к регистру). Ошибки компиляции имеют критическое значение, потому что делают весь код неработоспособным. Это значит, что вы не сможете запустить игру, пока эти ошибки не будут исправлены.

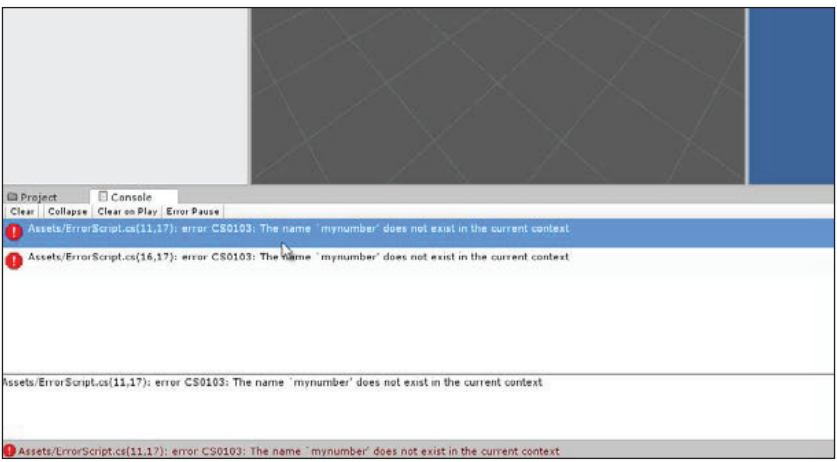


Рис. 2.1. Ошибки компиляции в окне консоли

Если ошибки компиляции не появились в консоли, как это ожидалось, убедитесь, что включен фильтр вывода ошибок. Для этого щелкните на ярлыке фильтра ошибок (изображение красного восклицательного знака) в правом верхнем углу окна консоли. Окно консоли имеет три фильтра: комментариев (**A**), предупреждений (**B**) и ошибок (**C**), как показано на рис. 2.2, для переключения между режимами сокрытия и отображения соответствующих сообщений. Эти

переключатели определяют видимость всех типов сообщений в окне консоли. Комментарии относятся к сообщениям, которые вы, как программист, выводите в окно консоли из программного кода с помощью оператора `Debug.log`. Мы познакомимся с этим приемом чуть ниже (для вывода можно также использовать функцию `Print`). Предупреждения определяют фрагменты кода, потенциально способные вызвать проблемы, или присутствие «мусора». Эти участки кода проходят синтаксический контроль и не вызывают ошибок компиляции, но если предупреждения проигнорировать, они могут создать проблемы при выполнении и привести к непредвиденным результатам или к расточительному использованию ресурсов. Сообщения об ошибках связаны с любыми найденными ошибками компиляции, которые делают компиляцию всего сценария невозможной, как, например, ошибки в листинге 2.1.

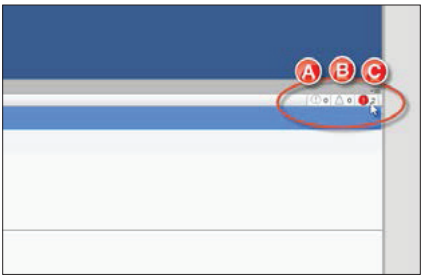


Рис. 2.2. Включение и выключение фильтров в окне консоли

Если в консоль выводится больше одной ошибки, они, как правило, перечисляются в порядке обнаружения компилятором, то есть сверху вниз. Это считается лучшим способом отображения ошибок, потому что более ранние ошибки могут привести к возникновению более поздних. То есть, исправление ранних ошибок может сделать ненужным исправление более поздних. Для исправления ошибки дважды щелкните на сообщении в окне консоли, в результате автоматически откроется редактор MonoDevelop, в котором будет выделена строка с ошибкой, или строка, где ошибка была впервые обнаружена. Отметьте, что MonoDevelop выделит строку, где ошибка была впервые обнаружена, хотя сама ошибка не обязательно будет находиться именно в этой строке. В зависимости от проблемы может понадобиться внести изменения в другую строку, а не в ту, которая была выделе-

на. Если дважды щелкнуть на верхней (первой) ошибке в консоли, появившейся при попытке скомпилировать листинг 2.1, откроется MonoDevelop и выделит строку 11. Данную ошибку можно исправить двумя способами: либо переименовать `mynumber` в `MyNumber` в строке 11, либо переименовать переменную `MyNumber` в `mynumber` в строке 6. Теперь рассмотрим пример в листинге 2.2.

Листинг 2.2. Файл сценария ErrorScript.cs после исправления ошибок

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class ErrorScript : MonoBehaviour
05 {
06   int MyNumber = 5;
07
08   // Этот метод выполняет инициализацию
09   void Start () {
10
11     MyNumber = 7;
12   }
13
14   // Вызывается при отображении каждого кадра
15   void Update () {
16     MyNumber = 10;
17   }
18 }
```

В листинге 2.2 исправлены ошибки, допущенные в листинге 2.1. Но, теперь вместо сообщений об ошибках появляется предупреждение (как показано на рис. 2.3). Оно указывает, что переменная `MyNumber` нигде не используется. Ей присваивается значение в строках 11 и 16, но это значение не используется в приложении. Это предупреждение можно проигнорировать, оно никак не влияет на работоспособность кода. Предупреждения надо рассматривать как рекомендации, сделанные компилятором на основании анализа кода. Как с ними поступить – вам решать, но я рекомендую устранять и ошибки, и предупреждения везде, где это целесообразно.

Отладка с помощью Debug.log – определяемые программистом сообщения

Пожалуй, самым старым и самым известным методом отладки в Unity является использование оператора `Debug.log` для вывода диагностических сообщений в консоль, отражающих ход выполне-

Рис. 2.3. Старайтесь устранять и ошибки и предупреждения

ния программы и текущие свойства объектов. Этот метод является универсальным и доступным, потому что может использоваться практически в любой **интегрированной среде разработки** (Integrated Development Environment, IDE), а не только в MonoDevelop. Кроме того, все объекты Unity, в том числе векторы и объекты, представляющие цвет, имеют удобную функцию `ToString`, позволяющую выводить в консоль значения их членов (таких как *X*, *Y*, и *Z*) в удобочитаемом виде. Для иллюстрации рассмотрим пример в листинге 2.3. Этот пример демонстрирует важный для отладки процесс, а именно вывод сообщения о состоянии при создании экземпляра. Этот сценарий, при прикреплении к объекту сцены, выводит в консоль его координаты вместе с поясняющим сообщением.

Листинг 2.3. Отладочный сценарий

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class CubeScript : MonoBehaviour
05 {
06   // Этот метод выполняет инициализацию
07   void Start () {
08     Debug.log ("Object created in scene at position: " +
09       + transform.position.ToString());
10   }
11 }
```

На рис. 2.4 показан вывод, произведенный этим кодом, после присоединения сценария к объекту игры. Само сообщение, что выводится функцией `Debug.log`, включается в главный список сообщений

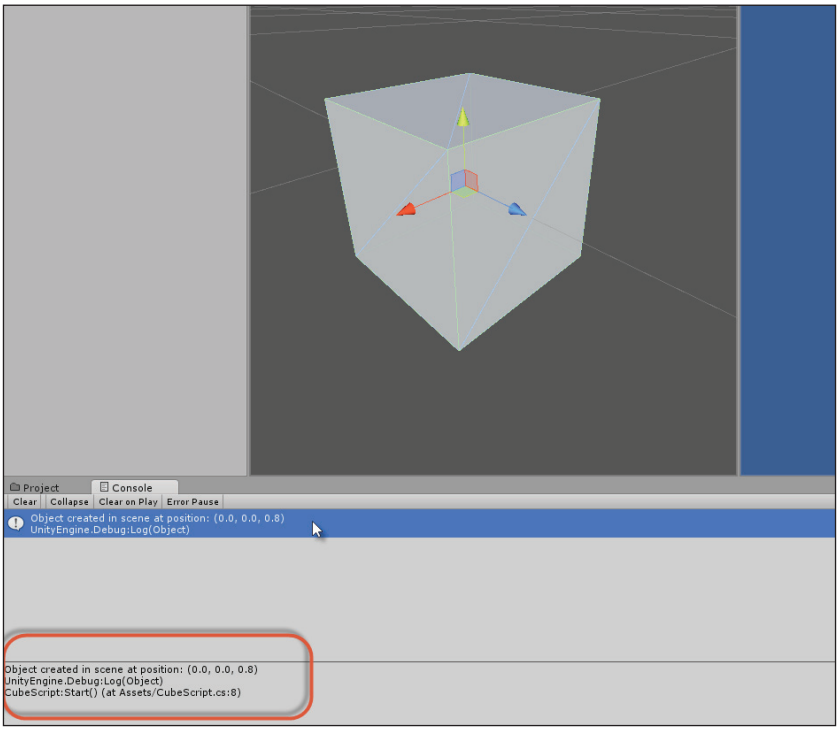


Рис. 2.4. В сообщениях Debug.log объекты можно преобразовывать в строки, и, кроме того, в окне консоли также отображаются имя файла и номер строки

в консоли. Если выбирать это сообщение с помощью мыши, в консоли появятся имя файла сценария и номер строки с оператором.

Основным препятствием к использованию оператора `Debug.log` для отладки являются стремление к сохранению чистоты кода и желание избежать излишней сложности. Во-первых, применение операторов `Debug.log` требует явно добавлять их в сценарии. По окончании отладки вам потребуется либо вручную удалить операторы `Debug.log`, либо оставить их там, что приведет к снижению производительности и к путанице, особенно если вы использовали операторы `Debug.log` во многих местах. Во-вторых, хотя оператор `Debug.log` полезен для решения конкретных проблем и мониторинга значений переменных, в конечном счете этот оператор совершенно не подходит для получения общей картины о ходе выполнения сценария и отслеживания ошибок,

чье присутствие было обнаружено, но местоположение остается неизвестным. Впрочем, из этих критических замечаний не следует, что нужно полностью отказаться от оператора Debug.log. Рассматривайте их как рекомендации по правильному использованию оператора. Оператор Debug.log прекрасно справляется с задачей, когда ошибку или проблему можно проследить от основного подозреваемого объекта и необходимо посмотреть, как изменяются его свойства, особенно в моменты таких событий, как OnStart.

Удаление операторов Debug.log. После окончания разработки игры, не забудьте удалить или закомментировать все операторы Debug.log.

Переопределение метода ToString

Пример в листинге 2.3 демонстрирует удобство метода ToString для отладки, при использовании в сочетании с Debug.log. Метод ToString позволяет преобразовать объект в читаемую строку, которую можно вывести в консоль. В C# каждый класс наследует метод ToString по умолчанию. Это означает, что с помощью наследования и полиморфизма можно переопределить метод ToString, откорректировать его и привести строку в более читаемый вид, точнее отражающий состояние членов класса. Если вы возьмете в привычку переопределять метод ToString в каждом своем классе, их будет гораздо проще отлаживать. Рассмотрим следующий пример в листинге 2.4, где переопределяется метод ToString.

Листинг 2.4. Переопределение метода ToString

```
01 using UnityEngine;
02 using System.Collections;
03 //-----------------------------------------
04 // Пример класса злого огра
05 public class EnemyOgre : MonoBehaviour
06 {
07   //-----------------------------------------
08   // Виды нападений огра
09   public enum AttackType {PUNCH, MAGIC, SWORD, SPEAR};
10   // Текущий вид нападения
11   public AttackType CurrentAttack = AttackType.PUNCH;
12   // Здоровье
13   public int Health = 100;
14   // Задержка перед восстановлением (после повреждения)
15   public float RecoveryTime = 1.0f;
16   // Скорость перемещения огра - в метрах в секунду
17   public float Speed = 1.0f;
```

```
18   // Имя огра
19   public string OgreName = "Harry";
20   //-------------------------------------------
21   // Переопределение метода ToString
22   public override string ToString ()
23   {
24     // Вернуть строку, представляющую объект
25     return string.Format ("***Class EnemyOgre*** OgreName:
26       {0} | Health: {1} | Speed: {2} | CurrentAttack: {3} |
27       RecoveryTime: {4}", OgreName, Health, Speed,
28       CurrentAttack, RecoveryTime);
29   }
30 //-------------------------------------------
31   void Start()
32   {
33     Debug.log (ToString());
34   }
35 //-------------------------------------------
36 }
37 //-------------------------------------------
```

На рис. 2.5 показан вывод в окне консоли, произведенный сценарием.

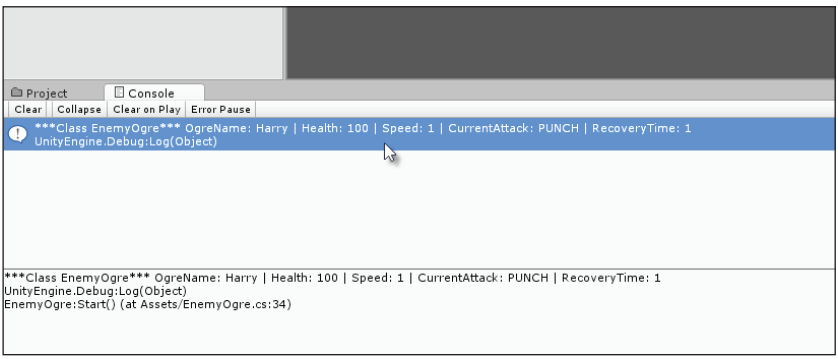


Рис. 2.5. Переопределение метода `ToString` для вывода нестандартного отладочного сообщения

Функция String.Format. Строка 25 в листинге 2.4 использует функцию `String.Format` для создания результирующей строки. Эту функцию удобно использовать, когда нужно получить одну строку, включающую обычный текст и значения переменных различных типов. Лексемы `{0}`, `{1}`, `{2}` ... внутри строкового аргумента функция `String.Format` заменит на следующие далее аргументы функции, порядковые номера которых соответствуют числам в фигурных скобках. Так, подстрока `{0}` будет заменена на резуль-

тат `OgreName.ToString()`. Более подробную информацию о функции `String.Format` можно найти по адресу http://msdn.microsoft.com/ru-ru/library/system.string.format%28v=vs.110%29.aspx.

В Unity имеется возможность выделить фрагменты кода, предназначенные для отладки, что позволит установкой определенного флажка выполнять отладочный код. При отладке игры, например, можно использовать два варианта кода: код для рабочей версии релиза и код для отладки. Представьте, что нужно выявить и исправить ошибку в коде. Для этого вы вставляете операторы `Debug.log`, выводящие значения переменных и свойств классов. Можно даже добавить дополнительные строки кода, операторы `if` и циклы, чтобы проверить альтернативные ситуации и выявить реакцию объектов на них. После внесения изменений в код на какое-то время проблема кажется решенной, вы удаляете дополнительный отладочный код и продолжаете тестирование. Но через некоторое время обнаруживается, что проблема появилась вновь или возникла похожая. Сейчас код отладки снова необходим, и его надо было бы сохранить. Вы обещаете себе в следующий раз закомментировать отладочный код, а не удалять его полностью. Это позволит просто убрать символы комментариев, если код понадобится снова. Но комментировать и раскомментировать код также утомительно, особенно если в нем много строк и они разбросаны по нескольким файлам в разных местах. Решить эту проблему можно с помощью глобальных символов. В сущности, глобальный символ представляет собой специальный флаг, сообщающий препроцессору, какие фрагменты кода он должен включить в процесс компиляции или убрать, то есть обеспечивает компиляцию по условию. При установке флажка в значение `true` Unity автоматически скомпилирует одну версию кода, а при установке в значение `false` – другую. Это позволит вам иметь два варианта вашего кода при одном и том же наборе исходных файлов: один – для отладки и один – для окончательной версии. Давайте посмотрим, как это выглядит на практике. Взгляните на листинг 2.5.

Листинг 2.5. Пример использования глобального символа для отладки

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class CubeScript: MonoBehaviour
05 {
06   // Этот метод выполняет инициализацию
07   void Start ()
08   {
```

```
09      #if SHOW_DEBUG_MESSAGES
10      // выполняется, ТОЛЬКО если символ SHOW_DEBUG_MESSAGES определен
11      Debug.log ("Pos: " + transform.position.ToString());
12      #endif
13
14      // выполняется всегда, так как находится
15      // за пределами блока #if #endif
16      Debug.log ("Start function called");
17    }
18 }
```

Строки 09–12 окружены условными директивами препроцессора `#if` и `#endif`. Эти директивы определяют условия, которые проверяются не во время выполнения кода, как в обычных операторах `if`, а во время компиляции. Во время компиляции Unity проверяет, существует ли глобальный символ `SHOW_DEBUG_MESSAGES`. Если условие выполняется, в результат компиляции будут добавлены строки 10 и 11, в противном случае компилятор проигнорирует эти строки, рассматривая их как комментарии. Используя эту особенность, можно окружить весь отладочный код директивами `#if #endif` и включать и отключать фрагменты кода во всех исходные файлах просто определяя глобальный символ `SHOW_DEBUG_MESSAGES`. Остается только решить, как задать глобальный символ. Чтобы определить глобальный символ, выберите в меню приложения пункт **Edit** ⇒ **Project Settings** ⇒ **Player** (Правка ⇒ Настройки проекта ⇒ Проигрыватель). Введите имя символа в поле **Scripting Define Symbols** (Определить символы для сценариев) и нажмите клавишу **Enter** после ввода имени для подтверждения, как показано на рис. 2.6.

Удаление и добавление новых символов. Ввести имя глобального символа – это все, что нужно, чтобы изменения в коде вступили в силу во всех исходных текстах. Вы можете удалить имя, чтобы удалить определение глобального символа, а также добавить символ / перед именем (например, `/SHOW_DEBUG_MESSAGE`), чтобы отключить глобальный символ, что облегчит повторное его включение. Можно также добавить несколько глобальных символов, разделив их имена точкой с запятой (например, `DEFINE1; DEFINE2; DEFINE3 ...`).

Визуальная отладка

Часто бывает достаточно обычной отладки с применением абстрактных или текстовых представлений данных (полученных, например, с помощью `Debug.log`), но иногда удобнее использовать другие способы. Порою лучше один раз увидеть, чем сто раз услышать. Так, например, при программировании функций, определяющих поле

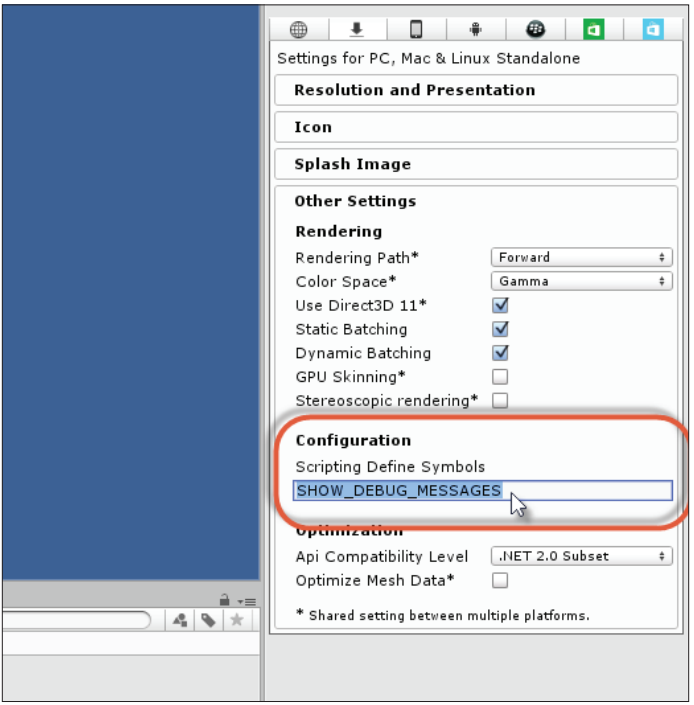


Рис. 2.6. Добавление глобального символа в редакторе Unity для условной компиляции кода

зрения врагов и других персонажей, которые позволяют им увидеть игрока и прочие объекты, полезно было бы иметь живое графическое представление в окне просмотра, отражающее их поле зрения. Поле зрения должно быть представлено в виде граничных линий или кубического каркаса. Аналогично, если объект движется по определенной траектории, было бы хорошо провести цветную линию этой траектории. Цель такой визуализации заключается не в создании наглядных подсказок игроку в законченной игре, а в облегчении процесса отладки с помощью получения более полного отражения работы игры. Создание таких наглядных визуальных представлений является частью визуальной отладки. Unity изначально предоставляет нам в пользование несколько визуальных представлений, например каркас коллайдера или границы поля зрения камер. Однако мы можем создавать свои собственные визуальные представления для своих объектов, о чем рассказывается далее в этом разделе.

Как уже упоминалось ранее, многие объекты Unity, такие как коллайдеры, триггеры объема, NavMesh Agents, камеры и источники света, уже оснащены собственными визуальными представлениями. Визуальные представления по умолчанию отображаются в окне просмотра сцены **Scene**, если вы не выключили их или не уменьшили их размер до нуля. То есть, если вы добавили встроенный объект и не увидели его визуальное представление в окне просмотра сцены, откройте панель **Gizmo** (Визуальное представление), щелкнув на кнопке **Gizmos** (Визуальные представления) в панели инструментов вкладки **Scene** (Сцена). Включите визуальные представления, которые хотите видеть, и отрегулируйте их размер ползунком **Size** (Размер), как показано на рис. 2.7.

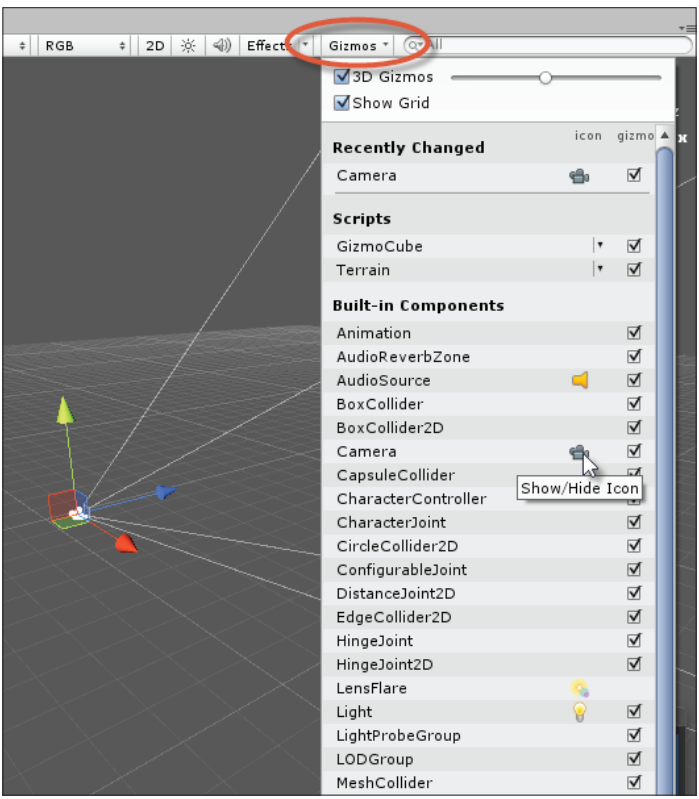


Рис. 2.7. Включение визуального представления в окне просмотра сцены

Визуальные представления во вкладке Game. По умолчанию визуальные представления не отображаются во вкладке **Game** (Игра). Вы легко сможете это исправить, открыв панель **Gizmo** (Визуальное представление), щелкнув на кнопке **Gizmo** (Визуальное представление) справа вверху во вкладке **Game** (Игра). Панель **Gizmo** (Визуальное представление) действует точно так же, как панель **Gizmos** (Визуальные представления) во вкладке **Scene** (Сцена), изображенной на рис. 2.7.

Рассмотрим пример в листинге 2.6. Он содержит класс, который можно подключить к объекту, чтобы добавить в него определенного типа визуальное представление. Более подробную информацию можно найти по адресу http://docs.unity3d.com/ScriptReference/Gizmos.html. Этот класс прорисовывает сферический каркас заданного радиуса с объектом в центре, который представляет область, доступную объекту для атаки. Кроме того, он отображает вектор направления движения объекта, обеспечивая визуальную индикацию траектории объекта. Прорисовка визуального представления выполняется в методе обработки события `OnDrawGizmos` класса `MonoBehaviour`, при условии, что значение переменной `DrawGizmos` равно `true`.

Листинг 2.6. Класс для рисовании визуального представления

```
using UnityEngine;
using System.Collections;

public class GizmoCube : MonoBehaviour
{
  // Выводить отладочную информацию?
  public bool DrawGizmos = true;

  // Вызывается для рисования визуального представления.
  // Будет рисовать всегда.
  // Если нужно нарисовать визуальные представления только
  // для выбранных объектов, тогда вызывайте
  // OnDrawGizmosSelected
  void OnDrawGizmos()
  {
    if(!DrawGizmos) return;

    // Установить цвет визуального представления
    Gizmos.color = Color.blue;

    // Нарисовать вектор движения
    Gizmos.DrawRay(transform.position,
      transform.forward.normalized * 4.0f);

    // Установить цвет визуального представления
    // Нарисовать сферу, описывающую куб
```

```
    // Если куб – это враг, он сможет определить присутствие
    // игрока, как только тот попадет внутрь сферы
    Gizmos.color = Color.red;
    Gizmos.DrawWireSphere(transform.position, 4.0f);

    // Восстановить белый цвет
    Gizmos.color = Color.white;
  }
}
```

На рис. 2.8 показано, как визуальное представление помогает при отладке.

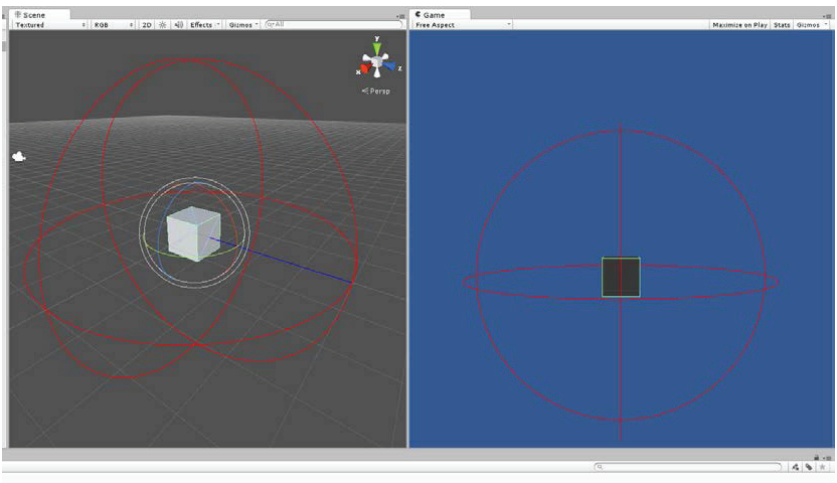


Рис. 2.8. Визуальное представление

Регистрация ошибок

После сборки игры и передачи ее тестировщикам, где бы они не находились, в том же офисе, что и вы, или разбросаны по всему миру, вам понадобится средство для записи сведений об ошибках и исключениях, происходящих во время выполнения игры. Одним из таких средств являются файлы регистрации, или файлы журналов (logfiles). Файлы регистрации – обычные текстовые файлы, предназначенные для чтения человеком. Они создаются на локальном компьютере во время выполнения игры и содержат сведения об ошибках. Объем информации для записи в файл должен быть тщательно продуман. Слишком детальная информация сделает файл запутанным, а слиш-

ком краткая – бесполезным. Однако будем считать, что нужный баланс достигнут, и тестировщики отправляют вам регистрационные файлы для просмотра. Это, как мы надеемся, позволит вам быстро выявлять ошибки и эффективно исправлять их, не внося новых ошибок! В Unity существует много способов регистрации. Один из них основан на встроенном классе `Application`, получающем уведомления об исключениях путем делегирования. Рассмотрим следующий пример в листинге 2.7:

Листинг 2.7

```
01 //---------------------------------------------
02 using UnityEngine;
03 using System.Collections;
04 using System.IO;
05 //---------------------------------------------
06 public class ExceptionLogger : MonoBehaviour
07 {
08   // Внутренняя ссылка на объект потока записи
09   private System.IO.StreamWriter SW;
10
11   // Имя файла регистрации
12   public string LogFileName = "log.txt";
13
14   //---------------------------------------------
15   // Этот метод выполняет инициализацию
16   void Start ()
17   {
18     // Сделать постоянно хранимым в памяти
19     DontDestroyOnLoad(gameObject);
20
21     // Создать объект записи в строку
22     SW = new System.IO.StreamWriter(
23       Application.persistentDataPath + "/" + LogFileName);
24
25     Debug.log(Application.persistentDataPath + "/" + LogFileName);
26   }
27   //---------------------------------------------
28   // Зарегистрировать обработчик исключений
29   void OnEnable()
30   {
31     Application.RegisterLogCallback(HandleLog);
32   }
33   //---------------------------------------------
34   // Отключить обработчик исключений
35   void OnDisable()
36   {
37     Application.RegisterLogCallback(null);
38   }
```

```
39   //-----------------------------------------------
40   // Записать информацию об исключении в файл
41   void HandleLog(string logString, string stackTrace, LogType type)
42   {
43     // Если исключение или ошибка, записать в файл
44     if(type == LogType.Exception || type == LogType.Error)
45     {
46       SW.WriteLine("Logged at: " + System.DateTime.Now.ToString() +
47         " - Log Desc: " + logString + " - Trace: " + stackTrace +
48         " - Type: " + type.ToString());
49     }
50   }
51   //-----------------------------------------------
52   // Вызывается при уничтожении объекта
53   void OnDestroy()
54   {
55     // Закрыть файл
56     SW.Close();
57   }
58   //-----------------------------------------------
59 }
60 //-------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 2.7:

- **Строка 22**: создается новый объект `StreamWriter` для записи отладочных строк в файл на локальном компьютере. Файл создается в папке `Application.persistentDataPath`, которая всегда доступна для записи.
- **Строка 31**: вызывается метод `Application.RegisterLogCallBack` со ссылкой на функцию `HandleLog` в качестве аргумента. Это связано с делегированием. Проще говоря, ссылка на функцию `HandleLog` нужна, чтобы вызвать ее при возникновении ошибки или исключения и записать данные в файл регистрации.
- **Строка 45**: вызывается метод `WriteLine` объекта `StreamWriter` для вывода данных в файл при возникновении ошибки. Сведения об ошибке передаются средой Unity через аргументы функции `HandleLog`: `logString`, `stackTrace` и `LogType`. Класс `StreamWriter` является частью фреймворка Mono, открытой реализации Microsoft .NET Framework. Более подробную информацию о классе `StreamWriter` можно найти по адресу http://msdn.microsoft.com/ru-ru/library/system.io.streamwriter%28v=vs.110%29.aspx.

Одним из самых быстрых способов проверить наше средство регистрации ошибок – внести в код ошибку деления на ноль. Не забудьте вставить в код строку `Debug.log(Application.persistentDataPath)`, чтобы вывести путь к файлу в окно консоли. Это поможет быстро найти файл с помощью про-

водника Windows или Mac Finder. Обратите внимание, что вместо абсолютного пути используется переменная `persistentDataPath`, значение которой меняется в зависимости от операционной системы.

Рисунок 2.9 демонстрирует, как выводятся сведения об ошибках в файл.

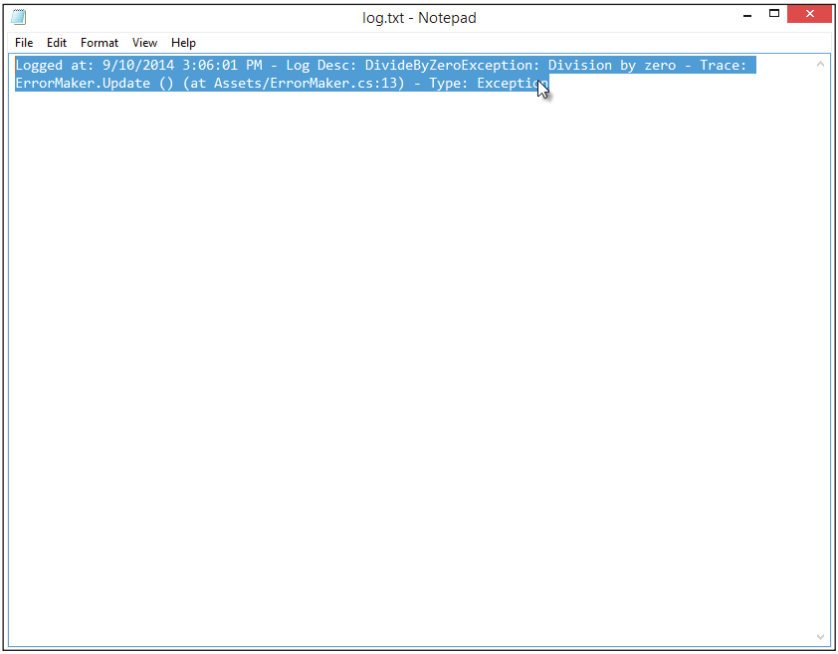


Рис. 2.9. Вывод ошибок в текстовый файл помогает упростить отладку и исправление ошибок

Что такое «делегирование» в C#? Представьте, что у вас есть возможность создать переменную и присвоить ей ссылку на функцию, а не обычное значение. В этом случае вы сможете вызвать переменную как обычную функцию. Позднее вы сможете даже присвоить этой переменной ссылку на другую функцию. Примерно так работает делегирование. Делегаты практически эквивалентны указателям на функции в C++. То есть, делегат является специальным типом данных, который может содержать ссылку для вызова функции. Они идеально подходят для создания системы обратного вызова и уведомления о событиях. Например, при наличии списка или массива

делегатов, многие классы смогут зарегистрироваться в качестве получателей обратных вызовов, добавив себя в список. Более подробную информацию о делегатах в C# можно найти по адресу http:// msdn.microsoft.com/ru-ru/library/ms173171.aspx. Рассмотрим пример использования делегатов C# в Unity (см. листинг 2.8).

Листинг 2.8. Пример использования делегатов C# в Unity

```
using UnityEngine;
using System.Collections;
//-----------------------------------------------
public class DelegateUsage : MonoBehaviour
{
  // Определить типа делегата со списком параметров
  public delegate void EventHandler(int Param1, int Param2);
  //-----------------------------------------------
  // Объявить массив ссылок на функции типа EventHandler
  // емкостью до 10 ссылок

  public EventHandler[] EH = new EventHandler[10];

  //-----------------------------------------------
  /// <summary>
  /// Awake вызывается перед началом работы. Добавляет делегата
  /// HandleMyEvent в список
  /// </summary>
  void Awake()
  {
    // Добавить обработчика события (HandleMyEvent) в список делегатов
    EH[0] = HandleMyEvent;
  }
  //-----------------------------------------------
  /// <summary>
  /// Выполняет обход списка делегатов и вызывает обработчики по одному
  /// </summary>
  void Start()
  {
    // Цикл по всем делегатам в списке
    foreach(EventHandler e in EH)
    {
      // Вызвать обработчика, если ссылка не равна null
      if(e!=null)
        e(0,0); // Это вызов обработчика
    }
  }
  //-----------------------------------------------
  /// <summary>
  /// Пример обработчика события. Позволяет ссылаться на него
  /// как на делегат типа EventHandler
```

```
/// </summary>
/// <param name="Param1">Пример параметра</param>
/// <param name="Param2">Пример параметра</param>
void HandleMyEvent (int Param1, int Param2)
{
  Debug.Log ("Event Called");
}
//---------------------------------------------------
```

Отладка с помощью редактора

Некоторые жалуются, что Unity не имеет встроенных инструментов отладки, но это не совсем верно. В Unity можно запустить игру и редактировать сцену одновременно с ее выполнением. Можно даже просматривать и редактировать общедоступные и закрытые свойства в инспекторе объектов. Это позволяет получить полную и наглядную картину происходящего в игре во время выполнения, выявлять и наблюдать широкий спектр всевозможных ошибок. Эту форму отладки не следует недооценивать. Чтобы полностью использовать возможности отладки в редакторе, включите режим отладки в инспекторе объектов, щелкнув на ярлыке контекстного меню в правом верхнем углу инспектора и выбрав пункт **Debug** (Отладка), как показано ни рис. 2.10.

Расположите окна просмотра так, чтобы одновременно видеть оба окна – **Scene** (Сцена) и **Game** (Игра) – в режиме **Play** (Играть), вместе с панелью **Stats** (Статистики). Для этого снимите флажок **Maximize on Play** (Распахнуть на время игры) на панели инструментов во вкладке **Game** (Игра), если он установлен. Затем поместите закладки **Scene** (Сцена) и **Game** (Игра) рядом друг с другом, либо разнесите их по разным мониторам, если есть такая возможность. Настоятельно рекомендую использовать несколько мониторов, если ваш бюджет это позволяет. Но и одного монитора вполне достаточно, если вы потратите дополнительное время на настройку положения и размеров каждого из окон под свои потребности. Также желательно сделать видимым окно консоли и скрыть панель проекта, чтобы предотвратить случайный выбор и перемещение ресурсов, как показано на рис. 2.11. Кроме того, можно настроить режим компоновки графического интерфейса Unity. Более подробную информацию об этом можно найти по адресу http://docs.unity3d.com/Manual/CustomizingYourWorkspace.html.

Когда вы будете готовы приступить к отладке в редакторе, щелкните на кнопке **Play** (Играть) в панели инструментов и используйте паузу для остановки игры и проверки значений свойств нужного

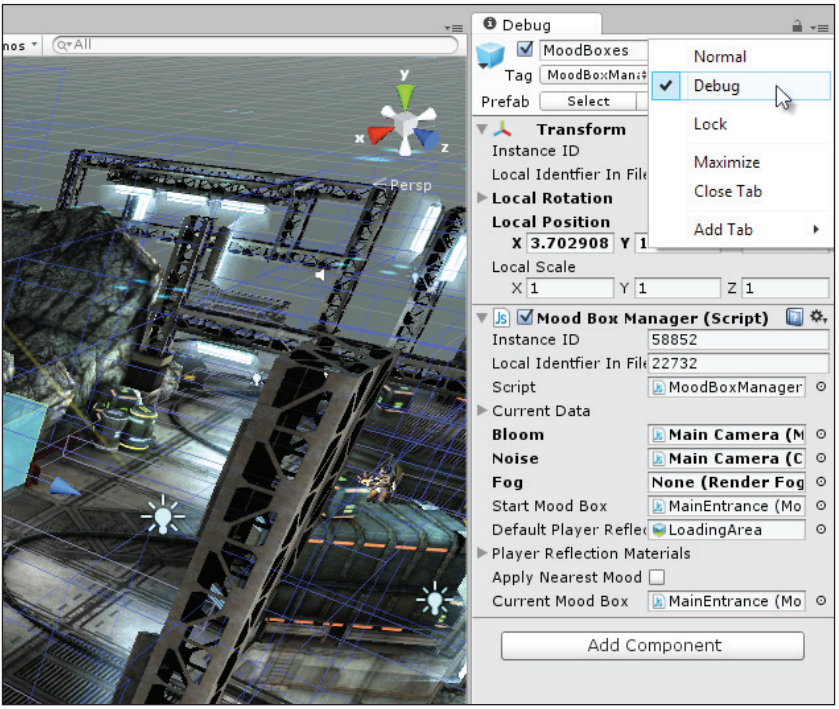


Рис. 2.10. Включение режима отладки в инспекторе объектов

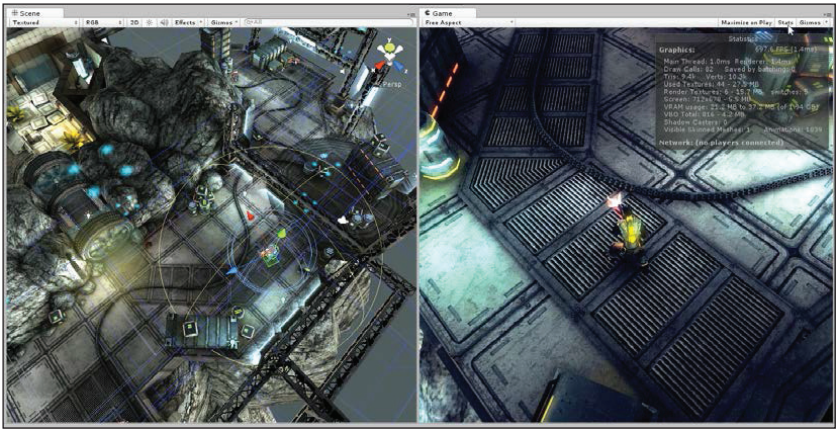


Рис. 2.11. Отладка игры в редакторе на единственном мониторе

объекта в инспекторе объектов. Помните, что вы все еще можете использовать инструменты позиционирования (положение, направление и масштаб) в игре, чтобы изменить местоположение игрока и его врагов, задавая разные значения и наблюдая, к каким результатам это приводит. Самое главное, однако, что все изменения в инспекторе объектов во время игры носят временный характер и будут отменены при ее завершении. Поэтому, если нужно, чтобы изменения сохранялись, производите их в режиме редактирования. Можно, конечно, копировать и вставлять значения, переключаясь между режимами **Play** (Играть) и **Edit** (Правка) с помощью контекстного меню компонента, как это показано на рис. 2.12. Однако гораздо удобнее переключаться между этими режимами с помощью горячей комбинации **Ctrl**+**P** (еще одна комбинация – **Ctrl**+**Shift**+**P** – позволяет включать и выключать режим паузы). Полный список горячих клавиш в Unity можно найти по адресу http://docs.unity3d.com/Manual/UnityHotkeys.html.

Профилирование

Еще один инструмент, предназначений для отладки и оптимизации, – окно **Profiler** (Профилировщик), доступное только в версии Unity Pro. Чтобы увидеть его, щелкните на вкладке **Profiler** (Про-

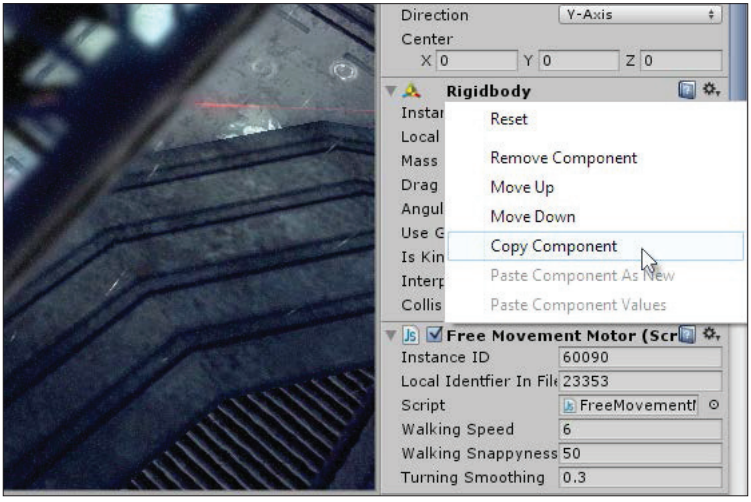


Рис. 2.12. Копирование и вставка значений свойств компонента из контекстного меню компонента

филировщик) в меню **Window** (Окно) приложения, как показано на рис. 2.13. Профилировщик позволяет получить статистическую информацию о распределения нагрузки между разными частями игры и элементами оборудования, такими как процессор и видеокарта. С помощью профилировщика можно определить, например, сколько времени расходуется камерой на отображение сцены, и сравнить со временем на фактические расчеты, или на воспроизведение аудио, или на что-то другое. Это позволит измерить производительность, сравнить результаты и оценить, что нужно улучшить. Профилирование не предназначено конкретно для выявления ошибок в коде. Но, если наблюдаются проблемы с производительностью игры, такие как замедление и замирание, оно поможет найти участок, нуждающийся в оптимизации. По этой причине профилировщик – это тот инструмент, к помощи которого следует обратиться, если производительность станет проблемой для игры, и вы должны будете провести грамотный анализ и выявить место для внесения улучшений.

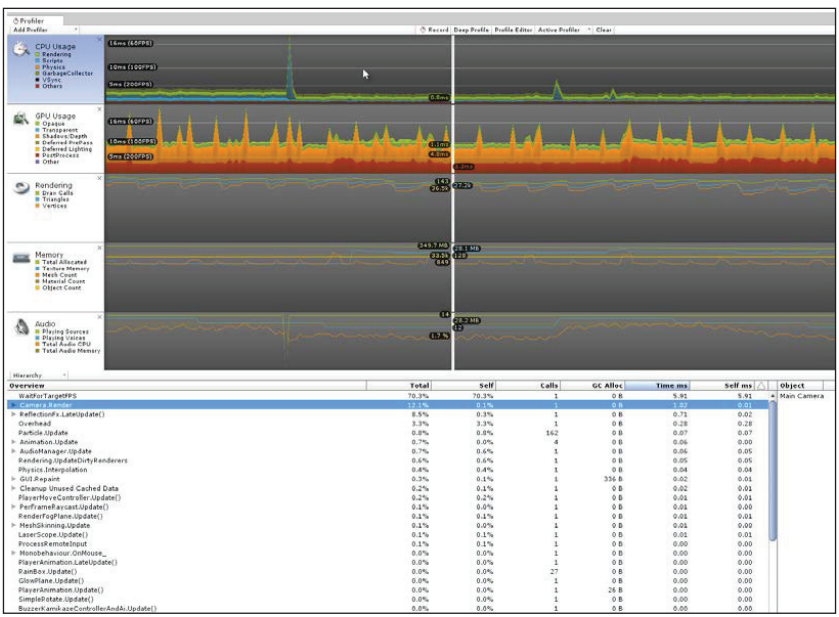


Рис. 2.13. Профилирование обычно используется для диагностики потерь производительности

Если запустить игру с открытым окном **Profiler** (Профилировщик), графики в нем заполнятся статистической информацией о последних кадрах. Профилировщик обычно записывает информацию не обо всех кадрах с начала игры, а только о последних, что разумно с точки зрения экономии памяти. Существует возможность переключения в режим «глубокого профилирования», его переключатель доступен в верхней панели инструментов, в окне **Profiler** (Профилировщик). Этот режим позволяет (в теории) получить дополнительную информацию об игре, но я рекомендую избегать его использования, так как это может вызвать проблемы с производительностью в редакторе Unity при использовании «тяжелых» ресурсов и «тяжелого» кода, вплоть до зависания всего редактора. Используйте только режим по умолчанию. В этом режиме обычно бывает желательно отключить визуализацию сигнала **VSync** в разделе **CPU Usage** (Загрузка процессора), чтобы получить более точное представление о других характеристиках, таких как отображение и сценарии, как показано на рис. 2.14. Для этого просто щелкните на ярлыке **VSync** в области легенды графика.

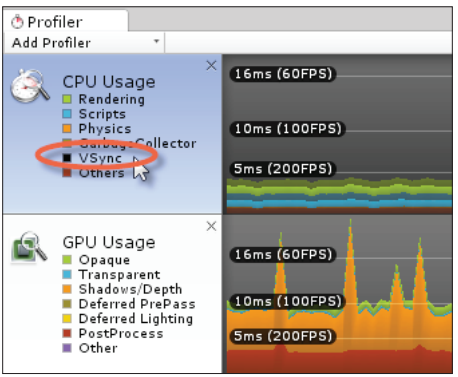


Рис. 2.14. Отключение визуализации сигнала **VSync** в разделе **CPU Usage** (Загрузка процессора)

Горизонтальная ось графика соответствует кадрам – самым последним, добавленным в буфер памяти. Эта ось постоянно заполняется новыми данными в процессе выполнения игры. Вертикальная ось соответствует затратам времени или вычислительных ресурсов: большие значения соответствуют более высоким требованиям к производительности и более затратным по времени кадрам. После заполнения

графика некоторым количеством данных в режиме **Play** (Играть), можно приостановить игру и проанализировать ее состояние. Выберите отдельные кадры из графика, чтобы получить более подробную информацию о производительности игры в этих кадрах. Когда вы это сделаете, панель **Hierarchy** (Иерархия) в нижней части окна **Profiler** (Профилировщик) заполнится данными о коде, выполненном в выбранном кадре. При просмотре графика желательно отследить внезапные всплески (пики или шипы), как показано на рис. 2.15. Они указывают на кадры с неожиданной интенсивной деятельностью. Иногда это могут быть разовые случаи, которые неизбежны из-за аппаратных операций или вполне объяснимы и не являются источником проблем с производительностью, такие как смена сцен или загрузка экранов. Но иногда они могут указывать на проблемы, особенно если повторяются регулярно. То есть, при диагностике проблем с производительностью пики – это именно то, с чего надо начинать исследования.

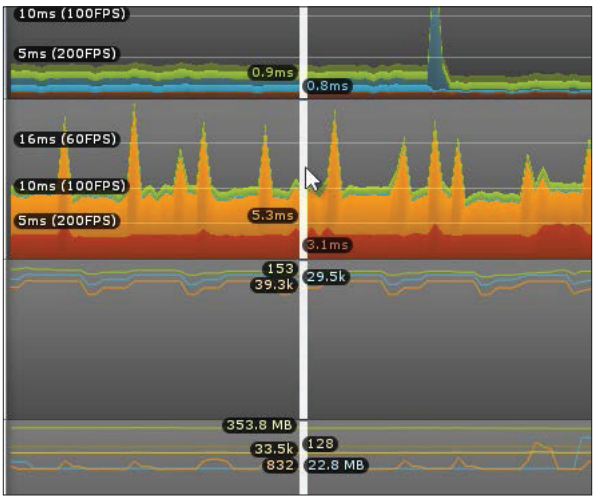
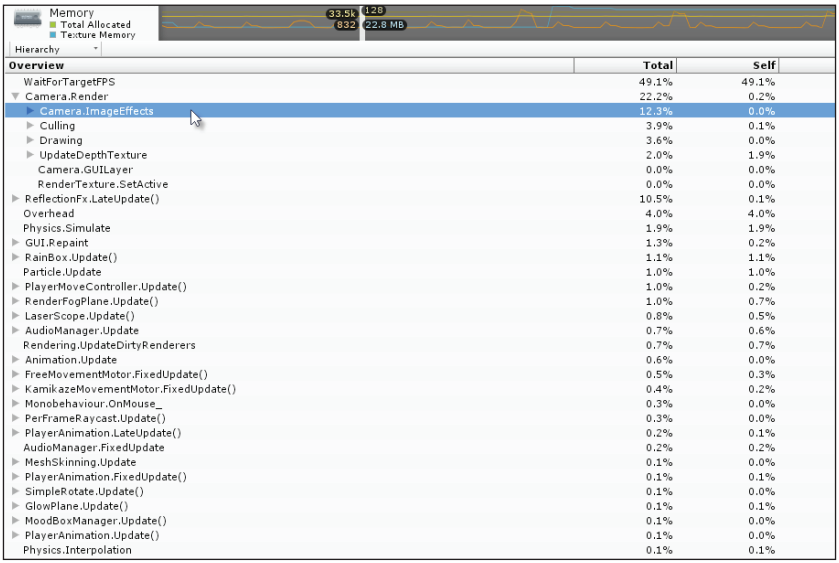


Рис. 2.15. Выбор кадра на графике в окне профилировщика

В панели **Hierarchy** (Иерархия) перечислены все основные функции и события, выполнявшиеся и обрабатывавшиеся в выбранном кадре. Для каждой функции приводится несколько важнейших характеристик, таких как **Total** (Всего), **Self** (Сама функция), **Time ms** (Время, мсек) и **Self ms** (Сама функция, мсек), как показано на рис. 2.16.



Overview	Total	Self
WaitForTargetFPS	49.1%	49.1%
Camera.Render	22.2%	0.2%
Camera.ImageEffects	12.3%	0.0%
Culling	3.9%	0.1%
Drawing	3.6%	0.0%
UpdateDepthTexture	2.0%	1.9%
Camera.GUILayer	0.0%	0.0%
RenderTexture.SetActive	0.0%	0.0%
ReflectionFx.LateUpdate()	10.5%	0.1%
Overhead	4.0%	4.0%
Physics.Simulate	1.9%	1.9%
GUI.Repaint	1.3%	0.2%
RainBox.Update()	1.1%	1.1%
Particle.Update	1.0%	1.0%
PlayerMoveController.Update()	1.0%	0.2%
RenderFogPlane.Update()	1.0%	0.7%
LaserScope.Update()	0.8%	0.5%
AudioManager.Update	0.7%	0.6%
Rendering.UpdateDirtyRenderers	0.7%	0.7%
Animation.Update	0.6%	0.0%
FreeMovementMotor.FixedUpdate()	0.5%	0.3%
KamikazeMovementMotor.FixedUpdate()	0.4%	0.2%
Monobehaviour.OnMouse_	0.3%	0.0%
PerFrameRaycast.Update()	0.3%	0.0%
PlayerAnimation.LateUpdate()	0.2%	0.1%
AudioManager.FixedUpdate	0.2%	0.2%
MeshSkinning.Update	0.1%	0.0%
PlayerAnimation.FixedUpdate()	0.1%	0.1%
SimpleRotate.Update()	0.1%	0.0%
GlowPlane.Update()	0.1%	0.1%
MoodBoxManager.Update()	0.1%	0.0%
PlayerAnimation.Update()	0.1%	0.0%
Physics.Interpolation	0.1%	0.1%

Рис. 2.16. Характеристики функций, вызывавшихся в кадрах

Обсудим эти характеристики подробнее:

- **Total** (Всего) и **Time ms** (Всего мсек): столбец **Total** (Всего) отражает отношение времени, затраченного на выполнение функции, ко времени, затраченному на весь кадр. Значение **49,1%**, например, означает, что 49,1 процента общего времени, затраченного на выбранный кадр, было поглощено функцией, сюда входит и время, затраченное функциями, вызывавшимися внутри нее. Столбец **Time ms** (Время, мсек) отображает затраченное время в миллисекундах. Вместе эти два значения позволяют получить относительную и абсолютную оценку времени, затраченного на вызов функций в каждом кадре и на весь кадр.
- **Self** (Сама функция) и **Self ms** (Сама функция, мсек): значения в столбцах **Total** (Всего) и **Time ms** (Всего мсек) отражают затраты времени на выполнение функции для выбранного кадра, но они включают также время выполнения функций, вызванных из функции. Значения **Self** (Сама функция) и **Self ms** (Сама функция, мсек) отражают только время выполнения самой функции, без учета времени ожидания завершения вызванных из нее других функций. Эти характеристики особенно

важны для выделения функций, вызывающих проблемы с производительностью.

Более подробную информацию о профилировании в Unity можно найти по адресу http://docs.unity3d.com/Manual/ProfilerWindow.html.

Отладка с помощью MonoDevelop – начало

Ранее мы уже познакомились с методом отладки `Debug.log` для вывода вспомогательных сообщений в консоль в критические моменты, возникающие в ходе выполнения программы. Несмотря на удобство, этот метод имеет ряд существенных недостатков. Во-первых, в больших программах множество операторов `Debug.log` очень легко может наводнить консоль большим количеством сообщений. Это усложнит задачу выделения нужных сообщений. Во-вторых, изменение кода вставкой операторов `Debug.log` только для мониторинга выполнения программы и выявления ошибок, считается плохой практикой. В идеале должна иметься возможность переходить в режим отладки без внесения изменений в код. Таким образом, у нас есть веские причины для поиска альтернативных способов отладки. В этом нам поможет MonoDevelop. В частности, последние версии MonoDevelop могут напрямую подключаться к запущенному процессу Unity. При этом мы получаем доступ к множеству инструментов отладки, широко используемых при разработке других видов программного обеспечения, таких как точки останова и трассировка. В настоящее время при установке связи между MonoDevelop и Unity может возникать ошибка, но только у некоторых пользователей некоторых систем. Тем не менее, при нормальной работе MonoDevelop может предложить богатые средства отладки, что позволяет отойти от практики вставки операторов `Debug.log`.

Начнем знакомство с возможностями отладки в MonoDevelop с точек останова. При отладке часто бывает нужно проверить состояние программы по достижении заданной строки. Точка останова позволяет отметить одну или несколько строк в исходном файле и останавливать выполнение программы в первой точке останова. Это дает возможность изучить код и состояние переменных, а также проверить и отредактировать их значения. Также поддерживается возможность продолжить выполнение с этой точки. Это позволяет перейти к следующим строкам, в соответствии с обычной логикой выполне-

ния программы. Вы получаете возможность проверить код в каждой строке, по ее достижении. Рассмотрим практический пример. В листинге 2.9 представлен простой сценарий. При подключении к объекту, сценарий получает список всех объектов в сцене (в том числе и свой объект), а затем, в вызове функции `Start`, перемещает их в начало координат (0, 0, 0).

Листинг 2.9. Пример сценария для отладки

```
using UnityEngine;
using System.Collections;
public class DebugTest : MonoBehaviour
{
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Получить все объекты в сцене
    Transform[] Objs = Object.FindObjectsOfType<Transform>();

    // Обойти все объекты в цикле
    for(int i=0; i<Objs.Length; i++)
    {
      // Переместить объект в начало координат
      Objs[i].position = Vector3.zero;
    }
  }
}
```

Давайте установим точку останова в выделенной строке. Она приостановит выполнение программы по достижении этой строки. Чтобы установить точку останова, поместите курсор мыши на выделенную строку, щелкните правой кнопкой мыши на сером поле слева и выберите в контекстном меню пункт **New Breakpoint** (Новая точка останова), или в меню приложения MonoDevelop выберите пункт **Run** ⇒ **New Breakpoint** (Выполнить ⇒ Точка останова), или нажмите клавишу **F9** (либо щелкните левой кнопкой мыши на номере строки), как показано на рис. 2.17.

Строка с точкой останова будет подсвечена красным цветом. Чтобы остановить сценарий в этой точке останова, подключите MonoDevelop к действующему процессу Unity. Для этого запустите редактор Unity вместе с MonoDevelop и в меню приложения MonoDevelop выберите пункт **Run** ⇒ **Attach to Process** (Выполнить ⇒ Подключиться к процессу), как показано на рис. 2.18.

После этого появится диалог **Attach to Process** (Подключение к процессу), где в списке процессов должен присутствовать редактор

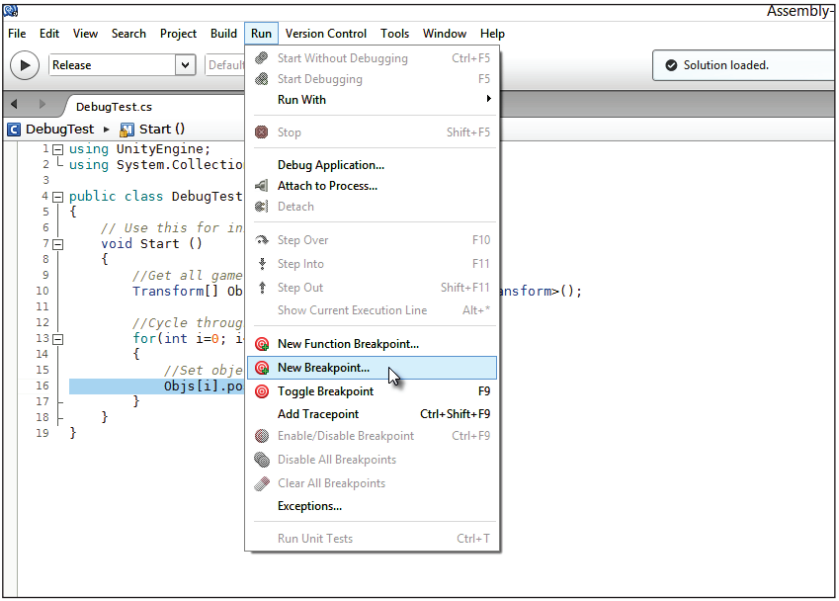


Рис. 2.17. Создание новой точки останова в MonoDevelop

Unity. В раскрывающемся списке **Debugger** (Отладчик), слева внизу, должен быть выбран пункт **Unity Debugger** (Отладчик Unity). Выберите процесс **Unity Editor** и щелкните на кнопке **Attach** (Подключить), как показано на рис. 2.19.

Когда MonoDevelop подключится к процессу Unity, в нижней части интерфейса MonoDevelop появятся две новые панели, **Watch** (Просмотр) и **Immediate** (Непосредственная отладка), как показано на рис. 2.20. Эти панели предоставляют дополнительную отладочную информацию при выполнении игры в редакторе Unity, как будет показано в следующем разделе.

Затем вернитесь в редактор Unity и подключите файл сценария `DebugTest.cs` из листинга 2.9 к объекту в сцене, при этом желательно, чтобы сцена включала также другие объекты (любые, например кубы или цилиндры). Запустите игру с помощью кнопки **Play** (Играть) в панели инструментов Unity, как показано на рис. 2.21.

После щелчка на кнопке **Play** (Играть), когда будет достигнута точка останова (режим останова), редактор Unity замрет и управление перейдет в окно MonoDevelop, где строка с точкой останова будет подсвечена желтым цветом, что указывает на текущий шаг выполне-

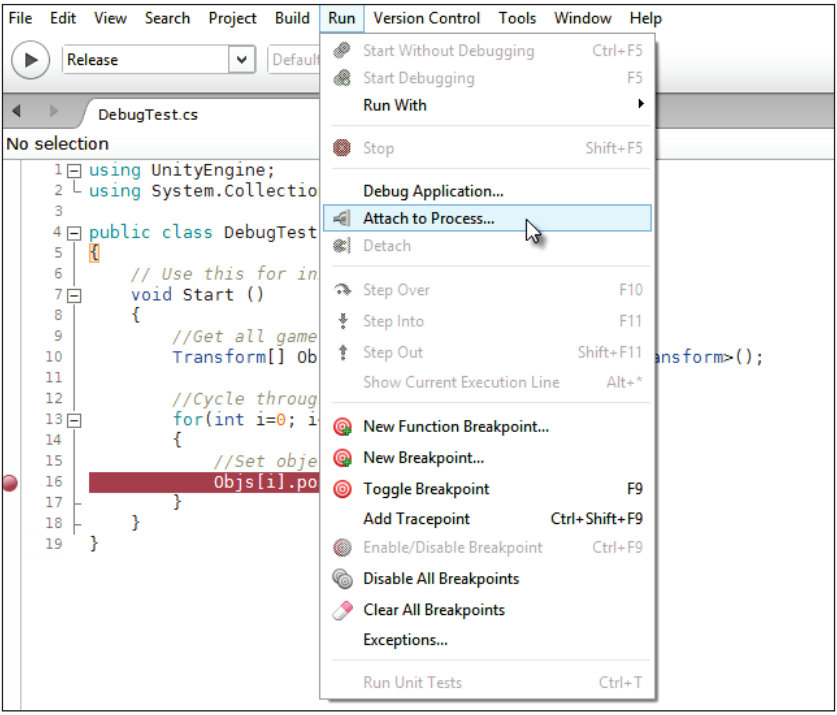


Рис. 2.18. Подключение к процессу

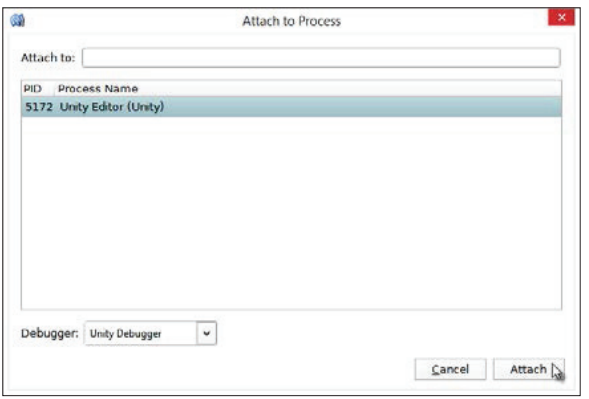


Рис. 2.19. Выбор редактора Unity в диалоге окне **Attach to Process** (Подключение к процессу)

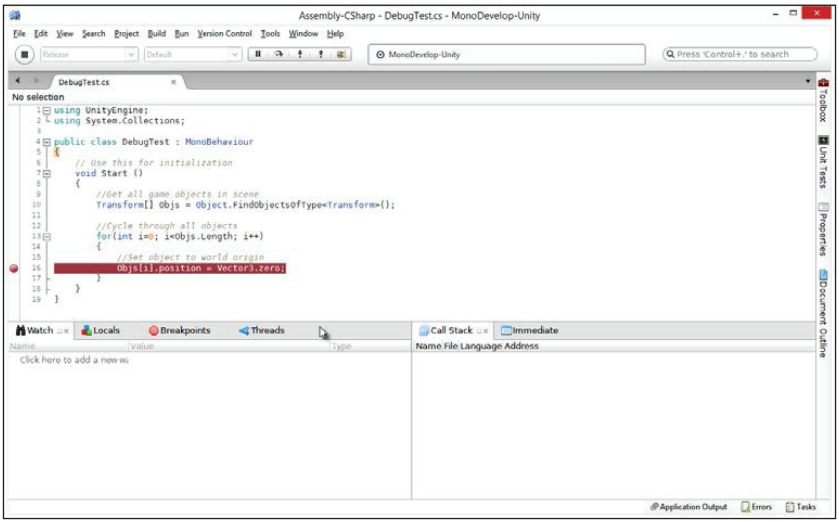


Рис. 2.20. При подключении MonoDevelop к процессу Unity появляются две новые панели

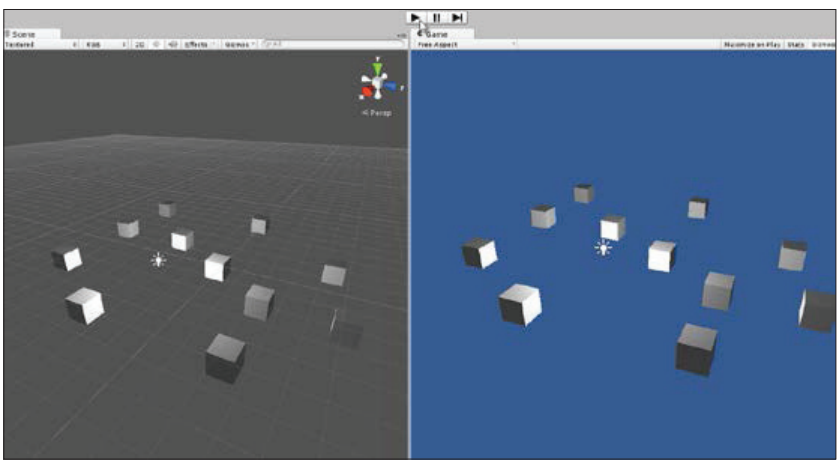


Рис. 2.21. Запуск в редакторе Unity игры, подготовленной к отладке в MonoDevelop

ния, как показано на рис. 2.22. В этом режиме вы не сможете ни использовать редактор Unity, ни переключаться между окнами просмотра, ни даже редактировать параметры в инспекторе объектов, как это

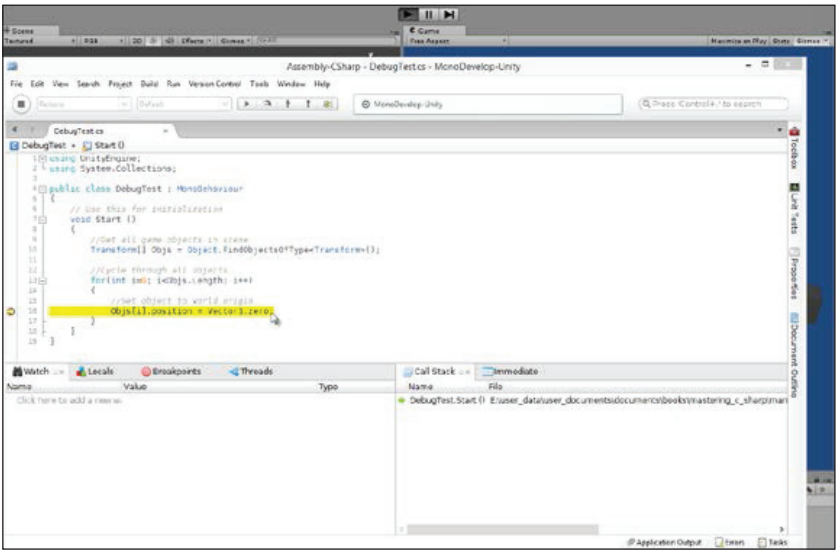


Рис. 2.22. Вход в режим останова в MonoDevelop

было возможно при отладке в редакторе. MonoDevelop ждет от вас только команды на продолжение выполнения. В следующих нескольких разделах будут рассмотрены некоторые полезные инструменты отладки, которые можно использовать в режиме останова.

Отладка с помощью MonoDevelop – окно Watch

Окно **Watch** (Просмотр) предназначено для просмотра значений переменных на текущем шаге выполнения. Это могут быть значения локальных или глобальных переменных. Чтобы быстро просмотреть состояние переменной во время останова, достаточно выделить ее в редакторе и навести на нее указатель мыши. Если оставить указатель мыши в таком положении на несколько секунд, автоматически появится всплывающее окно, позволяющее полностью исследовать переменную, как показано на рис. 2.23. Вы можете распахнуть класс и изучить состояние всех его переменных.

Используя метод наведения можно исследовать практически любые переменные в любом активном объекте. Однако, часто бывает желательно закрепить наблюдение за переменной и даже за группой

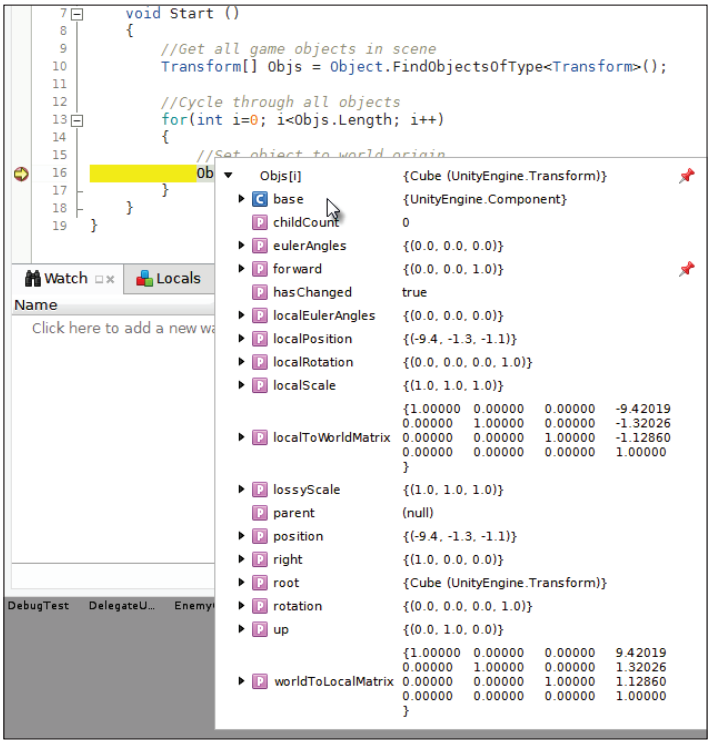


Рис. 2.23. Просмотр переменной
во всплывающем окне во время останова

переменных, чтобы видеть их значения собранными вместе в одном списке. Для этого можно использовать панель **Watch** (Просмотр) в нижней части окна MonoDevelop. Чтобы добавить новую переменную в эту панель, щелкните правой кнопкой мыши на списке в окне **Watch** (Просмотр) и выберите в контекстном меню пункт **Add watch** (Добавить для просмотра), как показано на рис. 2.24.

При добавлении нового наблюдения, в поле **Name** (Имя) можно ввести любое допустимое выражение или имя переменной, и результат будет показан в столбце **Value** (Значение), как показано на рис. 2.25. Значения в области **Watch** (Просмотр), соответствуют текущей выполняемой строке и меняются по ходу программы. Помните, что добавить можно любую допустимую переменную, на которую можно сослаться в текущей области видимости, в том числе `name`, `tag`, `transform.position` и т. д.

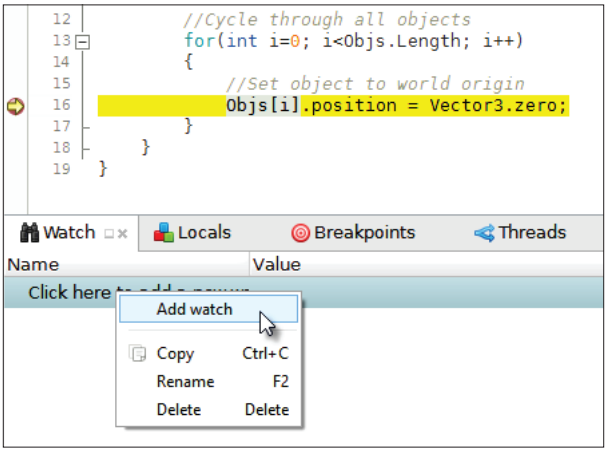


Рис. 2.24. Добавление новой переменной в окне **Watch** (Просмотр)

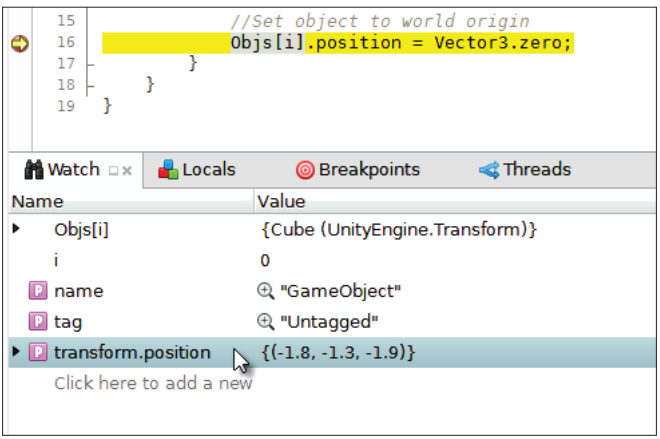


Рис. 2.25. Переменные, наблюдаемые в окне **Watch** (Просмотр)

Окно **Watch** (Просмотр) можно использовать для просмотра значений любых переменных и выражений, относящихся к активному классу или строке кода. Это означает, что можно просматривать значения глобальных и любых других переменных, связанных с другими классами или объектами, если они доступны. Однако, если вас интересуют только локальные переменные, то есть переменные, область ви-

димости которых ограничивается текущим блоком кода, вместо окна **Watch** (Просмотр) можно использовать окно **Locals** (Локальные). Это окно автоматически добавляет для наблюдения все локальные переменные – их не нужно добавлять вручную. Вкладка окна **Locals** (Локальные) по умолчанию располагается сразу вслед за вкладкой **Watch** (Просмотр), как показано на рис. 2.26.

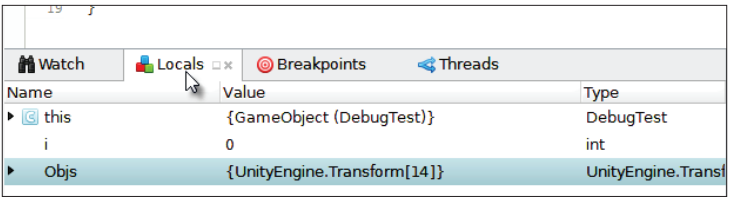


Рис. 2.26. Просмотр локальных переменных в окне **Locals** (Локальные)

Если вы не увидите каких-то окон для отладки в интерфейсе MonoDevelop, например **Watch** (Просмотр) или окна **Locals** (Локальные), их можно вывести или скрыть, выбрав пункт **View** ⇒ **Debug Windows** (Вид ⇒ Окна для отладки) в меню приложения MonoDevelop (см. рис. 2.27).

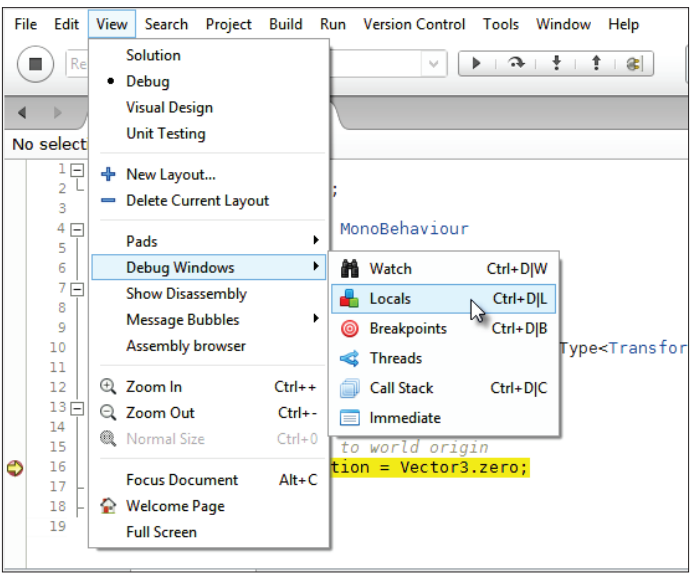


Рис. 2.27. Включение окон для отладки

Главным достоинством окон **Watch** (Просмотр) и **Locals** (Локальные) является доступность значений переменных не только для чтения, но и для записи. То есть, вы можете не только просматривать значения переменных, но и изменять их. Для этого просто дважды щелкните на поле **Value** (Значение) в окне **Watch** (Просмотр) или **Locals** (Локальные) и введите новое значение (см. рис. 2.28).

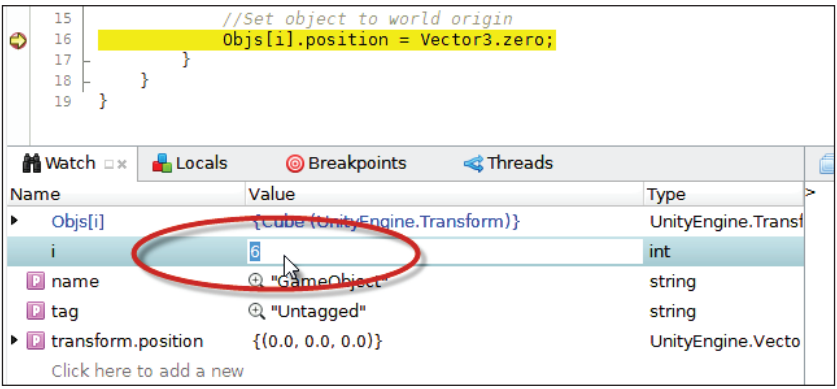


Рис. 2.28. Редактирование значений в окне **Watch** (Просмотр)

Отладка с помощью MonoDevelop – продолжение и пошаговый режим

После достижения точки останова и проверки кода часто бывает желательно выйти из режима паузы и продолжить выполнение программы. Для этого достаточно просто передать управление программой среде Unity. Это позволит продолжить выполнение в обычном режиме, пока не встретится следующая точка останова, если она есть. То есть сценарий продолжит выполнение без остановок до достижения следующей точки останова. Чтобы возобновить выполнение из MonoDevelop, нажмите клавишу **F5** или щелкните на кнопке **Play** (Играть) в панели инструментов MonoDevelop. Или же выберите пункт **Run** ⇒ **Continue Debugging** (Выполнить ⇒ Продолжить отладку) в меню приложения MonoDevelop, как показано на рис. 2.29.

Но иногда требуется выполнять код по шагам, строку за строкой, чтобы оценить ход выполнения программы в каждой строке и увидеть, как меняются значения переменных. Наблюдать за ходом выполнения программы позволяет режим пошагового выполнения.

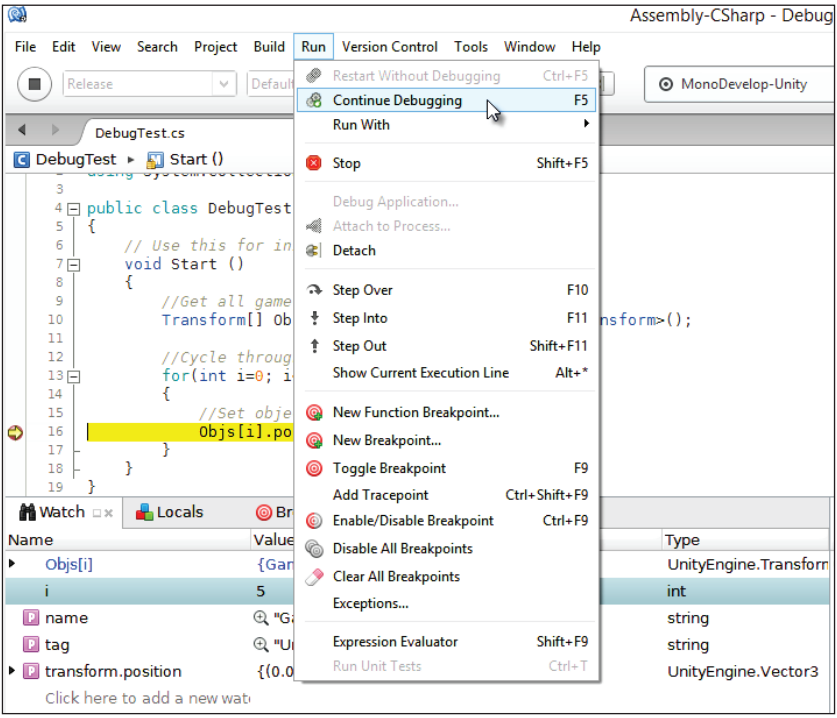


Рис. 2.29. Выход из режима останова и возобновление выполнения с помощью пункта меню **Run ⇒ Continue Debugging** (Выполнить ⇒ Продолжить отладку)

Есть три основных вида пошаговой отладки: «перешагнуть» (Step Over), «шагнуть в» (Step Into) и «шагнуть из» (Step Out). Команда отладчика «перешагнуть» вызывает переход к следующей строке кода и остановку для проверки, как если бы следующая строка содержала точку останова. Если выполняемая строка содержит вызов функции, отладчик выполнит его как обычно, а затем перейдет к следующей строке. То есть он «перешагнет» через функцию. Функция будет вызвана и выполнена, но это произойдет в режиме обычного выполнения, и режим останова включится на следующей строке кода, после вызова функции. Чтобы выполнить команду отладчика «перешагнуть», нажмите клавишу **F10**, или выберите пункт **Run ⇒ Step Over** (Выполнить ⇒ Перешагнуть) в меню приложения, или щелкните на кнопке **Step Over** (Перешагнуть) в панели инструментов MonoDevelop, как показано на рис. 2.30.

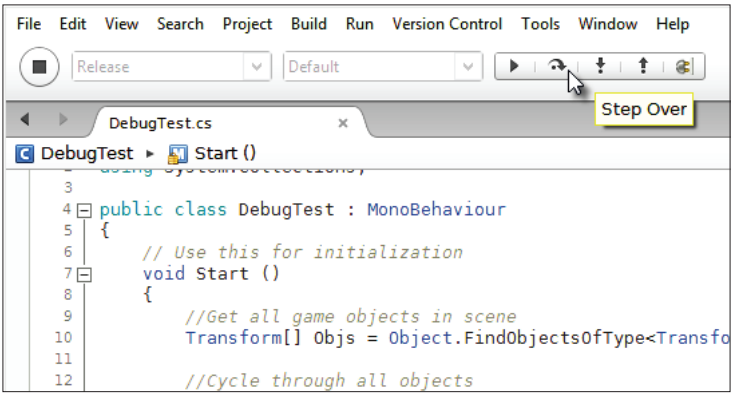


Рис. 2.30. Команда «перешагнуть» выполнит переход к следующему оператору, без входа во вложенную функцию

Если строка содержит вызов функции, команда **Step Into** (Шагнуть в) **(F11)** позволяет продолжить отладку, войдя внутрь этой функции. Выполнение остановится на первой строке функции. Эту команду удобно использовать для наблюдения за взаимодействием функций. Чтобы выйти из функции и продолжить выполнение, можно воспользоваться командой **Step Out** (Шагнуть из) **(Shift+F11)**, которая возобновит выполнение и остановится на строке, следующей за вызовом функции.

Отладка с помощью MonoDevelop – стек вызовов

Сложные программы обычно содержат много функций и их вызовов. Во время выполнения одной функции могут вызываться другие функции, и эти функции могут, в свою очередь, вызвать другие функции, создавая сложную цепочку вызовов функций из функций. Это значит, что при достижении точки останова внутри функции никогда нельзя точно знать, откуда данная функция была вызвана. Точка останова сообщает только, что выполнение программы достигло указанной строки, но ничего не говорит о том, каким путем программа пришла в эту строку. Иногда это легко выяснить, иногда очень сложно, особенно когда функция вызывается в циклах, в условных инструкциях или во вложенных циклах и условных инструкциях. Рассмотрим пример в листинге 2.10, который был получен из приме-

ра в листинге 2.9. Здесь класс содержит несколько функций, которые вызывают другие функции.

Листинг 2.10. Пример сценария для отладки

```
01 using UnityEngine;
02 using System.Collections;
03
04 public class DebugTest : MonoBehaviour
05 {
06   // Этот метод выполняет инициализацию
07   void Start ()
08   {
09     // Получить все объекты в сцене
10     Transform[] Objs = Object.FindObjectsOfType<Transform>();
11
12     // Обойти все объекты в цикле
13     for(int i=0; i<Objs.Length; i++)
14     {
15       // Переместить объект в начало координат
16       Objs[i].position = Vector3.zero;
17     }
18
19     // Вызвать функцию 01
20     Func01();
21   }
22   //-------------------------------------
23   // Вызывает функцию Func02
24   void Func01()
25   {
26     Func02();
27   }
28   //-------------------------------------
29   // Вызывает функцию Func03
30   void Func02()
31   {
32     Func03();
33   }
34   //-------------------------------------
35   // Выводит сообщение
36   void Func03()
37   {
38     Debug.log ("Entered Function 3");
39   }
40   //-------------------------------------
41 }
```

Если установить точку останова в строке 38 (как показано на рис. 2.31), выполнение будет приостановлено по ее достижении. Просмотрев этот пример, можно заметить, что один маршрут к этой точке

начинается в функции `Start`, вызывающей функцию `Func01`, функция `Func01` вызывает функцию `Func02`, а затем функция `Func02` вызывает функцию `Func03`. Но откуда мы знаем, что это единственный путь? Технически возможно, например, что функция `Func03` была вызвана из другого класса, в другом месте проекта. Итак, как же узнать путь, каким мы достигли этой функции на данном этапе во время отладки? С помощью уже знакомых нам инструментов – никак. Но мы можем использовать окно **Call Stack** (Стек вызовов). Это окно по умолчанию отображается в правом нижнем углу интерфейса MonoDevelop, в нем перечислены все вызовы функций, которые были сделаны до достижения текущей функции. Этот список и приведет нас обратно к первой, или начальной функции. Окно списка вызовов отражает весь пройденный путь в виде списка имен функций, от текущей до первой, или начальной функции. Список вызовов упорядочен от активной, или последней функции в начале списка до начальной, или первой функции в конце списка. С помощью списка можно также получить доступ к любой из функций, чтобы просмотреть значения их переменных, как показано на рис. 2.31.

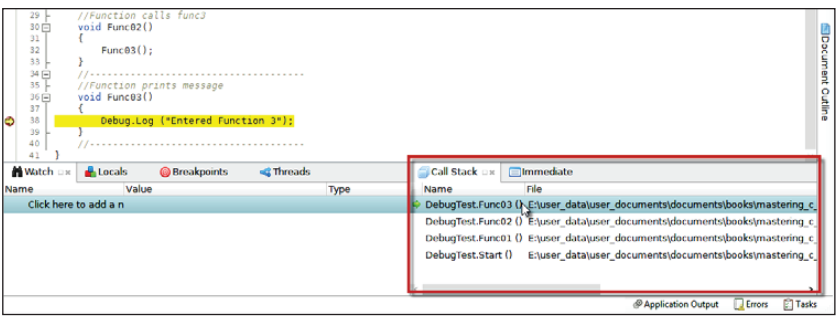


Рис. 2.31. Использование стека вызовов для трассировки вызванных функций

Отладка с помощью MonoDevelop – окно Immediate

Окно **Immediate** (Непосредственная отладка) похоже на окно **Console** (Консоль) в играх, его можно найти во многих шутерах от первого лица, таких как *Unreal, Half Life* или *Call of Duty*. Окно **Immediate** (Непосредственная отладка) по умолчанию находится в правом

нижнем углу интерфейса MonoDevelop. Оно становится активным в режиме останова. В нем можно вводить выражения и операторы, которые сразу же будут выполняться, как если бы были частью программного кода на этом шаге. Можно получать и устанавливать значения активных переменных, а также выполнять дополнительные операции. Можно написать любое допустимое выражение, например `2 + 2` или `10 * 5`. Результаты этих выражений будут выведены в следующей строке в окне **Immediate** (Непосредственная отладка), как показано на рис. 2.32.

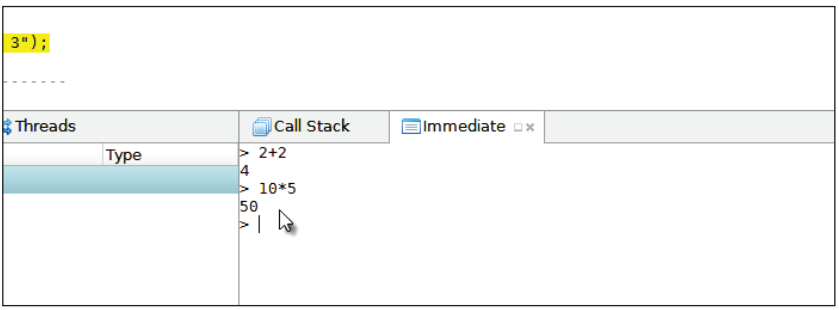


Рис. 2.32. Вычисление выражений в окне **Immediate** (Непосредственная отладка)

Конечно же, вы не ограничены вводом простых операторов, включающих основные арифметические операции, такие как сложение и вычитание. Вы можете вводить любые выражения, включающие активные переменные (см. рис. 2.33).

В целом окно **Immediate** (Непосредственная отладка) особенно удобно для тестирования кода, ввода альтернативных операторов и просмотра результатов их вычисления.

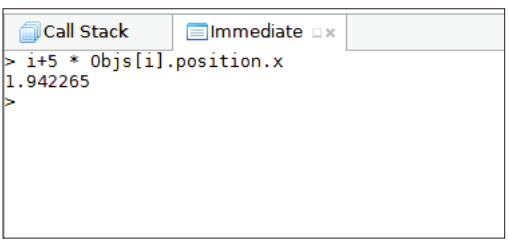


Рис. 2.33. Ввод более сложных выражений в окне **Immediate** (Непосредственная отладка)

Отладка с помощью MonoDevelop – точки останова с условием

Точки останова очень важны при отладке и представляют точки, где выполнение приложений в режиме отладки останавливается. Часто все, что вам нужно, – это задать точку останова и начать отладку! Однако иногда точки останова в обычном их виде начинают раздражать. Примером может служить точка останова внутри цикла. Иногда нужно, чтобы точка останова прерывала выполнение, только после заданного числа итераций, а не в первой же итерации. По умолчанию точка останова внутри цикла будет останавливать выполнение в каждой итерации, и если цикл длинный, такое поведение станет утомительным. Чтобы решить эту проблему, можно использовать точку останова с условием, которое определит состояние, при котором точка останова должна начать работать. Чтобы задать условие для точки останова, щелкните правой кнопкой мыши на точке останова и выберите в контекстном меню пункт **Breakpoint Properties** (Свойства точки останова), как показано на рис. 2.34.

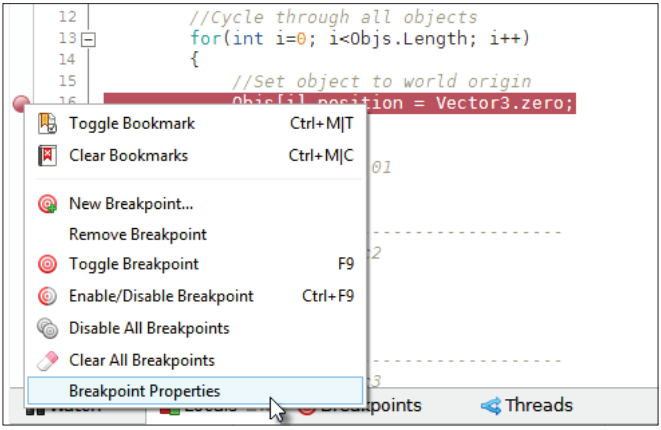


Рис. 2.34. Доступ к свойствам точки останова для определения условия

После выбора пункта **Breakpoint Properties** (Свойства точки останова) появится диалог со свойствами точки останова, где можно определить условие. В разделе **Condition** (Условие) выберите вариант **Break when condition is true** (Останавливать при выполнении

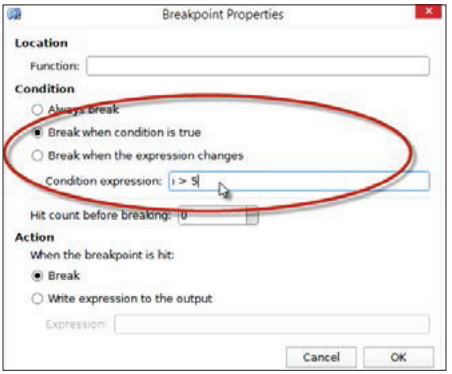


Рис. 2.35. Определение условия для точки останова

условия), а затем введите условие в поле **Condition expression** (Условное выражение). Для цикла вполне подойдет условие `i>5`, которое вызовет остановку выполнения, когда итератор цикла превысит значение `5`. Конечно, имя переменной `i` должно быть заменено на имя фактической переменной.

Отладка с помощью MonoDevelop – точки трассировки

Точки трассировки могут предложить удобную альтернативу оператору `Debug.log`, применение которого сопровождается нежелательным изменением кода при отладке. Точки трассировки определяются так же, как точки останова, то есть они устанавливаются в определенных строках в файле с исходным кодом. Для их использования не требуется изменять программный код, но (в отличие от точек останова) они не останавливают выполнение программы при отладке. Вместо этого они автоматически выполняют заданную инструкцию. Как правило, они выводят сообщение в окно **Application Output** (Вывод приложения), а не в консоль Unity. Чтобы установить точку трассировки в строке 16 листинга 2.10, установите курсор в строку 16 и выберите пункт **Run** ⇒ **Add Tracepoint** (Выполнить ⇒ Добавить точку трассировки) в меню приложения (или нажмите **Ctrl**+**Shift**+**F9**), как показано на рис. 2.36.

После выбора пункта **Add Tracepoint** (Добавить точку трассировки) MonoDevelop выведет диалог **Add Tracepoint** (Добавить точку

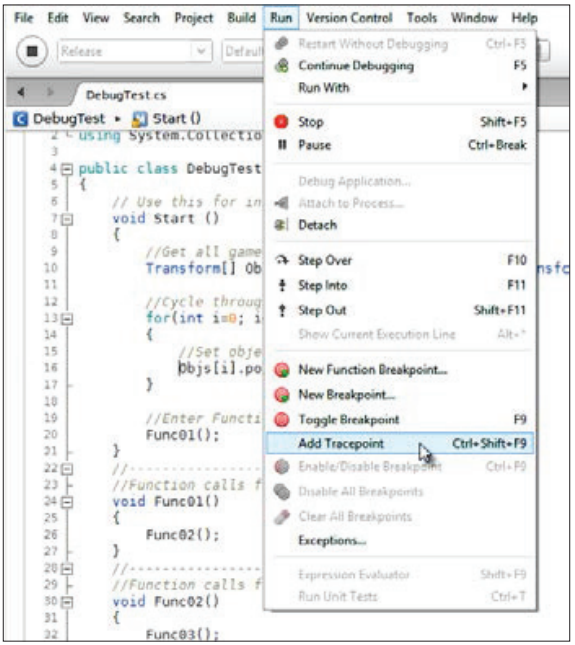


Рис. 2.36. Добавление точки трассировки в MonoDevelop

трассировки). Поле **Trace Text** (Текст трассировочного сообщения) предназначено для ввода текста, который будет напечатан в окне **Application Output** (Вывод приложения) при прохождении точки трассировки во время выполнения. В текст можно вставлять пары открывающих и закрывающих фигурных скобок с выражениями между ними. Таким способом можно выводить значения переменных, например: `"Loop counter is {i}"`, как показано на рис. 2.37.

После щелчка на кнопке **ОК** в выбранную строку будет добавлена точка трассировки. В редакторе MonoDevelop строки с точками трассировки отмечаются ромбиками, в отличие от кружков, отмечающих точки останова (см. рис. 2.38).

После установки точки трассировки в выбранной строке и запуска приложения, игра будет работать в нормальном режиме, непосредственно в редакторе Unity. При достижении точки трассировки приложение не станет останавливаться или входить в режим останова, как это было при достижении точки останова. Вместо этого в окно

Application Output (Вывод приложения) будет выведено сообщение. По умолчанию это окно находится в нижней части интерфейса MonoDevelop (см. рис. 2.39).

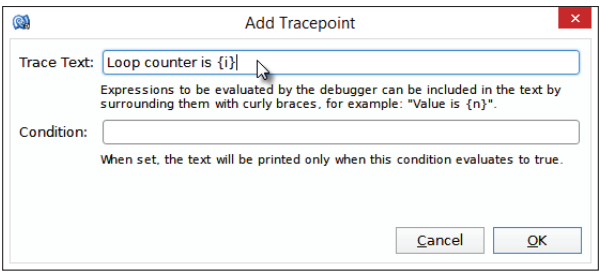


Рис. 2.37. Определение текста для вывода в точке трассировки

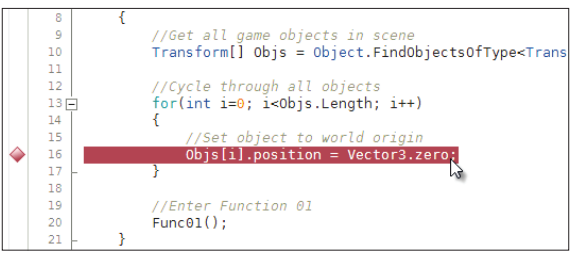


Рис. 2.38. Установка точки трассировки

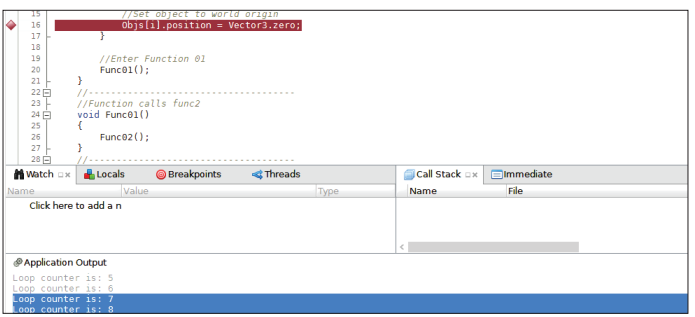


Рис. 2.39. Точки трассировки позволяют выводить в окно **Application Output** (Вывод приложения)сообщения, подобно операторам `Debug.log`

Точки трассировки – это удобная и эффективная альтернатива оператору `Debug.log`, и самое важное их преимущество в отсутствии необходимости вносить изменения в код, как этого требует использование оператора `Debug.log`. К сожалению, они не выводят сообщения непосредственно в консоль Unity. Вместо этого сообщения появляются в окне **Application Output** (Вывод приложения), в MonoDevelop. Тем не менее, следует признать, что точки трассировки могут быть мощным и полезным средством поиска и исправления ошибок.

Итоги

Эта глава была посвящена отладке, поиску и исправлению ошибок в игре. Существует много способов достижения этой цели, особенно в Unity. В частности, здесь был описан способ отладки с помощью оператора `Debug.Log`, наверное, самый простой из всех видов отладки. Он заключается во вставке операторов `Debug.log` в критические места в коде для вывода диагностических сообщений в консоль Unity. Далее мы познакомились с глобальными символами, позволяющими выделить и изолировать блоки кода для окончательной и отладочной версий. Это позволяет выполнять при отладке определенный код, когда установлен заданный флаг. Затем мы остановились на регистрации ошибок. В этой главе было показано, как создать класс регистратора ошибок, интегрирующийся со встроенным классом приложения Unity с помощью делегирования. Мы узнали о профилировщике в Unity, доступном только в версии Unity Pro, который предоставляет статистические данные о распределении времени и расходовании системных ресурсов. Кроме того, мы исследовали приемы визуальной отладки в редакторе, помогающие получить более четкое визуальное представление о сцене и факторах, влияющих на поведение объектов. Наконец, мы рассмотрели средства отладки в MonoDevelop, не требующие внесения изменений в код. В их число входят точки останова, точки трассировки, пошаговое выполнение и наблюдение. Далее мы рассмотрим работу с игровыми объектами.

Глава 3

Синглтоны, статические члены, игровые объекты и миры

Каждый уровень, или игровой мир в Unity представляет собой сцену, а сцена является коллекцией игровых объектов, помещенных в декартову систему координат с осями *x*, *y* и *z*. Расстояния в сцене измеряются в единицах Unity, соответствующих (условно) метрам. Чтобы освоить разработку сценариев для Unity, важно понимать, как устроены сцены и объекты, и как работают механизмы взаимодействий объектов. То есть, важно знать, как отдельные и независимые объекты в сцене могут воздействовать друг на друга, чтобы выполнить то, чего вы от них ждете. Как следствие, основное внимание в этой главе будет уделено встроенным методам поиска объектов, получения ссылок на них и доступа к объектам в сцене. Здесь также будут рассмотрены дополнительные понятия, такие как статические члены и синглтоны (объекты-одиночки), служащие для создания объектов, путешествующих между сценами и сохраняющих свои данные. Конечно, в этой главе будут обсуждаться не только методы сами по себе, но и произведена оценка их применимости в практических ситуациях, их производительность и эффективность.

Игровые объекты

Игровые объекты во многих отношениях являются основными единицами, или сущностями в сцене. Естественнее было бы называть их *реквизитами* в повседневном смысле. Не имеет значения, какие

именно реквизиты и какое их поведение понадобится в играх, в любом случае для их реализации вы должны будете использовать игровые объекты. Игровые объекты не всегда видны игроку, достаточно часто они бывают невидимыми. Звуки, коллайдеры и управляющие классы – вот некоторые примеры невидимых игровых объектов. С другой стороны, многие игровые объекты видимы: меши, анимированные меши, спрайты и т. д. В любом случае, видимые и невидимые игровые объекты в сцене являются совокупностью взаимосвязанных компонентов. **Компонент** – это, по существу, класс, производный от `MonoBehaviour`. Его можно прикрепить к игровому объекту и определять его поведение. Каждый игровой объект имеет, по крайней мере, один общий компонент, который невозможно удалить, а именно компонент `Transform` (или `RectTransform` для объектов графического интерфейса). Этот компонент отвечает за позиционирование, ориентацию и масштабирование объекта. Например, если создать в сцене новый пустой игровой объект, выбрав пункт **GameObject ⇒ Create Empty** (GameObject ⇒ Создать пустой объект) в меню приложения, как показано на рис. 3.1, новый игровой объект получит только один компонент `Transform`. То есть, даже новый пустой игровой объект не является, строго говоря, пустым, он пустой только как игровой объект. Объект всегда нуждается в компоненте `Transform` для определения своего местоположения в сцене.

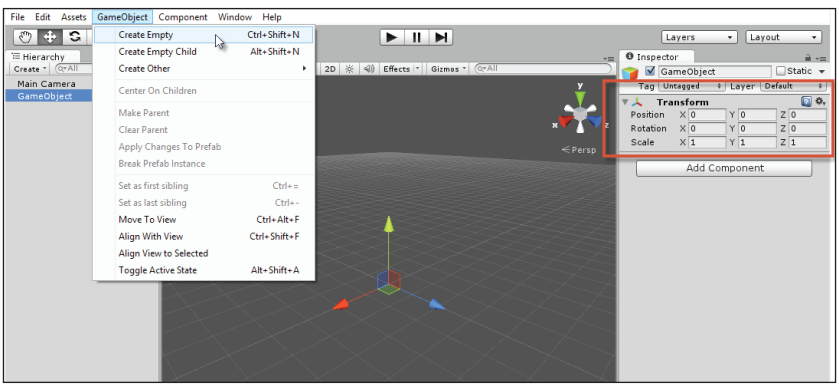


Рис. 3.1. Все игровые объекты оснащены компонентом `Transform`

Конечно, игровой объект может иметь множество компонентов, и поведение объекта определяется сочетанием компонентов и их взаимодействием. Вы можете добавлять к объекту предварительно

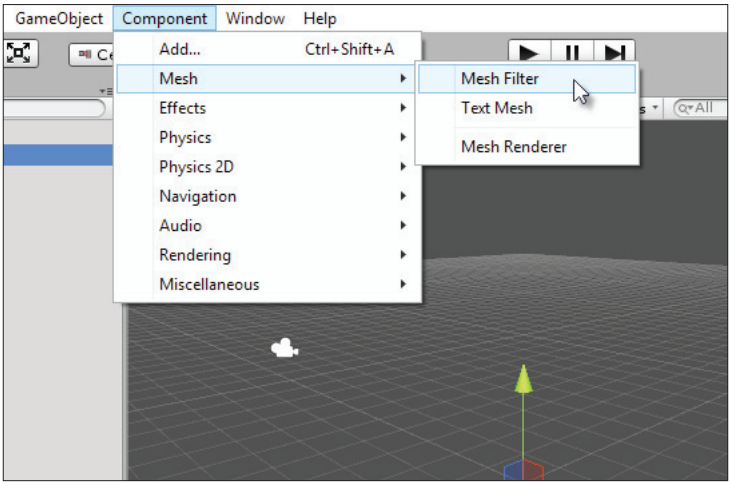


Рис. 3.2. Прикрепление компонента к игровому объекту

подготовленные компоненты, с помощью меню **Component** (Компонент), или свои собственные, прикрепляя свои сценарии к объекту.

Итак, игровые объекты состоят из компонентов. На более высоком уровне сцены – это коллекции игровых объектов внутри одного игрового мира. Кроме того, сами объекты состоят в тесных связях друг с другом, определяемых иерархией сцены. Объекты могут быть дочерними по отношению к другим объектам, которые, в свою очередь, могут иметь своих родителей (`transform.parent`). Эта связь влияет на перемещение объектов и их трансформацию. Проще говоря, значения свойств компонента `Transform` объекта передаются вниз по иерархии и добавляются к значениям свойств компонентов `Transform` всех дочерних объектов. Благодаря этому дочерний игровой объект всегда сохраняет свою позицию относительно родительского объекта – позиция родителя всегда определяет позицию дочернего объекта. Но, если объект не имеет родителя, он всегда будет позиционироваться относительно начала координат игрового мира (0, 0, 0). На рис. 3.3 показана панель иерархии.

Взаимодействия компонентов

Как мы уже определили, игровой объект является совокупностью компонентов и ничем более. Тогда возникает вопрос о взаимодействии и общении компонентов друг с другом. Каждый компонент

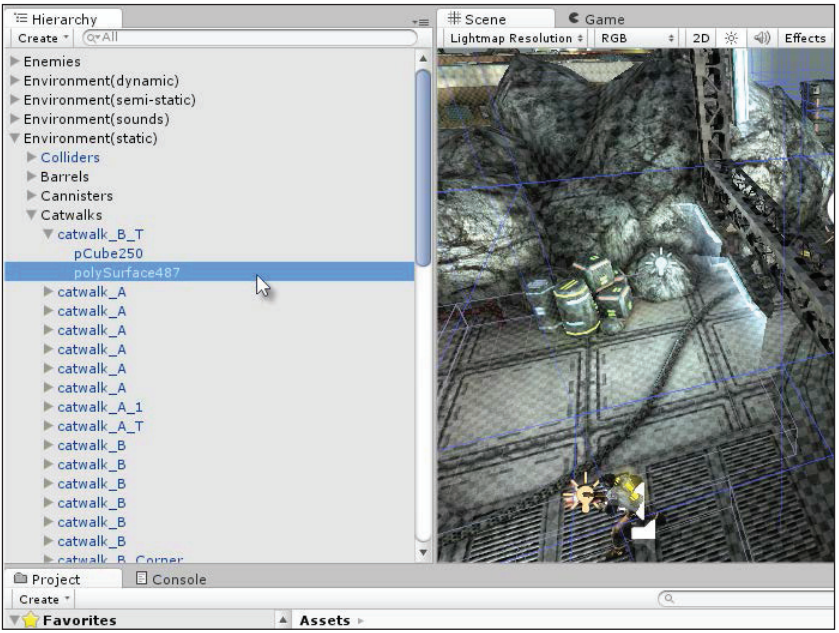


Рис. 3.3. Игровой объект занимает свое место в иерархии сцены

реализован в виде автономного файла сценария и обособлен от других компонентов, но компонент должен взаимодействовать с другими. В частности, ему необходим доступ к переменным и функциям других компонентов, прикрепленных к одному игровому объекту, и, возможно, это нужно делать в каждом кадре. В этом разделе мы посмотрим, как взаимодействуют компоненты.

Часто для вызова функций в других компонентах используются `SendMessage` и `BroadcastMessage`, как это было описано в главе 1 «Основы C# в Unity». Эти функции не зависят от типов. В частности, эти функции можно использовать в любой точке сценария для вызова по именам *любых* методов, *любых* компонентов, подключенных к тому же игровому объекту, независимо от их типа. Эти функции вообще не интересуют типы компонентов. Это делает `SendMessage` и `BroadcastMessage` очень удобным. Однако, они имеют два недостатка. Во-первых, эти функции действуют по принципу «все или ничего»: они либо вызывают метод по имени во всех компонентах, либо не вызывают ничего. Мы не можем выбрать, какому компоненту отправить сообщение, потому что сообщения всегда адресованы всем

компонентам. Во-вторых, оба метода (`SendMessage` и `BroadcastMessage`) полагаются на механизм рефлексии, что может вызвать проблемы с производительностью при их частом использовании, например при применении этих функций в событии `Update` или, что еще хуже, в событии `OnGUI`. Поэтому всегда старайтесь использовать альтернативные способы там, где это возможно. Давайте рассмотрим эти способы в следующих разделах.

Функция GetComponent

Если необходим прямой доступ к определенному компоненту игрового объекта и известен его тип, попробуйте использовать функцию `GetComponent`, как показано в листинге 3.1. Эта функция дает доступ к первому компоненту соответствующего типа, подключенному к игровому объекту. После получения ссылки на компонент, он становится доступным как обычный объект, и с помощью этой ссылки можно читать/изменять общедоступные переменные компонента и вызывать его методы.

Листинг 3.1. Пример использования функции GetComponent

```
using UnityEngine;
using System.Collections;
//-------------------------------------------------------
public class MyCustomComponent : MonoBehaviour
{
  // Ссылка на трансформацию объекта
  private Transform ThisTransform = null;
  //-------------------------------------------------------
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Получить сохраненную ссылку на трансформацию
    ThisTransform = GetComponent<Transform>();
  }
  //-------------------------------------------------------
  // Вызывается при отображении каждого кадра
  void Update ()
  {
     // Изменить позицию
    if(ThisTransform !=null)
    {
      ThisTransform.localPosition +=
        Time.deltaTime * 10.0f * ThisTransform.forward;
    }
  }
//-------------------------------------------------------
}
//-------------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 3.1:

- **Строки 07 и 13**: переменная `ThisTransform` объявлена как закрытая. Ей присваивается ссылка на компонент `Transform`, прикрепленный к игровому объекту, полученная с помощью вызова `GetComponent` в событии `Start`. В частности, чтобы получить доступ именно к компоненту `Transform`, можно также использовать унаследованное свойство `transform`, например: `ThisTransform = transform;`.
- **Строка 22**: здесь ссылка `ThisTransform` используется для записи в `localPosition` новой позиции игрового объекта. Опять же, если говорить конкретно о компоненте `Transform`, можно было бы использовать `transform.localPosition`. Однако, при этом был бы выполнен вызов внутренней функции, так как `transform` является свойством C#, а не обычной переменной. Более полную информацию о свойствах C# можно найти в главе 1 «Основы C# в Unity». По этой причине использование функции `GetComponent` в событиях `Start` или `Awake` для получения ссылок на компоненты, считается одним из самых эффективных способов доступа к внешним компонентам, особенно если ссылка на компонент регулярно используется, например, в событии `Update`.

Свойства localPosition и position. Компонент `Transform` предоставляет два основных поля для позиционирования: `position` и `localPosition`. Установка любого из них изменяет положение объекта, но по-разному. Свойство `position` всегда определяет положение объекта в глобальном пространстве. Поэтому значение, присваиваемое этой переменной в сценарии, может не соответствовать значению, отображаемому инспектором объектов в компоненте `Transform`. Если объект является потомком другого объекта, то есть, если его позиция определяется положением родителя, тогда Unity сместит объект относительно родителя настолько, насколько это будет необходимо, чтобы расположить его в указанном месте глобального пространства. Свойство `localPosition`, напротив, соответствует значению, отображаемому в компоненте `Transform` в инспекторе объектов. В частности, оно определяет положение объекта как смещение относительно родителя или начала координат игрового мира, если объект не имеет родителя. В последнем случае свойства `position` и `localPosition` будут идентичны.

Более подробную информацию о функции `GetComponent` можно найти в электронной документации: http://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html.

Документация по Unity также доступна в MonoDevelop, в виде пункта меню **Help ⇒ Unity API Reference** (Справка ⇒ Справочник по Unity API).

Получение нескольких компонентов

Иногда бывает необходимо получить список из нескольких компонентов: полный список всех компонентов или список компонен-

тов определенного типа. Сделать это можно с помощью функции GetComponents. Рассмотрим листинг 3.2. Так же как при использовании функции GetComponent, вызов GetComponents лучше поместить в одно событий, вызываемых лишь раз, такое как Start или Awake.

Листинг 3.2. Пример использования функции GetComponents

```
01 using UnityEngine;
02 using System.Collections;
03 //-------------------------------------------------------
04 public class MyCustomComponent : MonoBehaviour
05 {
06   // Массив ссылок на все компоненты
07   private Component[] AllComponents = null;
08   //-------------------------------------------------------
09   // Этот метод выполняет инициализацию
10   void Start ()
11   {
12     // Получить список всех компонентов данного объекта
13     AllComponents = GetComponents<Component>();
14
15     // Обойти компоненты в списке и вывести их в консоль
16     foreach(Component C in AllComponents)
17     {
18       // Вывести в консоль
19       Debug.log (C.ToString());
20     }
21   }
22 }
23 //-------------------------------------------------------
```

Еще о компонентах. Unity предлагает дополнительные разновидности функций GetComponent и GetComponents, упрощающие взаимодействия между компонентами, которые могут принадлежат разным объектам. В их число входят: функция GetComponentsInChildren, позволяющая получить список всех компонентов всех дочерних объектов, а также функция GetComponentsInParent, возвращающая все компоненты родительского объекта. Более подробную информацию о функции GetComponents можно найти в электронной документации: http://docs.unity3d.com/ScriptReference/Component.GetComponents.html.

Компоненты и сообщения

Функции семейства GetComponent способны удовлетворить практически любые потребности, связанные со взаимодействиями между компонентами. Они лучше справляются с этой задачей, чем SendMessage или BroadcastMessage, при правильном их использовании. Однако, бывает желательно вызвать метод SendMessage только *для одного компонента* выбранного игрового объекта, не зная ничего о типе компо-

нента. Эту задачу можно было бы решить, используя делегирование и интерфейсы (рассматриваются в следующей главе). Однако здесь мы рассмотрим прием, подобный вызову метода SendMessage. Такая возможность была бы особенно полезна для создания расширений. Например, представьте, что игра содержит несколько видов врагов и есть возможность добавлять новые их виды, каждый из которых имеет свою реализацию. Несмотря на их различия, при сохранении игры текущие сведения обо всех врагах должны быть записаны в файл. Для этого имеет смысл предусмотреть в реализации каждого врага метод OnSave, реализованный в специальном компоненте. Естественно, при сохранении игры было бы желательно вызвать только функцию OnSave этого компонента с помощью SendMessage. Но, если имеются другие компоненты с функцией OnSave, их функции с таким именем также будут ошибочно вызваны. Чтобы предотвратить это, можно прибегнуть к помощи метода Invoke. Рассмотрим пример в листинге 3.3.

Листинг 3.3. Пример использования функции Invoke

```
01 using UnityEngine;
02 using System.Collections;
03 //-------------------------------------------------------
04 public class MyCustomComponent : MonoBehaviour
05 {
06   // Ссылка на компонент, функция которого должна вызываться
07   public MonoBehaviour Handler = null;
08
09   //-------------------------------------------------------
10   // Этот метод выполняет инициализацию
11   void Start ()
12   {
13     // Вызвать функцию непосредственно
14     Handler.Invoke("OnSave",0.0f);
15   }
16 }
17 //-------------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 3.3:

- **Строка 07**: здесь объявляется общедоступная переменная Handler. В инспекторе объектов на нее можно перетащить какой-либо компонент. Это будет компонент, которому адресовано сообщение. Обратите внимание, что типом переменной может быть MonoBehaviour или производный от него класс. Это означает, что независимость от типа достигнута – нам не нужно знать тип компонента заранее.

- **Строка 14**: метод `Invoke` класса `MonoBehaviour` предназначен для вызова любого метода по его имени. Второй аргумент, десятичное число, определяет время задержки в секундах перед вызовом функции. Если указать задержку, равную `0`, метод будет вызван немедленно.

Более подробную информацию о функции `Invoke` можно найти в электронной документации: http://docs.unity3d.com/ScriptReference/MonoBehaviour.Invoke.html.

Игровые объекты и игровой мир

Еще одной важной задачей в Unity является поиск объектов в сцене, особенно если экземпляры объектов создаются во время выполнения. Задачи, такие как «Получить объект игрока» или «Получить список всех врагов в сцене», используются во многих операциях, от повторного создания врагов и сбора предметов до смены положения игрока и проверки столкновений между объектами. Для получения ссылок на конкретные игровые объекты в Unity предусмотрен набор функций, связанных с классом `GameObject`. Эти функции удобны, но затратны, поэтому постарайтесь вызывать их только в однократно выполняемых событиях, таких как `Start` и `Awake`, если это возможно. Давайте рассмотрим их вместе с другими методами, предназначенными для работы с найденными объектами.

Поиск игровых объектов

Найти объект в сцене можно с помощью двух функций: `GameObject.Find` и `GameObject.FindObjectWithTag`. Почти всегда предпочтительнее использовать последнюю из них, из соображений производительности. Тем не менее, давайте начнем с `GameObject.Find`. Эта функция просматривает содержимое сцены до первого объекта с именем, точно совпадающим с заданным (с учетом регистра), и возвращает найденный объект. Для поиска должно использоваться имя объекта, как оно отображается в панели иерархии. К сожалению, функция выполняет операцию сравнения строк в поисках совпадения, которая выполняется достаточно медленно. Кроме того, она хорошо подходит только для сцен, где все объекты гарантированно имеют уникальные имена, без повторов. Тем не менее, функция `GameObject.Find` остается полезной для поиска объекта по его имени:

```
// Поиск объекта с именем Player
ObjPlayer = GameObject.Find ("Player");
```

Метод Find класса GameObject. Если заглянуть в определение класса `GameObject`, можно увидеть, что `Find` – это статическая функция. То есть, чтобы вызвать ее, не нужно создавать экземпляр класса `GameObject`. Ее можно вызвать, записав инструкцию `GameObject.Find`. Статические члены и глобальная область видимости будут рассматриваться далее в этой главе.
Более подробную информацию о функции `GameObject.Find` можно найти в электронной документации: http://docs.unity3d.com/ScriptReference/GameObject.Find.html.

Функция `GameObject.Find` выполняется медленно, поэтому применяйте ее только в событиях, вызываемых один раз, таких как `Awake` или `Start`.

Намного эффективнее выполняется поиск по тегу. Каждый объект в сцене имеет свойство-тег, которому по умолчанию присваивается значение `Untagged`. Тег – это уникальный идентификатор, которым можно отметить один или несколько объектов, объединив их в коллекцию. Естественно, чтобы получить возможность искать объекты по тегам, сначала нужно присвоить теги объектам. Сделать это можно программно, используя общедоступное свойство `GameObject.tag`. Однако чаще для этого используется редактор Unity. Чтобы присвоить тег выбранному объекту в редакторе Unity, щелкните на раскрывающемся списке **Tag** (Тег) в инспекторе объектов и выберите тег. Так же есть возможность создавать свои теги, выбрав пункт **Add Tag...** (Добавить тег...). На практике часто используют теги **Player** (Игрок), **Enemy** (Враг), **Weapon** (Оружие), **Bonus** (Бонус), **Prop** (Реквизит), **Environment** (Окружение), **Light** (Источник света), **Sound** (Звук) и **GameController** (Контроллер игры). Взгляните на рис. 3.4.

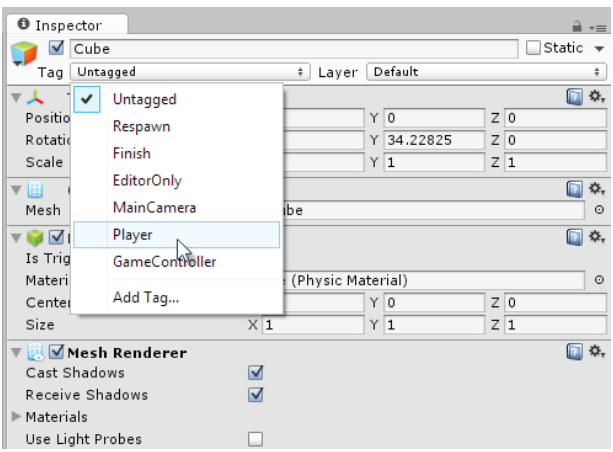


Рис. 3.4. Присваивание тега объекту

После присваивания тегов одному или нескольким объектам в сцене можно эффективно находить объекты по тегу. Функция `GameObject.FindGameObjectWithTag` ищет и возвращает первый найденный объект с указанным тегом. Функция `GameObject.FindObjectsWithTag` возвращает массив всех подходящих объектов. Рассмотрим пример в листинге 3.4. Обратите внимание, что хотя функция `FindGameObjectsWithTag` принимает строковый аргумент, Unity внутренне преобразует строку в число для увеличения скорости сравнения тегов.

Листинг 3.4. Поиск объектов по тегам

```
using UnityEngine;
using System.Collections;
//-------------------------------------------------------
public class ObjectFinder : MonoBehaviour
{
  // Имя тега для поиска оъектов
  public string TagName = "Enemy";

  // Массив найденных объектов
  public GameObject[] FoundObjects;

  //-------------------------------------------------------
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Найти объекты с указанным тегом
    FoundObjects = GameObject.FindGameObjectsWithTag(TagName);
  }
}
//-------------------------------------------------------
```

Иногда бывает желательно присвоить несколько тегов одному объекту. К сожалению, Unity не поддерживает такую возможность. Тем не менее, вы можете обойти данное ограничение, создав пустые дочерние объекты для основного объекта и присвоив им требуемые теги. Выполняя поиск объектов по тегу в этом случае нужно просто помнить, что надо получить ссылку на родительский объект, который, собственно, вам и нужен.

Сравнение объектов

Функции поиска в классе `GameObject` удобны для поиска конкретных объектов, но бывают ситуации, когда нужно сравнить два уже найденных объекта. Как правило, бывает нужно сравнить имена или теги двух объектов. Сравнить теги можно с помощью функции `CompareTag`:

```
// Сравнить тег данного объекта с тегом объекта Obj_Y
bool bMatch = gameObject.CompareTag(Obj_Y.tag);
```

Кроме того, иногда требуется сравнить два объекта на равенство, чтобы определить, один и тот же это объект или это два разных объекта с одинаковыми тегами. Это особенно важно при программировании алгоритмов принятия поведенческих решений. Например, решая вопрос, следует ли вражескому персонажу сражаться или бежать от игрока, неплохо было бы знать, сколько еще вражеских персонажей смогут поддержать его. Чтобы ответить на этот вопрос, как мы уже знаем, поиск всех врагов в сцене можно выполнить по тегу. Однако, результаты такого поиска будут также включать первого врага, и его необходимо исключить из результатов. Листинг 3.5 демонстрирует, как решить эту задачу с помощью метода `GetInstanceID`.

Листинг 3.5. Сравнение объектов

```
01 // Найти все объекты с указанным тегом
02 FoundObjects = GameObject.FindGameObjectsWithTag(TagName);
03
04 // Находит указанный объект в списке и исключает его
05 foreach(GameObject O in FoundObjects)
06 {
07   // Если это один и тот же объект
08   if(O.GetInstanceID() == gameObject.GetInstanceID())
09     continue; // Перейти к следующей итерации
10
11   // [...] Выполнить необходимые операции
12 }
```

Получение ближайшего объекта

Имеется массив игровых объектов игры, полученный, например, в результате поиска. Как найти объект, ближайший к заданному, в смысле расстояния? Пример в листинге 3.6 демонстрирует, как решить эту задачу с помощью функции `Vector3.Distance`, вычисляющей расстояние (в метрах) между любыми двумя точками в сцене.

Листинг 3.6. Поиск ближайшего объекта

```
// Возвращает ближайший игровой объект
GameObject GetNearestGameObject(GameObject Source, GameObject[]
DestObjects)
{
  // Извлечь ссылку на первый объект
  GameObject Nearest = DestObjects[0];

  // Вычислить расстояние
  float ShortestDistance = Vector3.Distance(
          Source.transform.position,
```

```
          DestObjects[0].transform.position);

  // Обойти в цикле все объекты
  foreach(GameObject Obj in DestObjects)
  {
    // Вычислить расстояние
    float Distance = Vector3.Distance(
            Source.transform.position, Obj.transform.position);

    // Если этот объект ближе к заданному, запомнить его
    if(Distance < ShortestDistance)
    {
      // Ближе, запомнить
      Nearest = Obj;
      ShortestDistance = Distance;
    }
  }

  // Вернуть ближайший объект
  return Nearest;
}
```

Поиск любого объекта определенного типа

Иногда бывает нужно получить список всех компонентов определенного типа, независимо от того, к каким игровым объектам они подключены. Это могут быть компоненты, подключенные ко всем врагам, всем собираемым предметам, все компоненты определения местоположения, все коллайдеры и т. д. Решить эту задачу можно с помощью функции `Object.FindObjectsOfType`, как показано в листинге 3.7, предназначенной для получения списка всех экземпляров заданного типа, за исключением неактивных объектов. Но, из-за невысокой скорости работы этой функции старайтесь избегать ее в событиях, связанных с кадрами, таких как `Update`. Ее имеет смысл использовать в редко вызываемых событиях, таких как события `Start` и `Awake`.

Листинг 3.7. Получение списка всех объектов заданного типа

```
void Start()
{
  // Получить список всех коллайдеров в сцене
  Collider[] Cols = Object.FindObjectsOfType<Collider>();
}
```

Проверка препятствий между игровыми объектами

Часто возникает задача проверить отсутствие препятствий между двумя выбранными игровыми объектами, такими как игрок и враг,

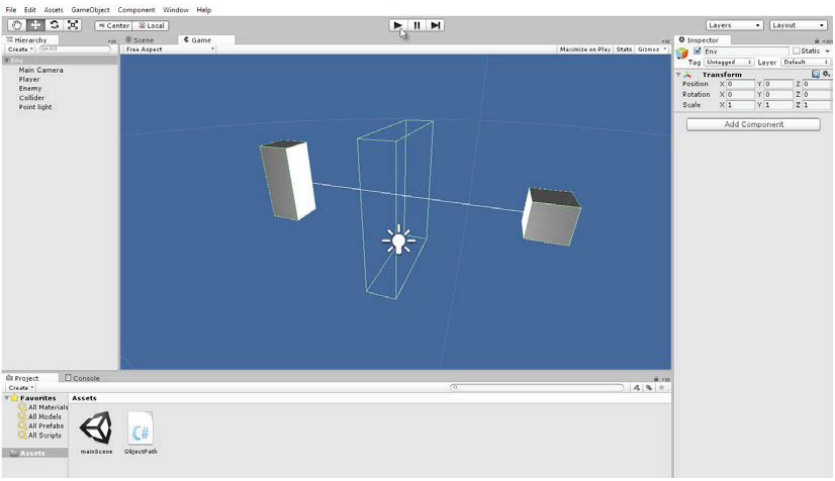


Рис. 3.5. Проверка отсутствия препятствий между двумя игровыми объектами с помощью `Physics.LineCast`

иначе говоря, проверить присутствие коллайдеров, пересекающих воображаемую линию между объектами. Это может пригодиться в системах, определяющих прямую видимость, как будет показано ниже, или, в более общем случае, для выбора объектов в реализациях искусственного интеллекта и др.

Существует множество способов решения этой задачи. Один из них основан на использовании функции `Physics.LineCast`, как показано в листинге 3.8.

Листинг 3.8. Определение отсутствия препятствий между объектами

```
01 using UnityEngine;
02 using System.Collections;
03 // Проверяет отсутствие препятствий на воображаемой линии между
      объектами
04 public class ObjectPath : MonoBehaviour
05 {
06    // Ссылка на вражеский объект
07    public GameObject Enemy = null;
08
09    // Маска слоев для ограничения определения препятствий на линии
10    public LayerMask LM;
11    //-------------------------------------------------------
12    // Вызывается при отображении каждого кадра
13    void Update ()
14    {
```

```
15      // Проверить отсутствие препятствий между объектами
16      if(!Physics.Linecast(transform.position,
                             Enemy.transform.position, LM))
17      {
18        // Препятствий нет
19        Debug.log ("Path clear");
20      }
21    }
22    //-----------------------------------------------------
23    // Показать вспомогательную линию в режиме отладки
24    void OnDrawGizmos()
25    {
26      Gizmos.DrawLine(transform.position, Enemy.transform.position);
27    }
28    //-----------------------------------------------------
29  }
```

Ниже приводится несколько комментариев к листингу 3.8:

- **Строка 07**: этот класс должен быть прикреплен к объекту `Player`, а в его общедоступную переменную `Enemy` должна быть записана ссылка на объект, отсутствие препятствий на пути к которому должно быть проверено.
- **Строка 10**: переменная `LayerMask` – это битовая маска, определяющая, какие слои в сцене следует учитывать при проверке. Более подробную информацию о битовых масках можно найти в электронной документации: http://docs.unity3d.com/Manual/Layers.html.
- **Строка 16**: функция `Physics.Linecast` определяет, возможно ли соединить два объекта в сцене непрерывной прямой линией. Обратите внимание, что если два проверяемых объекта снабжены коллайдерами, такими, например, как коллайдер `BoxCollider`, они будут рассматриваться как препятствия. Другими словами, собственный коллайдер объекта может повлиять на результаты функции `LineCast`. Поэтому используйте переменную `LayerMask` для включения и исключения нужных слоев.

Проект примера использования функции `Physics.LineCast` можно найти в сопроводительных файлах к книге, в папке `Chapter03/LineCast`.

Доступ к иерархии объектов

Панель **Hierarhy** (Иерархия) в Unity является графическим представлением иерархических связей между игровыми объектами в сцене. Эти отношения играют важную роль, потому что дочерние объек-

ты наследуют трансформации своих родителей. Однако возможности определять и редактировать иерархические взаимосвязи только в редакторе, как правило, не достаточно. Часто бывает нужно связать два объекта программно, а также перебрать в цикле все дочерние объекты выбранного объекта для их обработки или вызова их функций. Давайте сначала посмотрим, как установить отношение родитель-потомок между объектами. В следующем примере (листинг 3.9) демонстрируется, как присоединить один объект к другому в качестве дочернего с помощью компонента `Transform`.

Листинг 3.9. Присоединение объекта

```
using UnityEngine;
using System.Collections;
//-----------------------------------------------------
public class Parenter : MonoBehaviour
{
  // Ссылка на дочерний объект
  private GameObject Child;
  // Ссылка на родительский объект
  private GameObject Parent;
  //-----------------------------------------------------
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Получить ссылки на родительский и дочерний объекты
    Child = GameObject.Find("Child");
    Parent = GameObject.Find("Parent");

    // Установить связь между ними
    Child.transform.parent = Parent.transform;
  }
  //-----------------------------------------------------
}
//-------------------------------------------------------
```

Теперь посмотрим, как перебрать в цикле все дочерние объекты выбранного родительского объекта. Сделать это можно с помощью того же компонента `Transform`, как показано в листинге 3.10.

Листинг 3.10. Обход дочерних объектов

```
using UnityEngine;
using System.Collections;
//-----------------------------------------
public class CycleChildren : MonoBehaviour
{
  //-----------------------------------------
```

```
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Обойти все дочерние объекты
    for(int i=0; i<transform.childCount; i++)
    {
      // Вывести имя потомка в консоль
      Debug.log (transform.GetChild(i).name);
    }
  }
  //-----------------------------------------
}
//-----------------------------------------
```

Игровой мир, время и обновление

Сцена в Unity – это коллекция игровых объектов в одном трехмерном пространстве, а также во времени. Каждая игра нуждается в определении единого понятия времени для синхронизации анимации и обновлений, потому что анимация – это изменение с течением времени. В Unity для работы со временем применяется класс `Time`. Умение работать с этим классом очень важно для создания предсказуемых и последовательных движений в играх. Подробнее мы обсудим эту тему чуть ниже.

Каждая игра имеет свою частоту смены кадров, которая измеряется в **кадрах в секунду** (Frames Per Second, сокращенно FPS). Увидеть значение этого показателя можно в панели **Stats** (Статистики), во вкладке **Game** (Игра). Частота кадров определяет, сколько раз в секунду Unity может выполнить код игры, предназначенный для вывода изображений с камер на экране. Каждая такая итерация называется **кадром** (frame). Частота кадров может резко меняться с течением времени и на разных компьютерах. Она зависит от мощности компьютера, наличия других процессов, сложности сцены для текущего кадра и других факторов. Это значит, что никогда нельзя полагаться на неизменность частоты кадров во времени на одном и том же компьютере, а тем более при смене компьютера. Взгляните на рис. 3.6.

Для работы с кадрами в Unity предусмотрены три вида событий, которые могут быть реализованы в любом классе `MonoBehaviour` для выполнения непрерывного обновления с течением времени. С этими событиями – `Update`, `FixedUpdate` и `LateUpdate` – мы уже знакомы, теперь рассмотрим их более подробно и предметно:

- `Update`: событие `Update` вызывается *один* раз для каждого кадра в каждом активном компоненте каждого активного игро-

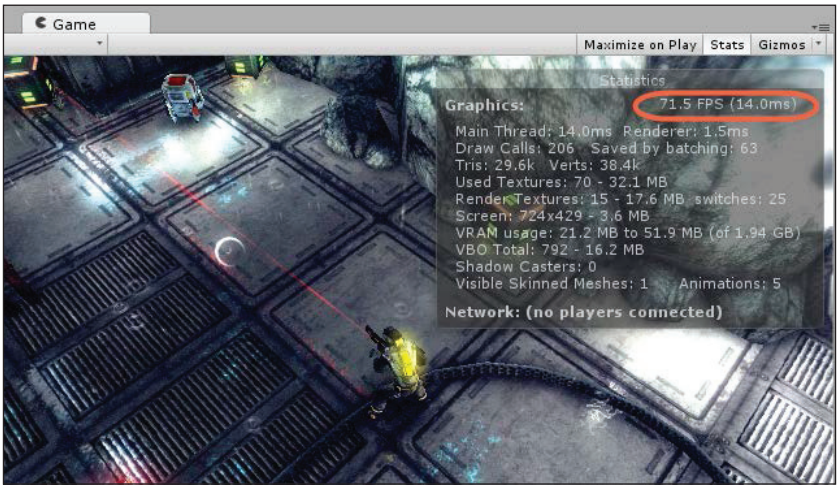


Рис. 3.6. Частота кадров важна
для создания основанных на времени действий и анимаций

вого объекта. Если объект выключен с помощью метода `MonoBehaviour.SetActive`, событие `Update` не будет вызываться для этого объекта, пока он не будет активирован. Проще говоря, событие `Update` наиболее точно соответствует понятию кадра в Unity и применяется для выполнения повторяющихся действий или вызова функций обновления или наблюдения, например, за событиями ввода: нажатиями клавиш или щелчками мышью. Обратите внимание, что в пределах кадра очередность обработки событий `Update` в разных компонентах не определена, то есть нельзя утверждать, что функция `Update` объекта X будет вызвана раньше, чем функция `Update` объекта Y, или наоборот.

- `FixedUpdate`: это событие не привязано к кадрам и может вызываться несколько раз в кадре. Однако, оно вызывается регулярно и нормированно, через фиксированные интервалы времени. Чаще всего событие `FixedUpdate` используется для эмуляции физических характеристик объектов. Если, к примеру, нужно обновлять скорость или свойства компонента `Rigidbody` с течением времени, событие `FixedUpdate` подойдет для этого лучше, чем `Update`.
- `LateUpdate`: это событие вызывается в каждом кадре, как и событие `Update`. С той лишь разницей, что `LateUpdate` всегда вы-

зывается после событий Update и FixedUpdate. Это значит, что при вызове события LateUpdate можно быть уверенным, что события Update и FixedUpdate уже были вызваны для всех объектов в текущем кадре. Это делает LateUpdate удобным местом для изменения положения камеры, особенно в играх от третьего лица, так как гарантируется, что положение камеры всегда будет совпадать с последним положением объекта в текущем кадре.

Нюансы использования событий Update, FixedUpdate и LateUpdate в сочетании с понятиями времени и частоты кадров оказывают значительное влияние на программирование движений в играх. В частности, существуют два основных направляющих принципа, рассматриваемых в следующих двух подразделах.

Правило № 1 – важность событий обновления кадров

Кадры должны сменяться много раз в секунду, иначе игра будет замедляться и дергаться. В каждом кадре для каждого активного экземпляра класса MonoBehaviour один раз вызывается событие Update. Это значит, что сложность расчетов (и производительность) в каждом кадре в значительной мере зависит от происходящего внутри событий Update. Больший объем возможностей требует больше времени на обработку и увеличивает нагрузку на процессор или на графический сопроцессор. В больших сценах, с множеством объектов и компонентов, обновление кадров легко может выйти из-под контроля, если не позаботиться о снижении нагрузки на события Update, тщательно планируя их код. Важно рационально относиться к Update и любым другим регулярным вызываемым событиям, связанным с кадрами. Проще говоря, они должны содержать только самый необходимый код, например для чтения ввода игрока или наблюдения за перемещением курсора. Серьезно уменьшить нагрузку на функции Update вам поможет событийное программирование. Событийное программирование и система событий будут рассмотрены в следующей главе.

Правило № 2 – движение должно основываться на времени

Так как нельзя гарантировать неизменность частоты кадров (она будет колебаться с течением времени и отличаться на разных компьютерах), нужно писать код для реализации движений и изменений так, чтобы вне зависимости от частоты кадров они выглядели одинаково

у всех игроков. Рассмотрим простой случай равномерного перемещения объекта кубической формы. Один из способов (не самый лучший) реализации такого движения приведен в листинге 3.11.

Листинг 3.11. Реализация (не самая лучшая) равномерного перемещения объекта

```
using UnityEngine;
using System.Collections;
public class Mover : MonoBehaviour
{
  // Расстояние перемещения объекта в каждом кадре
  public float AmountToMove = 1.0f;
  // Вызывается при отображении каждого кадра
  void Update ()
  {
    // Переместить куб вдоль оси x
    transform.localPosition += new Vector3(AmountToMove,0,0);
  }
}
```

Этот код работает и в каждом кадре перемещает объект, к которому он прикреплен, на расстояние, определяемое значением переменной `AmountToMove`. Проблема – в его зависимости от частоты кадров. Так как частота кадров меняется и отличается для разных компьютеров, каждый пользователь будет видеть разную картину, в нашем случае это движение куба с разными скоростями. Это плохо, потому что нельзя предсказать, как игра будет работать у конкретного пользователя. Чтобы исправить это, нужно связать движение со временем, а не со сменой кадров. Частота кадров меняется, но ход времени постоянен, одна секунда – всегда одна секунда. Чтобы добиться нужного результата, используем переменную `DeltaTime`, которая является свойством класса `Time`. Рассмотрим следующий пример в листинге 3.12. Это исправленная версия примера из листинга 3.11.

Листинг 3.12. Исправленная реализация равномерного перемещения объекта

```
using UnityEngine;
using System.Collections;
public class Mover : MonoBehaviour
{
  // Скорость перемещения
  public float Speed = 1.0f;

  // Вызывается при отображении каждого кадра
  void Update ()
```

```
  {
    // Переместить куб в соответствии с заданной скоростью
    transform.localPosition +=
      transform.forward * Speed * Time.deltaTime;
  }
}
```

Переменная `DeltaTime` хранит вещественное число, выражающее время в секундах, прошедшее с момента предыдущего вызова функции `Update`. Значение `0.5`, например, означает, что с момента смены кадров прошло полсекунды и т. д. Поэтому переменную `DeltaTime` можно использовать как множитель. Умножая значение скорости на значение переменной `DeltaTime` в каждом кадре, мы получим расстояние перемещения для объекта, так как *расстояние* = *скорость* × *время*. То есть, переменная `DeltaTime` обеспечивает независимость движения объекта от частоты кадров.

Неуничтожаемые объекты

По умолчанию Unity рассматривает каждый объект, как существующий в пространстве и времени только одной активной сцены. Разница между сценами подобна разнице между отдельными вселенными. Как следствие, объекты не могут существовать вне сцены, которой они принадлежат, то есть они уничтожаются при смене активной сцены. Как правило, это именно то, что нужно, потому что сцены обычно совершенно независимы и должны быть отделены друг от друга. Тем не менее, даже в этом случае есть объекты, которые не должны уничтожаться. Примерами объектов, которые *должны переноситься из сцены в сцену*, могут служить: персонаж игрока, система подсчета баллов или класс `GameManager`. Это, как правило, объекты высшего порядка, существование которых не должно ограничиваться одной конкретной сценой – они должны присутствовать в нескольких сценах. Вопрос сохранения объекта при смене сцены легко разрешим с помощью функции `DontDestroyOnLoad`, но применение этой функции принесет с собой важные последствия, на которых нужно остановиться подробно. Взгляните на следующий пример в листинге 3.13.

Листинг 3.13. Неуничтожаемые объекты

```
using UnityEngine;
using System.Collections;
//-----------------------------------------
```

```
// Этот объект будет перемещаться между сценами
public class PersistentObj : MonoBehaviour
{
  //-----------------------------------------
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Сделать объект неуничтожаемым
    DontDestroyOnLoad(gameObject);
  }
}
//-----------------------------------------
```

Переход объектов из сцены в сцену необходим, но *путешествующие объекты* берут с собой багаж, который путешествует по сценам вместе с ними. Это значит, что также не будут уничтожаться все дочерние объекты переносимого объекта и любые ресурсы, которые он использует, например меши, текстуры, звуки и прочее. Это не является проблемой само по себе, но важно, чтобы вы знали об этом. По этой причине многие неуничтожаемые объекты создаются максимально легковесными, то есть из пустых игровых объектов, без дочерних объектов, только с самыми основными компонентами, необходимыми для их функционирования. Это гарантирует, что только самые важные данные будут перенесены в другие сцены.

Смена сцен. Для смены активной сцены в Unity используется функция `Application.LoadLevel`. Существуют несколько разновидностей этой функции: `LoadLevelAsync`, `LoadLevelAdditive` и `LoadLevelAdditiveAsync`. Более подробную информацию о функциях загрузки сцен можно найти по адресу http://docs.unity3d.com/ScriptReference/Application.html.

Как мы видели выше, для защиты объекта от разрушения при смене сцен, вызывается его функция `DontDestroyOnLoad`. Но при этом иногда возникает проблема дублирования объектов. В частности, если позднее произойдет возврат к исходной сцене, где был создан неуничтожаемый объект, сценарий создаст его копию, то есть в сцене будут присутствовать неуничтоженный оригинальный объект, пришедший из предыдущей сцены, и новый, вновь созданный для загруженной сцены. Причем эта проблема будет усугубляться с каждым возвратом к сцене, так как каждый раз будет создаваться новая копия объекта. Такое дублирование, как правило, вовсе не то, что вам нужно. Обычно нужен только один экземпляр объекта: один игрок, один диспетчер игры или одно табло с баллами. Чтобы добиться этого, нужно создать объект-одиночку, или синглтон (singleton), как описывается в следующем разделе.

Синглтоны и статические переменные

Правила создания экземпляров некоторых классов могут принципиально отличаться от правил создания экземпляров других классов. Большинство классов представляют собой шаблоны с наборами свойств и методов, которые могут быть многократно воссозданы в сцене в виде игровых объектов. Класс вражеского персонажа можно использовать для создания множества экземпляров противника, а класс электрической батареи – для множества объектов батарей и т. д. Однако некоторые классы, такие как `GameManager`, `HighScoreManager`, `AudioManager` или `SaveGameManager`, должны существовать в единственном экземпляре. Такая особенность объединяет их в единую группу. Проще говоря, в каждый момент времени может существовать один и только один экземпляр такого класса. Существование нескольких экземпляров либо бессмысленно, либо сделает функционирование объекта невозможным или вредным. Эти виды объектов называют синглтонами[1]. Синглтоны часто являются неуничтожаемыми объектами, сохраняющимися при смене сцен, хотя это и не обязательно. Единственная существенная особенность синглтонов (что и делает их синглтонами, то есть, одиночками) – это невозможность одновременного существования более одного экземпляра. Давайте теперь создадим синглтон в виде экземпляра класса `GameManager`.

Практически каждая игра имеет в своем составе или экземпляр класса `GameManager`, или экземпляр класса `GameController`, как правило, это неуничтожаемые синглтоны. Класс `GameManager` отвечает в игре за все высокоуровневые функции. Он должен определять, приостановлена ли игра, была ли достигнута победа и иметь надежный способ знать, что происходит в игре в любой момент времени. Рассмотрим типичный пример реализации класса `GameManager` в листинге 3.14.

Листинг 3.14. Типичный пример реализации класса GameManager

```
using UnityEngine;
using System.Collections;
//----------------------------------------
// Пример класса диспетчера игры
public class GameManager : MonoBehaviour
{
  //----------------------------------------
  // Высший результат
```

[1] От англ. singleton – одиночка. – *Прим. ред.*

```
  public int HighScore = 0;

  // Признак приостановки игры
  public bool IsPaused = false;

  // Признак поддержки ввода пользователя
  public bool InputAllowed = true;
  //---------------------------------------
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Сделать диспетчера игры неуничтожаемым
    DontDestroyOnLoad(gameObject);
  }
  //---------------------------------------
}
//---------------------------------------
```

Этот объект будет сохраняться при смене сцен, но как сделать его (или любой другой объект) синглтоном? Ответ на этот вопрос демонстрирует листинг 3.15.

Листинг 3.15. Создание синглтона

```
01 using UnityEngine;
02 using System.Collections;
03 //---------------------------------------
04 // Пример класса диспетчера игры - синглтона
05 public class GameManager : MonoBehaviour
06 {
07   //---------------------------------------
08   // Свойство C# для доступа к экземпляру синглтона
09   // Доступно только для чтения - метод set отсутствует
10   public static GameManager Instance
11   {
12     // вернуть ссылку на экземпляр
13     get
14     {
15       return instance;
16     }
17   }
18
19   //---------------------------------------
20   private static GameManager instance = null;
21   //---------------------------------------
22   // Высший результат
23   public int HighScore = 0;
24
25   // Признак приостановки игры
26   public bool IsPaused = false;
```

```
27
28   // Признак поддержки ввода пользователя
29   public bool InputAllowed = true;
30   //-----------------------------------------
31   // Этот метод выполняет инициализацию
32   void Awake ()
33  {
34    // Проверить наличие экземпляра класса
35    // Если присутствует - уничтожить текущий экземпляр
36    if(instance)
37    {
38      DestroyImmediate(gameObject);
39      return;
40    }
41
42    // Сделать активным и единственным данный экземпляр
43    instance = this;
44
45    // Сделать диспетчера игры неуничтожаемым
46     DontDestroyOnLoad(gameObject);
47   }
48   //-----------------------------------------
49 }
50 //-----------------------------------------
```

Ниже приводится несколько комментариев к листингу 3.15.

- **Строки 10–20**: в класс `Manager` добавлена закрытая переменная `instance`, объявленная как статическая (`static`). Это означает, что переменная совместно используется всеми экземплярами класса, а не является собственностью каждого отдельного экземпляра. Это позволяет при создании каждого нового экземпляра определить присутствие созданного ранее экземпляра. Эта переменная сделана общедоступной с помощью свойства `Instance`, имеющего только один метод `get`, то есть это свойство предназначено лишь для чтения.
- **Строки 36–43**: здесь, в событии `Awake` (вызывается при создании объекта), выполняется проверка переменной `instance`, чтобы определить присутствие в текущей сцене ранее созданного экземпляра класса. Если такой экземпляр существует, текущий объект удаляется, потому что допускается существование лишь одного экземпляра этого класса и он уже существует. Таким образом, объект класса `GameManager` будет сохраняться при смене сцен, и всегда будет существовать только один, первый созданный экземпляр.

События Awake и Start. Класс `GameManager` в листинге 3.15 использует функцию `Awake` вместо `Start`. Разница между функциями `Start` и `Awake` заключается в следующем:

- функция `Awake` всегда вызывается перед функцией `Start`;
- функция `Awake` всегда вызывается при создании объекта. Функция `Start` вызывается в первом кадре, где игровой объект становится активным. Если игровой объект начинает сцену в неактивном состоянии, функция `Start` не будет вызвана, пока объект не будет активирован. Для активных по умолчанию объектов функция Start вызывается при запуске сцены, после функции `Awake`.

Если потребуется кэшировать ссылки на компоненты в локальных переменных класса, как, например, ссылка на компонент `Transform` была помещена в переменную `ThisTransform`, лучше использовать для этого событие `Awake`, а не `Start`. При обработке события `Start` предполагается, что все локальные ссылки на объекты уже подготовлены.

Главное преимущество наличия глобального статического свойства `Instance` в классе `GameManager` в том, что оно непосредственно доступно в любом другом файле сценария, без необходимости использовать какие-либо локальные переменные или ссылки на объекты. Это означает, что каждый класс будет иметь непосредственный доступ ко всем свойствам экземпляра класса `GameManager` и использовать функционал высокого уровня. Например, чтобы изменить счет игры из другого класса, достаточно использовать код из листинга 3.16.

Листинг 3.16. Простота использования статического свойства класса

```
using UnityEngine;
using System.Collections;
//-----------------------------------------
public class ScoreSetter : MonoBehaviour
{
  //-----------------------------------------
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Изменить счет в GameManager
    GameManager.Instance.HighScore = 100;
  }
  //-----------------------------------------
}
//-----------------------------------------
```

Более подробную информацию о синглтонах можно найти по адресу http://unitypatterns.com/singletons/.

Итоги

В этой главе были рассмотрены игровые объекты, сцены, компоненты и их совместное использование. Эти вопросы, на первый взгляд, могут показаться простыми, но их понимание и способность их использования для управления объектами совершенно необходимо для разработки игр в Unity. В частности, мы рассмотрели игровой объект, как набор компонентов, взаимодействующих для создания единого поведения. Подчеркнули особую значимость компонента `Transform`. Мы также рассмотрели сцены. Сцена представляет собой единство времени и пространства, внутри которого существуют игровые объекты. Как правило, сцена является изолированной средой, объекты которой не могут существовать за ее пределами. Кроме того, каждая сцена выполняется во времени, что делает возможным изменения и анимацию. Время может быть измерено с помощью свойства `DeltaTime`, которое используется как множитель и позволяет добиться независимости от частоты кадров. Наконец, мы исследовали шаблон проектирования «Одиночка» (синглтон), использующий статические переменные в определении класса, для которого можно создать только один активный экземпляр. В следующей главе мы перейдем к событийному программированию.

Глава 4

Событийное программирование

События Update для объектов MonoBehaviour кажутся удобным местом для выполнения операций, регулярно повторяющихся в течение длительного периода времени, охватывающего несколько кадров и, возможно, несколько сцен. При создании устойчивого поведения во времени, например искусственного интеллекта для врагов или непрерывного движения, может показаться, что практически нет альтернативы заполнению функции Update многочисленными операторами if и switch, служащими для ветвления кода в разных направлениях, в зависимости от того, что должны сделать объекты в текущий момент времени. Но, если рассматривать событие Update как место по умолчанию для реализации длительных действий, это приведет к серьезным проблемам с производительностью в крупных и сложных играх. При более глубоком анализе нетрудно догадаться, почему это именно так. Как правило, игра заполнена массой действий, которые происходят одновременно и в одном месте, и реализация их всех только с помощью функций Update просто невозможна. Рассмотрим персонажи врагов в состоянии покоя, они должны знать, когда игрок входит в зону их прямой видимости и выходит из нее, когда уровень их здоровья слишком низок, когда заканчиваются боеприпасы, когда они находятся на опасной местности, когда они получают повреждения, когда движутся или стоят на месте и многое другое. При размышлении об этом спектре состояний кажется, что все они требуют постоянного и непрерывного отслеживания, потому что враги всегда должны быть готовы мгновенно отреагировать на изменения, вызванные действиями игрока. Это и есть главная причина, почему функция Update выглядит самым подходящим местом в такой ситуации, но есть и лучшее решение, а именно событийное (или событийно-ориентированное) программирование[1].

[1] https://ru.wikipedia.org/wiki/Событийно-ориентированное_программирование. – *Прим. ред.*

Взглянув на игру с точки зрения событий, вы сможете значительно увеличить ее производительность. Эта глава посвящена событиям и управлению игрой с их помощью.

События

Игровые миры представляют собой полностью детерминированные системы. В Unity сцена представляет общее декартово трехмерное пространство и время, внутри которых существует конечный набор игровых объектов. События в этом пространстве происходят, только когда логика игры и код позволяют это. Например, объекты могут двигаться, только когда некоторый код управляет их перемещениями, или при выполнении определенных условий, таких как нажатие клавиши на клавиатуре. Заметим, что поведение объектов не является случайным, оно всегда чем-то обусловлено: объекты движутся, только когда происходят события ввода с клавиатуры. Здесь установлена важная связь между действиями, одно действие влечет за собой другое. Эти связи и называются событиями; каждая единичная связка представляет одно событие. События не активны, они пассивны; представляют собой моменты возможности действий, но не действия сами по себе, например нажатие клавиши, щелчок мышью, объект вошел внутрь коллайдера, игрок подвергся нападению и т. д. Все эти события не сообщают программе, что она должна сделать, а только уведомляют о произошедшем. Событийное программирование начинается с признания за событиями роли всеобщего понятия, и рассмотрения почти каждого обстоятельства в игре как экземпляра события, то есть как события во времени. Не как понятия события вообще, а как конкретного события, происходящего в определенный момент времени. При таком подходе к событиям все действия в игре можно рассматривать как прямые отклики на происходящие события. В частности, события связаны с откликами; событие происходит и вызывает отклик (реакцию). Кроме того, реакция может пойти дальше и, в свою очередь, стать событием, которое запустит следующие отклики и т. д. Другими словами, игровой мир является полной и интегрированной системой событий и откликов на них. После того как общий подход определен, возникает вопрос, как это может помочь улучшить производительность, по сравнению с простым применением функции `Update`, для выполнения изменений в каждом кадре. Ответ заключается в поиске способов уменьшить частоту возникновения событий. Эта стратегия может показаться сырой, но

в ней заключена суть метода. Для иллюстрации рассмотрим пример реализации стрельбы вражеского персонажа в игрока во время боя.

В процессе игры враг должен отслеживать множество показателей. Во-первых, свое здоровье, потому что когда его уровень сильно понижается, ему нужно заняться поиском аптечки для его восстановления. Во-вторых, патроны, потому что когда они заканчиваются, ему также нужно будет заняться их поиском и, кроме того, беречь патроны, стреляя в игрока, только когда он оказывается в области прямой видимости. Вот так, просто размышляя об этой проблеме, мы уже наметили некоторые связи между действиями, которые можно рассматривать как события. Но прежде чем развить эту тему, давайте посмотрим, как реализовать такое поведение с помощью функции `Update` (см. листинг 4.1). А затем обсудим, как события смогут помочь улучшить эту реализацию.

Листинг 4.1. Упрощенная реализация с помощью функции Update

```
// Вызывается при отображении каждого кадра
void Update ()
{
  // Проверить здоровье врага
  // Убит?
  if(Health <= 0)
  {
    // Сымитировать смерть
    Die();
    return;
  }
  // Уровень здоровья слишком низкий?
  if(health <= 20)
  {
    // Уровень низкий, отправиться на поиски аптечки
    RunAndFindHealthRestore();
    return;
  }
  // Проверить патроны
  // Патроны кончились?
  if(Ammo <= 0)
  {
    // Отправиться на поиски патронов
    SearchMore();
    return;
  }
  // Здровье в порядке, патроны есть.
  // Игрок в поле зрения? Если да – выстрелить
  if(HaveLineOfSight)
  {
    FireAtPlayer();
  }
}
```

В листинге 4.1 приведена «тяжелая» функция Update, наполненная множеством проверок состояний и вызовов ответных реакций. В сущности, функция Update пытается объединить в себе обработку событий и реакцию на них, а это излишне затратно. Если проследить связи для наблюдаемых показателей (проверка здоровья и проверка боеприпасов), можно найти пути значительного улучшения кода. Например, количество боеприпасов изменяется только в двух случаях: когда оружие стреляет и когда подбирается новый боезапас. Точно так же и здоровье меняется только в двух случаях: когда враг успешно атакован игроком и когда враг подобрал аптечку. В первом случае происходит уменьшение, во втором – увеличение.

Поскольку это единственные моменты, когда происходит изменение свойств (события), только в эти моменты значения этих свойств и нужно проверять. Рассмотрим следующий пример (листинг 4.2) управления вражеским персонажем, который содержит свойства C# и значительно уменьшившуюся функцию Update.

Листинг 4.2. Улучшенная реализация на основе событий

```
using UnityEngine;
using System.Collections;

public class EnemyObject : MonoBehaviour
{
  //-------------------------------------------------------
  // Свойства C# для закрытых переменных
  public int Health
  {
    get{return _health;}
    set
    {
      // Привести уровень здоровья в диапазон 0-100
      _health = Mathf.Clamp(value, 0, 100);

      // Убит?
      if(_health <= 0)
      {
        OnDead();
        return;
      }

      // Проверить уровень здоровья
      // и сгенерировать событие при необходимости
      if(_health <= 20)
      {
        OnHealthLow();
        return;
```

```
      }
    }
  }
  //-------------------------------------------------------
  public int Ammo
  {
    get{return _ammo;}
    set
    {
      // Привести боезапас в диапазон 0-50
      _ammo = Mathf.Clamp(value,0,50);

      // Проверить отсутствие патронов
      if(_ammo <= 0)
      {
        // Вызвать событие опустошения боезапаса
        OnAmmoExpired();
        return;
      }
    }
  }
  //-------------------------------------------------------
  // Внутренние переменные: уровень здоровья и количество патронов
  private int _health = 100;
  private int _ammo = 50;
  //-------------------------------------------------------
  // Вызывается при отображении каждого кадра
  void Update ()
  {
  }
  //-------------------------------------------------------
  // Это событие вызывается при низком уровне здоровья
  void OnHealthLow()
  {
    // Реализовать здесь отклик на событие
  }
  //-------------------------------------------------------
  // Это событие вызывается, когда враг погибает
  void OnDead()
  {
    // Реализовать здесь отклик на событие
  }
  //-------------------------------------------------------
  // Это событие вызывается, когда заканчивается боезапас
  void OnAmmoExpired()
  {
    // Реализовать здесь отклик на событие
  }
  //-------------------------------------------------------
}
```

Класс врага в листинге 4.2 был переработан в стиле событийного программирования, и теперь такие свойства, как `Ammo` и `Health`, проверяются не внутри функции `Update`, а в моменты присваивания им новых значений. Поэтому события возникают, только когда они необходимы. Приняв за основу событийный подход, мы достигли оптимизации производительности и навели чистоту в нашем коде. Мы избавились от лишних проверок в функции `Update` и позволили операциям присваивания новых значений запускать код событий, зная, что они будут вызываться только в соответствующие моменты времени.

Управление событиями

Событийное программирование может значительно упростить нам жизнь. Но, начав разработку, основанную на событиях, мы сталкиваемся с чередой новых проблем, требующих радикального решения. В частности, мы видели в листинге 4.2, как использовать свойства C# для проверки здоровья и наличия боеприпасов, и при обнаружении соответствующих изменений инициировать другие события (например, `OnDead`), когда это необходимо. Это прекрасно работает в принципе, по крайней мере персонаж врага будет уведомляться о событиях, происходящих в нем самом. Однако, что если врагу нужно знать о смерти другого врага или он должен узнать, когда было убито определенное количество других врагов? В этом конкретном случае мы могли бы вернуться к классу врага в листинге 4.2 и изменить его, вызвав событие `OnDead` не только для текущего экземпляра, но и для всех других врагов с помощью такой функции, как `SendMessage`, как мы это видели в предыдущих главах. Но это не решит нашей проблемы в общем виде. А теперь давайте сформулируем сразу идеальный подход: нам нужно, чтобы каждый объект отслеживал необходимые типы событий и получал уведомления о них так же легко, как если бы событие было связано непосредственно с ним самим. Теперь мы столкнулись с вопросом создания оптимизированной системы, позволяющей с легкостью контролировать события. Проще говоря, нам нужен класс управления событиями `EventManager`, который позволит объектам получать уведомления о конкретных событиях. Эта система основана на трех основных идеях, а именно:

- `EventListener`: определение «получатель событий» применимо к любому объекту, который должен получать уведомления о событии, даже если это событие связано с ним самим. На практике почти каждый объект будет получателем, по меньшей мере,

одного события. Враг, например, среди всего прочего нуждается в уведомлениях о низком уровне здоровья и о малом количестве боеприпасов. В этом случае он будет получателем, по меньшей мере, двух отдельных событий. То есть, всякий раз, когда объект ожидает, что ему сообщат о событии, он становится получателем.

- `EventPoster`: когда объект обнаруживает, что произошло событие, он должен уведомить всех получателей, или разместить объявление о событии. В листинге 4.2 класс врага обнаруживает события снижения уровня здоровья и исчерпания боезапаса, используя свойства, и вызывает внутренние события, если это требуется. Но чтобы это в полной мере соответствовало обобщенному подходу, требуется, чтобы объект инициировал события на глобальном уровне.
- `EventManager`: и наконец, всеобъемлющий синглтон `EventManager`, который сохраняется при переходе между уровнями и доступен глобально. Этот объект эффективно связывает получателей с отправителями. Он принимает уведомления о событиях, посылаемых отправителями, и сразу же передает их соответствующим получателям в форме событий.

Основы управления событиями с помощью интерфейсов

Первым, или начальным субъектом в системе обработки событий является получатель, который должен уведомляться о конкретных событиях по мере их наступления. Потенциально получателем может быть объект любого типа, он просто ждет уведомлений об определенных событиях. Проще говоря, получатель должен быть зарегистрирован в `EventManager` в качестве получателя одного или более конкретных событий. Затем, когда событие действительно происходит, получатель должен быть уведомлен посредством вызова функции. Тут возникает технический вопрос об определении менеджером `EventManager` типа получателя, которого он должен уведомить о событии, потому что получателем может быть объект любого типа. Конечно, этот вопрос можно обойти, задействовав функцию `SendMessage` или `BroadcastMessage`. И действительно, в Интернете можно найти бесплатные системы обработки событий, такие как `NotificationCenter`, основанные на этих функциях. Однако в этой главе вместо них мы будем использовать интерфейсы и полиморфизм, так как функции `SendMessage` и `BroadcastMessage` основаны на тяжеловесном механизме

рефлексии (подробнее этот механизм рассматривается в главе 8, «Настройка редактора Unity»). В частности, мы создадим интерфейс, из которого будем обращаться ко всем объектам получателей.

Более подробную информацию о бесплатной системе обработки событий `NotificationCenter` (в версии для C#) можно найти по адресу http://wiki.unity3d.com/index.php? title=CSharpNotificationCenter.

В языке C# интерфейс похож на пустой абстрактный базовый класс. Подобно классу, интерфейс объединяет коллекцию методов и функций в единый блок. Но, в отличие от класса, интерфейс позволяет определить только прототипы функций, содержащие имя функции, тип возвращаемого значения и типы аргументов. Он не позволяет определять реализацию функций. Причина в том, что интерфейс определяет лишь общий набор функций, которые будут унаследованы производным классом. Производный класс может реализовать все или некоторые функции интерфейса, а наличие интерфейса позволяет другим объектам вызывать эти функции, основываясь на полиморфизме, не зная конкретного типа производного класса. Это делает интерфейс подходящим кандидатом для создания объектов получателей. С помощью интерфейса `Listener`, от которого будут произведены все объекты, каждый объект получает возможность стать получателем события.

Следующий пример в листинге 4.3 демонстрирует простой интерфейс `Listener`.

Листинг 4.3. Интерфейс Listener

```
using UnityEngine;
using System.Collections;
//-----------------------------------------------------------
// Перечисление определяет все возможные события
// Дополнительные события должны добавляться сюда
public enum EVENT_TYPE {GAME_INIT,
                        GAME_END,
                        AMMO_EMPTY,
                        HEALTH_CHANGE,
                        DEAD};
//-----------------------------------------------------------
// Интефейс Listener для реализации в классах получателей событий
public interface IListener
{
  // Вызывается, когда возникает событие
  void OnEvent(EVENT_TYPE Event_Type,
               Component Sender, Object Param = null);
}
//-----------------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 4.3:

- **Строки 06–10**: это перечисление определяет полный список всех событий, возникающих в игре. В примере перечислены всего пять событий: `GAME_INIT`, `GAME_END`, `AMMO_EMPTY`, `HEALTH_CHANGE` и `DEAD`. В вашей игре их наверняка будет много больше. На самом деле нет особой необходимости использовать перечисления для кодирования событий, можно просто обойтись целыми числами. Но я использовал перечисление для улучшения читабельности событий в коде.
- **Строки 13–18**: объявление интерфейса получателя с именем `IListener`. Он поддерживает только одно событие `OnEvent`. Эта функция будет унаследована всеми производными классами и вызываться диспетчером всякий раз, когда происходит событие, на которое зарегистрирован получатель. Обратите внимание, что `OnEvent` – это прототип функции, он не имеет тела.

Более подробную информацию об интерфейсах C# можно найти по адресу http://msdn.microsoft.com/ru-ru/library/ms173156.aspx.

Теперь, с помощью интерфейса `IListener` мы можем сделать получателем любой объект, используя только наследование класса, то есть любой объект может объявить себя получателем и получать события. Например, вновь созданный компонент `MonoBehaviour` в листинге 4.4 можно превратить в получателя. Этот код, как это уже было в предыдущих главах, использует множественное наследование, то есть наследует два класса. Более подробную информацию о множественном наследовании можно найти по адресу http://www. dotnetfunda.com/articles/show/1185/multiple-inheritance-in-csharp.

Листинг 4.4. Определение получателя событий

```
using UnityEngine;
using System.Collections;

public class MyCustomListener : MonoBehaviour, IListener
{
  // Этот метод выполняет инициализацию
  void Start () {}

  // Вызывается при отображении каждого кадра
  void Update () {}
  //---------------------------------------
  // Реализовать функцию OnEvent для приема событий
```

```
  public void OnEvent(EVENT_TYPE Event_Type, Component Sender,
                      Object Param = null)
  {
  }
  //-----------------------------------------
}
```

Создание класса EventManager

Как мы убедились, любой объект можно превратить в получателя. Но получатели должны зарегистрироваться в объекте диспетчера событий. Обязанностью диспетчера является вызов событий у получателей, когда события действительно происходят. Давайте теперь обратимся к созданию самого диспетчера и деталям его реализации. Класс диспетчера будет называться `EventManager`, как показано в листинге 4.5. Этот класс, будучи неуничтожаемым синглтоном, будет подключаться к пустому игровому объекту в сцене и непосредственно доступен для всех других объектов через статическое свойство. Подробнее об этом классе и его использовании рассказывается в комментариях, следующих за примером.

Листинг 4.5. Диспетчер событий

```
001 using UnityEngine;
002 using System.Collections;
003 using System.Collections.Generic;
004 //-----------------------------------
005 // Синглтон EventManager для отправки событий получателям
006 // Работает с реализациями IListener
007 public class EventManager : MonoBehaviour
008 {
009   #region свойства C#
010   //-----------------------------------
011   // Общий доступ к экземпляру
012   public static EventManager Instance
013   {
014     get{return instance;}
015     set{}
016   }
017   #endregion
018
019   #region переменные
020   // Экземпляр диспетчера событий (синглтон)
021   private static EventManager instance = null;
022
023   // Массив получателей (все зарегистрировавшиеся объекты)
024   private Dictionary<EVENT_TYPE, List<IListener>> Listeners =
        new Dictionary<EVENT_TYPE, List<IListener>>();
```

```
025   #endregion
026   //-------------------------------------------------------------
027   #region методы
028   // Вызывается перед началом работы для инициализации
029   void Awake()
030   {
031     // Если экземпляр отсутствует, сохранить данный экземпляр
032     if(instance == null)
033     {
034       instance = this;
035       DontDestroyOnLoad(gameObject);
036     }
037     else
038       DestroyImmediate(this);
039   }
040   //-------------------------------------------------------------
041   /// <summary>
042   /// Функция добавления получателя в массив
043   /// </summary>
044   /// <param name="Event_Type">Событие, ожидаемое получателем</param>
045   /// <param name="Listener">Объект, ожидающий события</param>
046   public void AddListener(EVENT_TYPE Event_Type, IListener Listener)
047   {
048     // Список получателей для данного события
049     List<IListener> ListenList = null;
050
051     // Проверить тип события. Если существует - добавить в список
052     if(Listeners.TryGetValue(Event_Type, out ListenList))
053     {
054       // Список существует, добавить новый элемент
055       ListenList.Add(Listener);
056       return;
057     }
058
059     // Иначе создать список как ключ словаря
060     ListenList = new List<IListener>();
061     ListenList.Add(Listener);
062     Listeners.Add(Event_Type, ListenList);
063   }
064   //-------------------------------------------------------------
065   /// <summary>
066   /// Посылает события получателям
067   /// </summary>
068   /// <param name="Event_Type">Событие для вызова</param>
069   /// <param name="Sender">Вызываемый объект</param>
070   /// <param name="Param">Необязательный аргумент</param>
071   public void PostNotification(EVENT_TYPE Event_Type,
        Component Sender, Object Param = null)
072   {
073     // Послать событие всем получателям
```

```
074
075     // Список получателей только для данного события
076     List<IListener> ListenList = null;
077
078     // Если получателей нет - выйти
079     if(!Listeners.TryGetValue(Event_Type, out ListenList))
080       return;
081
082     // Получатели есть. Послать им событие
083     for(int i=0; i<ListenList.Count; i++)
084     {
085       if(!ListenList[i].Equals(null))
086         ListenList[i].OnEvent(Event_Type, Sender, Param);
087     }
088   }
089   //-------------------------------------------------------------
090   // Удаляет событие из словаря, включая всех получателей
091   public void RemoveEvent(EVENT_TYPE Event_Type)
092   {
093     // Удалить запись из словаря
094     Listeners.Remove(Event_Type);
095   }
096   //-------------------------------------------------------------
097   // Удаляет все избыточные записи из словаря
098   public void RemoveRedundancies()
099   {
100     // Создать новый словарь
101     Dictionary<EVENT_TYPE, List<IListener>> TmpListeners =
        new Dictionary<EVENT_TYPE, List<IListener>>();
102
103     // Обойти все записи в словаре
104     foreach(KeyValuePair<EVENT_TYPE, List<IListener>> Item in Listeners)
105     {
106       // Обойти всех получателей, удалить пустые ссылки
107       for(int i = Item.Value.Count-1; i>=0; i--)
108       {
109         // Если ссылка пустая, удалить элемент
110         if(Item.Value[i].Equals(null))
111           Item.Value.RemoveAt(i);
112       }
113
114       // Если в списке остались элементы, добавить его в словарь tmp
115       if(Item.Value.Count > 0)
116         TmpListeners.Add (Item.Key, Item.Value);
117     }
118
119     // Заменить объект Listeners новым словарем
120     Listeners = TmpListeners;
121   }
122   //-------------------------------------------------------------
```

```
123   // Вызывается при смене сцены. Очищает словарь
124   void OnLevelWasLoaded()
125   {
126     RemoveRedundancies();
127   }
128   //-------------------------------------------------------------
129   #endregion
130 }
```

Более подробную информацию о событии `OnLevelWasLoaded` можно найти по адресу http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnLevelWasLoaded.html.

Ниже приводится несколько комментариев к листингу 4.5:

- **Строка 003**: обратите внимание на добавление пространства имен `System.Collections.Generic`, которое дает доступ к дополнительным классам фреймворка Mono, в том числе к классу `Dictionary`. Этот класс используется в нескольких местах в классе `EventManager`. Более подробно фреймворк Mono и его классы будут рассмотрены ниже, в главе 6 «Работа с фреймворком Mono». Если коротко, то класс `Dictionary` является двумерным массивом особого вида, позволяющим хранить базу данных в виде набора пар ключ–значение. Более подробную информацию о классе `Dictionary` можно найти по адресу http://msdn.microsoft.com/ru-ru/library/xfhwa508%28v=vs.110%29.aspx.
- **Строка 007**: класс `EventManager` является производным от класса `MonoBehaviour` и должен прикрепляться к пустому игровому объекту в сцене, где он будет существовать как неуничтожаемый синглтон.
- **Строка 024**: закрытая переменная `Listeners` объявлена с типом данных `Dictionary`. Она представляет собой структуру для хранения хэш-таблицы с парами ключ–значение, в которой можно осуществлять поиск как в базе данных. Пара ключ–значение для класса `EventManager` принимает форму `EVENT_TYPE` и `List<Component>`. Если коротко, это значит, что будет сохранен список типов событий (таких как `HEALTH_CHANGE`) и для каждого типа можно задать ноль, один или несколько компонентов, которые будут получателями и должны уведомляться о событиях. В сущности, свойство `Listeners` является главной структурой данных, на которой основана поддержка получателей классом `EventManager`. Более подробную информацию о фреймворке Mono и его классах можно найти в главе 6, «Работа с фреймворком Mono».

- **Строки 029–039**: функция `Awake` отвечает за превращение класса `EventManager` в синглтон, то есть, она гарантирует, что в сцене будет существовать единственный экземпляр класса `EventManager` и сохраняться при смене сцен. За более подробной информацией о неуничтожаемых синглтонах обращайтесь к главе 3 «Синглтоны, статические переменные, игровые объекты и игровой мир».
- **Строки 046–063**: метод `AddListener` класса `EventManager` следует вызвать из объекта-получателя один раз для каждого события, которое он должен получать. Метод принимает два аргумента: тип события (`Event_Type`) и ссылку на объект-получатель (производный от `IListener`), желающий получать уведомления о событиях этого типа. Функция `AddListener` обращается к словарю получателей `Listeners` и генерирует новую пару ключ–значение для сохранения связи между событием и получателем.
- **Строки 071–088**: функцию `PostNotification` можно вызвать из любого объекта, независимо от того, является он получателем или нет, в момент обнаружения события. При вызове этой функции объект `EventManager` перебирает все записи в словаре, находит всех получателей событий данного типа и уведомляет их, вызывая методы `OnEvent` получателей через интерфейс `IListener`.
- **Строки 098–127**: заключительные методы класса `EventManager` несут ответственность за поддержание целостности данных в структуре получателей при смене сцены и сохранении объекта `EventManager`. Хотя экземпляр `EventManager` сохраняется при смене сцен, объекты-получатели, ссылки на которые хранятся в переменной `Listeners`, могут быть уничтожены. Если это произойдет, после смены сцены некоторые получатели перестанут существовать, оставив в объекте `EventManager` недействующие записи. Метод `RemoveRedundancies` должен найти и удалить все такие записи. Событие `OnLevelWasLoaded` вызывается автоматически, когда происходит смена сцены.

Словари. Большое преимущество словарей не только в скорости доступа к ним, но и в возможности работать с ними через типы объектов и индексы массива. В обычном массиве каждый элемент доступен по целочисленному индексу, например `MyArray[0]` или `MyArray[1]`. Но со словарями дело обстоит иначе. В частности, доступ к элементам словаря можно получить, используя объекты с типом данных `EVENT_TYPE`, который представляет собой ключ для пары ключ–значение, например `MyArray[EVENT_TYPE.HEALTH_CHANGE]`. За более подробной информацией о словарях обращайтесь к офи-

циальной документации Microsoft по адресу http://msdn.microsoft.com/ru-ru/library/xfhwa508%28v=vs.110%29.aspx.

Директивы #region и #endregion для свертывания кода в MonoDevelop

Редактор MonoDevelop поддерживает две директивы препроцессора – `#region` и `#endregion` – которые (в сочетании с функцией свертывания кода) весьма полезны для улучшения читаемости кода и увеличения скорости перемещения по исходному коду. Они добавляют в исходный код организацию и структурность, не затрагивая его сущности и не влияя на его выполнение. Директива `#region` отмечает начало блока кода, директива `#endregion` – его окончание. После того как область отмечена, она становится сворачиваемой, точнее, она становится сворачиваемой в редакторе кода MonoDevelop, при условии, что функция свертывания кода включена. Сворачивание фрагментов может пригодиться для их сокрытия, чтобы сосредоточиться на чтении других, нужных вам сейчас областей, как показано на рис. 4.1.

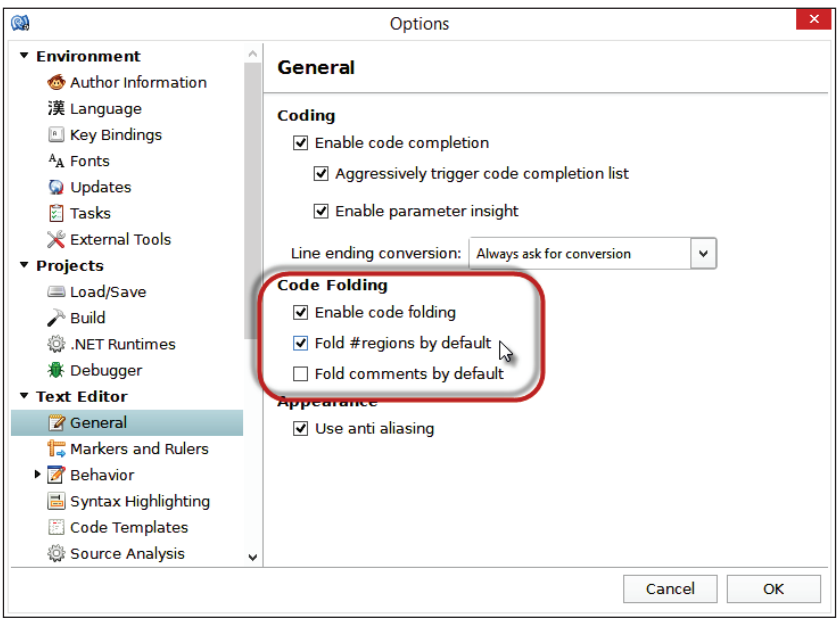


Рис. 4.1. Включение свертывания кода в редакторе MonoDevelop

Чтобы включить функцию свертывания кода в MonoDevelop, выберите пункт **Tools** ⇒ **Options** (Инструменты ⇒ Параметры) в меню приложения. После этого появится диалог **Options** (Параметры). В нем выберите вкладку **General** (Общие) в разделе **Text Editor** (Редактор текста) и установите флажки **Enable code folding** (Разрешить свертку кода) и **Fold #regions by default** (Сворачивать области #region по умолчанию).

Использование EventManager

Теперь давайте посмотрим, как применить класс `EventManager` на практике, в сцене с получателями и отправителями сообщений. Во-первых, для получения уведомлений о событии (любом событии) получатель должен зарегистрироваться в экземпляре синглтона `EventManager`. Обычно это делается при первой же возможности, например, в функции `Start`. Не используйте для этого функцию `Awake`, потому что она зарезервирована для внутренней инициализации объектов, а не для операций, которые выходят за пределы текущего объекта, для смены состояний и настройки других объектов. Взгляните на следующий пример в листинге 4.6 и обратите внимание, что в нем используется статическое свойство `Instance` для получения ссылки на активный синглтон `EventManager`.

Листинг 4.6. Регистрация получателя событий

```
// Вызывается в момент запуска
void Start()
{
  // Добавить себя как получателя события изменения уровня здоровья
  EventManager.Instance.AddListener(EVENT_TYPE.HEALTH_CHANGE, this);
}
```

После регистрации получателей одного или более событий объекты могут затем уведомить синглтон `EventManager` об обнаруженном событии, как показано в листинге 4.7.

Листинг 4.7. Уведомление о событии

```
public int Health
{
  get{return _health;}
  set
  {
    // Привести уровень здоровья в диапазон 0-100
    _health = Mathf.Clamp(value, 0, 100);

    // Послать уведомление об изменении уровня здоровья
    EventManager.Instance.PostNotification(EVENT_TYPE.EALTH_CHANGE,
                                           this, _health);
  }
}
```

Наконец, после отправки уведомления о событии, все подписавшиеся на него получатели будут автоматически уведомлены синглтоном EventManager. В частности, объект EventManager вызовет функцию OnEvent каждого получателя, давая возможность обработать событие и среагировать в случае необходимости, как показано в листинге 4.8.

Листинг 4.8. Обработка события получателем

```
// Вызывается, когда происходит событие
public void OnEvent(EVENT_TYPE Event_Type, Component Sender,
                    object Param = null)
{
  // Определить тип события
  switch(Event_Type)
  {
    case EVENT_TYPE.HEALTH_CHANGE:
      OnHealthChange(Sender, (int)Param);
      break;
  }
}
```

Пример использования EventManager можно найти в папке events, в пакете примеров для этой главы.

Альтернативный способ, основанный на делегировании

Интерфейсы являются эффективным и простым способом реализации систем обработки событий, но это не единственный способ. Также можно использовать механизм C#, называемый делегированием. Суть его в том, чтобы создать функцию и сохранить ссылку на нее в переменной. Эта переменная позволит обрабатывать функции как переменные ссылочного типа. То есть делегирование позволяет хранить ссылки на функции и использовать их для вызова этих функций. Другие языки, такие как C++, предлагают реализацию подобного подхода с помощью указателей на функции. При реализации системы событий с использованием делегирования нам не понадобятся интерфейсы. Рассмотрим следующий пример в листинге 4.9, который является альтернативной реализацией класса EventManager, использующей делегирование. Соответствующие изменения в кода выделены жирным, чтобы показать различия между реализациями на основе интерфейсов и делегирования. Помимо незначительных изменений, необходимых для работы с делегатами, все остальные функции остались неизменными.

Листинг. 4.9. Реализация поддержки событий на основе делегатов

```
001 using UnityEngine;
002 using System.Collections;
003 using System.Collections.Generic;
004 //-------------------------------------------------------------
005 // Перечисление определяет все возможные события
006 // Дополнительные события должны добавляться сюда
007 public enum EVENT_TYPE {GAME_INIT,
008                         GAME_END,
009                         AMMO_CHANGE,
010                         HEALTH_CHANGE,
011                         DEAD};
012 //-------------------------------------------------------------
013 // Синглтон EventManager для отправки событий получателям
014 // Реализация основана на делегатах
015 public class EventManager : MonoBehaviour
016 {
017   #region свойства C#
018   //-------------------------------------------------------------
019   // Общий доступ к экземпляру
020   public static EventManager Instance
021   {
022     get{return instance;}
023     set{}
024   }
025   #endregion
026
027   #region переменные
028   // Экземпляр диспетчера событий (синглтон)
029   private static EventManager instance = null;
030
031   // Тип делегата, обрабатывающего события
032   public delegate void OnEvent(EVENT_TYPE Event_Type,
        Component Sender, object Param = null);
033
034   // Массив получателей
035   private Dictionary<EVENT_TYPE, List<OnEvent>>
        Listeners = new Dictionary<EVENT_TYPE, List<OnEvent>>();
036   #endregion
037   //-------------------------------------------------------------
038   #region методы
039   // Вызывается перед началом работы для инициализации
040   void Awake()
041   {
042     // Если экземпляр отсутствует, сохранить данный экземпляр
043     if(instance == null)
044     {
045       instance = this;
046       DontDestroyOnLoad(gameObject);
```

```
047     }
048     else
049       DestroyImmediate(this);
050   }
051   //-------------------------------------------------------------
052   /// <summary>
053   /// Функция добавления получателя в массив
054   /// </summary>
055   /// <param name="Event_Type">Событие, ожидаемое получателем</param>
056   /// <param name="Listener">Объект, ожидающий события</param>
057   public void AddListener(EVENT_TYPE Event_Type, OnEvent Listener)
058   {
059     // Список получателей для данного события
060     List<OnEvent> ListenList = null;
061
062     // Проверить тип события. Если существует - добавить в список
063     if(Listeners.TryGetValue(Event_Type, out ListenList))
064     {
065       // Список существует, добавить новый элемент
066       ListenList.Add(Listener);
067       return;
068     }
069
070     // Иначе создать список как ключ словаря
071     ListenList = new List<OnEvent>();
072     ListenList.Add(Listener);
073     Listeners.Add(Event_Type, ListenList);
074   }
075   //-------------------------------------------------------------
076   /// <summary>
077   /// Посылает события получателям
078   /// </summary>
079   /// <param name="Event_Type">Событие для вызова</param>
080   /// <param name="Sender">Вызываемый объект</param>
081   /// <param name="Param">Необязательный аргумент</param>
082   public void PostNotification(EVENT_TYPE Event_Type,
        Component Sender, object Param = null)
083   {
084     // Послать событие всем получателям
085
086     // Список получателей только для данного события
087     List<OnEvent> ListenList = null;
088
089     // Если получателей нет - выйти
090     if(!Listeners.TryGetValue(Event_Type, out ListenList))
091       return;
092
093     // Получатели есть. Послать им событие
094     for(int i=0; i<ListenList.Count; i++)
095     {
```

```
096       if(!ListenList[i].Equals(null))
097         ListenList[i](Event_Type, Sender, Param);
098     }
099   }
100   //-------------------------------------------------------------
101   // Удаляет событие из словаря, включая всех получателей
102   public void RemoveEvent(EVENT_TYPE Event_Type)
103   {
104     // Удалить запись из словаря
105     Listeners.Remove(Event_Type);
106   }
107   //-------------------------------------------------------------
108   // Удаляет все избыточные записи из словаря
109   public void RemoveRedundancies()
110   {
111     // Создать новый словарь
112     Dictionary<EVENT_TYPE, List<OnEvent>>
          TmpListeners = new Dictionary<EVENT_TYPE, List<OnEvent>>();
113
114     // Обойти все записи в словаре
115     foreach(KeyValuePair<EVENT_TYPE, List<OnEvent>> Item in Listeners)
116     {
117       // Обойти всех получателей, удалить пустые ссылки
118       for(int i = Item.Value.Count-1; i>=0; i--)
119       {
120         // Если ссылка пустая, удалить элемент
121         if(Item.Value[i].Equals(null))
122           Item.Value.RemoveAt(i);
123       }
124
125       // Если в списке остались элементы, добавить его в словарь tmp
126       if(Item.Value.Count > 0)
127         TmpListeners.Add (Item.Key, Item.Value);
128     }
129
130     // Заменить объект Listeners новым словарем
131     Listeners = TmpListeners;
132   }
133   //-------------------------------------------------------------
134   // Вызывается при смене сцены. Очищает словарь
135   void OnLevelWasLoaded()
136   {
137     RemoveRedundancies();
138   }
139   //-------------------------------------------------------------
140   #endregion
141 }
```

Более подробную информацию о делегировании в C# можно найти в документации Microsoft по адресу http://msdn.microsoft.com/ru-ru/library/aa288459%28v=vs.71%29.aspx.

Ниже приводится несколько комментариев к листингу 4.9:

- **Строки 005–011**: перечисление типов событий было перенесено из файла с интерфейсом `IListener` в файл с реализацией класса `EventManager`. Поскольку прием делегирования устраняет необходимость в интерфейсах, в частности необходимость в интерфейсе `IListener`, перечисление стало возможным поместить непосредственно в исходный файл с реализацией диспетчера.
- **Строка 032**: общедоступная функция `OnEvent` объявлена с типом делегата. Обратите внимание, что это объявление является гибридным, поскольку сочетает в себе объявление в стиле переменной с прототипом функции. Оно определяет прототип функции, которая может быть присвоена переменной. Переменной может быть назначена любая функция с такой же структурой, из любого класса или любого другого файла сценария. То есть, функция `OnEvent` становится переменной с типом делегата и далее будет использована для создания внутреннего словаря.
- **Строка 035**: объявление локальной переменной-словаря `listeners`, в котором для каждого типа событий будет храниться массив делегатов (вместо интерфейсов). Каждый делегат является ссылкой на функцию, которая должна быть вызвана, когда произойдет событие.
- **Строка 097**: функция `PostNotification` класса `EventManager` предназначена для вызова всех делегатов (функций получателей), когда происходит событие. Это происходит в строке 097 с оператором `ListenList[i](Event_ Type, Sender, Param);`. Этот оператор вызывает делегата как функцию, как показано на рис. 4.2.

Реализацию класса `EventManager`, использующую делегатов, можно найти в папке `events_delgateversion`, в пакете примеров для этой главы.

События класса MonoBehaviour

Перед завершением этой главы рассмотрим некоторые предопределенные события, предлагаемые Unity. Класс `MonoBehaviour` уже содержит широкий спектр событий, которые вызываются автоматически при определенных условиях. Эти функции, или события, начинаются с префикса `On` и включают такие события, как `OnGUI`, `OnMouseEnter`, `OnMouseDown`, `OnParticleCollision` и др. В этом разделе рассматриваются некоторые детали этих общих типов событий.

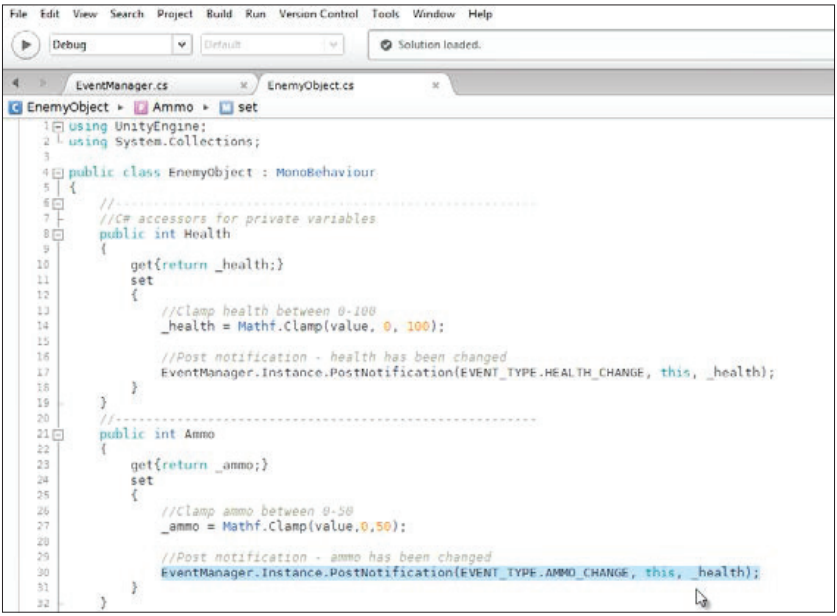


Рис. 4.2. Исследование проекта EventManager

Полный список событий, поддерживаемых классом `MonoBehaviour`, можно найти в документации Unity по адресу http://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

События мыши и сенсорной панели

Одним из интересных наборов событий является множество событий от мыши и сенсорного панели. Оно включает в себя события `OnMouseDown`, `OnMouseEnter` и `OnMouseExit`. В ранних версиях Unity эти события были связаны только с мышью. Но недавно они были связаны и сенсорной панелью. Это значит, что сенсорная панель будет теперь по умолчанию регистрироваться так же, как мышь. Событие `OnMouseDown` вызывается один раз при нажатии кнопки мыши, когда ее курсор находится над объектом. Событие, однако, не вызывается, пока кнопка не будет отпущена. Событие `OnMouseEnter` также вызывается один раз, когда курсор входит в область объекта, а событие `OnMouseExit` вызывается, когда курсор покидает границы объекта. Вызов этих событий определяется коллайдером объекта, события вызываются при нахождении курсора мыши в пределах его объема. Это значит, что ни одно из событий мыши не срабатывает для объектов, не имеющих коллайдера.

Однако иногда событии мыши не срабатывают и для объектов с коллайдерами, потому что другие объекты (с коллайдерами) заслоняют их при обзоре с активной камеры. То есть интерактивные объекты находятся на заднем плане. Можно, конечно, решить вопрос (по крайней мере, во многих случаях), просто установив для объектов переднего плана слой `IgnoreRaycast`, что сделает их нечувствительными к операциям отслеживания лучей.

Чтобы назначить слой `IgnoreRaycast`, просто выберите объект в сцене, а затем щелкните на раскрывающемся списке **Layers** (Слои) в инспекторе объектов и выберите пункт **Ignore Raycast** (Не участвовать в отслеживании лучей), как показано на рис. 4.3.

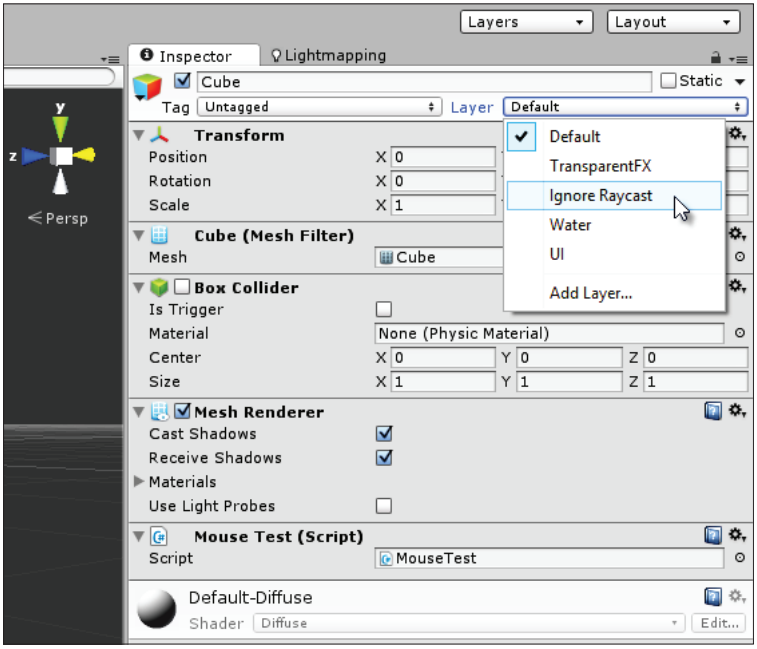


Рис. 4.3. Назначение объекту слоя Ignore Raycast

Но такое простое решение помогает не всегда. Часто сцена включает множество камер и объектов с коллайдерами, и иногда бывает неясно, какие именно объекты должны обрабатывать события от мыши. В этих случаях может потребоваться вручную обрабатывать события от мыши, как показано в листинге 4.10, ниже, где события вызываются

вручную на основе входных данных. В сущности, этот код использует систему отслеживания лучей Raycast для перенаправления вручную обнаруженных событий мыши. Также в этом примере используются **сопрограммы**, которые будут рассмотрены ниже.

Листинг 4.10. Обработка событий от мыши вручную

```
using UnityEngine;
using System.Collections;
//---------------------
public class ManualMouse : MonoBehaviour
{
  //---------------------
  // Коллайдер, подключенный к данному объекту
  private Collider Col = null;

  //---------------------
  // Вызывается перед началом работы для инициализации
  void Awake()
  {
    // Сохранить ссылку на коллайдер
    Col = GetComponent<Collider>();
  }

  //---------------------
  // Сопрограмма Start
  void Start()
  {
    StartCoroutine(UpdateMouse());
  }

  //---------------------
  public IEnumerator UpdateMouse()
  {
    // Признак пересечения
    bool bIntersected = false;

    // Кнопка нажата или отпущена
    bool bButtonDown = false;

    // Бесконечный цикл
    while(true)
    {
      // Получить экранные координаты X и Y указателя мыши
      // Может потребоваться использовать другую камеру
      Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
      RaycastHit hit;

      // Проверить столкновение луча с коллайдером
```

```
      if (Col.Raycast(ray, out hit, Mathf.Infinity))
      {
        // Луч пересек объект
        if(!bIntersected)
          SendMessage("OnMouseEnter",
            SendMessageOptions.DontRequireReceiver);

        bIntersected = true;

        // Проверить события от мыши
        if(!bButtonDown && Input.GetMouseButton(0))
        {
          bButtonDown = true; SendMessage("OnMouseDown",
            SendMessageOptions.DontRequireReceiver);
        }
        if(bButtonDown && !Input.GetMouseButton(0))
        {
          bButtonDown = false; SendMessage("OnMouseUp",
            SendMessageOptions. DontRequireReceiver);
        }
      }
      else
      {
        // Прежде указатель входил в границы объекта и теперь вышел
        if(bIntersected)
          SendMessage("OnMouseExit",
            SendMessageOptions.DontRequireReceiver);

        bIntersected = false;
        bButtonDown = false;
      }
      // Ждать следующего кадра
      yield return null;
    }
  }
  //---------------------
}
//---------------------
```

Сопрограммы. Сопрограммы это особый вид функций. Они ведут себя подобно потокам выполнения, поскольку кажутся работающими параллельно или асинхронно с основным циклом игры, то есть в фоновом режиме. Выполнение кода не приостанавливается и не ждет завершения сопрограмм, как это происходит с обычными функциями. По этой причине сопрограммы отлично подходят для создания асинхронных действий. Технически все сопрограммы должны возвращать значение типа `IEnumerator`, содержать хотя бы один оператор `yield` и запускаться с помощью функции `StartCoroutine`. Оператор `yield` – это особый оператор, он приостанавливает выполнение сопрограммы, пока его состояние не изменится. Оператор `yield`, возвращающий результат вызова `WaitForSeconds(x)`, приостановит выполнение на

x секунд, а затем продолжит его со следующей строки. Оператор `yield`, возвращающий `null`, приостановит выполнение в текущем кадре и возобновит выполнение со следующей строки в следующем кадре. Более подробную информацию о сопрограммах и их использовании можно найти в документации Unity по адресу http://docs.unity3d. com/Manual/Coroutines.html.

Фокус приложения и пауза

Три дополнительных события `MonoBehaviour` отличаются тем, что часто вызывают замешательство и удивление. Это события: `OnApplicationPause`, `OnApplicationFocus` и `OnApplicationQuit`.

Событие `OnApplicationQuit` посылается всем объектам сцены перед завершением игры, но до уничтожения сцены и ее содержимого. Если игра тестируется в редакторе, событие `OnApplicationQuit` вызывается при остановке воспроизведения. Примечательно, что событие `OnApplicationQuit` не вызывается на устройствах, работающих под управлением iOS, где, как правило, приложения не закрываются, а приостанавливаются, ожидая, пока пользователи занимаются другими делами, что позволяет им вернуться и возобновить работу с того места, где они остановились. Если вам нужно получать уведомление о событии `OnApplicationQuit` при приостановке, вы должны установить соответствующий флаг в окне настроек проигрывателя **Player Settings** (Настройки проигрывателя). Чтобы получить доступ к ним, выберите пункт **Edit** ⇒ **Project Settings** ⇒ **Player** (Правка ⇒ Настройки проекта ⇒ Проигрыватель) в меню приложения, а затем в инспекторе объектов, раскройте вкладку **Other Settings** (Прочие настройки) для iOS, установите флажок **Exit on Suspend** (Завершать при приостановке), как показано на рис. 4.4.

Событие `OnApplicationFocus` передается всем объектам в сцене, когда окно игры теряет фокус, например, когда оно деактивируется при переключении на другую программу. Это событие может быть значительным событием для игр, особенно для многопользовательских, где действия и события в общем мире продолжаются, даже если один или несколько игроков не принимают в игре участия. В этих случаях, возможно, потребуется приостановить или возобновить определенные действия либо приглушить или усилить звуковое сопровождение игры.

Событие `OnApplicationPause` является неоднозначным событием, потому что понятие паузы в Unity четко не определено. С моей точки зрения существует два вида пауз, а именно абсолютная и относительная паузы. При абсолютной паузе все действия и события в игре полностью приостанавливаются. В этом состоянии нет течения времени,

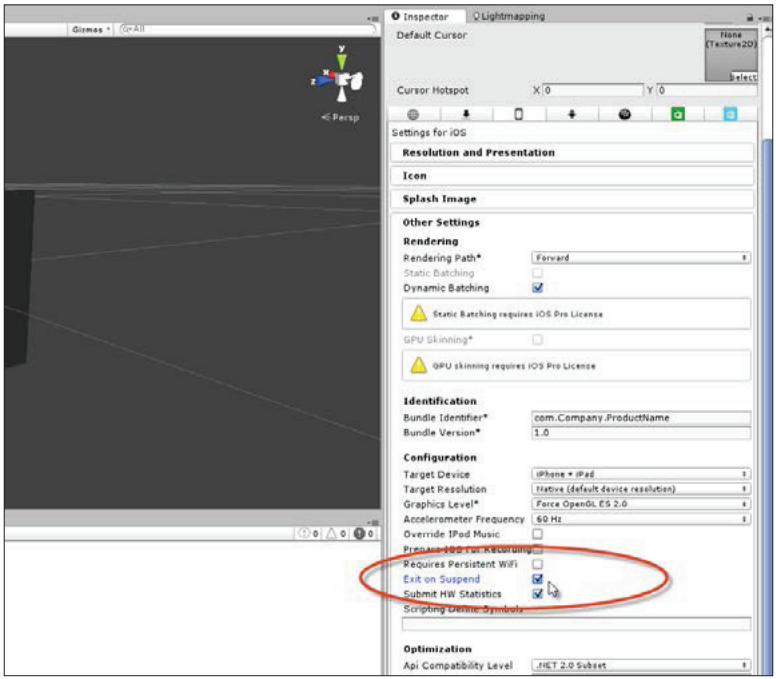


Рис. 4.4. Включение флага **Exit on Suspend** (Завершать при приостановке) для iOS

и ничто не может двигаться. Относительная пауза применяется чаще. Здесь игра осознает себя и понимает, что находится в состоянии паузы. Она останавливает некоторые события, но позволяет другим событиям выполняться, чтобы продолжить взаимодействие с вводом пользователя, который имеет возможность возобновить игру. Событие `OnApplicationPause` имеет отношение к первому виду пауз. Это событие будет вызываться при выполнении нескольких условий. Они будут рассмотрены ниже.

Во-первых, событие `OnApplicationPause` вызывается, только если сброшен флаг **Run In Background** (Запускать в фоне) во вкладке **Player Settings** (Настройки проигрывателя), в группе **Resolution** (Разрешение), как показано на рис. 4.5. Выключение этого параметра приведет к автоматической приостановке игры, когда ее окно потеряет фокус. Это значит, что событие `OnApplicationPause` будет вызвано после события `OnApplicationFocus`.

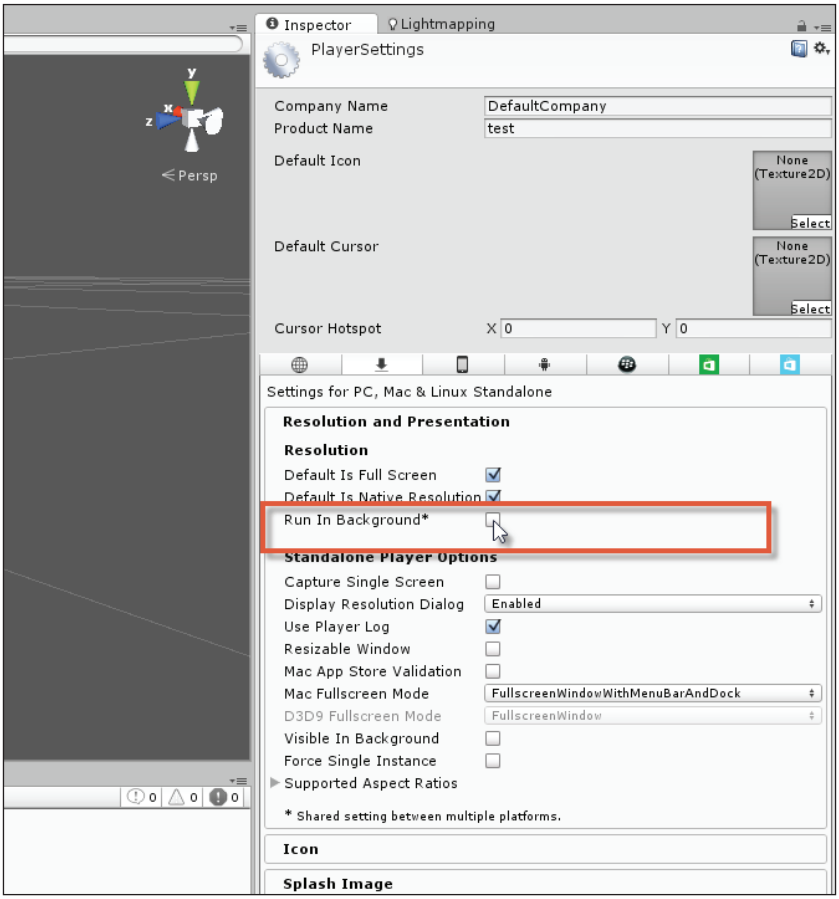


Рис. 4.5. Выключение параметра **Run In Background** (Запускать в фоне)

В iOS событие `OnApplicationPause` будет вызываться всякий раз, когда приложение сворачивается или переносится на задний план.

Не полагайтесь на событие `OnApplicationPause` при реализации собственной реакции на относительную паузу. Используйте для этого переменную `Time.timeScale` или создайте более полноценную систему, которая сможет сама выбирать, какие элементы приостанавливать.

Итоги

Эта глава была посвящена преимуществам, которые дает система управления событиями, основанная на классе `EventManager`. Для реализации диспетчера событий можно использовать интерфейсы или делегирование – оба метода являются мощными и расширяемыми. Мы увидели, как легко до бесконечности добавлять и добавлять новые операции в функцию `Update`, и как это приводит к серьезным проблемам производительности. Мы проанализировали связи между действиями и переработали код в стиле событийного программирования. По сути, события являются сырьем для систем управления событиями. Они обеспечивают связь между одним действием (причина), а другим (реакция). Для управления событиями, мы создали класс `EventManager` – интегрированный класс или систему, связывающую отправителей с получателями. Он принимает уведомления о событиях от отправителей, а затем сразу же вызывает соответствующие функции получателей. В следующей главе мы рассмотрим камеры и отображение сцены.

Глава 5

Камеры и отображение сцены

Эта глава посвящена некоторым из множества действий с камерами и отображением сцен, а также интересным комбинациям их совместного использования. Вообще говоря, камера – это точка обзора отображаемой сцены. Это точка в трехмерном пространстве, из которой открывается вид на сцену. Камера определяет перспективу и поле зрения, захватывает и разбивает текстуры до пикселей. После этого они визуализируются на экране, смешиваясь с изображениями, полученными с любых других камер. Таким образом, камеры, сцены и их отображение тесно связаны. В этой главе мы увидим, как оживить камеру и создать эффект ее полета, как перемещать камеру вдоль криволинейных траекторий, и рассмотрим, как объекты могут узнать, видны ли они для определенной камеры. Кроме того, мы увидим, как вручную обрабатывать изображения с камер для создания дополнительных эффектов, а также как настроить ортографические камеры для отображения пикселей двухмерной текстуры в двухмерной игре и графическом пользовательском интерфейсе. Итак, начнем.

Визуальное представление камеры

Когда камера выбрана во вкладке **Scene** (Сцена) и разрешено отображение визуального представления камеры, ребра усеченной пирамиды ясно показывают, где находится камера и что она может видеть из этой точки обзора, с учетом других ее свойств, таких как угол обзора, как показано на рис. 5.1.

Визуальные представления камер особенно полезны при их размещении для получения наилучшего обзора сцены. Однако иногда может понадобится обратное, то есть разместить объекты в поле зрения не выбранной в настоящий момент камеры. Например, может понадобится поместить отдельные объекты в поле зрения камеры

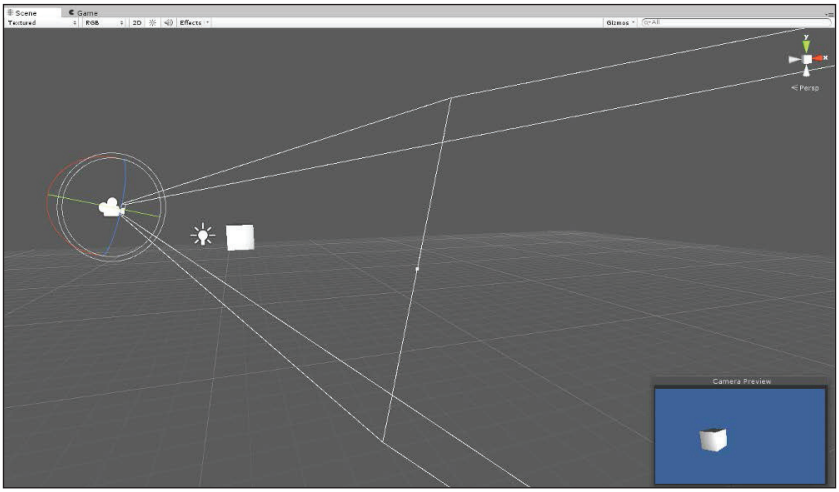


Рис. 5.1. При выборе камеры во вкладке **Scene** (Сцена), она отображается в виде значка и усеченной пирамиды

и убедиться, что они видны для нее. Сделать это сложно, потому что по умолчанию визуальные представления для камер не отображаются, если они не выбраны. Это значит, что при перемещении объектов придется постоянно выбирать камеру, чтобы проверить, попадают ли объекты в поле зрения камеры после перемещения, и откорректировать их позиции, если это потребуется. Решению этой задачи очень помогла бы возможность просматривать визуальные представления камер постоянно, даже если они не выбраны, но, такая возможность отсутствует, по крайней мере, на момент написания этой книги. Чтобы обойти этот недостаток, можно написать сценарий, представленный в листинге 5.1.

Листинг 5.1. Сценарий отображения визуального представления камер

```
01 using UnityEngine;
02 using System.Collections;
03 //-------------------------------------------------------
04 [ExecuteInEditMode]
05 [RequireComponent(typeof(Camera))]
06 //-------------------------------------------------------
07 public class DrawFrustumRefined : MonoBehaviour
08 {
09   //-------------------------------------------------------
10   private Camera Cam = null;
```

```
11   public bool ShowCamGizmo = true;
12   //-------------------------------------------------------
13   void Awake()
14   {
15     Cam = GetComponent<Camera>();
16   }
17   //-------------------------------------------------------
18   void OnDrawGizmos()
19   {
20     // Отображать визуальное представление?
21     if(!ShowCamGizmo) return;
22     // Получить размеры окна Game (Игра)
23     Vector2 v = DrawFrustumRefined.GetGameViewSize();
24     float GameAspect = v.x/v.y; // Вычислить отношение сторон окна
25     float FinalAspect = GameAspect / Cam.aspect;
26
27     Matrix4x4 LocalToWorld = transform.localToWorldMatrix;
28     Matrix4x4 ScaleMatrix = Matrix4x4.Scale(new Vector3(Cam.aspect *
       (Cam.rect.width / Cam.rect.height), FinalAspect,1));
29     Gizmos.matrix = LocalToWorld * ScaleMatrix;
30     Gizmos.DrawFrustum(transform.position, Cam.fieldOfView,
       Cam.nearClipPlane, Cam.farClipPlane, FinalAspect);
31     Gizmos.matrix = Matrix4x4.identity; // Сбросить матрицу
32   }
33   //-------------------------------------------------------
34   // Возвращает размеры окна игры
35   public static Vector2 GetGameViewSize()
36   {
37     System.Type T =
         System.Type.GetType("UnityEditor.GameView,UnityEditor");
38     System.Reflection.MethodInfo GetSizeOfMainGameView =
         T.GetMethod("GetSizeOfMainGameView",System.Reflection.
         BindingFlags.NonPublic | System.Reflection.BindingFlags. Static);
39     return (Vector2)GetSizeOfMainGameView.Invoke(null,null);
40   }
41   //-------------------------------------------------------
42 }
43 //---------------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 5.1.

- **Строки 27–31**: функция `Gizmos.DrawFrustum` принимает аргументы, такие как позиция и угол поворота, выраженные в глобальных (мировых) координатах. Это значит, что все аргументы позиционирования должны быть предварительно преобразованы с помощью матрицы преобразования локальных координат в глобальные координаты. Это преобразование выполняет метод `localToWorldMatrix` класса `Transform`. Кроме того, аргумент `FinalAspect` требует расчета соотношения между шириной и высотой окна просмотра и шириной и высотой окна игры.

- **Строки 35–40**: функция `GetGameViewSize` возвращает двухмерный вектор, выражающий фактические размеры окна Game (Игра) в пикселях. Она получает эти значения, используя недокументированные возможности редактора. Следует подчеркнуть, что код, использующий «недокументированные» возможности, может перестать работать в будущих версиях редактора.

На рис. 5.2 изображена полученная пирамида, представляющая поле зрения камеры.

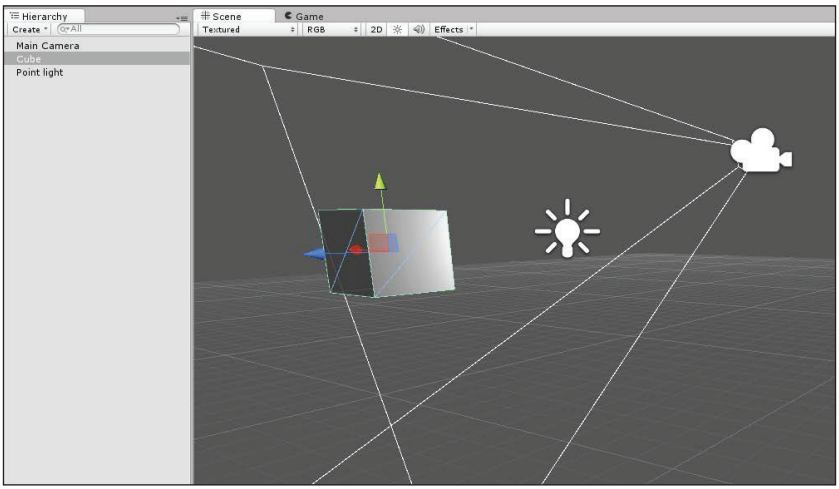


Рис. 5.2. Визуальное представление невыбранной камеры

Быть на виду

Есть много случаев, когда во время игры возникают вопросы о видимости объекта, как фактической, так и гипотетической. Относительно фактической видимости есть несколько вопросов, ответы на которые мы хотели бы получить, в том числе видим ли объект X для камеры Y прямо сейчас, видим ли объект X для любой камеры прямо сейчас или когда объект X становится видимым или невидимым для конкретной камеры или для любой камеры. Что касается гипотетической видимости, вопрос может быть сформулирован так: будет ли объект X виден, если камеру Y переместить в положение Z. В вопросах о фактической видимости нас заботит действительная видимость объектов в теку-

щем кадре, зависящая от текущей позиции камеры, а в отношении гипотетической видимости нас беспокоит, что было бы, если бы камера была бы перемещена в определенное положение. И оба эти варианта видимости имеют важное значение для игры. Зная, что объекты (такие как враги) действительно видны для камеры, важно определить их поведение и реакцию. Это объясняется тем, что когда объекты не видны, многие действия и расчеты можно приостановить, уменьшив нагрузку за счет отказа от их обработки. Кроме того, информация о видимости объекта, если камера будет перемещена, позволит предвидеть, какие объекты, если они имеются, станут видимыми со следующего кадра, чтобы подготовить их заранее. Теперь, прежде чем перейти к рассмотрению ответов на эти вопросы с помощью сценариев, рассмотрим видимость в узком смысле этого понятия.

Видимость базируется на двух основных понятиях: поле зрения и препятствия. Каждая камера, работающая в режиме перспективы, имеет поле зрения в форме усеченной пирамиды – трапециевидный объем, простирающийся от камеры и содержащий область, определяемую углом поля зрения и отсекающими плоскостями (ближней и дальней). Усеченная пирамида математически определяет область сцены, которую камера потенциально может видеть прямо сейчас. Замечание «потенциально» важно, потому что даже когда активный и видимый объект находится в пределах поля зрения камеры, это не обязательно означает, что он виден для камеры. Дело в том, что объекты в поле зрения могут перекрываться другими объектами, то есть находящиеся ближе к камере объекты могут заслонять объекты, расположенные за ними, полностью или частично. По этой причине правильная проверка видимости должна включать в себя, по меньшей мере, два процесса: во-первых, определение нахождения объекта внутри поля зрения, а во-вторых, определение наличия препятствия, скрывающего объект. Только если объект пройдет обе проверки, его можно признать видимым для камеры, и то при условии, что он не скрыт пользовательскими шейдерами или другими эффектами постобработки. Проще говоря, есть много причин, почему полная проверка видимости является сложным процессом, но здесь я ограничусь двухступенчатой проверкой, которой достаточно для большинства задач.

Определение видимости объекта

Наверное, самым простым и прямолинейным способом проверки видимости объектов в Unity является определение видимости и неви-

димости для некоторой камеры. Два сопутствующих этому события – OnBecameVisible и OnBecameInvisible – автоматически вызываются для любого объекта с компонентом отображения, таким как MeshRenderer или SkinnedMeshRenderer. Это, конечно, не касается пустых объектов, даже попадающих в поле зрения камеры, поскольку они (технически говоря) не содержат видимых частей, несмотря на то, что находятся в пространстве. Вы можете обрабатывать эти события, как показано в листинге 5.2:

Листинг 5.2

```
//---------------------------------------------
using UnityEngine;
using System.Collections;
//---------------------------------------------
public class ViewTester : MonoBehaviour
{
  //---------------------------------------------
  void OnBecameVisible()
  {
    Debug.log ("Became Visible");
  }
  //---------------------------------------------
  void OnBecameInvisible()
  {
    Debug.log ("Became Invisible");
  }
  //---------------------------------------------
}
//---------------------------------------------
```

Есть несколько важных аспектов событий OnBecameVisible и OnBecameInvisible, на которые стоит обратить внимание. Во-первых, видимость здесь означает лишь, что объект попал в поле зрения камеры, но он может быть перекрыт другими объектами, находящимися ближе к камере, и если это так, его может быть не видно вообще. Во-вторых, события относятся ко всем камерам, а не к конкретной камере. Событие OnBecameVisible вызывается один раз, чтобы сообщить, что объект, которого раньше нигде не было видно, вступил в поле зрения, по крайней мере, одной камеры. Точно так же событие OnBecameInvisible вызывается один раз и говорит о том, что объект, который ранее мог быть виден, теперь оставил поле зрения всех камер. И наконец, весьма неприятная особенность этих функций состоит в том, что они отслеживают также видимость камеры сцены. Это означает, что при тестировании игры с открытой и видимой вкладкой **Scene** (Сцена), если

объект виден во вкладке **Scene** (Сцена), он будет считаться видимым. Проще говоря, методы `OnBecameVisible` и `OnBecameInvisible` полезны, только если действия и решения зависят от общей видимости или невидимости в сцене, где видимости соответствует присутствие в поле зрения камеры. Другими словами, эти события хорошо подходят для переключения поведения, зависящего от видимости, например испуг персонажа и другие виды взаимодействий между персонажами.

Более подробную информацию о событиях `OnBecameVisible` и `OnBecameInvisible` можно найти в электронной документации Unity по адресам: http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnBecameVisible.html и http://docs.unity3d.com/ScriptReference/MonoBehaviour.OnBecameInvisible.html.

Подробнее о видимости

Другая важная проверка видимости, не связанная с входом объекта в поле зрения камеры и выходом из него определяет, виден ли объект в настоящий момент для определенной камеры. В отличие от событий `OnBecameVisible` и `OnBecameInvisible`, которые вызываются один раз при входе в поле зрения и выходе из него, этот вид проверки связан только с текущим состоянием объекта и не требует никаких предварительных знаний о нем. Для его реализации можно использовать событие `OnWillRenderObject`. Это событие вызывается регулярно, один раз в каждом кадре для каждой камеры, для которой он виден, все время, пока объект остается видимым для этой камеры. Под «видимостью» здесь понимается «находится в поле зрения». Опять же, проверка заслонения другими объектами не применяется. Взгляните на листинг 5.3, и обратите внимание, что внутри этого события можно использовать свойство `Camera.current`, чтобы получить ссылку на камеру, для которой объект в настоящее время виден, в том числе и на камеру обзора сцены.

Листинг 5.3. Событие OnWillRenderObject

```
void OnWillRenderObject()
{
  Debug.log (Camera.current.name);
}
```

Проверка поля зрения – отображаемые компоненты

Часто встроенных событий Unity, с которыми мы познакомились выше, недостаточно для проверки видимости или нахождения объектов в поле зрения камер. В частности, может быть нужно всего лишь

проверить, виден ли объект только одной определенной камере; попадет ли невидимый объект в поле зрения, если станет видимым; видна ли камере указанная точка в пространстве; или будет ли камере виден определенный объект, если переместить его в новое положение. Все проверки видимости важны в разных ситуациях, и все они требуют ручной доработки. Для реализации этих проверок придется написать достаточно много кода. В следующих разделах мы создадим несколько статических функций и поместим их в выделенный для этого класс `CamUtility`. Давайте начнем с проверки нахождения конкретного отображаемого компонента внутри поля зрения некоторого объекта `Camera`, как показано в листинге 5.4.

Листинг 5.4. Проверка попадания объекта в поле зрения камеры

```
01 using UnityEngine;
02 using System.Collections;
03 //-----------------------------------------------------------
04 public class CamUtility
05 {
06   //-----------------------------------------------------------
07   // Проверяет попадание отображаемого объекта в поле зрения камеры
08   // Возвращает true, если попадает, и false - в противном случае
09   public static bool IsRendererInFrustum(Renderer Renderable,
     Camera Cam)
10   {
11     // Сконструировать плоскости, ограничивающие поле зрения камеры
12     // Каждая плоскость представляет грань усеченной пирамиды
13     Plane[] planes = GeometryUtility.CalculateFrustumPlanes(Cam);
14
15     // Проверить попадание объекта внутрь усеченной пирамиды
16     return GeometryUtility.TestPlanesAABB(planes, Renderable.bounds);
17   }
18   //-----------------------------------------------------------
19 }
```

В строках 10–17 с помощью класса `GeometryUtility` создается массив плоскостей, ограничивающих область поля зрения камеры. Плоскости в трехмерном пространстве подобны прямым в двумерном: они отмечают воображаемую поверхность в трехмерном пространстве. Массив включает шесть плоскостей, размещенных в пространстве так, чтобы образовать поле зрения камеры трапециевидной формы. Этот массив затем используется функцией `TestPlanesAABB`, где аббревиатура **AABB** расшифровывается как **Axially Aligned Bounding Box**, что можно перевести как «объем, ограниченный осями». Эта функция проверяет, находится ли поверхность, ограничивающая отображаемый меш, внутри поля зрения камеры, определяемого плоскостями.

Проверка поля зрения – точки

Конечно, не всегда будет нужна проверка видимости целых объектов. Иногда требуется проверить видимость одной точки. Тому могут быть две основные причины. Во-первых, возможно достаточно знать, виден ли такой объект, как частица или точка прицеливания. Во-вторых, бывает желательно знать не только видима ли точка, но и где на экране она будет показана камерой. Такая проверка реализована в листинге 5.5. Она выясняет, находится ли точка в пределах поля зрения камеры, и возвращает ее координаты в нормализованном виде (между 1 и 0), в окне просмотра.

Листинг 5.5. Проверка попадания точки в поле зрения камеры

```
//-----------------------------------------------------------
// Определяет попадание точки в поле зрения камеры
// Возвращает true, если попадает, и false – в противном случае
// Выходной параметр ViewPortLoc определяет координаты
public static bool IsPointInFrustum(Vector3 Point, Camera Cam,
                                    out Vector3 ViewPortLoc)
{
  // Создать новые границы с нулевыми размерами
  Bounds B = new Bounds(Point, Vector3.zero);

  // Сконструировать плоскости, ограничивающие поле зрения камеры
  // Каждая плоскость представляет грань усеченной пирамиды
  Plane[] planes = GeometryUtility.CalculateFrustumPlanes(Cam);

  // Проверить попадание точки внутрь усеченной пирамиды
  bool IsVisible = GeometryUtility.TestPlanesAABB(planes, B);

  // Присвоить координаты в видимой области
  ViewPortLoc = Vector3.zero;

  // Если точка видима, вернуть ее координаты
  if(IsVisible)
    ViewPortLoc = Cam.WorldToViewportPoint(Point);
  return IsVisible;
}
//-----------------------------------------------------------
```

Проверка поля зрения – заслонение

Как уже упоминалось, проверка видимости в строгом смысле является двухэтапным процессом. Все проверки видимости до сих пор представляли собой только проверку присутствия объекта в поле зрения камеры. Часто этого достаточно, но иногда нет, потому что объекты в поле зрения могут перекрывать друг друга, то есть ближние к каме-

ре объекты могут закрывать дальние полностью или частично. Это не всегда является проблемой, потому что чаще всего основной интерес состоит в том, чтобы определить – достаточно ли близко находится камера для выбора соответствующей реакции (например, реакции искусственного интеллекта). Целью является не проверка видимости как таковой, а близость камеры к объектам. В этих случаях заслонение одних объектов другими не имеет значения, важно только, что объекты находятся в поле зрения. Но иногда возможность заслонения необходимо учитывать, например, при отображении элементов интерфейса или всплывающих уведомлений, или чтобы узнать – видит ли игрок конкретные объекты. Важность можно проиллюстрировать возможностью отображения элемента графического интерфейса за глухой стеной. Иногда эти ситуации можно обойти, творчески используя коллайдеры, триггеры и тщательно размещая объекты, а иногда действительно нет выбора, кроме дальнейшей фильтрации объектов в поле зрения проверкой заслонения одних объектов другими. Такая проверка среди объектов внутри поля зрения является операцией, которая при некоторых способах реализации может привести к значительной потере производительности. По этой причине лучшим считается использование простого метода `Physics.LineCast`, определяющего пересечение воображаемой линии, соединяющей объект с камерой, другими коллайдерами. Этот способ, как правило, работает хорошо, но имеет некоторые ограничения. Во-первых, он предполагает, что все видимые объекты имеют коллайдеры; любые исключения из этого правила не будут обнаружены методом `LineCast`. Во-вторых, коллайдеры только приблизительно повторяют границы мешей и лишь окружают вершины мешей, это может послужить причиной ошибки метода `LineCast`, например когда меши имеют внутренние отверстия, окружающий коллайдер предотвратит проникновения в них луча `LineCast`. Наконец, в соединении с прозрачными материалами, которые делают видимыми объекты, расположенные за ними, метод `LineCast` всегда терпит неудачу. Рассмотрим пример в листинге 5.6.

Листинг 5.6. Определение видимости объекта с помощью метода LineCast

```
//-----------------------------------------------------------
// Определяет видимость объекта
public static bool IsVisible(Renderer Renderable, Camera Cam)
{
  // Если находится в поле зрения, протянуть линию
  if(!CamUtility.IsRendererInFrustum(Renderable, Cam))
```

```
    return false;

  // Проверить, не пересекается ли линия , соединяющая камеру с объектом
  If(!Physics.Linecast(Renderable.transform.position,
                       Cam.transform.position);
    return false; // Линия пересекается, объект невидим

  return true; // Объект видим
}
//---------------------------------------------------------
```

Видимость для камеры – впереди или позади

В некоторых играх, таких как стратегии реального времени или казуальные игры, горизонт камеры (или дальняя плоскость отсечения) не имеет большого значения, потому что камера всегда видит все, что находится перед ней. В этих случаях, объекты оказываются вне поля зрения, только когда находятся за плоскостями *x* и *y*, а расстояние по локальной оси *z* не имеет значения. То есть скрытые объекты не видны только потому, что камера не направлена прямо на них. Однако, когда камера ориентирована правильно, расстояние до объектов не играет никакого значения для их видимости. В таких ситуациях проверку видимости часто можно свести к быстрым и простым проверкам ориентации. То есть, вопрос: «Находится ли объект в поле зрения и не заслонен другим объектом?», – можно заменить вопросом: «Находится ли объект перед камерой или позади нее?». Здесь и ответ будет другим, так как вопрос будет связан не с видимостью, а с ориентацией – будут ли камера и объект ориентированы так, что объект находится перед камерой или позади нее. Для такой проверки можно использовать скалярное произведение векторов. Скалярное произведение умножает два вектора и возвращает скалярное числовое значение. Это значение описывает угловое отношение между векторами. В следующем листинге 5.7 представлен класс `CamFieldView`, который можно прикрепить к камере, и он будет обнаруживать, может ли камера видеть целевой объект, то есть, находится ли целевой объект перед камерой в пределах ее ограниченного поля зрения.

Листинг 5.7. Класс CamFieldView

```
using UnityEngine;
using System.Collections;
//-------------------------------------------------
public class CamFieldView : MonoBehaviour
{
  //-------------------------------------------------
```

```
  // Угол поля зрения камеры
  // Измеряется в градусах от направления вперед (влево или вправо)
  public float AngleView = 30.0f;

  // Целевой объект
  public Transform Target = null;

  // Локальная трансформация
  private Transform ThisTransform = null;
  //-------------------------------------------------
  // Используется для инициализации
  void Awake ()
  {
    // Получить локальную трансформацию
    ThisTransform = transform;
  }

  //-------------------------------------------------
  // Вызывается при отображении каждого кадра
  void Update ()
  {
    // Обновить видимое пространство между камерой и целью
    Vector3 Forward = ThisTransform.forward.normalized;
    Vector3 ToObject = (Target.position - ThisTransform.position).
                       normalized;

    // Вычислить скалярное произведение
    float DotProduct = Vector3.Dot(Forward, ToObject);
    float Angle = DotProduct * 180f;

    // Проверить попадание в поле зрения
    if(Angle >= 180f-AngleView)
    {
      Debug.log ("Object can be seen");
    }
  }
  //-------------------------------------------------
}
//-------------------------------------------------
```

Ортографические камеры

Каждый вновь созданный объект камеры в Unity по умолчанию настроен на режим перспективы, если вы не изменили параметры по умолчанию. Камеры этого вида наиболее близко соответствует реальным камерам, размещенным в трехмерном пространстве, имеющим изогнутую линзу и использующим метод фиксации изображения на плоской двумерной поверхности, или экране. Главным признаком та-

кой камеры является ракурс, как называется искажение, применяемое к отображаемым объектам. В частности, отображаемые объекты становятся меньше, с увеличением расстояния, форма и внешний вид объектов изменяются тем больше, чем дальше они от центра зрения, и все параллельные линии сходятся где-то в удаленной точке, либо на линии горизонта, либо на какой-то другой линии. Но, кроме перспективных камер, существуют еще и ортографические камеры. Они полезны для создания двухмерных игр и настоящих изометрических игр, в отличие от игр, похожих на изометрические. В ортографических камерах объектив строго плоский, ракурса не существует, то есть параллельные линии остаются параллельными, объекты не уменьшаются с расстоянием, объекты не искажаются при отдалении от центра зрения и т. д. Вы легко можете включить ортографический режим камеры, заменив параметр **Perspective** (Перспективная) параметром **Orthographic** (Ортографическая) в поле **Projection** (Проекция), в инспекторе объектов, как показано на рис. 5.3.

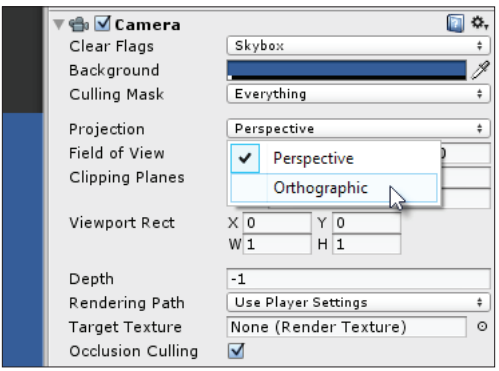


Рис. 5.3. Превращение перспективной камеры в ортографическую

После изменения типа камеры с перспективной на ортографическую, форма поля зрения камеры тоже изменится с объемной трапециевидной на прямоугольную. Все, видимое пространство будет иметь форму прямоугольника, ближние объекты по-прежнему будут перекрывать более далекие, но ощущение глубины утратится, как показано на рис. 5.4. Такая камера больше подходит для двухмерных игр.

Главной проблемой при работе с ортографическими камерами является создание соотношения 1:1 между единицами расстояний

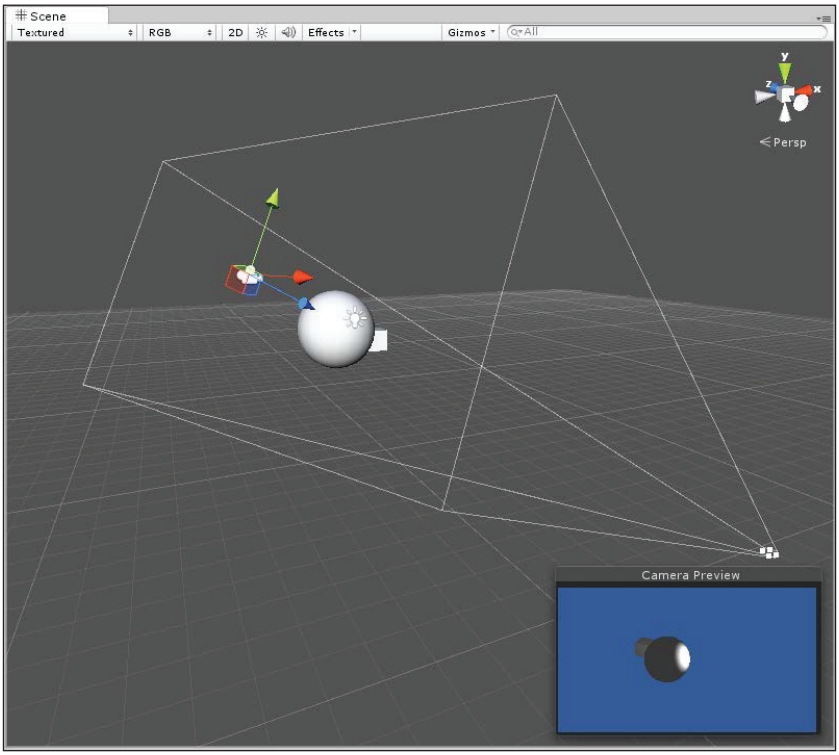


Рис. 5.4. Поле зрения ортографической камеры имеет вид прямоугольника

(в сцене) и пикселями (на экране). Эта проблема возникает потому, что в двухмерных играх и графических интерфейсах желательно отображать графику на экране в ее изначальных и правильных размерах, как они заданы в файлах текстур. В большинстве трехмерных игр ракурс и перспектива искажают текстуры, то есть при проецировании на поверхность трехмерных объектов они отображаются не прямо, как в программе для редактирования фотографий, а в перспективе. В случае с двухмерными играми и спрайтами ситуация иная. Эта графика отображается напрямую. Поэтому желательно сохранить ее стандартные размеры, пиксель в пиксель. Такой вид визуализации называется безупречным, потому каждый пиксель в текстуре выводится на экран без изменений. Но реализация его требует особого подхода. Проще говоря, для отображения 1 игровой единицы в 1 пик-

сель значение в поле **Size** (Размер) во вкладке **Camera** (Камера) должно быть равно половине вертикального разрешения дисплея. То есть, если игра будет запущена в окне с разрешением 1024×768, поле **Size** (Размер) должно содержать значение 364, потому что 768 / 2 = 364, как показано на рис. 5.5.

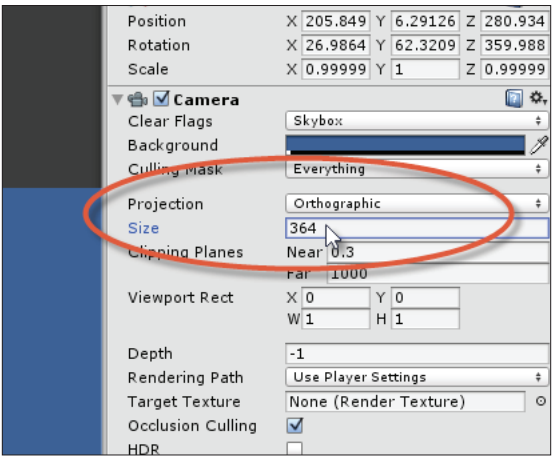


Рис. 5.5. Поле **Size** (Размер) управляет преобразованием единиц расстояний в игре в пиксели экрана

Вы можете установить значение в поле **Size** (Размер) непосредственно в редакторе, но в этом случае игра будет работать правильно только в окне с постоянным неизменным разрешением. Если пользователь имеет возможность изменять размер окна игры или разрешение, вам нужно будет обновить размер камеры программно, как это показано в листинге 5.8.

Листинг 5.8. Изменение размера камеры программно

```
01 //---------------------------------------------------------
02 using UnityEngine;
03 using System.Collections;
04 //---------------------------------------------------------
05 [RequireComponent(typeof(Camera))]
06 //---------------------------------------------------------
07 public class OrthoCam : MonoBehaviour
08 {
09   // закрытая ссылка на компонент камеры
```

```
10   private Camera Cam = null;
11
12   // Число пикселей в единице размера Units
13   public float PixelsToWorldUnits = 200f;
14   //-------------------------------------------------------
15   // Этот метод выполняет инициализацию
16   void Awake ()
17   {
18     // Получить ссылку на камеру
19     Cam = GetComponent<Camera>();
20   }
21   //-------------------------------------------------------
22   // Вызывается при отображении каждого кадра
23   void LateUpdate ()
24   {
25     // Изменить ортографический размер
26     Cam.orthographicSize = Screen.height / 2f / PixelsToWorldUnits;
27   }
28   //-------------------------------------------------------
29 }
30 //---------------------------------------------------------
```

Обратите внимание на переменную `PixelsToWorldUnits` в строке 13, определяющую масштаб ортографического размера в соответствии с полем **Pixels To Units** (Пикселей в единице размера) импортированной текстуры спрайтов, как показано на рис. 5.6. Это гарантирует, что спрайты отобразятся с правильными размерами при выводе на экран. Это вызвано необходимостью масштабировать все спрайты в соответствии с этим значением для отображения пикселей текстуры в мировые единицы измерения.

Вывод изображения с камеры и постобработка

В официальной документации Unity вывод изображения с камеры и постобработка упоминаются сравнительно редко. Тем не менее, это не следует воспринимать как свидетельство, что данная тема не требует никаких пояснений. Напротив, камеры и объекты Unity предоставляют широкие возможности управления отображением сцены. Все связанные с этим вопросы называются постобработкой. В частности, это относится ко всем дополнительным правкам и изменениям, внесенным в изображение с камеры, не относящимся к обычной визуализации. Они включают в себя эффекты размытия, регулировки цвета, эффект рыбьего глаза и т. д. Следует отметить,

что все эти функции доступны только в профессиональной версии Unity. По этой причине пользователи бесплатной версии не смогут на практике применить сведения, почерпнутые из этого раздела. Тем не менее, пользователям профессиональной версии доступен широкий спектр возможностей вывода изображений с камер, один из примеров которых показан на рис. 5.7. В этом разделе будет рассмотрено создание системы внесения изменений в изображение с камеры, обеспечивающей плавный переход изображения с одной камеры в изображение с другой. Переход означает не просто врезку изображения с одной камеры в изображение с другой камеры, что (случайно) может быть достигнуто при изменении значения поля глубины камеры. Это означает, что изображение с первой камеры будет постепенно становиться все прозрачнее, открывая изображение со второй камеры, и наоборот. Итак, начнем.

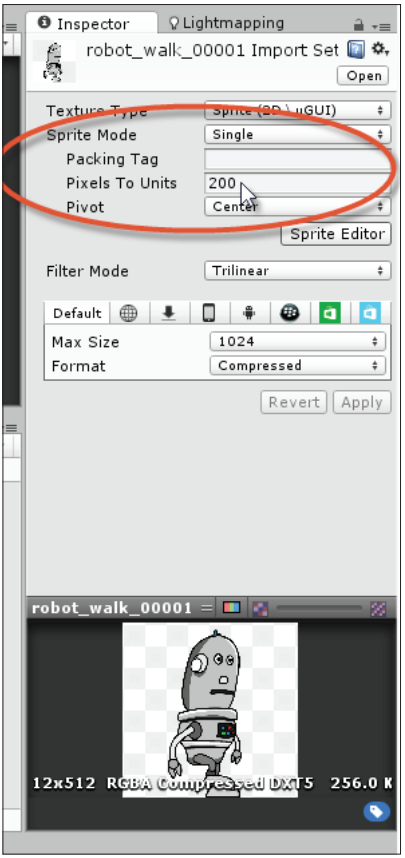


Рис. 5.6. Значение поля **Pixels to Units** (Пикселей в единице размера) для текстуры спрайтов

Создайте проект со сценой, содержащей две отдельные зоны или области, как показано на рис. 5.7. Пример проекта можно найти в пакете примеров к книге, в папке `Cameras`. Каждой зоне в сцене должна быть придана отдельная камера, всего две камеры в сцене, и обе камеры должны быть деактивированы. Это позволит избежать автоматического вывода изображений с камер. Здесь мы будем управлять изображениями вручную, это позволит скомбинировать изображения и наложить изображение с одной камеры на изображение с другой.

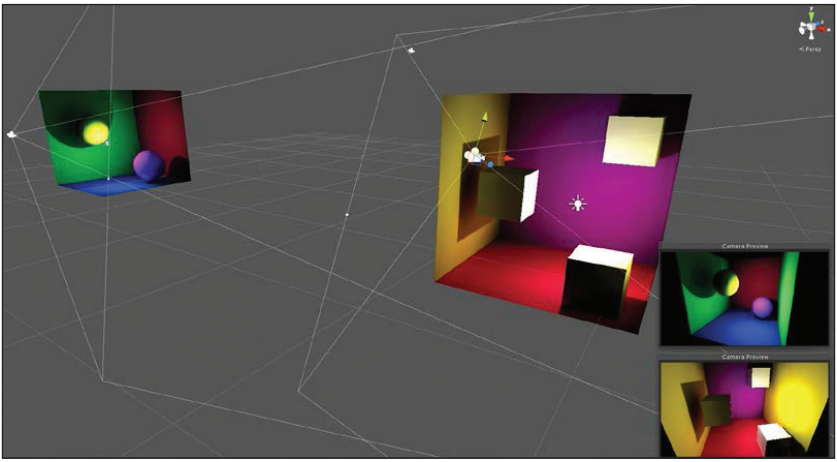


Рис. 5.7. Создание сцены с несколькими камерами

Компоненты `AudioListener` обеих камер должны быть удалены, так как Unity разрешает присутствие только одного активного компонента `AudioListener` в сцене.

Далее создайте третью камеру с тегом `MainCamera` в начале координат сцены и установите ей пустую маску отбраковки. Убедитесь, что камера активна, но ничего не отображает. Она будет представлять собой центральную, основную камеру сцены, соединяющую изображения с двух других камер, как показано на рис. 5.8.

Теперь сцена имеет три камеры: две неактивные камеры в разных местах (камеры *X* и *Y*) и одну основную камеру сцены (камера *Z*). При подключении кода из листинга 5.9 к камере *Z*, он обеспечит плавную смену изображений с камер *X* и *Y* после нажатии клавиши пробела.

Листинг 5.9. Реализация наложения изображений с двух камер

```
001 // Класс для наложения изображений с камер 0 и 1
002 // Предполагает присутствие в сцене двух дополнительных камер
003 //--------------------------------------
004 using UnityEngine;
005 using System.Collections;
006 //--------------------------------------
007 public class CameraFader : MonoBehaviour
008 {
009   //--------------------------------------
010   // Все камеры в сцене для объединения изображений
```

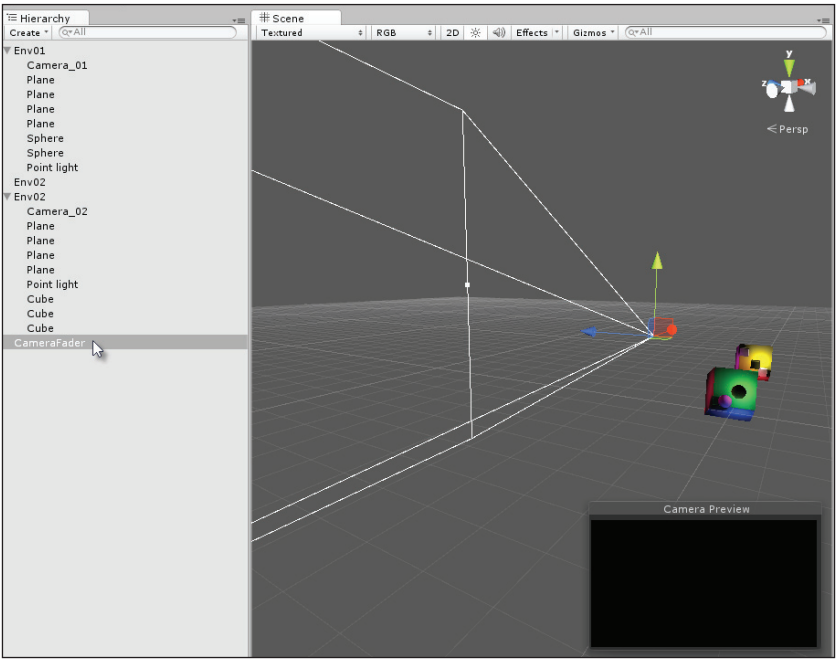


Рис. 5.8. Создание третьей, основной камеры

```
011   public Camera[] Cameras;
012
013   // Цвет для умножения
014   public Color[] CamCols = null;
015
016   // Время растворения/проявления в секундах
017   public float FadeTime = 2.0f;
018
019   // Материал для окончательного отображения
020   public Material Mat = null;
021   //-------------------------------------
022   // Этот метод выполняет инициализацию
023   void Start ()
024   {
025     // Присвоить текстуры каждой камере
026     foreach(Camera C in Cameras)
027       C.targetTexture = new
          RenderTexture(Screen.width, Screen.height, 24);
          // Создаст текстуру
028   }
029   //-------------------------------------
```

```
030  // Вызывается один раз в каждом кадре после получения изображения
031  // с камеры, но перед выводом его на экран
032  // Парная функция: OnPreRender
033  void OnPostRender()
034  {
035    // Определить прямоугольник экрана
036    Rect ScreenRct = new Rect(0,0,Screen.width,Screen.height);
037
038    // Исходный прямоугольник
039    Rect SourceRect = new Rect(0,1,1,-1);
040
041    // Вывести изображение  каждой камеры в их текстуры
042    for(int i = 0; i<Cameras.Length; i++)
043    {
044      // Вывести изображение
045      Cameras[i].Render();
046
047      // Нарисовать текстуры на кране с помощью камеры
048      GL.PushMatrix();
049      GL.LoadPixelMatrix();
050      Graphics.DrawTexture(ScreenRct,
         Cameras[i].targetTexture, SourceRect, 0,0,0,0, CamCols[i]);
051      GL.PopMatrix(); // Сбросить матрицу
052    }
053  }
054  //---------------------------------------
055  // Эта функция вызывается после OnPostRender,
056  // когда пиксели изображения выводятся на экран
057  // src = текущее изображение с камеры
058  // dst = текстура для вывода на экран
059  void OnRenderImage(RenderTexture src, RenderTexture dst)
060  {
061    // Вывести на экран окончательные пиксели с материалом Mat
062    Graphics.Blit(src, dst, Mat);
063  }
064  //---------------------------------------
065  // Смена цвета в течение периода TotalTime
066  // Изменить альфа-канал изображения сверху
067  public IEnumerator Fade(Color From, Color To, float TotalTime)
068  {
069    float ElapsedTime = 0f;
070
071    // Выполнять цикл, пока время не истекло
072    while(ElapsedTime <= TotalTime)
073    {
074      // Изменить цвет
075      CamCols[1] = Color.Lerp(From, To, ElapsedTime/TotalTime);
076
077      // Ждать следующего кадра
078      yield return null;
```

```
079
080       // Обновить время
081       ElapsedTime += Time.deltaTime;
082     }
083
084     // Применить конечный цвет
085     CamCols[1] = Color.Lerp(From, To, 1f);
086   }
087   //--------------------------------------
088   // Пример тестирования функциональности камеры
089   // Нажмите клавишу пробела чтобы включить плавную смену изображений
090   void Update()
091   {
092     // Растворить или проявить изображение с камеры
093     if(Input.GetKeyDown(KeyCode.Space))
094     {
095       StopAllCoroutines();
096
097       // Растворить или проявить?
098       if(CamCols[1].a <= 0f)
099         StartCoroutine(Fade(CamCols[1], new Color(0.5f,0.5f,0.5f,1f),
            FadeTime));          // Проявить
100       else
101         StartCoroutine(Fade(CamCols[1], new Color(0.5f,0.5f,0.5f,0f),
            FadeTime)); // Растворить
102      }
103   }
104   //--------------------------------------
105 }
```

Ниже приводится несколько комментариев к листингу 5.9:

- **Строки 011–020**: класс `CamerFader` отвечает за плавную смену изображений с камер `Camera[0]` и `Camera[1]`. Для этого создано несколько переменных. Массив `Cameras` содержит список камер, в нашем случае две камеры. Массив `CamCols` связан с массивом `Cameras`. Он определяет цвета, с которыми будут смешиваться изображения с камер, это позволяет с помощью альфа-канала сделать изображение прозрачным. Переменная `FadeTime` определяет общее время в секундах, за которое происходит полная смена изображений. Наконец, переменная `Mat` содержит ссылку на любой доступный материал, который будет применен к окончательному изображению основной камеры, то есть к изображению, собранному из изображений со всех других камер.
- **Строки 023–038**: в методе `Start` для каждой камеры создается текстура `RenderTexture`, которая присваивается свойству `TargetTexture` камеры. В сущности, это означает, что каждой

камере присваивается внутренняя текстура, куда будет выводиться изображение.

- **Строки 033–052**: событие `OnPostRender` вызывается автоматически для любых активных камер в сцене, в каждом кадре, после завершения его отображения. Это дает возможность включить изображения с дополнительных камер, наложив их поверх других изображений, полученных как обычно. Здесь вызывается метод `Render` каждой камеры из массива `Cameras`. Этот метод выводит изображение с камеры, но не на экран, а в текстуру. После получения текстур функция `Graphics.DrawTexture` прорисовывает на экране текстуры `RenderTexture` каждой камеры, в порядке их вхождения в массив, одну поверх другой. Обратите внимание, что при каждом вызове функция `DrawTexture` присоединяет цвет `CamCols` к текстуре, он является множителем для альфа-канала, обеспечивая создание эффекта прозрачности.
- **Строки 059–063**: событие `OnRenderImage`, так же как `OnPostRender`, вызывается автоматически для активных камер один раз в кадр, после события `OnPostRender`, до вывода изображений с камер на экран. Это событие принимает два аргумента – `src` и `dst`. Аргумент `src` должен содержать ссылку на текстуру, в которой находится окончательное изображение, полученное после выполнения события `OnPostRender`, аргумент `dst` должен содержать ссылку текстуру для отображения на экране после завершения события `OnRenderImage`. Проще говоря, эта функция дает возможность изменить пиксели изображения вручную или с помощью шейдеров. Здесь функция `Graphics.Blit` вызывается для копирования из `src` в `dst` с использованием шейдера, основанного на материале `Mat`.
- **Строки 067–085**: функция `Fade` является сопрограммой и производит перехода от цвета `From` к цвету `To` за время `TotalTime`. Эта сопрограмма используется для изменения альфа-канала цвета от значения `0` до значения `1`, которые, соответственно, определяют прозрачность и непрозрачность.

Рисунок 5.9 показывает эффект перехода изображений.

Дрожание камеры

А теперь рассмотрим эффект, доступный в бесплатной версии Unity: дрожание камеры! Для боев, стрельбы и приключений вообще, эффект дрожания камеры очень важен. Он передает толчок, опасность,

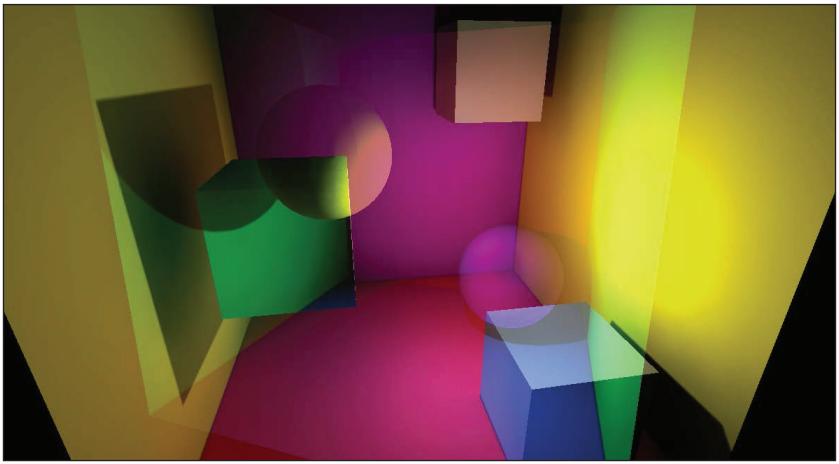


Рис. 5.9. Плавная смена изображений с камер

действие, динамизм и волнение в форме кинетической обратной связи. Этот эффект можно использовать вместе со множеством других эффектов, имитируя всепроникающее движение и эмоции в определенном месте сцены. В этом смысле эффект дрожания камеры может избавить от массы работы по созданию всеобъемлющей анимации, как показано на рис. 5.10.

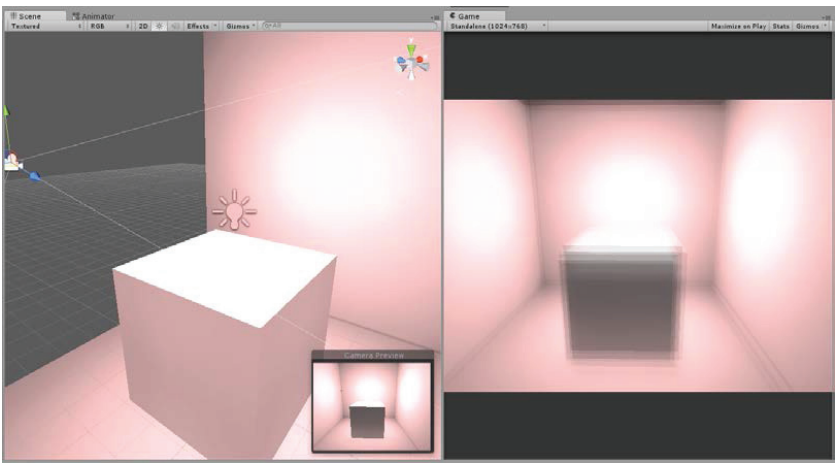


Рис. 5.10. Эффект дрожания камеры

Есть много способов создать эффект дрожания камеры, но все они основаны на колебаниях положения камеры в диапазоне между минимальным и максимальным пределами, с использованием некоторой функции получения случайных значений. Иногда случайность используется в чистом виде, а иногда она сглаживается с помощью функции демпфирования, чтобы создать более медленное или более плавное дрожание. Рассмотрим пример в листинге 5.10, который можно прикрепить к любой камере для создания эффекта дрожания.

Листинг 5.10. Реализация эффекта дрожания камеры

```
using UnityEngine;
using System.Collections;
//--------------------
public class CameraShake : MonoBehaviour
{
  private Transform ThisTransform = null;

  // Общее время дрожжания в секундах
  public float ShakeTime = 2.0f;

  // Амплитуда дрожжания - смещение в любом направлении
  public float ShakeAmount = 3.0f;

  // Скорость перемещения камеры
  public float ShakeSpeed = 2.0f;

  //----------------------
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Получить компонент трансформации
    ThisTransform = GetComponent<Transform>();

    // Начать дрожжание
    StartCoroutine(Shake());
  }

  //----------------------
  // Дрожжание камеры
  public IEnumerator Shake()
  {
    // Сохранить исходную позицию камеры
    Vector3 OrigPosition = ThisTransform.localPosition;

    // Счетчик прошедшего времени в секундах
```

```
    float ElapsedTime = 0.0f;

    // Повторять, пока время дрожжания не истекло
    while(ElapsedTime < ShakeTime)
    {
      // Выбрать точку в сфере
      Vector3 RandomPoint = OrigPosition +
        Random.insideUnitSphere * ShakeAmount;

      // Изменить позицию
      ThisTransform.localPosition = Vector3.Lerp(
        ThisTransform.localPosition, RandomPoint,
        Time.deltaTime * ShakeSpeed);

      // Приостановиться до следующего кадра
      yield return null;

      // Обновить время
      ElapsedTime += Time.deltaTime;
    }

    // Восстановить исходную позицию камеры
    ThisTransform.localPosition = OrigPosition;
  }

  //----------------------
}
//----------------------
```

Камеры и анимация

Пролет камеры через сцену – это анимационный эффект, в котором камера перемещается и поворачивается с течением времени по определенным позициям для создания кинематографического эффекта. Обычно пролет камеры применяется для создания заставок в играх, хотя и не всегда. Он может пригодиться для создания камер персонажей от третьего лица и для обзора сверху по определенному заранее маршруту. Часто для получения эффекта пролета камеры предварительно создается траектория ее полета в редакторе анимаций Unity или в пакетах сторонних производителей, таких как Maya, Blender и 3DS Max. Однако, иногда бывает достаточно простого программного управления камерой, чтобы плавно провести ее от заданной позиции через ряд точек по предопределенному маршруту. В этом разделе будут рассмотрены все три подхода.

Сопровождающие камеры

Пожалуй, одним из самых распространенных видов камер являются сопровождающие камеры, то есть камеры, которые отслеживают заданный объект в сцене и следуют за ним. Такие камеры держатся на некотором расстоянии от объекта, как показано на рис. 5.11. Так действует камера персонажа от третьего лица, реализующая обзор из-за плеча или сверху.

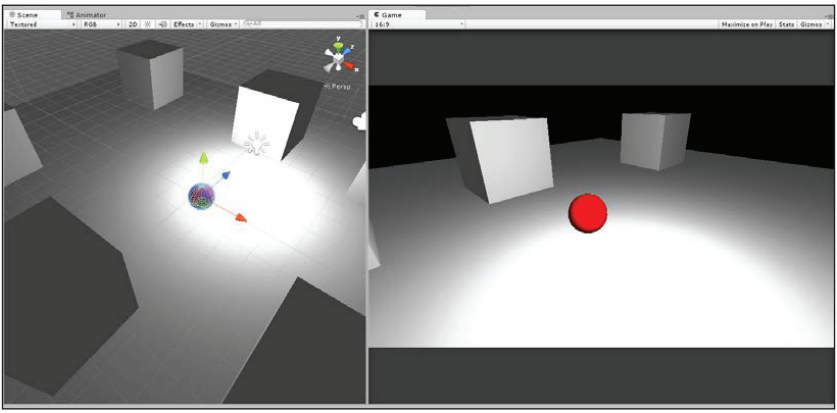


Рис. 5.11. Создание камеры, плавно следующей за объектом

Этот проект можно найти в пакете с примерами к книге, в папке `Camera_Smooth_Damp`.

Для таких камер простого следования, как правило, бывает недостаточно. Если бы требовалось только это, можно было бы просто сделать объект родителем камеры и на этом закончить. Однако, как правило, желательно добиться некоторой степени сглаживания, или демпфирования движения камеры, то есть сделать смену скоростей плавной, что позволит камере постепенно замедляться до полной остановки при достижении цели, а не останавливаться внезапно и резко. Для этого можно использовать функции `Quaternion.Slerp` и `Vector3.SmoothDamp`. Рассмотрим следующий пример в листинге 5.11, содержащий класс, который можно подключить к любой камере, чтобы заставить ее плавно следовать за объектом.

Листинг 5.11. Реализация сопровождающей камеры

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

```
//-------------------------------------------------------------------
public class CamFollow : MonoBehaviour
{
  //-----------------------------------------------------------------
  // Цель для преследования
  public Transform Target = null;

  // Сылка на локальную трансформацию
  private Transform ThisTransform = null;

  // Линейное расстояние до цели (в мировых единицах измерения)
  public float DistanceFromTarget = 10.0f;

  // Высота камеры над целью
  public float CamHeight = 1f;

  // Демпфирование вращения
  public float RotationDamp = 4f;

  // Демпфирование позиции
  public float PosDamp = 4f;

  //-----------------------------------------------------------------
  void Awake()
  {
    // Получить трансформацию для камеры
    ThisTransform = GetComponent<Transform>();
  }
  //-----------------------------------------------------------------
  // Вызывается при отображении каждого кадра
  void LateUpdate ()
  {
    // Получить скорость
    Vector3 Velocity = Vector3.zero;

    // Вычислить угол поворота интерполяцией
    ThisTransform.rotation =
      Quaternion.Slerp(ThisTransform.rotation,
      Target.rotation, RotationDamp * Time.deltaTime);

    // Вычислить новую позицию
    Vector3 Dest = ThisTransform.position =
      Vector3.SmoothDamp(ThisTransform.position,
        Target.position, ref Velocity,
        PosDamp * Time.deltaTime);

    // Отдалить от цели
    ThisTransform.position = Dest - ThisTransform.forward *
      DistanceFromTarget;

    // Вычислить высоту
```

```
    ThisTransform.position =
      new Vector3(ThisTransform.position.x, CamHeight,
                  ThisTransform.position.z);

    // Направление на цель
    ThisTransform.LookAt(Dest);
  }
  //-------------------------------------------------------------------
}
```

Более подробную информацию о функции `Quaternion.Slerp` можно найти по адресу http://docs.unity3d.com/ScriptReference/Quaternion.Slerp.html, а о функции `Vector3.SmoothDamp` – по адресу http://docs.unity3d.com/ScriptReference/Vector3.SmoothDamp.html.

Управление движением камеры

Для заставок, фона меню или организации обычного пролета камеры может понадобиться реализовать движение камеры по прямой, с изменением скорости, при этом камера должна плавно начинать и заканчивать свое движение. Это значит, что камера должна набрать скорость в начале пути, а затем постепенно снизить скорость в конце пути. Для этого, можно использовать эффект, подготовленный с помощью редактора анимаций в Unity, или использовать анимационные кривые, обеспечивающие высокую степень гибкости и контроля над преобразованиями объекта с течением времени, как показано на рис. 5.12.

Чтобы создать сценарий управления скоростью движения камеры с течением времени, в том числе для реализации движения по кри-

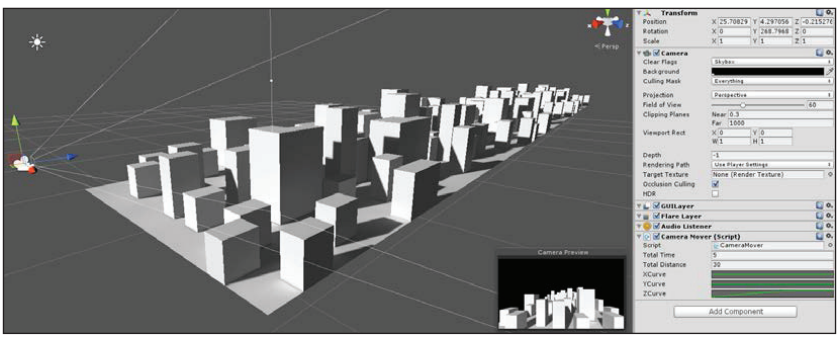


Рис. 5.12. Перемещение камеры
с применением анимационных кривых

волинейной траектории и сглаживания или регулирования скорости, можно использовать сценарий из листинга 5.12.

Листинг 5.12. Управление движением камеры

```
//-----------------------------
using UnityEngine;
using System.Collections;
//-----------------------------
public class CameraMover : MonoBehaviour
{
  //-----------------------------
  // Время длительности анимационного эффекта
  public float TotalTime = 5.0f;

  // Расстояние перемещения по каждой оси
  public float TotalDistance = 30.0f;

  // Кривые для перемещения
  public AnimationCurve XCurve;
  public AnimationCurve YCurve;
  public AnimationCurve ZCurve;

  // Преобразование для данного объекта
  private Transform ThisTransform = null;

  //-----------------------------
  void Start()
  {
    // Получить компонент трансформации
    ThisTransform = GetComponent<Transform>();

    // Запустить анимацию
    StartCoroutine(PlayAnim());
  }
  //-----------------------------
  public IEnumerator PlayAnim()
  {
    // Время, прошедшее с начала анимации
    float TimeElapsed = 0.0f;

    while(TimeElapsed < TotalTime)
    {
      // Нормализовать время
      float NormalTime = TimeElapsed / TotalTime;

      // Вычислить смещение по осям X, Y и Z
      Vector3 NewPos = ThisTransform.right.normalized *
          XCurve.Evaluate(NormalTime) * TotalDistance;

      NewPos += ThisTransform.up.normalized *
```

```
            YCurve.Evaluate(NormalTime) * TotalDistance;

        NewPos += ThisTransform.forward.normalized *
            ZCurve.Evaluate(NormalTime) * TotalDistance;

        // Изменить позицию
        ThisTransform.position = NewPos;

        // Ждать следующего кадра
        yield return null;

        // Обновить время
        TimeElapsed += Time.deltaTime;
      }
    }
    //------------------------------
}
//------------------------------
```

Проект, демонстрирующий использование анимационных кривых для перемещения камеры, можно найти в пакете примеров, сопровождающих книгу, в папке `Camera_Anim_Curves`.

Чтобы задействовать класс `CameraMover`, прикрепите сценарий к камере и в инспекторе объектов щелкните на каждом из полей определения кривых по осям **X**, **Y** и **Z** для настройки изменения скорости камеры с течением времени. Щелкая на графике, можно отредактировать его, добавляя точки и изменяя кривые для выбранной оси. Обратите внимание, что оси **X**, **Y** и **Z** привязаны к локальным осям объекта (вперед, вверх и вправо), а не к глобальным осям (*x*, *y* и *z*). С их помощью задается относительное движение объекта, как показано на рис. 5.13.

Более подробную информацию об анимационных кривых можно найти в электронной документации Unity по адресу http://docs.unity3d.com/Manual/ AnimatorCurves.html.

Траектория камеры – iTween

Одним из полезных свойств, которое, как ни странно, до сих пор не реализовано в Unity, являются программируемые траектории движения. Под этим термином понимается возможность объекта, например камеры, плавно следовать по траектории, проложенной с помощью сферической интерполяции, когда путь определяется серией связанных игровых объектов. Эта функция уже существует в том смысле, что движение камеры можно определить с помощью анимаций, созданных в редакторе Unity. Однако, желательно иметь более гибкий

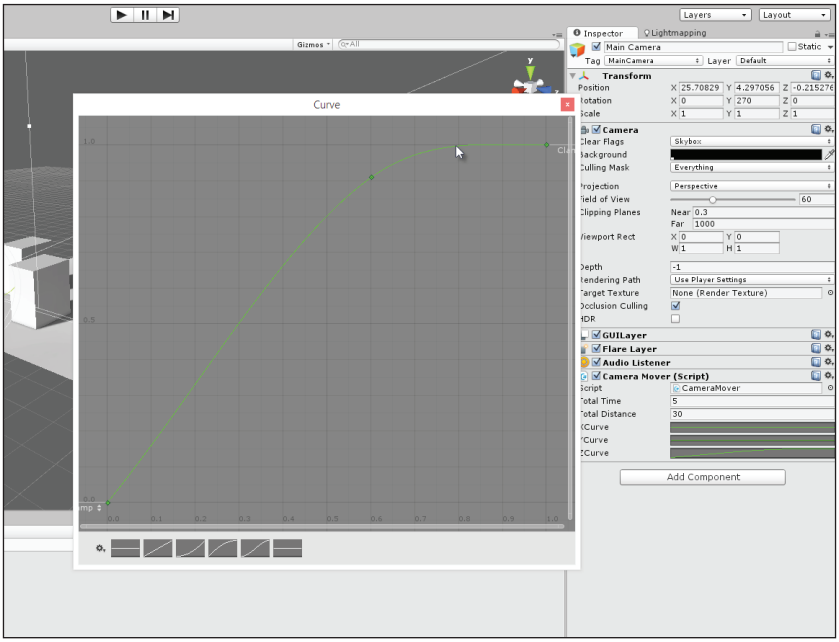


Рис. 5.13. Прокладка траектории движения с помощью кривых анимации

способ программного управления траекторией движения, определяемой набором точек, позиции которых можно корректировать программно, в определенные моменты времени. Эта функция особенно полезна, например, для космических игр, где полет вражеских кораблей четко следует гладким, криволинейным траекториям, соответствующим положению космического корабля игрока, как показано на рис. 5.14. Есть много способов достижения этой цели в Unity, но быстрое решение состоит в использовании бесплатного расширения iTween Боба Беркебайла (Bob Berkebile), которое можно загрузить и импортировать непосредственно с сайта Asset Store Unity. Более подробную информацию о расширении iTween можно найти по адресу http://itween.pixelplacement.com/index.php.

В дополнение к самому пакету iTween можно также скачать бесплатное расширение для iTween – визуальный редактор траекторий iTween (Visual iTween Path Editor), доступный по адресу http://pixelplacement.com/2010/12/03/visual-editor-for-itween-motion-paths/.

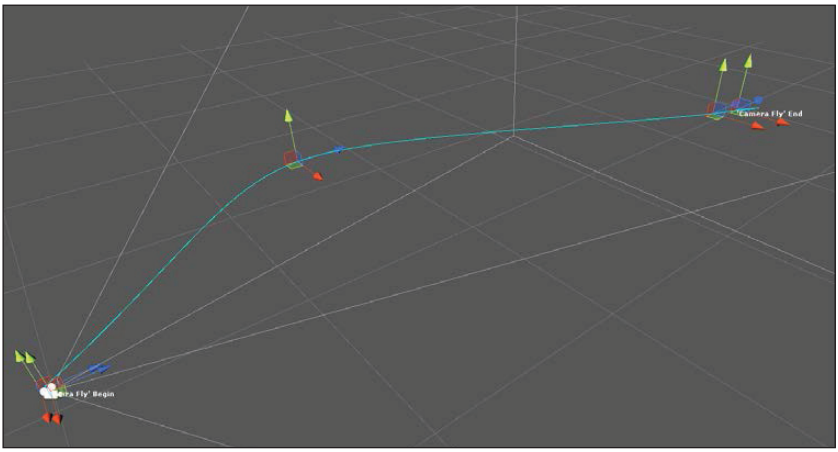


Рис. 5.14. Создание траекторий движения камеры с помощью iTween

Импортировав оба пакета iTween, их можно использовать для анимации движения объекта вдоль траектории. Чтобы реализовать пролет камеры, перетащите сценарий `iTweenPath` на объект камеры. Он позволяет создать независимый и именованный путь, состоящий из нескольких путевых точек, как показано на рис. 5.15.

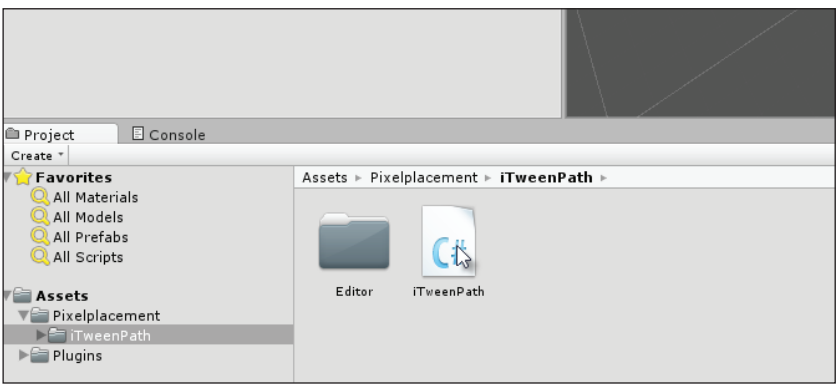


Рис. 5.15. Сценарий iTweenPath позволяет определить траекторию с помощью путевых точек

Чтобы определить несколько путевых точек, введите общее их число в поле **Node Count** (Число узлов), а затем выберите визуальное представление каждого узла в окне просмотра сцены и поместите их в нужные места. Обратите внимание на линию, соединяющую точки – это траектория камеры (см. рис. 5.16).

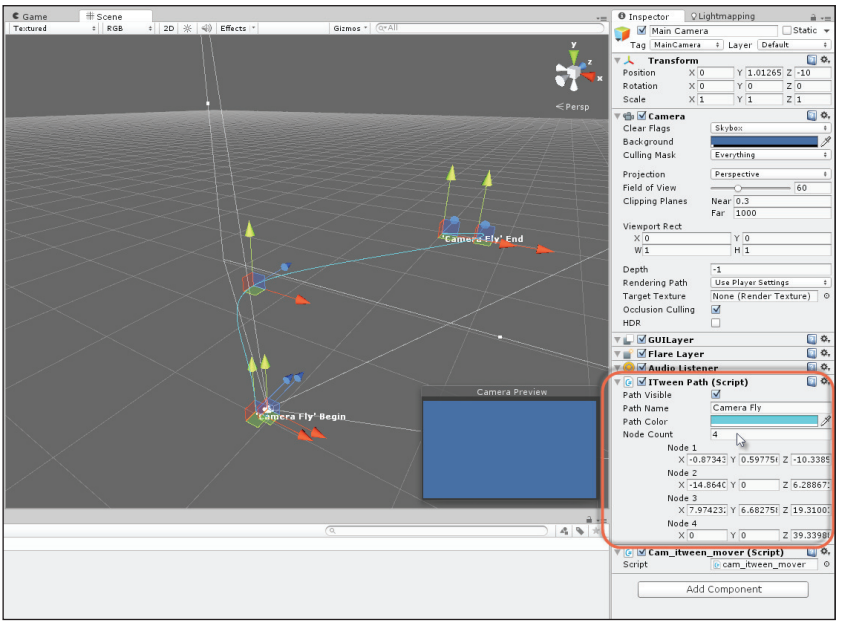


Рис. 5.16. Определение путевых точек для траектории

Затем, чтобы заставить камеру двигаться вдоль траектории, добавьте к камере сценарий, представленный в листинге 5.13.

Листинг. 5.13. Сценарий, управляющий движением камеры вдоль траектории

```
using UnityEngine;
using System.Collections;

public class cam_itween_mover : MonoBehaviour
{
  // Этот метод выполняет инициализацию
  void Start ()
  {
    iTween.MoveTo(gameObject, iTween.Hash("path",
```

```
        iTweenPath.GetPath("Camera Fly") , "time", 4f, "easetype",
        iTween.EaseType.easeInOutSine));
    }
}
```

Более подробную информацию о расширении iTween и его использовании можно найти по адресу http://itween.pixelplacement.com/gettingstarted.php.

Итоги

В этой главе все внимание было сосредоточено на решении задач, связанных с камерами. Камеры имеют большое значение в Unity, как и в любом другом игровом движке, потому что предназначены для визуализации сцены на экране. Большая часть функциональности камеры встроена в Unity, и, как следствие, это ведет к потере гибкости и возможностей управления камерами, многие из которых плохо документированы. В частности, здесь мы впервые рассмотрели возможность постоянного отображения визуального представления камер в окне сцены, даже если камера не выбрана. Во-вторых, мы узнали, как определить, какие объекты видны для камеры, а какие нет. Это реализуется несколькими видами проверок, таких как проверка на нахождение поле зрения и выявление заслоняющих препятствий. В-третьих, мы увидели, как создавать и настраивать ортографические камеры, отображающие двухмерные элементы без искажений. В-четвертых, мы узнали, как изменять и улучшать изображения, полученные с камер через промежуточные текстуры. Для этого потребовалось переопределить несколько важных событий камеры и организовать смешивание изображений с камер для создания эффекта растворения/проявления изображений при смене камер. В-пятых, мы увидели, как реализовать движение камеры, например, дрожание камеры. И наконец, мы узнали, как определить траекторию движения камеры, то есть организовать ее перемещение камеры по заданному маршруту, определенному серией путевых точек, или простое сопровождение объекта. Далее мы займемся исследованием фреймворка Mono.

Глава 6

Работа с фреймворком Mono

Unity поддерживает два основных языка программирования, а именно C# и JavaScript. Разработчики должны выбрать один из них и последовательно применять его на протяжении всего проекта, то есть писать все сценарии на выбранном языке. Отказ от этого правила (смешивание файлов сценариев на разных языках) обычно вызывает головную боль и излишнюю путаницу. Однако определившись с языком, выбрав, например, C#, вы понимаете, что он не предоставляет всего необходимого для создания игр. Язык программирования C# сам по себе не может ни загружать, ни разбирать файлы XML для поддержки сохранения игры, не может создавать объекты окна и компоненты пользовательского интерфейса, выполнять расширенный поиск и запросы к сложным наборам данных и коллекций. Для этих дополнительных функций и многих других приходится обращаться к внешним библиотекам. Некоторые библиотеки можно приобрести непосредственно в магазине Unity, но они, как правило, используются для конкретных целей. Однако, Unity распространяется вместе с Mono – бесплатным, кросс-платформенным фреймворком с открытым исходным кодом на основе библиотеки Microsoft .NET Framework, и предоставляет множество классов, доступных в данной библиотеке. В .NET Framework имеются классы для работы со строками и файлами, поиска и сортировки данных, отслеживания динамических списков, анализа файлов XML и многие другие. Это значит, что в виде Mono вам доступен обширный набор инструментов, позволяющих эффективно и продуктивно управлять данными в приложении. В этой главе рассматриваются некоторые из возможностей Mono, доступных в приложениях Unity, такие как поддержка списков и стеков, **язык интегрированных запросов (Language Integrated Query**, **LINQ)**, регулярные выражения, счетчики и т. д.

На рис. 6.1 показана домашняя страница проекта Mono Framework.

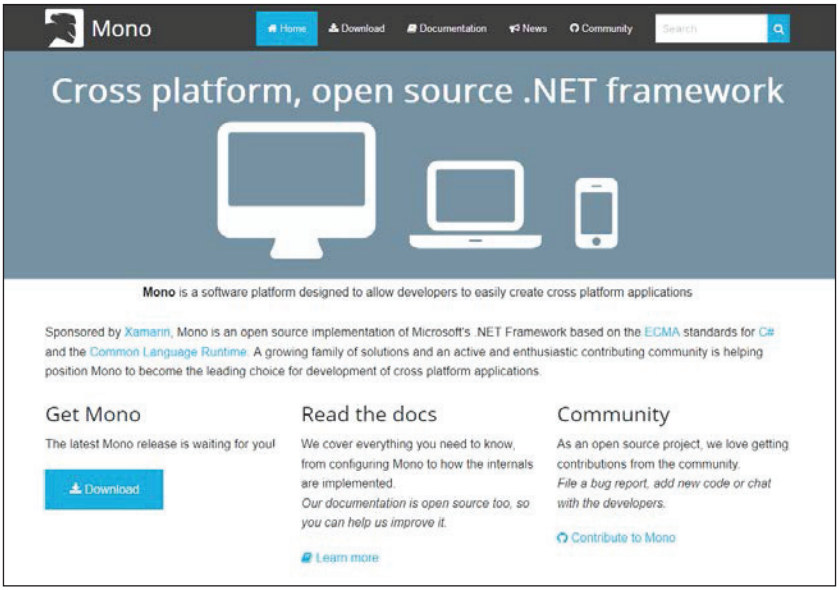


Рис. 6.1. Фреймворк Mono
поставляется вместе с игровым движком Unity

Списки и коллекции

Пожалуй, самой распространенной задачей при программировании игр является хранение списков данных. Природа этих данных может быть разной: счет игры, статистика игроков, статистика врагов, списки предметов, оружия, бонусов, уровней и многое другое. Там, где это только возможно, следует выбирать для хранения данных статические массивы, из-за высокой скорости их обработки. Статические массивы были подробно рассмотрены в главе 1 «Основы C# в Unity». Проще говоря, статические массивы создаются заранее, и их максимальная емкость определяется с самого начала. В них можно добавлять и удалять элементы, но общий размер статических массивов нельзя изменить. Если их максимальная емкость полностью не используется, пространство будет потрачено впустую. Статические массивы являются отличным выбором для хранения постоянных списков данных, например всех уровней в игре, всего оружия, которое может быть собрано, всех бонусов, которые могут быть собраны, и т. д.

Однако не менее часто используются и динамические массивы, которые могут увеличиваться и уменьшаться, по мере добавления и удаления данных, например, при создании и уничтожении врагов, появления и исчезновения предметов или оружия, и т. д. Фреймворк Mono предлагает множество классов для поддержки списков данных. Основных таких классов три: `List`, `Stack` и `Dictionary`. Каждый из них полезен для определенных целей.

Класс List

Если необходим неупорядоченный, последовательный список элементов любого одного типа данных, то есть список, который растет и сжимается в соответствии с объемом хранимых данных, класс `List` является идеальным выбором. В список особенно удобно добавлять и удалять элементы и последовательно перебирать их элементы. Кроме того, объекты класса `List` могут изменяться в инспекторе объектов Unity. В следующем листинге 6.1 представлено содержимое файла `Using_List.cs`.

Листинг 6.1. Файл Using_List.cs

```
01 using UnityEngine;
02 using System.Collections;
03 using System.Collections.Generic;
04 //-----------------------------------------
05 // Пример класса для хранения данных о врагах
06 [System.Serializable]
07 public class Enemy
08 {
09   public int Health = 100;
10   public int Damage = 10;
11   public int Defense = 5;
12   public int Mana = 20;
13   public int ID = 0;
14 }
15 //-----------------------------------------
16 public class Using_List : MonoBehaviour
17 {
18   //-----------------------------------------
19   // Список активных врагов в сцене
20   public List<Enemy> Enemies = new List<Enemy>();
21   //-----------------------------------------
22   // Этот метод выполняет инициализацию
23   void Start ()
24   {
25     // Добавить 5 врагов в список
26     for(int i=0; i<5; i++)
27       Enemies.Add (new Enemy());
         // Метод Add вставляет элемент в конец списка
```

```
28
29      // Удалить врага в начале списка (с индексом 0)
30      Enemies.RemoveRange(0,1);
31
32      // Обойти элементы списка
33      foreach (Enemy E in Enemies)
34      {
35        // Вывести значение свойства ID врага
36        Debug.log (E.ID);
37      }
38    }
39 }
40 //----------------------------------------
```

Более подробную информацию об использовании класса `List` можно найти в пакете примеров для книги, в папке `Chapter06\Collections`. Документацию с описанием класса `List` можно найти в MSDN: http://msdn.microsoft.com/ru-ru/library/ 6sh2ey19%28v=vs.110%29.aspx.

Ниже приводится несколько комментариев к листингу 6.1:

- **Строка 03**: чтобы использовать класс `List`, необходимо подключить пространство имен `System.Collections.Generic`.
- **Строка 06**: если тип данных списка наследует класс `System.Serializable`, список будет доступен в инспекторе объектов.
- **Строка 20**: вы можете объявить и тут же инициализировать новый экземпляр списка в одном операторе при объявлении членов класса.
- **Строка 27**: новые объекты добавляются в конец списка с помощью метода `Add`.
- **Строка 30**: элементы могут удаляться с помощью нескольких методов. Метод `RemoveRange` удаляет несколько последовательных элементов из списка. Другими методами удаления являются: `Remove`, `RemoveAll` и `RemoveAt`.
- **Строка 33**: есть возможность обойти все элементы списка с помощью цикла `foreach`.
- **Строки 27–33**: обычно при обходе элементов в цикле операции добавления и удаления элементов не выполняются.

На рис. 6.2 показано, как выглядит экземпляр класса `List` в инспекторе объектов.

Класс `List` поддерживает несколько методов удаления элементов, по одному или группами, предназначенных для использования вне итераций по спискам. Тем не менее, иногда бывает удобнее и проще удалять элементы во время их обхода в цикле, например когда нужно удалить элемент после его обработки. Классический случай – удаление всех объектов ссылочного типа в сцене, например врагов, а также

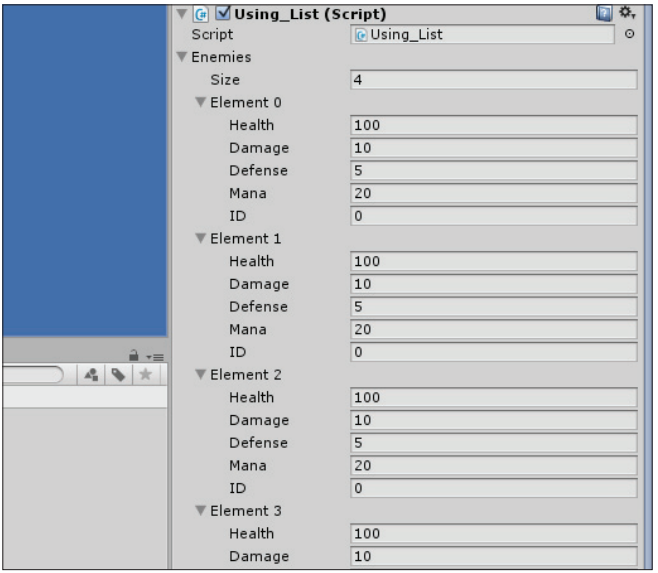


Рис. 6.2. Экземпляр класса List в инспекторе объектов

связанных с ними элементов списка, чтобы избежать присутствия пустых ссылок. Удаление элементов в цикле может вызвать проблемы, потому что при удалении итератор перестанет отслеживать правильное положение элемента в списке, и общее количество итераций цикла не будет соответствовать длине списка. Чтобы совместить обход элементов списка в цикле и удаление элементов, перебор элементов списка следует проводить в обратном порядке, от конца к началу, как показано в листинге 6.2.

Листинг 6.2. Удаление элементов списка в цикле

```
// Удалит все элементы списка в цикле
void RemoveAllItems()
{
  // Обойти список в обратном порядке
  for(int i = Enemies.Count-1; i>=0; i--)
  {
    // Вызвать функцию элемента перед удалением
    Enemies[i].MyFunc();

    // Удалить элемент из списка
    Enemies.RemoveAt(i);
  }
}
```

Класс Dictionary

Класс List является, пожалуй, одним из самых полезных классов в фреймворке Mono для хранения данных в памяти. Однако не будем забывать и о классе Dictionary (аналоге класса std::map в C ++). Этот класс особенно полезен, когда нужно нечто большее, чем простой список элементов. Если нужна возможность быстро находить элементы по ключу, лучше всего использовать класс Dictionary. Для каждого элемента списка следует указать ключ, или идентификатор, однозначно определяющий элемент. Класс Dictionary позволяет получить мгновенный доступ к элементу, основываясь исключительно на знании его ключа. Это делает работу с классом Dictionary похожей на работу с обычным словарем, например, если нужно найти толкование некоторых слов в большом словаре или базе данных слов, само слово будет ключом, а толкование – значением.

Конечно, можно реализовать аналог класса Dictionary с помощью нескольких объектов класса List. Но класс Dictionary работает очень быстро, почти молниеносно. Вы можете хранить большие объемы данных в словаре без ощутимых потерь производительности. Это делает его весьма полезным, когда нужен быстрый поиск данных по ключевым значениям. Применение класса Dictionary показано в листинге 6.3.

Листинг 6.3. Применение класса Dictionary

```
01 using UnityEngine;
02 using System.Collections;
03 using System.Collections.Generic;
04
05 public class Using_Dictionary : MonoBehaviour
06 {
07   // База данных слов. Пары ключ/значение: <Word, Score>
08   public Dictionary<string, int> WordDatabase = new
         Dictionary<string, int>();
09
10   // Этот метод выполняет инициализацию
11   void Start ()
12   {
13     // Определить несколько слов
14     string[] Words = new string[5];
15     Words[0]="hello";
16     Words[1]="today";
17     Words[2]="car";
18     Words[3]="vehicle";
19     Words[4]="computers";
20
```

```
21      // добавить в словарь с числом очков
22      foreach(string Word in Words)
23        WordDatabase.Add(Word, Word.Length);
24
25      // Выбрать слово из списка с использованием ключа
26      // Используется синтаксис массивов!
27      Debug.log ("Score is: " + WordDatabase["computers"].ToString());
28    }
29 }
```

Ниже приводится несколько комментариев к листингу 6.3:

- **Строка 03**: как и для работы с классом `List`, необходимо подключить пространство имен `System.Collections.Generic`.
- **Строка 08**: словарь объявляется и создается в одной строке; в отличие от класса `List`, словари не отображаются в инспекторе объектов Unity.
- **Строки 13 – 23**: класс `Dictionary` заполняется с помощью метода `Add`.
- **Строка 27**: обращаться к элементам класса `Dictionary` можно так же, как к элементам массива, используя значение ключа вместо индекса массива.

Более подробную информацию об использовании словарей можно найти в главе 4, «Событийное программирование», в разделе об управлении событиями с помощью класса `EventManager`.

Класс Stack

Если вы разрабатываете карточную игру, где игроки снимают верхнюю карту с колоды, если нужно отменить сделанный ход, если вы программируете поиск пути, создаете сложную систему произнесения заклинаний или игру-головоломку Ханойская башня (https://ru.wikipedia.org/wiki/Ханойская_башня), очень возможно, что вам понадобится стек. Стек – это особый вид списка, основанный на модели «**Последним вошел – первым вышел**» (**Last in, first out**, **LIFO**). Концепция его базируется на стопке листов бумаги. Вы можете добавлять элементы в список, и они будут укладываться один на другой в вертикальную стопку, в которой последний элемент всегда будет сверху. Затем вы можете снимать элементы с вершины стека (удалять их из списка) по одному. Порядок, в котором вы получаете элементы, всегда обратен порядку, в котором они были добавлены.

Вот почему класс `Stack` особенно полезен для отката или отмотки назад. Рассмотрим листинг 6.4 с примером использования класса `Stack`.

Листинг 6.4. Пример использования класса Stack

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
//----------------------------------------
[System.Serializable]
public class PlayingCard
{
  public string Name;
    public int Attack;
    public int Defense;
  }

//----------------------------------------
public class Using_Stack : MonoBehaviour
{
  //----------------------------------------
  // Стопка карт
  public Stack<PlayingCard> CardStack = new Stack<PlayingCard>();

  //----------------------------------------
  // Этот метод выполняет инициализацию
  void Start ()
  {
    // Создать массив карт
    PlayingCard[] Cards = new PlayingCard[5];

    // Создать несколько карт
    for(int i=0; i<5; i++)
    {
      Cards[i] = new PlayingCard();
      Cards[i].Name = "Card_0" + i.ToString();
      Cards[i].Attack = Cards[i].Defense = i * 3;

      // Втолкнуть карту в стек
      CardStack.Push(Cards[i]);
    }

    // Удалить карты из стека
    while(CardStack.Count > 0)
    {
      PlayingCard PickedCard = CardStack.Pop();

      // Вывеси имя выбранной карты
      Debug.log (PickedCard.Name);
    }
  }
  //----------------------------------------
}
//----------------------------------------
```

Интерфейсы IEnumerable и IEnumerator

При работе с коллекциями данных, будь то списки, словари, стеки или что-то другое, часто бывает нужно обойти все элементы коллекции или лишь некоторые из них, опираясь на определенные критерии. В некоторых случаях нужно перебрать в цикле все элементы последовательности или некоторые из ее элементов. Чаще бывает нужно обойти элементы в прямой последовательности, но иногда желательно сделать это в обратном порядке. Вы можете сделать это с помощью стандартного оператора цикла `for`. Однако, при этом возникнут некоторые неудобства, от которых помогут избавиться интерфейсы `IEnumerable` и `IEnumerator`. Давайте посмотрим, что за неприятности могут возникать. Рассмотрим цикл в листинге 6.5.

Листинг 6.5. Обход элементов коллекции

```
// Создать переменную с общим счетом
int Total = 0;

// Обойти список объектов слева направо
for(int i=0; i<MyList.Count; i++)
{
  // Выбрать число из списка
  int MyNumber = MyList[i];

  // Увеличить общий счет
  Total += MyNumber;
}
```

С циклом `for` связаны три основные неприятности. Начнем с первых двух. Во-первых, синтаксис не особенно подходит циклу, во всем теле цикла `for` мы вынуждены использовать числовой итератор (`i`) для доступа к каждому элементу массива. Во-вторых, сам итератор не гарантирует защиты от выхода за границы последовательности. Он может увеличиваться, выходя за верхний предел массива, и уменьшаться, выходя за нижний предел, вызывая ошибку выхода за границы.

Эти проблемы можно решить до определенной степени применением цикла `foreach`, который не выходит за границы и использует простой синтаксис, как показано в листинге 6.6.

Листинг 6.6. Оператор цикла foreach

```
// Создать переменную с общим счетом
int Total = 0;

// Обойти список объектов слева направо
```

```
foreach(int Number in MyList)
{
  // Увеличить общий счет
  Total += Number;
}
```

Цикл `foreach` проще и лучше читается, но в нем содержится нечто большее, чем это кажется на первый взгляд. Цикл `foreach` работает только с классами, реализующими интерфейс `IEnumerable`, которые должны возвращать экземпляр интерфейса `IEnumerator`. Итак, объект, пригодный для работы в цикле `foreach`, должен реализовать эти два интерфейса. Возникает вопрос, для чего такая сложность при реализации простого обхода элементов. Дело в том, что интерфейсы `IEnumerable` и `IEnumerator` не только решают две первые проблемы – упрощают синтаксис и гарантируют отсутствие выхода за границы, – но также решают третью задачу. В частности, они позволяют обойти в цикле объекты, которые не являются массивами. То есть интерфейсы `IEnumerable` и `IEnumerator` позволяют перебирать объекты разных типов, как если бы они были массивами. Это, несомненно, мощные инструменты. Давайте рассмотрим их в действии на практическом примере.

Перебор врагов с помощью интерфейса IEnumerator

Возьмем, к примеру, ролевую игру на средневековую тему, где игровой мир населен злыми волшебниками (класс `Wizard`). Пусть эти волшебники появляются в случайных местах и в случайные моменты времени, они могут нанести вред игроку своими заклинаниями и прочими злодеяниями. Из-за случайности их появления мы не можем знать заранее, сколько волшебников находится в сцене в текущий момент и где они находятся. Однако, есть важные причины отыскать всех волшебников, например, чтобы деактивировать их, скрыть, заставить замереть, уничтожить или, может быть, пересчитать их, чтобы предотвратить перенасыщение сцены злыми волшебниками. Итак, независимо от случайного характера появления волшебников в сцене, есть веские причины иметь доступ ко всем волшебникам.

Мы уже видели в главе 2, «Отладка», один способ получения полного списка всех волшебников, как показано в листинге 6.7.

Листинг 6.7. Получение списка объектов определенного типа

```
// Получить список волшебников
Wizard[] WizardsInScene = Object.FindObjectsOfType<Wizard>();

// Обойти элементы списка
```

```
foreach (Wizard W in WizardsInScene)
{
  // Обратиться к каждому объету Wizard через W
}
```

Функция FindObjectsOfType имеет один существенный недостаток – она выполняется медленно и серьезно снижает производительность. Даже в документации к Unity (http://docs.unity3d.com/ScriptReference/Object.FindObjectsOfType.html) можно найти рекомендации не использовать ее в тех местах, где она будет часто вызываться.

Проект, демонстрирующий использование интерфейсов IEnumerator и IEnumerable, можно найти в пакете примеров к книге, в папке Chapter06\Enumerators.

Однако, тот же результат можно получить с помощью интерфейсов IEnumerable и IEnumerator, и это позволит избежать значительных потерь производительности. С помощью этих двух интерфейсов можно эффективно обойти всех волшебников в сцене, используя цикл foreach, как если бы они находились в массиве (см. листинг 6.8).

Листинг 6.8. Обход всех объектов с помощью интерфейсов IEnumerable и IEnumerator

```
01 using UnityEngine;
02 using System.Collections;
03 using System.Collections.Generic;
04 //-------------------------------------------------------
05 // Класс наследует IEnumerator
06 // Поддержка границ для безопасности итераций
07 public class WizardEnumerator : IEnumerator
08 {
09   // Текущий объект, на который указывает итератор
10   private Wizard CurrentObj = null;
11   //-------------------------------------------------------
12   // Переопределить метод перехода к следующему элементу
13   public bool MoveNext()
14   {
15     // Получить следующего волшебника
16     CurrentObj = (CurrentObj==null) ? Wizard.FirstCreated :
                                         CurrentObj.NextWizard;
17
18     // Вернуть следующего волшебника
19     return (CurrentObj != null);
20   }
21   //-------------------------------------------------------
22   // Переустановить итератор на первого волшебника
23   public void Reset()
24   {
```

```
25      CurrentObj = null;
26    }
27    //-----------------------------------------------------
28    // Свойство C# для доступа к текущему волшебнику
29    public object Current
30    {
31      get{return CurrentObj;}
32    }
33    //-----------------------------------------------------
34  }
35  //-------------------------------------------------------
36  // Пример класса, определяющего объект волшебника
37  // Наследует IEnumerable, что позволяет обходить объекты в цикле foreach
38  [System.Serializable]
39  public class Wizard : MonoBehaviour, IEnumerable
40  {
41    //-----------------------------------------------------
42    // Ссылка на последнего созданного волшебника
43    public static Wizard LastCreated = null;
44
45    // Ссылка на первого созданного волшебника
46    public static Wizard FirstCreated = null;
47
48    // Ссылка на следующего волшебника в списке
49    public Wizard NextWizard = null;
50
51    // Ссылка на предыдущего волшебника в списке
52    public Wizard PrevWizard = null;
53
54    // Имя данного волшебника
55    public string WizardName = "";
56    //-----------------------------------------------------
57    // Конструктор
58    void Awake()
59    {
60      // Обновить ссылку на первого созданного волшебника?
61      if(FirstCreated==null)
62        FirstCreated = this;
63
64      // Обновить ссылку на последнего созданного волшебника?
65      if(Wizard.LastCreated != null)
66      {
67        Wizard.LastCreated.NextWizard = this;
68        PrevWizard = Wizard.LastCreated;
69      }
70
71      Wizard.LastCreated = this;
72    }
73    //-----------------------------------------------------
74    // Вызывается перед уничтожением объекта
```

```
75   void OnDestroy()
76   {
77     // Переустановить ссылки, если уничтожается объект в середине цепочки
78     if(PrevWizard!=null)
79       PrevWizard.NextWizard = NextWizard;
80
81     if(NextWizard!=null)
82       NextWizard.PrevWizard = PrevWizard;
83   }
84   //-------------------------------------------------
85   // Возвращает данный класс как итератор
86   public IEnumerator GetEnumerator()
87   {
88     return new WizardEnumerator();
89   }
90   //-------------------------------------------------
91 }
92 //---------------------------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 6.8:

- **Строки 07** и **39**: объявляются два класса. Первый класс, `WizardEnumerator`, реализует интерфейс `IEnumerator`, и второй класс, `Wizard`, реализует интерфейс `IEnumerable`. Экземпляр класса `WizardEnumerator` просто перебирает коллекцию волшебников, запоминая текущего волшебника в процессе итераций. Для перебора всех волшебников в сцене он использует члены класса `Wizard`, как мы это увидим в следующих разделах.
- **Строки 13**, **23** и **29**: класс `WizardEnumerator` реализует методы и свойства `IEnumerator`, в частности методы `MoveNext` (выполняет переход к следующему волшебнику), `Reset` (переустанавливает итератор в начало, на первого волшебника) и `Current` (возвращает текущего волшебника).
- **Строка 39**: класс `Wizard` инкапсулирует свойства и методы волшебника и наследует два класса: `MonoBehaviour` и `IEnumerable`. То есть, все свойства обоих классов собраны вместе в этом производном классе. Он обеспечивает поддержку нескольких переменных, позволяющих в любой момент выполнить обход всех волшебников в сцене. Во-первых, класс `Wizard` содержит статические свойства `FirstCreated` и `LastCreated` (являются глобальными для всех экземпляров класса `Wizard`). Эти переменные устанавливаются при создании объектов (обратите внимание на строку 58 в функции `Awake`).
- **Строки 48** и **52**: класс `Wizard` также поддерживает переменные экземпляра `NextWizard` и `PrevWizard`. Они реализуют двусвяз-

ный список, то есть каждый экземпляр класса Wizard указывает на предыдущий и следующий экземпляры, что позволит последовательно обходить всех волшебников. Первый волшебник будет хранить в переменной PrevWizard значение null, а последний – то же значение null в переменной NextWizard. Эти переменные и делают возможным перебор мастеров без создания массива мастеров.

- **Строка 86**: метод GetEnumerator возвращает экземпляр класса Enumerator. Этого требует интерфейс IEnumerable для обхода в цикле foreach всех волшебников.

Вместе классы Wizard и WizardEnumerator обеспечивают простой и быстрый способ обхода объектов Wizard в цикле, без создания массива. Чтобы убедиться в этом, рассмотрим пример в листинге 6.9, перечисляющий всех волшебников в сцене.

Листинг 6.9. Перечисление всех волшебников в сцене

```
void Update()
{
  // Нажмите пробел, чтобы запустить перечисление волшебников
  if(Input.GetKeyDown(KeyCode.Space))
  {
    // Получить первого волшебника
    Wizard WizardCollection = Wizard.FirstCreated;

    // Если есть хотя бы один волшебник, начать обход
    if(Wizard.FirstCreated != null)
    {
      // Цикл по всем волшебникам
      foreach(Wizard W in WizardCollection)
        Debug.log (W.WizardName);
    }
  }
}
```

Также есть возможность перечислить всех волшебников без использования цикла foreach, а напрямую обращаясь к объекту Enumerator, как показано в листинге 6.10.

Листинг 6.10. Перечисление всех волшебников без использования цикла foreach

```
void Update()
{
  // Нажмите пробел, чтобы запустить перечисление волшебников
  if(Input.GetKeyDown(KeyCode.Space))
  {
```

```
    // Получить итератор (экземпляр типа Enumerator)
    IEnumerator WE = Wizard.FirstCreated.GetEnumerator();
    while(WE.MoveNext())
    {
      Debug.log(((Wizard)WE.Current).WizardName);
    }
  }
}
```

Строки и регулярные выражения

Работа с текстовыми данными важна, и важна по многим причинам. Если нужно вывести субтитры, показать в игре текст, а также реализовать поддержку нескольких языков, вам придется работать с текстом, в частности с текстовыми ресурсами. В Unity к тестовым ресурсам относятся любые текстовые файлы, включенные в проект Unity, и каждый тестовый ресурс интерпретируется как одна длинная строка, даже когда речь идет о нескольких строках (разделителем строк служит управляющий символ `\n`). Как только в коде появляются такие строки, тут же обычно возникает необходимость их обработки разными способами. Давайте рассмотрим некоторые главные операции со строками.

Null, пустые строки и пробелы

При обработке строк не всегда можно положиться на их корректность, иногда строки неправильно сформированы и дальнейшая работа с ними не имеет смысла. По этой причине часто бывает нужно проверить их перед обработкой. Обычно сначала проверяется – не передано ли в качестве строки значение `null`, а затем (если значение не `null`) определяется длина строки, потому что если ее длина равна `0`, значит строка пустая и, следовательно, неправильная.

Также часто бывает желательно исключить строки, состоящие только из пробелов, потому что обычно такие строки не содержат ничего, что требовало бы обработки. Можно проверять строки на соответствие (точнее, несоответствие) каждому из этих условий в отдельности, однако класс `string` в .NET предлагает возможность выполнить весь комплекс проверок сразу – с помощью метода `IsNullOrWhiteSpace`. Проблема, однако в том, что данный метод появился только в .NET 4.5, и Mono не поддерживает его. Это означает, что этот метод придется реализовать вручную, как показано в листинге 6.11.

Листинг 6.11. Реализация комплексной проверки строк

```
using UnityEngine;
using System.Collections;
//---------------------------------------------------------------
// Расширенный класс, добавляющий проверку на null и пробелы
public static class StringExtensions {
  public static bool IsNullOrWhitespace(this string s){
    return s == null || s.Trim().Length == 0;
  }
}
//---------------------------------------------------------------
public class StringOps : MonoBehaviour
{
  // Проверить строку
  public bool IsValid(string MyString)
  {
    // Проверить на null и пробельные символы
    if(MyString.IsNullOrWhitespace()) return false;
      // Выполнить другие проверки
      return true;
  }
}
//---------------------------------------------------------------
```

Сравнение строк

Часто бывает нужно сравнить две строки, обычно на равенство, чтобы определить, являются ли две строки идентичными. Сделать это можно с помощью оператора ==, например так: `string1 == string2`, но для большей скорости лучше использовать метод `theString.Equals`. Этот метод имеет несколько версий и все они выполняются с разной скоростью. В общем случае предпочтительнее выбирать версию с параметром, имеющим тип `StringComparison`. Если сравниваемые типы указаны явно, операция сравнения будет работать быстрее, как показано в листинге 6.12:

Листинг 6.12. Сравнение строк

```
// Сравнение строк
public bool IsSame(string Str1, string Str2)
{
  // Без учета регистра символов
  return string.Equals(Str1, Str2,
      System.StringComparison.CurrentCultureIgnoreCase);
}
```

Более подробную информацию о методе `String.Compare` можно найти в MSDN по адресу http://msdn.microsoft.com/ru-ru/library/system.string.compare%28v=vs.110%29.aspx.

Другой способ быстрого сравнения двух строк на равенство заключается в сравнении хэш-кодов, для этого строки преобразуются в уникальные целые числа, а затем выполняется сравнение этих целых чисел, а не самих строк, как показано в листинге 6.13.

Листинг 6.13. Сравнение строк по их хэш-кодам

```
// Сравнение строк по их хэш-кодам
public bool StringHashCompare(string Str1, string Str2)
{
  int Hash1 = Animator.StringToHash(Str1);
  int Hash2 = Animator.StringToHash(Str2);
  return Hash1 == Hash2;
}
```

Для получения хэш-кодов строк можно также использовать функцию `String.GetHashCode` из библиотеки Mono. За более подробной информацией обращайтесь по адресу http://msdn.microsoft.com/ru-ru/library/system.string.gethashcode%28v=vs.110%29.aspx.

Однако иногда нужно выяснить не равенство строк, а определить лексикографический порядок строк, то есть какая из строк будет располагаться впереди в алфавитном порядке. Реализовать это можно с помощью функции `String.Compare`. Однако, опять же, старайтесь использовать версию с параметром типа `StringComparison`, как показано в листинге 6.14. В этой версии число `-1` возвращается, если `Str1` располагается перед `Str2`, `1` – если `Str2` располагается перед `Str1`, и `0` – если две строки равны.

Листинг 6.14. Определение лексикографического порядка следования строк

```
// Определяет порядок сортировки
public int StringOrder (string Str1, string Str2)
{
  // Без учета регистра символов
  return string.Compare(Str1, Str2,
      System.StringComparison.CurrentCultureIgnoreCase);
}
```

Хотя функция `string.Compare` и возвращает `0`, сообщая о равенстве строк, не используйте ее с целью проверки равенства. Для этого лучше использовать функцию `string.Equals` или хэш-коды, так как они работают гораздо быстрее, чем `string.Compare`.

Форматирование строк

При создании элементов пользовательского интерфейса, таких как табло с результатами, список имен игроков, отображение денежных расчетов или индикаторы ресурсов, приходится выводить не только постоянный текст, но и вставлять в него числовые значения, например соединив слово «Score:» со строкой, содержащей фактический результат, который будут меняться со временем, в зависимости от действий игрока. Для этого можно использовать, например, метод `String.Format`, как показано в листинге 6.15.

Листинг 6.15. Конструирование строки из трех чисел

```
// Конструирование строки из трех чисел
public void BuildString(int Num1, int Num2, float Num3)
{
  string Output = string.Format(
      "Number 1 is: {0}, Number 2 is: {1}, Number 3 is: {2}",
      Num1, Num2, Num3);
  Debug.log (Output.ToString("n2"));
}
```

Цикл по символам строке

Мы уже знакомы с интерфейсами `IEnumerable` и `IEnumerator`. К счастью, эти интерфейсы поддерживаются строками и могут использоваться для обхода символов в них. Для этого можно использовать сам интерфейс `IEnumerator` или цикл `foreach`. Рассмотрим оба способа, как показано в листингах 6.16 и 6.17.

Листинг 6.16. Обход символов в цикле foreach

```
// Обход символов в цикле foreach
public void LoopLettersForEach(string Str)
{
  // Для каждого символа
  foreach(char C in Str)
  {
    // Вывести символ в консоль
    Debug.log (C);
  }
}
```

Листинг 6.17. Обход символов с помощью итератора

```
// Обход символов с помощью итератора
public void LoopLettersEnumerator(string Str)
{
  // Получить перечислитель
```

```
  IEnumerator StrEnum = Str.GetEnumerator();

  // Перейти к следующему символу
  while(StrEnum.MoveNext())
  {
    Debug.log ((char)StrEnum.Current);
  }
}
```

Создание строк

Чтобы код получился более читаемым, чистым, надежным и вообще полнее соответствовал стилистике .NET, применяйте следующее правило при создании строк. Избегайте инициализации строковых переменных таким способом: `string MyString = "";`. Вместо этого применяйте способ объявления строк с одновременным присваиванием им начальных значений с помощью функции `String.Empty`:

```
string MyString = string.Empty;
```

Поиск в строках

Если вы имеете дело с несколькими строками текста, прочитанными из файла, может потребоваться найти первое вхождение строки меньшего размера в строке большего размера, например, определенное слово в строке. Реализовать это можно с помощью метода `String.indexOf`. Если совпадение найдено, функция вернет положительное целое число, указывающее позицию первого символа найденного слова в строке, как смещение от начала строки до первой буквы слова. Если совпадений не найдено, функция вернет `-1`, как показано в листинге 6.18.

Листинг 6.18. Поиск слова в строке

```
// Ищет слово в строе и возвращает индекс
// первого найденного вхождения
public int SearchString(string LargerStr, string SearchStr)
{
  // Без учета регистра символов
  return LargerStr.IndexOf(SearchStr,
      System.StringComparison.CurrentCultureIgnoreCase);
}
```

Регулярные выражения

Иногда может потребоваться выполнить более сложный поиск в очень больших строках, например, найти все слова в строке, начинающиеся с определенной буквы, начинающиеся с `a` и заканчивающиеся на `t` и т. д. В этих случаях нужно, чтобы результаты поиска, если таковые

имеются, сохранялись в массиве. Реализовать это можно с помощью регулярных выражений. Регулярные выражения позволяют определить искомую строку, используя специальный синтаксис, задав шаблон поиска. Например, строка `[dw]ay` означает: «найти все слова, которые заканчиваются на `ay` и начинаются с `d` или `w`». То есть найти все вхождения слов `day` или `way`. Применить регулярное выражение к большей строке можно с помощью класса `Regex`. Фреймворк .NET предоставляет доступ к регулярным выражениям с помощью пространства имен `RegularExpressions`, как это показано в листинге 6.19.

Листинг 6.19. Регулярные выражения

```
01 //-----------------------------------------------------------
02 using UnityEngine;
03 using System.Collections;
04 // Подключить пространство имен с регулярными выражениями
05 using System.Text.RegularExpressions;
06 //-----------------------------------------------------------
07 public class RGX : MonoBehaviour
08 {
09   // Шаблон поиска
10   string search = "[dw]ay";
11
12   // Большая строка для поиска
13   string txt = "hello, today is a good day to do things my way";
14
15   // Этот метод используется для инициализации
16   void Start ()
17   {
18     // Выполнить поиск и вернуть первый результат в m
19     Match m = Regex.Match(txt, search);
20
21     // Пока обнаруживаются вхождения, продолжать поиск
22     while(m.Success)
23     {
24       // Вывести очередной результат в консоль
25       Debug.log (m.Value);
26
27       // Получить следующий результат
28       m = m.NextMatch();
29     }
30   }
31 }
32 //-----------------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 6.19:

- **Строка 05**: пространство имен `RegularExpressions` обязательно должно подключаться во всех исходных файлах, использующих поиск с помощью регулярных выражений.

- **Стоки 09** и **13**: строка `search` определяет регулярное выражение. Строка `txt` определяет большую строку, в которой нужно найти строку, соответствующую регулярному выражению. Строка `search` определяет поиск всех вхождений слов `day` и `way`.
- **Строка 19**: метод `Regex.Match` вызывается для поиска регулярного выражения `search` в строке `txt`. Результаты поиска сохраняются в локальной переменной `m`. С помощью этой переменной можно просмотреть все результаты в цикле.
- **Строка 25**: результаты в `m` будут включать три совпадения (не два) для строки `txt`: подстроку *day*, являющуюся частью слова to*day*, а также отдельные слова *day* и *way*.

Более подробную информацию о регулярных выражениях можно найти по адресу https://ru.wikipedia.org/wiki/Регулярные_выражения.

Произвольное количество аргументов

Хотя мы не вдавались в технические подробности .NET или Mono, наше исследование обеих этих библиотек коснулось нескольких функций, которые принимают, казалось бы, бесконечную цепь аргументов, например функция `String.Format`. Функции `String.Format` можно передать столько аргументов, сколько понадобится для включения в отформатированную строку. В этом разделе я хочу сделать небольшое отступление, чтобы показать, как писать функции, которые принимают и обрабатывают неограниченное число аргументов; они создаются очень легко. Взгляните на следующую функцию в листинге 6.20, которая предназначена для суммирования массивов целых чисел неограниченной длины:

Листинг6.20

```
01 public int Sum(params int[] Numbers)
02 {
03   int Answer = 0;
04
05   for(int i=0; i<Numbers.Length; i++)
06     Answer += Numbers[i];
07
08   return Answer;
09 }
```

Ниже приводится несколько комментариев к листингу 6.20:

- **Строка 01**: чтобы функция могла принимать неограниченное число аргументов, нужно добавить ключевое слово `params` и объявить аргумент-массив.

- **Строка 05**: аргумент `params` можно обрабатывать как обычный массив.

Язык интегрированных запросов

Очевидно, что игры работают с большим количеством данных, не только со строками, но также с объектами, базами данных, таблицами, документами и многими другими видами данных, их слишком много, чтобы перечислить их все. Однако, несмотря на обширность и разнообразие данных, всегда есть общая потребность фильтровать данные и просматривать ограниченные их подмножества, имеющие отношение к текущему моменту. Например, если имеется полный массив (или список) всех волшебников в сцене, можно было бы ограничить его, оставив только волшебников, чье здоровье составляет менее 50 процентов и чьи оборонные показатели меньше 5 пунктов. Это может понадобиться для организации массового бегства волшебников на поиски зелья, восстанавливающего здоровье, прежде чем они возобновят атаки на игрока. Давайте теперь рассмотрим реализацию этого сценария с помощью технологии языка интегрированных запросов Linq.

Законченный проект примера использования языка интегрированных запросов можно найти в пакете с примерами для книги, в папке `Chapter06\Linq\`.

Для начала создадим упрощенный пример класса злого волшебника, как показано в листинге 6.21. Этот класс включает переменные `Health` и `Defense`, имеющие решающее значение для реализации нашей логики поведения.

Листинг 6.21. Класс злого волшебника

```
//-----------------------------------------
using UnityEngine;
using System.Collections;
//-----------------------------------------
public class Enemy : MonoBehaviour
{
  public int Health = 100;
  public int Mana = 20;
  public int Attack = 5;
  public int Defense = 10;
}
//-----------------------------------------
```

Теперь, получив коллекцию всех объектов волшебников в сцене, ее можно отфильтровать в соответствии с нашими критериями, и получить меньший массив, как показано в листинге 6.22.

Этот код перебирает все элементы коллекции, пропускает их через условный оператор if, и, если они удовлетворяют условию, добавляет их в результирующий массив. Условием в нашем случае является уровень здоровья менее 50 процентов и защитные способности меньше 5 пунктов.

Листинг 6.22. Фильтрация объектов

```
// Получает отфильтрованный список врагов
public void FindEnemiesOldWay()
{
  // Получить список всех врагов
  Enemy[] Enemies = Object.FindObjectsOfType<Enemy>();

  // Отфильтрованный список
  List<Enemy> FilteredData = new List<Enemy>();

  // Обойти в цикле всех врагов и проверить
  foreach(Enemy E in Enemies)
  {
    if(E.Health <= 50 && E.Defense < 5)
    {
      // Соответствующий враг найден
      FilteredData.Add (E);
    }
  }

  // Теперь можно обработать отфильтрованные данные
  // Все элементы в FilteredData соответствуют критериям
  foreach(Enemy E in FilteredData)
  {
    // Обработать врага E
    Debug.log (E.name);
  }
}
```

Этот код выделяет из большего набора элементы, опираясь на заданные условия, и помещает их в меньший набор. Однако, язык интегрированных запросов Linq позволяет достичь тех же результатов меньшим количеством строк кода, и часто обеспечивает более высокую производительность. Язык интегрированных запросов Linq является специализированным языком высокого уровня для выполнения запросов к наборам данных, таким как массивы, объекты, базы данных и документы XML. Запросы Linq автоматически переводятся на язык, соответствующий используемому набору данных (например, SQL для баз данных). Целью является извлечение результатов в обычный массив.

В листинге 6.23 демонстрируется альтернативный подход с использованием Linq, решающий ту же задачу, что и функция в листинге 6.22.

Листинг 6.23. Фильтрация объектов с помощью Linq

```
01 using UnityEngine;
02 using System.Collections;
03 using System.Collections.Generic;
04 using System.Linq;
05 //-------------------------------------------------
06 public void FindEnemiesLinqWay()
07 {
08   // Получить список всех врагов
09   Enemy[] Enemies = Object.FindObjectsOfType<Enemy>();
10
11   // Выполнить поиск
12   Enemy[] FilteredData = (from EnemyChar in Enemies
13           where EnemyChar.Health <= 50 && EnemyChar.Defense < 5
14           select EnemyChar).ToArray();
15
16   // Теперь можно обработать отфильтрованные данные
17   // Все элементы в FilteredData соответствуют критериям
18   foreach(Enemy E in FilteredData)
19   {
20     // Обработать врага E
21     Debug.log (E.name);
22   }
23 }
24 //-------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 6.23:

- **Строки 03–04**: чтобы использовать Linq, необходимо подключить пространство имен `System.Collections.Linq`, а для использования объектов `List` следует подключить пространство имен `System.Collections.Generic`.
- **Строки 12–14**: основная часть кода, связанная с Linq, находится здесь. Она состоит из трех основных частей. Во-первых, указывается источник исходных данных, в данном случае – массив врагов `Enemies`. Во-вторых, определяются критерии поиска, а именно, `EnemyChar.Health <= 50 && EnemyChar.Defense < 5`. Затем, после определения критерия, выбирается объект для включения в множество результатов, в данном случае – `EnemyChar`. И, наконец, результаты преобразуются в массив с помощью функции `ToArray`.

Более подробную информацию о Linq можно найти в MSDN по адресу http://msdn.microsoft.com/ru-ru/library/bb397926.aspx.

Linq и регулярные выражения

Язык интегрированных запросов Linq, конечно, не обязательно должен работать в изоляции. Его можно объединить, например, с регулярными выражениями, для извлечения строк, соответствующих заданному шаблону, и преобразования результатов в массив. Такое соединение полезно при обработке файлов с данными, разделенными запятыми (CSV-файлов), – специально отформатированные текстовые файлы, значения в которых отделены друг от друга запятыми. Язык интегрированных запросов Linq и регулярные выражения можно использовать для быстрого и удобного переноса каждого из значений в отдельный элемент массива. Например, рассмотрим игру, где вновь созданным персонажам даются человеческие имена. Сами имена хранятся в формате CSV и делятся на две группы: мужские и женские. При создании мужских и женских персонажей им должны быть присвоены соответствующие имена, полученные из данных в формате CSV, как показано в листинге 6.24.

Листинг 6.24. Извлечение имен из списков

```
01 // Генерирует женские имена
02 // Регулярное выражение – шаблон поиска
03 // Извлечение всех имен, начинающихся с 'female:',
      но без включения префикса в результат
04 string search = @"(?<=\bfemale:)\w+\b";
05
06 // CSV-данные – имена персонажей
07 string CSVData =
   "male:john,male:tom,male:bob,female:betty,female:jessica,male:dirk ";
08
09 // Извлечь имена с префиксом 'female'.
10 string[] FemaleNames = (from Match m in Regex.Matches(CSVData,
search)
11                          select m.Groups[0].Value).ToArray();
12
13 // Вывести все найденные женские имена
14 foreach(string S in FemaleNames)
15   Debug.log (S);
16
17 // Выбрать случайное женское имя из коллекции
18 string RandomFemaleName =
     FemaleNames[Random.Range(0, FemaleNames.Length)];
```

Ниже приводится несколько комментариев к листингу 6.24:

- **Строка 04**: переменная `Search` определяет регулярное выражение для поиска по шаблону. В данном случае эта переменная

определяет все слова с префиксом `female:`. Однако сам префикс не будет включен в полученные строки.

- **Строка 07**: переменная `CSVData` определяет строку CSV с мужскими и женскими именами, отформатированную нужным образом. Эта строка, по сути, представляет собой базу данных или источник данных.
- **Строки 10–11**: здесь Linq используется в сочетании с регулярным выражением для извлечения из строки CSV всех женских имен без префиксов. Затем список преобразуется в массив строк `FemaleNames`.

Строки и символ @. Обратите внимание на строку 04 в листинге 6.24: перед строкой регулярного выражения стоит символ `@`. Этот префикс в C# определяется соглашением, позволяющим записать литеральную строку в исходный код. Такая строка может содержать управляющие последовательности (например, `\`), не нарушающие целостности строки и не вызывающие ошибок.

Работа с текстовыми ресурсами

Во всех рассмотренных нами примерах текст хранился непосредственно в строках, но в Unity существует возможность работы с текстовыми файлами. В частности, текст можно загружать из внешних источников. Сейчас я опишу, как это делается.

Текстовые ресурсы – статическая загрузка

Первый способ заключается в перетаскивании текстового файла в проект Unity, при этом текст будет импортирован в ресурс. Импортированный файл имеет тип `TextAssets`, как показано на рис. 6.3.

Вы можете получить доступ к файлу и его текстовым данным из любого файла сценария, объявив общедоступную переменную с типом данных `TextAsset`, как показано в листинге 6.25.

Листинг 6.25. Доступ к текстовому ресурсу

```
//-------------------------------------------------
using UnityEngine;
using System.Collections;
//-------------------------------------------------
public class TextFileAccess : MonoBehaviour
{
  // Ссылка на текстовый файл
  public TextAsset TextData = null;

  // Этот метод выполняет инициализацию
```

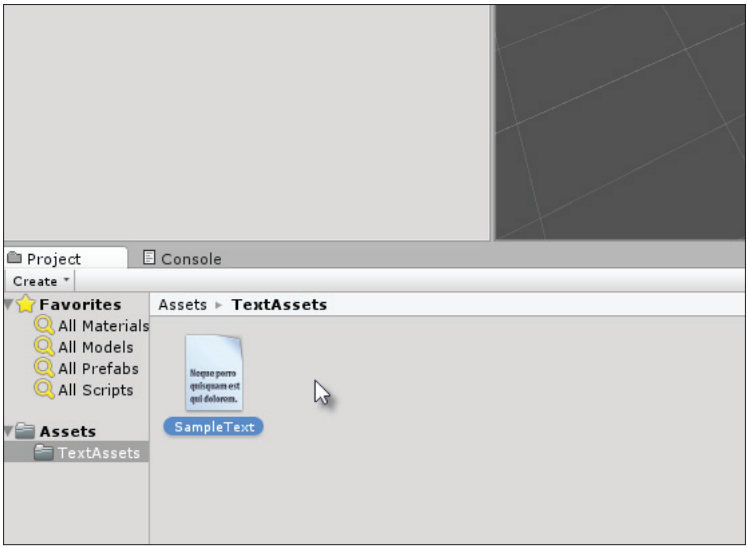


Рис. 6.3. Импорт текстовых файлов в Unity как ресурсов типа TextAssets

```
  void Start ()
  {
    // Вывести текст из файла
  }
}
//-------------------------------------------------
```

Этот пример подразумевает, что вы должны перетащить файл `TextAsset` в слот **Text Data** (Текстовые данные) сценария, в инспекторе объектов, как показано на рис. 6.4.

Текстовые ресурсы – загрузка из локальных файлов

Другой метод предназначен для загрузки внешних текстовых данных, то есть из файлов с локальных дисков. Текстовые данные, загружаемые таким способом, читаются динамически, из сценария, причем не обязательно при запуске сцены, а когда выполняется соответствующий код. Это значит, что для больших текстовых файлов со сложной обработкой замедление выполнения становится серьезным фактором. Поэтому предпочтительнее использовать статическую загрузку текстовых ресурсов. Я рекомендую выполнять динамическую загрузку и обработку всех ресурсов при запуске сцены, чтобы избежать замедления игры, как показано в листинге 6.26.

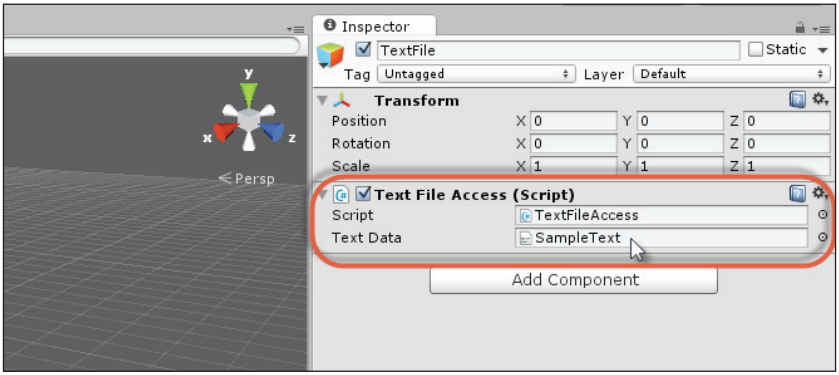


Рис. 6.4. Доступ к текстовому файлу из сценария

Листинг 6.26. Динамическая загрузка текстовых ресурсов

```
using UnityEngine;
using System.Collections;
using System.IO;

// Функция для загрузки текстовых данных из внешнего файла
public static string LoadTextFromFile(string Filename)
{
  // Если файл не найден, вернуть пустую строку
  if(!File.Exists(Filename)) return string.Empty;

  // Файл найден, загрузить текст из него
  return File.ReadAllText(Filename);
}
```

Функция в листинге 6.26 загружает весь текстовый файл в один строковый объект. Вы можете предпочесть построчную обработку текстового файла, особенно если это файл с настройками, где отдельные значения указаны в отдельных строках. Этот способ загрузки демонстрирует пример в листинге 6.27.

Листинг 6.27. Построчная загрузка текстового ресурса

```
// Функция для построчной загрузки текстовых данных в массив
public static string[] LoadTextAsLines(string Filename)
{
  // Если файл не найден, вернуть пустой массив
  if(!File.Exists(Filename)) return null;

  // Прочитать строки
  return File.ReadAllLines(Filename);
}
```

Текстовые ресурсы – загрузка из INI-файлов

В число поддерживаемых форматов текстовых файлов входят также INI-файлы. При создании игр в Unity этот формат используется нечасто, так как обычно разработчики применяют для хранения настроек класс `PlayerPreferences`. Тем не менее, INI-файлы обладают преимуществом хранения всех настроек приложения в одном месте и в одном формате, независимо от платформы. Это может стать веской причиной использовать INI-файлы. В листинге 6.28 приводится пример INI-файла с парами ключ/значение.

Листинг 6.28. Пример INI-файла

```
ApplicationName=MyTestApp
Date=1st Nov 2014
Author=Alan Thorn
Engine=Unity
Build=Production
```

Идеальной структурой для хранения данных из INI-файла является словарь, который также основан на парах ключ/значение. По этой причине лучше всего загружать INI-файл в словарь. Однако, ни Unity, ни Mono не предоставляют встроенной поддержки для этого, то есть, мы должны сами написать такую поддержку, как показано в листинге 6.29.

Листинг 6.29. Загрузка INI-файла в словарь

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

// Функция для чтения простых INI-файлов в словарь
public static Dictionary<string, string> ReadINIFile(string Filename)
{
  // Если файл не найден, вернуть null
  if(!File.Exists(Filename)) return null;

  // Создать новый словарь
  Dictionary<string, string> INIFile = new Dictionary<string, string>();

  // Создать новый объект чтения из потока
  using (StreamReader SR = new StreamReader(Filename))
  {
    // Переменная для хранения текущей строки
    string Line;

    // Продолжать читать допустимые строки
    while(!string.IsNullOrEmpty(Line = SR.ReadLine()))
    {
```

```
      // Удалить ведущие и конечные пробелы
      Line.Trim();

      // разбить строку на ключ и значение
      string[] Parts = Line.Split(new char[] {'='});

      // Добавить в словарь
      INIFile.Add(Parts[0].Trim(), Parts[1].Trim());
    }
  }

  // Вернуть словарь
  return INIFile;
}
```

Словарь, возвращаемый этой функцией, будет иметь ту же структуру, что и INI-файл. Соответственно, значения можно получать следующим образом: `Value = MyDictionary["Key"];`. Также есть возможность перебрать все ключи и значения из словаря в цикле `foreach`, как показано в листинге 6.30.

Листинг 6.30. Обход всех ключей и значений в словаре

```
// Создать словарь из INI-файла
Dictionary<string,string> DB = ReadINIFile(@"c:\myfile.ini");

// Обойти все записи в словаре
foreach(KeyValuePair<string, string> Entry in DB)
{
  // Цикл по каждой паре ключ/значение
  Debug.log("Key: " + Entry.Key + " Value: " + Entry.Value);
}
```

Текстовые ресурсы – загрузка из CSV-файлов

Ранее в этой главе мы видели, как обработать CSV-файл с мужскими и женскими именами. Давайте теперь посмотрим, как загрузить данные из CSV-файла на диске в массив строк, как показано в листинге 6.31.

Листинг 6.31. Загрузка CSV-файла в массив строк

```
// Функция читает содержимое файла CSV в массив строк
public static string[] LoadFromCSV(string Filename)
{
  // Если файл не найден, вернуть null
  if(!File.Exists(Filename)) return null;

  // Прочитать весь текст
  string AllText = File.ReadAllText(Filename);

  // Вернуть массив строк
  return AllText.Split(new char[] {','});
}
```

Текстовые ресурсы – загрузка из Интернета

Если вы разрабатываете многопользовательские игры и нужно получить доступ к данным игрока или к данным игры через Интернет, например чтобы сверить пароли с хэшем или обработать элементы вебстраницы, вам пригодится класс `WWW`, способный извлекать текстовые данные из Интернета, как показано в листинге 6.32.

Листинг 6.32. Использование класса WWW

```
// Извлекает текст из Интернета в строку
public IEnumerator GetTextFromURL(string URL)
{
  // Создать новый объект WWW
  WWW TXTSource = new WWW(URL);

  // Ждать загрузки данных
  yield return TXTSource;

  // Обработать полученный текст
  string ReturnedText = TXTSource.text;
}
```

Более подробную информацию о классе WWW можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/WWW.html.

Итоги

В этой главе рассматривался широкий спектр применений фреймворка Mono на практике. Для этого потребовалось три раздела. В первом мы исследовали общие структуры хранения данных, используемые в C#, такие как список, словарь и стек. Затем мы перешли к рассмотрению общих подходов к хранению и поиску данных, а также обработке строк. Мы также познакомились с применением регулярных выражений для поиска строк по шаблону и с языком интегрированных запросов Linq для фильтрации не только строк, но и коллекций объектов любых типов, доступных в Mono. И, наконец, мы рассмотрели различные методы импорта текстовых данных: из внутренних ресурсов проекта, из локальных файлов, а также из Интернета. В следующей главе мы переместимся в мир искусственного интеллекта и познакомимся с выбором маршрутов, конечными автоматами, зоной прямой видимости, системой принятия решений, операциями отслеживания лучей и многим другим.

Глава 7

Искусственный интеллект

Эта глава имеет конкретную практическую направленность. Здесь мы рассмотрим разработку проекта от начала и до конца – игры с лабиринтом, который населяют враждебные персонажи с искусственным интеллектом. Эти персонажи способны находить игрока, преследовать его, нападать и убегать, когда им понадобится восстановить свое здоровье, найдя и выпив зелье. На рис.7.1 показана сцена лабиринта.

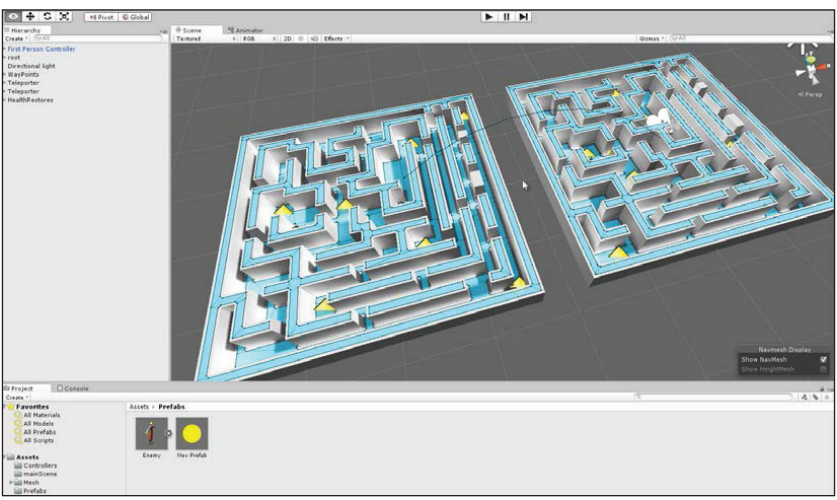


Рис. 7.1. Сцена лабиринта

Проект AI для этой книги можно найти в пакете примеров для этой главы, в папке `ai`.

При реализации проекта мы используем почти все концепции и идеи, которые рассмотрели в чистом виде выше, вместе с понятиями искусственного интеллекта, такими как **конечные автоматы**, навигационные меши, зоны прямой видимости и т. д. Чтобы получить максимальную пользу от чтения этой главы, я рекомендую создать новый пустой проект и пройти со мной все шаги разработки от начала до конца. Законченный проект для этой главы можно в пакете примеров для этой главы, в папке `ai`.

Искусственный интеллект в играх

Концепцию интеллекта можно рассматривать с разных позиций: психологической, научной, философской, духовной, социологической и т. д. Многие из них весьма существенны. Однако, в видеоиграх это, прежде всего, поведение, и именно в разумности поведения проявляется интеллект. Может быть поэтому слово «искусственный» является частью названия. Идея видеоигр заключается, прежде всего, в получении удовольствия и приобретении опыта. То есть, правдоподобность для любителей компьютерных игр означает соответствие событий, происходящих в игре, их реальному опыту. Всякий раз, когда не управляемые игроком персонажи, такие как злые волшебники, делают что-то «глупое» (безуспешно пытаются пройти сквозь твердые стены или бесцельно ходят взад и вперед, впав в ступор), игроки чувствуют, что что-то идет не так. Они видят неразумность персонажа, неадекватность его поведения конкретной ситуации, которая и не может быть объяснена происходящим в игре. «Неадекватное» или «глупое» поведение персонажа заставляет игрока признать, что он видит «глюк» игры, а это выводит его из игрового мира. Поэтому главная цель искусственного интеллекта в игре заключается в обеспечении способности персонажей адекватно реагировать на ситуации. В играх, где применяется искусственный интеллект врагов или противников, необходима настройка его сложности, искусственный интеллект должен быть не слишком простым и не слишком сложным. С этой точки зрения, создание искусственного интеллекта состоит не в построении математической модели человеческого разума и сознания, а только в создании поведения, соответствующего нашим ожиданиям. Речь пойдет о создании поведения персонажа, показанного на рис. 7.2, ожидаемого игроками. Следовательно, это будет искусственный интеллект в виде внешней формы без внутреннего содержания, но это философское наблюдение, и мы не будем его в дальнейшем касаться.

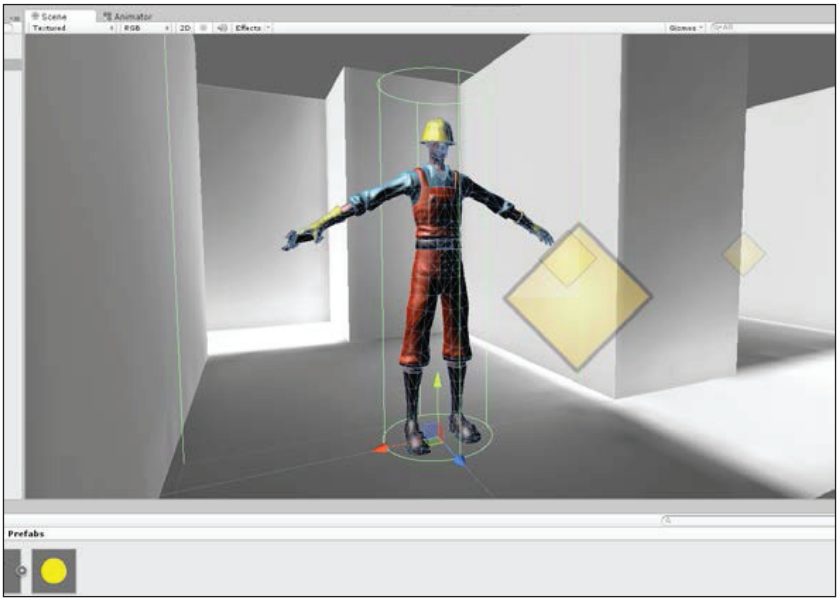


Рис. 7.2. Вражеский персонаж с искусственным интеллектом, который создан с помощью конструктора мешей Unity

В этой главе мы создадим игру, где игрок находится внутри лабиринта. Здесь игрок сможет атаковать врагов, а враги будут нападать на игрока. Меш врага опирается персонаж анимированного инженера, входящего в состав Unity, способный ходить, бегать и прыгать. Инженер будет исследовать окружающую среду, пытаясь найти игрока, и при его обнаружении настигать и нападать на него. Инженер сам может быть атакован и, получив повреждения, будет спасаться бегством и искать лечебное зелье для поправки своего здоровья. Итак, давайте начнем!

Начало проекта

Для начала создадим новый пустой проект Unity с новой сценой. В этом примере я импортировал несколько пакетов ресурсов Unity, выбрав в меню пункт **Asset** ⇒ **Import Package** (Ресурсы ⇒ Импортировать пакет). Вот эти пакеты: **Character Controller** (Контроллер персонажа), **Skyboxes** (Небо) и **Particles** (Частицы), как показано на рис. 7.3. Пакет **Character Controller** (Контроллер персонажа) содер-

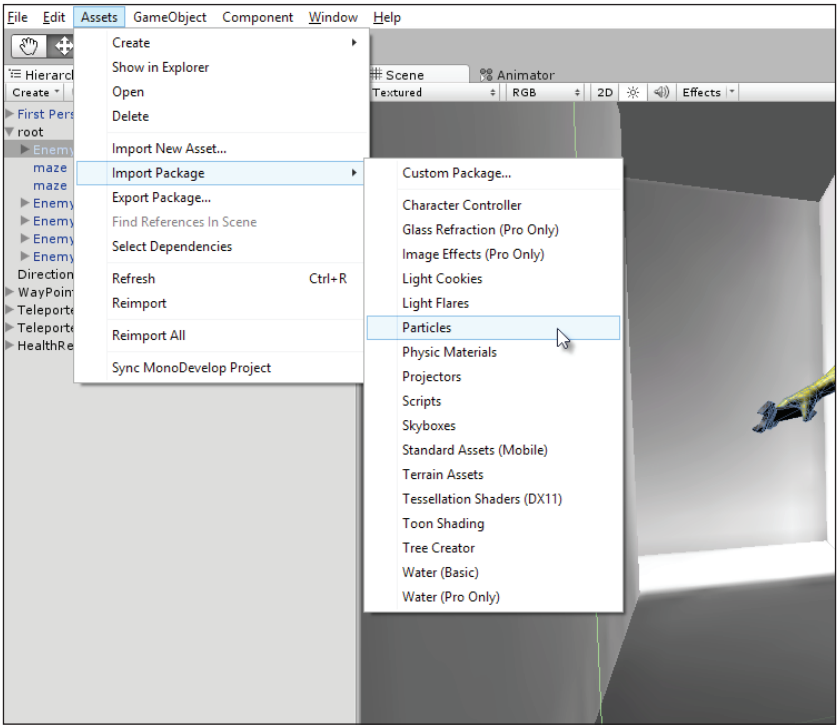


Рис. 7.3. Импорт ресурсов в проект

жит меш инженера и его анимации, а также заранее подготовленный контроллер игрока. Пакет **Skyboxes** (Небо) содержит привлекательные виды неба, а пакет **Particles** (Частицы) будет использован для создания устройства телепортации.

Кроме того, добавьте в сцену контроллер игрока и меш лабиринта (можно найти в пакете с примерами для этой книги, в папке `assets` для этой главы), и создайте некоторое освещение, чтобы изначально все выглядело красиво. Меш был создан в программе трехмерного моделирования, в данном случае Blender (http://www.blender.org/). Ни один из этих ресурсов не имеет прямого отношения к искусственному интеллекту, но они создают презентабельное окружение, соответствующее сценарию нашей игры. Детальное описание настройки освещения выходит за рамки этой книги, но необходимые функции становятся доступны при выборе пункта меню **Window** ⇒ **Lightmapping** (Окно ⇒ Освещение), как показано на рис. 7.4.

Более подробную информацию о настройке освещения можно найти в электронной документации Unity по адресу http://docs.unity3d.com/Manual/Lightmapping.html.

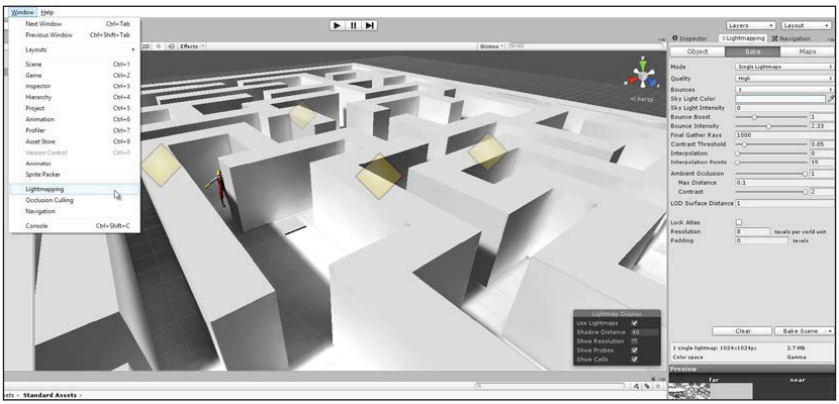


Рис. 7.4. Создание начальной сцены

Внедрение навигационного меша

Вражеские персонажи должны уметь разумно перемещаться по уровню, находить и преследовать игрока, а также искать средства для восстановления своего здоровья. Искусственному интеллекту недостаточно способности перемещаться по прямой между любыми двумя точками, так как ему могут помешать препятствия, такие как стены и другие персонажи. Искусственный интеллект должен изменять направление своего движения при встрече с другими объектами. Чтобы добиться такой разумности, будет использоваться навигационный меш. Он представляет собой ресурс невидимого меша, который Unity автоматически сгенерирует, опираясь на доступность для прохода горизонтальных поверхностей в уровне, то есть поверхностей, которые идентифицируются как пол. Сама навигационная сетка не содержит искусственного интеллекта. Навигационный меш является, скорее, математической моделью со всеми необходимыми данными, позволяющими объектам с искусственным интеллектом успешно рассчитывать маршрут и пройти весь путь, обходя препятствия, когда это требуется. Для создания навигационного меша уровня выберите пункт **Window ⇒ Navigation** (Окно ⇒ Навигация) в меню приложе-

ния. В результате появится вкладка **Navigation Mesh** (Навигационный меш), которая может быть пристыкована к инспектору объектов.

Описание принципов внедрения навигационного меша можно найти в электронной документации Unity по адресу http://docs.unity3d.com/Manual/Navmeshes.html.

При внедрении навигационного меша можно воспользоваться некоторыми настройками, изображенными на рис. 7.5.

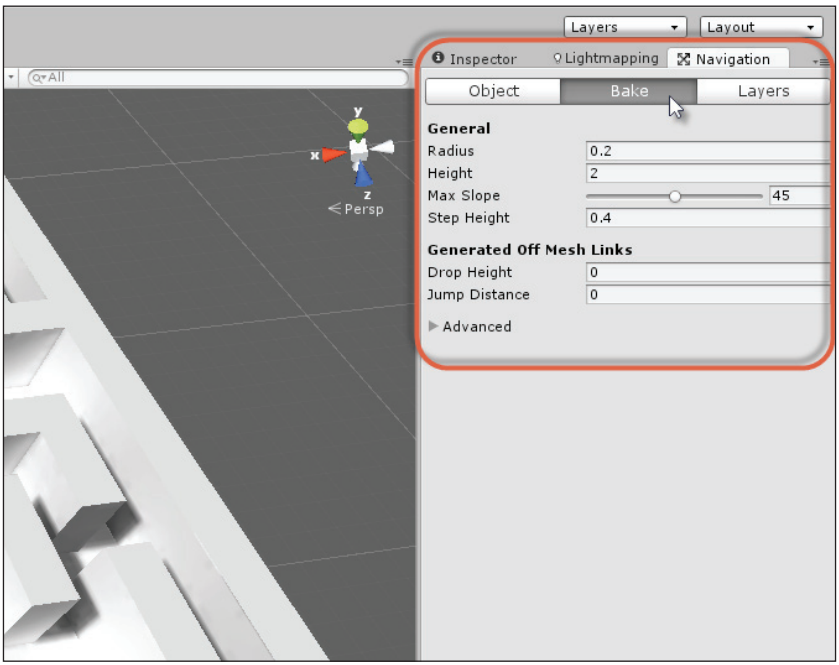


Рис. 7.5. Подготовка навигационного меша к внедрению

Во-первых, почти всегда требуется изменить значение параметра **Radius** (Радиус). Этот параметр определяет радиус воображаемой окружности вокруг ног персонажей, эта окружность задает их габариты при ходьбе. Если радиус слишком большой, навигационный меш отображается неправильно или разрывается, а если он слишком мал, генерация меша займет много времени, а кроме того, персонажи станут просачиваться сквозь стены. Экспериментальный подход, основанный на пробах, ошибках и корректировках, позволит получить значение, оптимальное для вашего проекта. Для этого примера лучше

всего подойдет значение 0,2. Если радиус слишком велик, навигационный меш будет порван в узких местах, что нехорошо, потому что персонажи не смогут проходить через зауженные проходы, как показано на рис. 7.6.

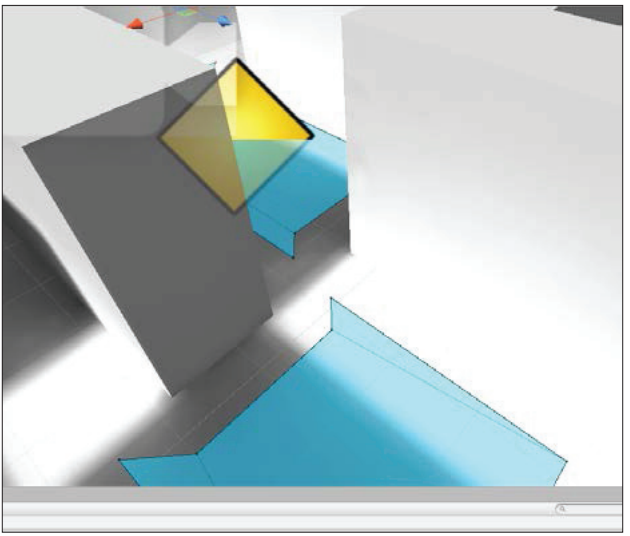


Рис. 7.6. Разрыв навигационного меша в узких местах

Во-вторых, навигационный меш (генерируется один раз) может оказаться расположен выше меша пола. Если это произойдет, вы можете уменьшить значение 1 в параметре **Height Inaccuracy %** (Неточность по высоте %) из группы **Advanced** (Дополнительные), как показано на рис. 7.7. Это предотвратит парение персонажей в воздухе. Помните, что после корректировки настроек нужно повторно внедрить навигационный меш, чтобы применить изменения.

На рисунках можно заметить, что сцена содержит два отдельных лабиринта (левый и правый), не соединенных между собой проходами. В этом примере персонажи имеют возможность свободно перемещаться между лабиринтами с помощью телепортации.

Для реализации соединения между разделенными навигационными мешами, позволяющего искусственному интеллекту прокладывать маршруты, мы могли бы использовать ссылки между мешами. Добавим новый меш, который должен послужить площадкой или

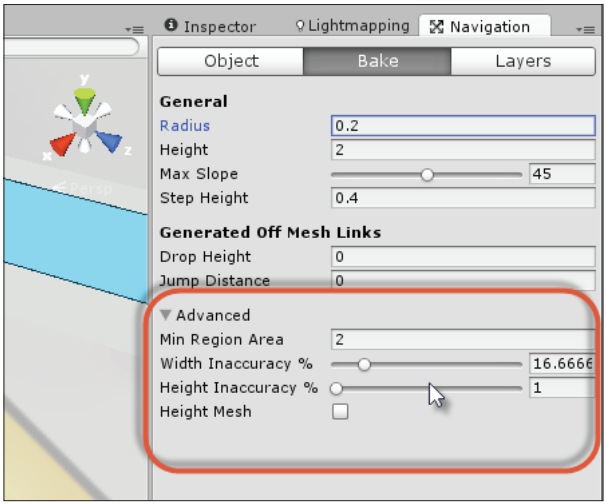


Рис. 7.7. Уменьшение **Height Inaccuracy %** (Неточность по высоте %) помогает приблизить сгенерированный навигационный меш к полу

платформой для телепортации. Для этого примера я использовал стандартный параллелепипед с системой частиц для создания нужного эффекта, но это не существенно. Затем присоедините компонент ссылки между мешами к объекту меша, как показано на рис. 7.8.

Повторите процедуру для создания второй площадки телепортации. Свяжите поле **Start** (Начало) трансформации каждого объекта с компонентом ссылки между мешами. При этом выбранная площадка телепортации станет стартовой точкой. Затем свяжите поле **End** (Конец) трансформации каждого объекта с местом назначения. В результате будет установлена связь между двумя площадками телепортации. Если панель навигации открыта в редакторе Unity и активна, после установки соединения в окне обзора сцены должна появиться стрелка соединения, как показано на рис. 7.9. Ссылки между мешами можно также создавать автоматически. За дополнительной информацией обращайтесь по адресу https://www.youtube.com/watch?v=w3-sSozYph4.

Заготовку проекта для этой главы, готовую к реализации искусственного интеллекта, можно найти в пакете примеров для этой книги, в папке `Start`.

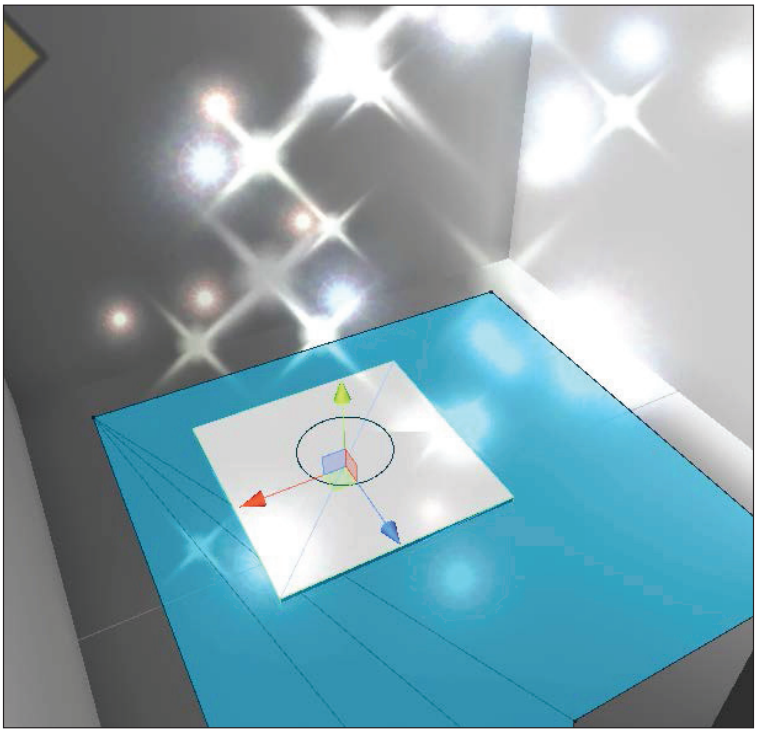


Рис. 7.8. Создание площадки телепортации
с помощью ссылки между мешами

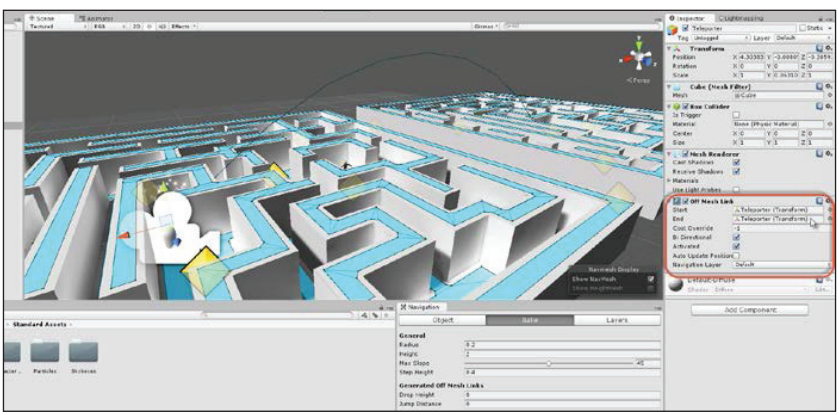


Рис. 7.9. Определение связей с помощью ссылок между мешами

Создание агента искусственного интеллекта

Теперь создадим агента искусственного интеллекта, который будет реагировать на действия игрока. Во-первых, агент должен иметь меш для отображения его в сцене. Для этого я использовал меш `Constructor`, который является частью пакета **Character Controllers** в Unity, импортированного ранее. Перетащите его из панели **Project** (Проект) в сцену и удалите компонент аниматора как показано на рис. 7.10. Анимация будет важна, но требуемый нам аниматор будет создан позже.

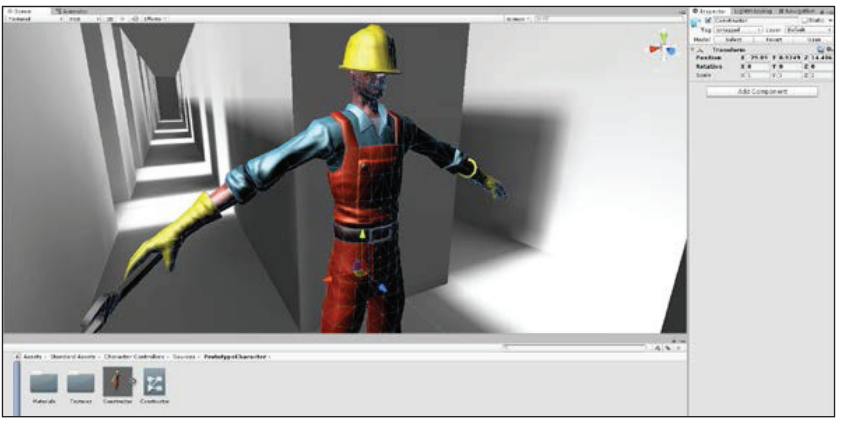


Рис. 7.10. Добавление меша `Constructor` для создания вражеского персонажа

Напомню, что мы не воспользовались готовым контроллером третьего лица, а использовали лишь меш `Constructor`.

Затем добавьте компонент `NavMeshAgent` к объекту, выбрав в меню пункт **Component** ⇒ **Navigation** ⇒ **Nav Mesh Agent** (Компонент ⇒ Навигация ⇒ Агент навигационного меша). Это позволит объекту использовать навигационный меш и прокладывать маршруты. Установите значения полей **Radius (Радиус)** и **Height** (Высота) компонента в соответствии с размерами меша. Установите значение поля **Stopping Distance** (Расстояние остановки) равным 2, оно определяет, как близко к месту назначения можно подойти, как показано на рис. 7.11. Конечно, для ваших собственных проектов значение поля **Stopping Distance** (Расстояние остановки), вероятно, будет другим.

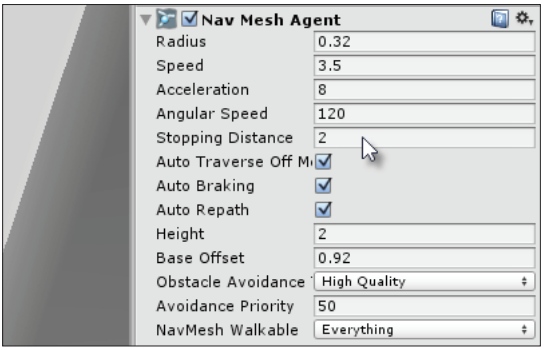


Рис. 7.11. Настройка компонента `NavMeshAgent` для прокладки маршрутов

Теперь добавим компонент `Rigidbody` и включим флажок **Is Kinematic** (Кинематический), как показано на рис. 7.12. Это позволит объекту стать триггером и частью физической системы, вызывая и принимая физические события. Однако, с установленным флажком **Is Kinematic** (Кинематический), среда Unity не переопределяет трансформацию объекта (позиция, направление и масштаб). Это позволит использовать `NavMeshAgent` исключительно для управления движением персонажа.

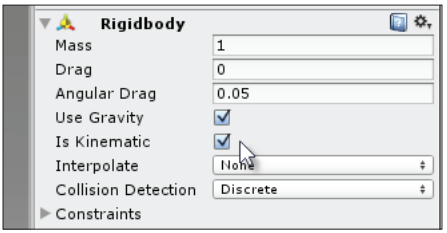


Рис. 7.12. Настройка компонента `Rigidbody` для взаимодействия с физической системой

Теперь добавьте к объекту компонент `BoxCollider` и включите флажок **Is Trigger** (Триггер), чтобы преобразовать его в триггер и запретить другим объектам проходить сквозь него. Это будет использоваться искусственным интеллектом для расчета области обзора агента. Искусственный интеллект будет просматривать эту область и только объекты, попадающие в эту область, будут учитываться при выборе

соответствующей реакции. Чтобы изменить размер области обзора агента, задайте значения полей **X**, **Y**, и **Z**, как показано на рис. 7.13.

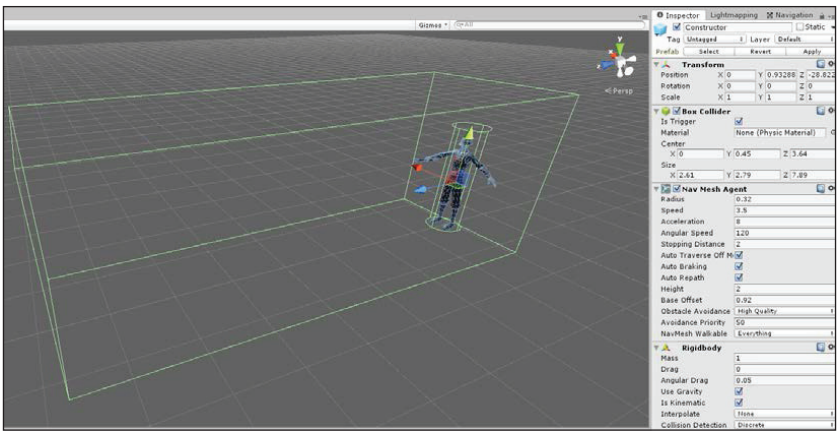


Рис. 7.13. Использование компонента `BoxCollider` для настройки области обзора агента

Наконец, создайте новый файл сценария `AI_Enemy.cs` на C# для реализации искусственного интеллекта врага. Этот сценарий будет содержать весь код искусственного интеллекта, и он будет написан в этой главе. После создания файла подключите его к объекту врага в сцене. Теперь можно переходить к программированию и построению графов искусственного интеллекта! Начнем с создания конечного автомата и подготовки состояний, которые определяют поведение врага.

Конечные автоматы в Mecanim

С этого момента мы сосредоточимся на разработке искусственного интеллекта враждебного персонажа на языке C#, и визуальном программировании графа Mecanim. Mecanim – это система анимации в Unity (http://docs.unity3d.com/Manual/MecanimAnimationSystem.html). В следующих разделах мы создадим полный класс, просматривая и обсуждая отдельные участки кода, и исходный код полного класса будет написан нами вместе. Его можно найти в файле `AI_Enemy.cs` законченного проекта.

Для начала давайте рассмотрим концепцию конечного автомата. Если задуматься о поведении вражеских персонажей, можно выде-

лить несколько моделей поведения. При запуске сцены враги бездействуют, а затем переходят к патрулированию. Во время патрулирования они могут наткнуться на игрока. В этом случае они начнут преследование игрока, пока игрок не окажется в области нападения. Когда игрок будет доступен для атаки, они нападают на игрока. Исключением из этих правил является ситуация, когда врагу нанесен серьезный ущерб и он близок к смерти. Оказавшись в критическом состоянии, враг, вместо того чтобы действовать агрессивно, будет спасаться бегством и искать зелье для восстановления здоровья, пока уровень здоровья не вернется в норму.

Получив набор возможных шаблонов поведения противника, мы можем определить ряд дискретных ключевых состояний. Это ожидание, патрулирование, погоня, нападение и бегство. В каждый момент времени враг может находиться в одном и только в одном из этих состояний, и каждое состояние определяет, как враг будет себя вести. Для реализации этой логики можно использовать граф конечного автомата. Он относится не к конкретному классу или типу объекта (например, `MonoBehaviour` или `ScriptableObject`), а является шаблоном проектирования или способом программирования. Конечный определяет конечное множество состояний (ожидание, патрулирование, погоня и т. д., как уже упоминалось выше) и логику связей между состояниями – когда и как одно состояние переходит в другое. Враг в нашей ситуации будет зависеть на самом деле от двух механизмов управления: программного кода на C# и графа анимаций Mecanim. Последний контролирует только анимационные эффекты, которые воспроизводятся мешами врагов в каждом состоянии. Построим сначала граф Mecanim.

Щелкните правой кнопкой мыши на панели **Project** (Проект) и создайте новый ресурс **Animator Controller** (Контроллер аниматора). Откройте ресурс в окне **Animator** (Аниматор), доступном в виде пункта **Window** ⇒ **Animator** (Окно ⇒ Аниматор) в меню приложения, как показано на рис. 7.14.

Граф анимации Mecanim определяет все возможные состояния анимации для меша, и они должны соответствовать состояниям, перечисленным выше, а именно ожидание, патрулирование, погоня, нападение и бегство. Для настройки анимации этих состояний выберите ресурс меша **Constructor** (Инженер) в панели **Project** (Проект) и сделайте все анимации повторяющимися, установив флажки **Loop Time** (Повторять анимацию) и **Loop Pose** (Повторять позы) в инспекторе объектов, как показано на рис. 7.15. Это предотвратит остановку анимаций персонажей после первого же воспроизведения.

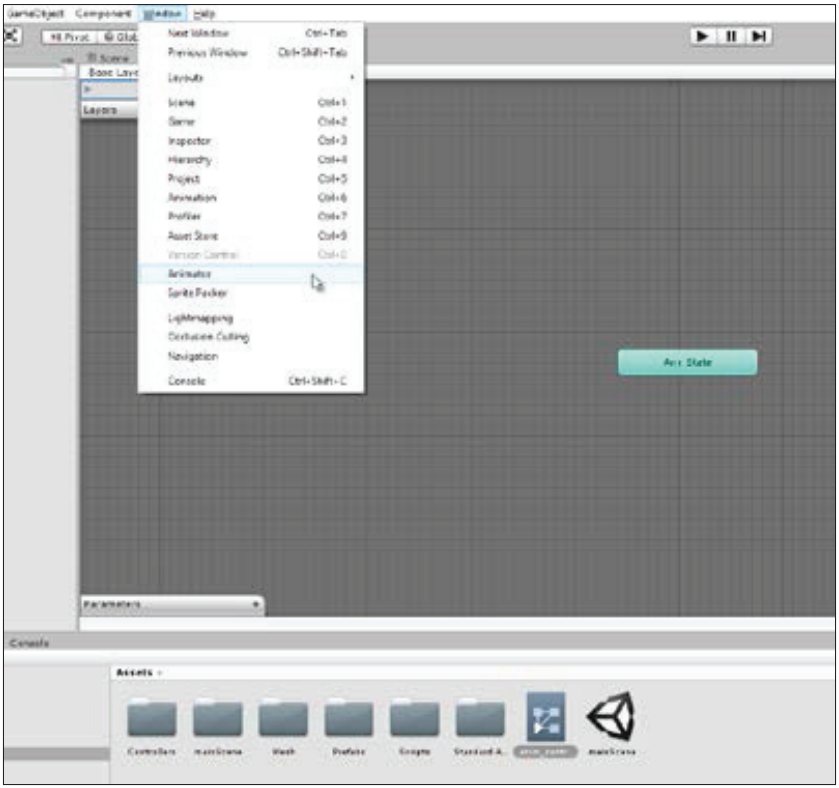


Рис. 7.14. Доступ к графу анимации

Теперь добавим в граф анимации состояний, по одной для каждого состояния. В состоянии ожидания **Idle** должна воспроизводиться анимация бездействия. В состоянии патрулирования **Patrol** должна воспроизводиться анимация ходьбы, так как персонаж должен ходить. Для состояний преследования **Chase** и бегства **Flee** подойдет анимация бега, а для состояния **Attack** – анимация прыжка. Модель **Constructor** не содержит анимации нападения, так что в данном примере ее заменит анимация прыжка.

Добавьте их в граф, перетаскивая каждую анимацию из панели проекта в редактор графа и давая соответствующие названия каждому состоянию, как показано на рис. 7.16.

В дополнение к уже ранее добавленным состояниям добавим еще одно пустое состояние. Это будет начальное состояние врага, или

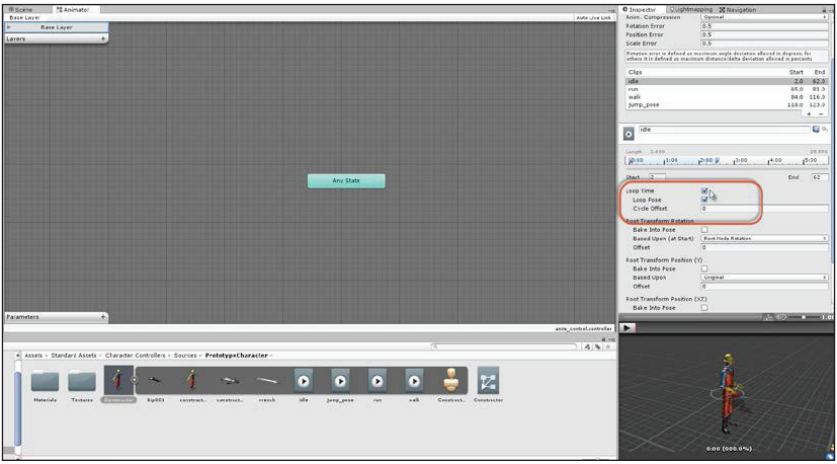


Рис. 7.15. Подготовка анимаций
для конечного автомата Mecanim

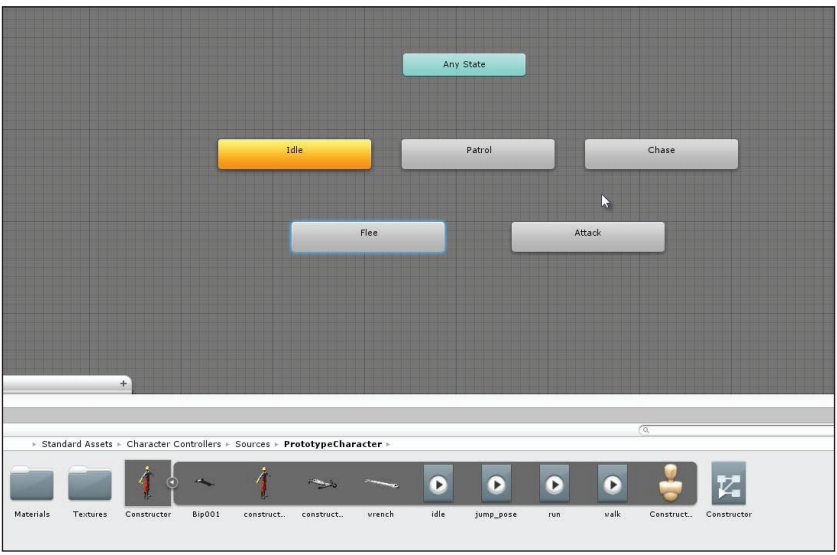


Рис. 7.16. Построение конечного автомата в окне **Animator**

состояние по умолчанию; это состояние не воспроизводит никакой анимации и представляет, по сути, состояние без состояния, то есть соответствует моменту, пока мы явно не определили состояние противника при запуске уровня. Чтобы создать пустое состояние по умолчанию, щелкните правой кнопкой мыши на пустом пространстве внутри редактора графа и выберите в контекстном меню пункт **Create State ⇒ Empty** (Создать состояние ⇒ Пустое) (переименуйте его в `Start` или `Init`), а затем сделайте его состоянием по умолчанию, щелкнув на нем правой кнопкой мыши и выбрав пункт **Set As Default** (Установить по умолчанию), как показано на рис. 7.17.

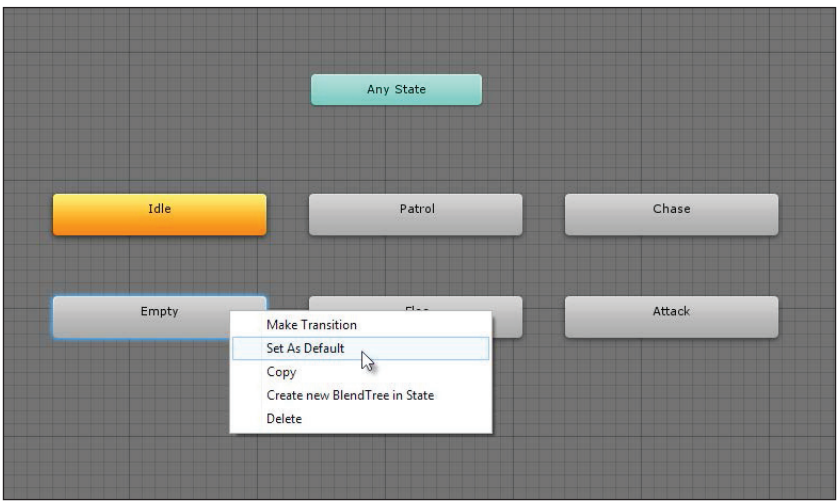


Рис. 7.17. Установка узла **Empty** как состояния по умолчанию

Граф теперь содержит по одной анимации для каждого состояния персонажа, но состояния не связаны между собой; каждое состояние изолировано. В частности, нет никакой логики перехода из одного состояния в другое. Чтобы исправить это, создайте пять новых триггеров, используя панель **Parameters** (Параметры) в нижнем левом углу окна **Mecanim**. Триггер – это специальная булева переменная. Unity автоматически присваивает триггерам значение `false` – смена их значений на `true` приведет к немедленным изменениям, например к смене состояния. Триггеры, как мы увидим ниже, доступны в программном коде на C#.

Теперь создайте пять триггеров: **Idle**, **Patrol**, **Chase**, **Attack** и **SeekHealth**, как показано на рис. 7.18.

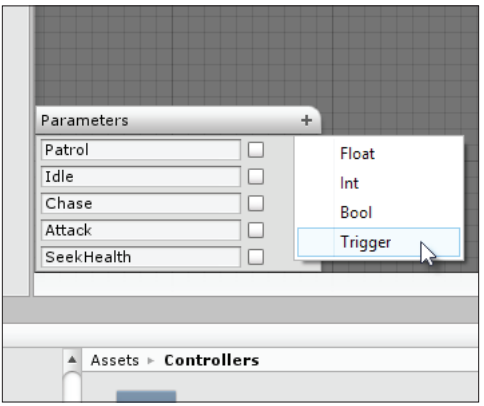


Рис. 7.18. Создание триггеров для анимации каждого состояния

С помощью состояний и триггеров можно определить связи между состояниями в графе. В частности, при установке триггера **Patrol** следует перейти от состояния **Idle** к состоянию **Patrol**; при установке триггера **Chase** нужно перейти от состояния **Patrol** к состоянию **Chase**, когда устанавливается триггер **Attack**, должен быть выполнен переход от состояния **Chase** к состоянию **Attack**, и т. д. Кроме того, связи между большинством состояний являются двунаправленными: состояние **Patrol** может перейти в состояние **Chase** (например, когда враг видит игрока), а из состояния **Chase** может вернуться в состояние `Patrol` (когда теряет игрока из виду). Чтобы связать два состояния, щелкните правой кнопкой мыши на состоянии, выберите в контекстном меню пункт **Make Transition** (Создать переход), а затем щелкните на состоянии назначения, с которым должно быть установлена связь.

Граф определяет теперь полный конечный автомат состояний для объекта противника. Привязка его к объекту врага в сцене реализуется очень просто.

Добавьте компонент **Animator** к объекту, а затем перетащите контроллер **Animator** из панели **Project** (Проект) в поле **Controller** (Контроллер) компонента **Animator**, как показано на рис. 7.20.

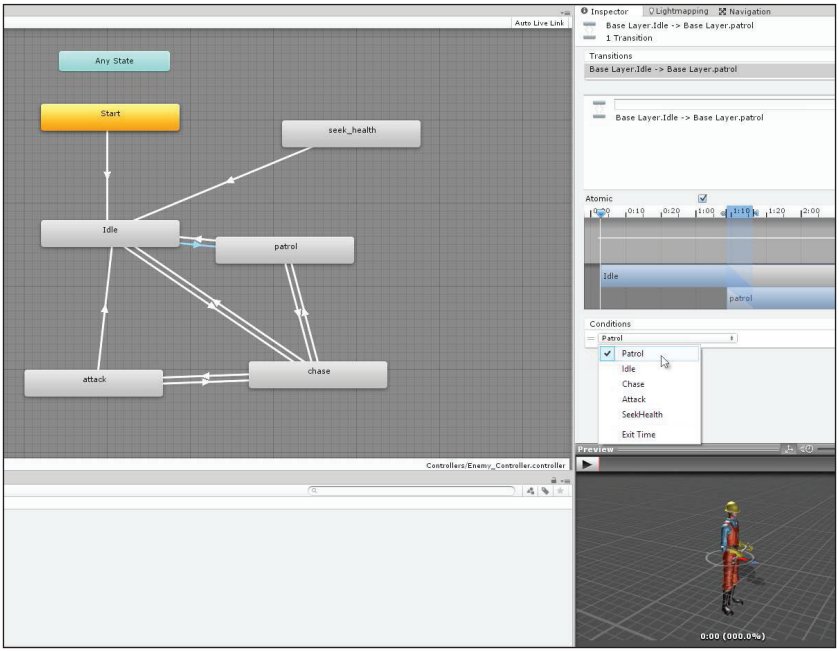


Рис. 7.19. Настройка связей между состояниями

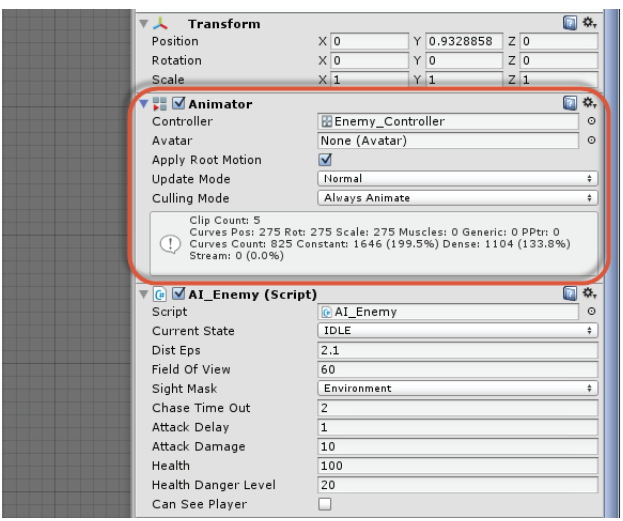


Рис. 7.20. Присоединение контроллера **Animator** к объекту врага

Конечный автомат состояний в C# – начало

Теперь, когда создание конечного автомата для анимаций завершено, мы должны обратить внимание на конечный автомат в C#, который управляет поведением противника, а также инициирует триггеры в графе Mecanim для анимации соответствующих действий (ходьбы и бега) в нужные моменты времени. Для начала добавьте следующее общедоступное перечисление в начало файла сценария `AI_Enemy.cs`, как показано в листинге 7.1. Это перечисление определяет все возможные состояния врага в конечном автомате и присваивает каждому состоянию свой уникальный хэш-код строки, то есть состоянию `IDLE` присвоено значение `2081823275`, которое является хэш-кодом строки `IDLE`, и т. д. Это понадобится позднее для работы с Mecanim, чтобы инициировать триггеры. Получить хэш-код строки можно с помощью функции `StringToHash` класса `Animator`, как показано ниже.

Листинг 7.1. Функция для получения хэш-кода строки

```
// Определение возможных состояний
public enum AI_ENEMY_STATE {IDLE = 2081823275,
                            PATROL = 207038023,
                            CHASE = 1463555229,
                            ATTACK = 1080829965,
                            SEEKHEALTH = -833380208};
```

Более подробную информацию можно найти по адресу http://docs.unity3d.com/ ScriptReference/Animator.StringToHash.html.

Опираясь на перечисление `AI_ENEMY_STATE`, класс `AI_Enemy` будет поддерживать в актуальном состоянии общедоступную переменную `CurrentState`, которая отражает состояние объекта врага в текущий момент. Значение этой переменной будет меняться с течением времени, по мере изменения состояния, как показано ниже:

```
// Текущее состояние врага
public AI_ENEMY_STATE CurrentState = AI_ENEMY_STATE.IDLE;
```

Как и большинство объектов, класс `AI_Enemy` содержит функцию `Awake`, где создает кэш ссылок на другие компоненты, в том числе на локальный компонент `Transform` объекта `NavMeshAgent`, а также на компоненты других объектов сцены, таких как объект игрока, как показано в листинге 7.2. Эти ссылки будут использоваться в других местах в сценарии.

Листинг 7.2. Функция Awake в классе AI_Enemy

```
// Получить ссылку на объект Animator
ThisAnimator = GetComponent<Animator>();

// Получить ссылку на агента навигационного меша
ThisAgent = GetComponent<NavMeshAgent>();

// Получить ссылку на свой компонент Transform
ThisTransform = transform;

// Получить ссылку на компонент Transform игрока
PlayerTransform = GameObject.FindGameObjectWithTag("Player").transform;

// Получить ссылку на коллайдер
ThisCollider = GetComponent<BoxCollider>();
```

Здесь используется прием кэширования в переменных `ThisAnimator`, `ThisTransform`, `ThisAgent` и `ThisCollider`. Это позволяет получить прямые ссылки на другие компоненты при запуске уровня и избежать необходимости вызывать функции доступа к свойствам (`get` и `set`) каждый раз, когда понадобится обратиться к объекту. То есть, использование `This.Transform` дает большую нагрузку, чем переменной кэша `ThisTransform`.

Каждому состоянию конечного автомата будут соответствовать отдельная сопрограмма. Сопрограмма будет выполнять бесконечный цикл, пока состояние остается активным, определяя поведение врага в этом состоянии. Основная работа конечного автомата состоит в выборе и инициализации соответствующего состояния при корректных условиях. Давайте начнем с создания сопрограммы для состояния `Idle` – состояния по умолчанию вражеского персонажа.

Создание состояния Idle

Объект врага начинает игру в состоянии простоя (состояние бездействия), к которому будет выполнен первый переход. В этом состоянии враги стоят там, где находятся, воспроизводя анимацию простоя. Переход в это состояние выполняется один раз при запуске сцены, но к нему можно вернуться при выходе из некоторых других состояний, как к промежуточному шагу перед переходом в новое состояние. В этом состоянии враг всегда должен воспроизводить анимацию ожидания только один раз, а затем, по завершении анимации, перейти в другое состояние. Враг автоматически должен перейти в состояние патрулирования, чтобы начать поиск игрока в сцене. Этот переход выполняется в два этапа. Во-первых, нужно начать воспроизведение анимации `Idle`. Во-вторых, мы должны получить уведомление о за-

вершении анимации Idle, чтобы начать переход в состояние Patrol. Рассмотрим реализацию состояния Idle в листинге 7.3.

Листинг 7.3. Реализация состояния Idle

```
01 //-------------------------------------------------
02 // Выполняется, когда объект находится в состоянии ожидания
03 public IEnumerator State_Idle()
04 {
05   // Установить текущее состояние
06   CurrentState = AI_ENEMY_STATE.IDLE;
07
08   // Активировать состояние ожидания с помощью графа Mecanim
09   ThisAnimator.SetTrigger((int) AI_ENEMY_STATE.IDLE);
10
11   // Остановить перемещение агента навигационного меша
12   ThisAgent.Stop();
13
14   // Бесконечный цикл на время пребывания в состоянии ожидания
15   while(CurrentState == AI_ENEMY_STATE.IDLE)
16   {
17     // Проверить видимость игрока
18     if(CanSeePlayer)
19     {
20       // игрок виден, начать преследование
21       StartCoroutine(State_Chase());
22       yield break;
23     }
24
25     // Ждать следующего кадра
26     yield return null;
27   }
28 }
29 //-------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 7.3:

- **Строка 03**: объявление сопрограммы State_Idle для состояния ожидания. За более подробной информацией о сопрограммах обращайтесь к электронной документации Unity по адресу http://docs.unity3d.com/Manual/Coroutines.html. Если коротко, то сопрограммы работают как асинхронные функции (как блок кода, который выполняется в фоновом режиме, параллельно с другими функциями). По этой причине бесконечный цикл в строке 15 не вызовет зависания, потому что сопрограмма работает как отдельный поток. Сопрограммы всегда возвращают результат с типом данных IEnumerator и всегда содержат в своем теле оператор yield.

- **Строка 09**: вызов функции `SetTrigger` аниматора, она получает хэш-код строки `Idle` в качестве аргумента и устанавливает триггер `Idle` в графе Mecanim, инициируя воспроизведение анимации ожидания. Таким способом конечный автомат C # связывается с конечным автоматом Mecanim. Обратите внимание, что в строке 12 вызывается функция `Stop` компонента `NavMeshAgent`, чтобы остановить любое движение, которое объект, возможно, выполняет. Это связано с тем, что в режиме воспроизведения анимации ожидания враг не должен двигаться.
- **Строка 15**: здесь функция `State_Idle` входит в бесконечный цикл, то есть этот цикл будет переходить из кадра в кадр, пока враг находится в состоянии `Idle`. Пока состояние ожидания активно, все, что есть в цикле, будет выполняться в каждом кадре, что позволит объекту обновлять и изменять свое поведение с течением времени.
- **Строка 18**: проверяется единственное условие выхода из состояния `Idle`, кроме ожидания завершения анимации `Idle`, – видимость игрока. Видимость игрока определяется булевой переменной `CanSeePlayer` (подробнее о прямой видимости рассказывается ниже). Если значение переменной `CanSeePlayer` равно `true`, вызовом функции `StartCoroutine` активируется состояние `Chase` и состояние ожидания прерывается вызовом оператора `yield break`.

Состояние `Idle` продолжается, пока выполняется бесконечный цикл, а он выполняется, пока не будет виден игрок. Однако состояние `Idle` должно быть временным, пока не завершится одноразовое воспроизведение анимации простоя, после этого мы должны получить сообщение о завершении воспроизведения. Для реализации уведомления можно использовать события анимации. Для этого выберите меш персонажа **Constructor** (Инженер) в панели **Project** (Проект) и откройте вкладку **Animation** (Анимация), чтобы найти анимацию бездействия в инспекторе объектов. Здесь распахните раздел **Events** (События), как показано на рис. 7.21.

Затем дважды щелкните на шкале времени анимации в точке, соответствующей моменту времени 1 (в конце), чтобы добавить вызов функции в этот момент, которая отправит сообщение объекту врага, когда анимация завершится, как показано на рис. 7.22. Для этого я добавил метод `OnIdleAnimCompleted` в класс `AI_Enemy`.

Функция `OnIdleAnimCompleted` будет автоматически вызвана по завершении анимации ожидания. Реализация этого метода представлена в листинге 7.4.

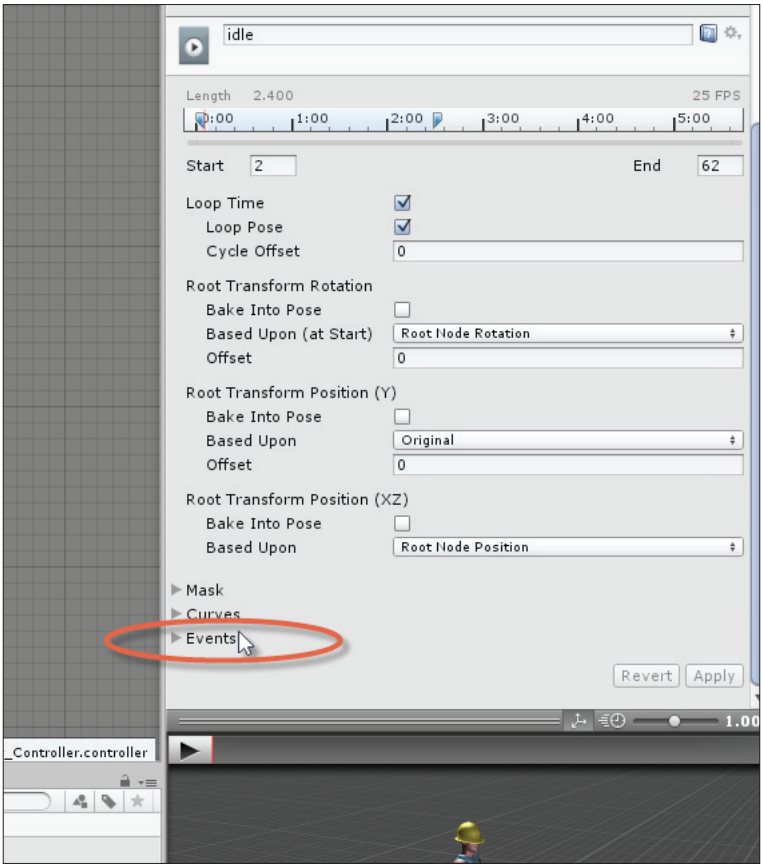


Рис. 7.21. Раздел **Events** (События) в инспекторе объектов

Листинг 7.4. Реализация метода OnIdleAnimCompleted

```
// Посылает событие по завершении анимации
public void OnIdleAnimCompleted()
{
  // Прервать активное состояние ожидания
  StopAllCoroutines();
  StartCoroutine(State_Patrol());
}
```

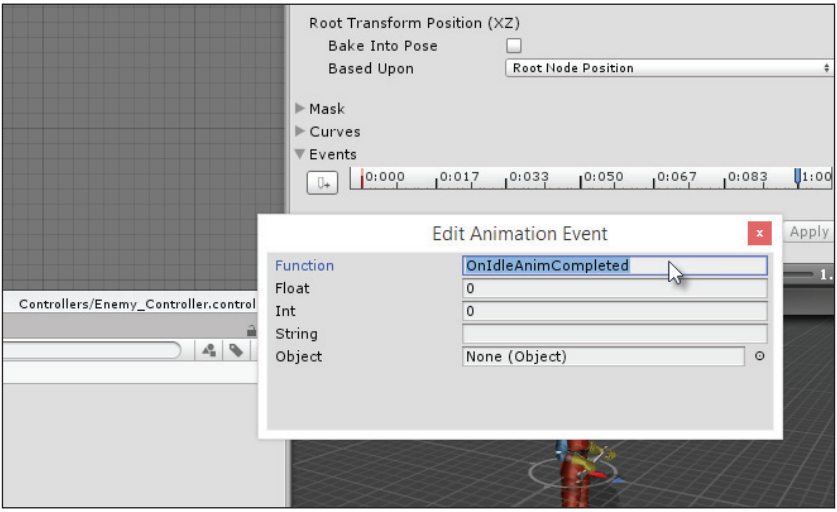


Рис. 7.22. Вызов функции по завершении анимации

Создание состояния Patrol

В состоянии патрулирования `Patrol` враг должен обходить окружающую территорию в поисках игрока. Персонаж может перейти в состояние `Patrol` из состояния `Idle`, после завершения анимации ожидания, а также из состояния `Chase`, если потерял игрока из вида во время погони. Состояние патрулирования основано на цикличной логике. В частности, враг должен выбрать случайный пункт назначения в меше навигации и отправиться в этот пункт. По прибытии в пункт назначения процесс повторяется. Единственным условием, которое выведет врага из этого состояния, является появление игрока в зоне видимости, что переведет его в состояние преследования `Chase`.

Хотя все выглядит просто, реализация этого состояния зависит от решения двух сложных вопросов: выбора случайного места и проверки видимости игрока. Начнем со случайного выбора места.

Во вкладке **Scene** (Сцена) я создал коллекцию пунктов назначения (пустых игровых объектов), которые отмечены тегом **Waypoint** (Путевая точка) и нужны только для обозначения мест в навигационном меше `NavMesh`. Вместе они представляют собой все возможные места, в которые враг может направляться во время патрулирования. Врагу остается только случайным образом выбрать одно из этих мест, как это показано на рис. 7.23.

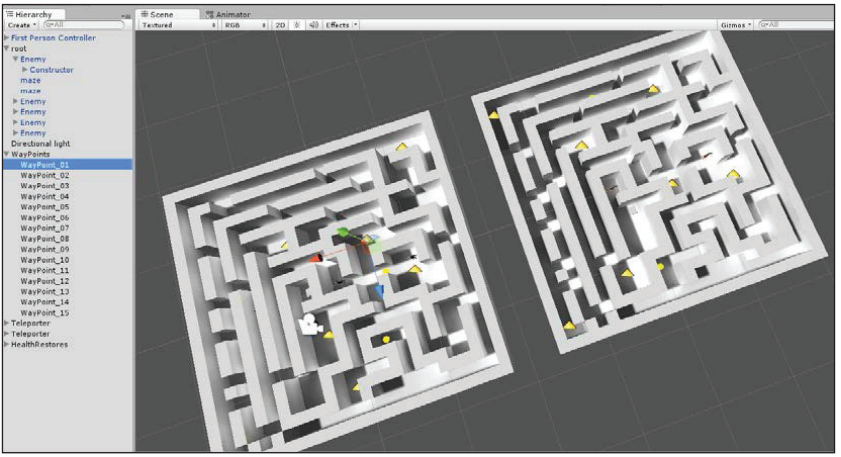


Рис. 7.23. Создание путевых точек во вкладке **Scene** (Сцена)

Для выбора пункта назначения в состоянии `Patrol` функция `Awake` в классе `AI_Enemy` должна получить список всех путевых точек в сцене. Мы можем сделать это с помощью Linq, как показано в листинге 7.5. В этом примере создается статический массив `Waypoints` всех объектов путевых точек в сцене.

Листинг 7.5. Получение списка путевых точек

```
01 // найти все игровые объекты с тегом Waypoint
02 GameObject[] Waypoints =
   GameObject.FindGameObjectsWithTag("Waypoint");
03
04 // Выбрать все компоненты трансформации из путевых точек с помощью Linq
05 WayPoints = (from GameObject GO in Waypoints
06                   select GO.transform).ToArray();
```

Имея список всех пунктов назначения, можно приступать к реализации состояния `Patrol`, как показано в листинге 7.6, которая периодически выбирает новые пункты назначения.

Листинг 7.6. Реализация состояния патрулирования

```
01 //--------------------------------------------------
02 // Выполняется, когда объект находится в состоянии патрулирования
03 public IEnumerator State_Patrol()
04 {
05   // Установить текущее состояние
06   CurrentState = AI_ENEMY_STATE.PATROL;
07
```

```
08   // Активировать состояние патрулирования
09   ThisAnimator.SetTrigger((int) AI_ENEMY_STATE.PATROL);
10
11   // Выбрать случайную путевую точку
12   Transform RandomDest = WayPoints[Random.Range(0, WayPoints.Length)];
13
14   // Отправиться к выбранной точке
15   ThisAgent.SetDestination(RandomDest.position);
16
17   // Бесконечный цикл на время пребывания в состоянии патрулирования
18   while(CurrentState == AI_ENEMY_STATE.PATROL)
19   {
20     // Проверить видимость игрока
21     if(CanSeePlayer)
22     {
23       // игрок виден, начать преследование
24       StartCoroutine(State_Chase());
25       yield break;
26     }
27
28     // Проверить достижения пункта назначения
29     if(Vector3.Distance(ThisTransform.position,
       RandomDest.position) <= DistEps)
30     {
31       // Заданная точка достигнута. Вернуться в состояние ожидания
32       StartCoroutine(State_Idle());
33       yield break;
34     }
35
36     // Ждать следующего кадра
37     yield return null;
38   }
39 }
40 //---------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 7.6:

- **Строка 12**: здесь функция `Random.Range` выбирает случайный пункт назначения из массива `Waypoints`. Этот пункт назначения передается как аргумент функции `SetDestination` компонента `NavMeshAgent`, который отправляет туда врага.
- **Строка 28**: функция `Vector3.Distance` определяет достижение цели. Это реализуется не проверкой на совпадение позиции врага с позицией пункта назначения, потому что при операциях с плавающей точкой возможны погрешности. Вместо этого выполняется проверка, не подошел ли враг к пункту назначения ближе, чем на заданное расстояние (`DistEps`), что и считается прибытием.

- **Строка 32**: если враг находится в пункте назначения, он переходит в состояние `Idle`. После завершения анимации ожидания враг снова входит в состояние `Patrol`.
- **Строка 21**: и снова состояние `Patrol` прерывается, если враг увидел игрока. В этом случае он переходит в состояние `Chase`.

Значение логической переменной `CanSeePlayer` определяет, видит ли враг игрока в текущем кадре. Значение этой переменной обновляется в каждом кадре. Процесс этот начинается в функции `Update`, как показано в листинге 7.7.

Листинг 7.7. Проверка видимости игрока

```
01 void Update()
02 {
03   // Допустить, что игрок невидим
04   CanSeePlayer = false;
05
06   // Если игрок вне границ видимости, выйти
07   if(!ThisCollider.bounds.Contains(PlayerTransform.position)) return;
08
09   // Если игрок в границах видимости, проверить его
     нахождение в прямой видимости
10   CanSeePlayer = HaveLineSightToPlayer(PlayerTransform);
11 }
```

Ключевой вопрос для функции `Update` – находится ли игрок внутри коллайдера, присоединенного к врагу. Этот коллайдер определяет область видимости врага. Если игрок находится внутри этой области, игрок может быть видим врагу. В этом случае требуется дальнейшая проверка. Она основана на функции `HaveLineSightToPlayer`. Эта функция возвращает логическое значение (`true/false`), указывающее, видит враг игрока или нет, как показано в листинге 7.8.

Листинг 7.8. Проверка нахождения игрока в прямой видимости

```
// Определяет видимость игрока в текущий момент
private bool HaveLineSightToPlayer(Transform Player)
{
  // Вычислить угол между линией зрения врага и направлением на игрока
  float Angle = Mathf.Abs(Vector3.Angle(ThisTransform.forward,
    (Player.position-ThisTransform.position).normalized));

  // Если угол больше угла поля зрения, игрок невидим
  if(Angle > FieldOfView) return false;

  // Проверить, не скрывает ли игрока непрозрачная стена
```

```
  if(Physics.Linecast(ThisTransform.position,
       Player.position, SightMask))
    return false;

  // Игрок видим
  return true;
}
```

Как мы знаем из предыдущих глав, определение видимости является двухступенчатым процессом. Во-первых, видимость определяется углом между вектором направления взгляда врага и нормированным вектором направления от врага к игроку. Если угол меньше, чем угол обзора врага, игрок находится перед врагом и он видим, если врага и игрока не разделяют препятствия, такие как стены. Вторая проверка выполняется с помощью функции `Physics.Linecast`, которая определяет, можно ли соединить врага и игрока непрерывной прямой линией. Если это возможно, значит никаких препятствий между ними нет и игрок виден.

Создание состояния Chase

Если враг видит игрока и не находится на расстоянии атаки, он побежит к игроку. Это состояние, при котором враг бежит к игроку с враждебными намерениями, является состоянием `Chase`. Из этого состояния есть два основных выхода. Если враг сокращает расстояние до расстояния, достаточного для нападения, его состояние `Chase` переходит в состояние `Attack`. Если нет, и игрок исчезает из поля зрения врага, враг продолжит преследование некоторое время, а затем откажется от погони, если игрок все еще не видим. Рассмотрим листинг 7.9.

Листинг 7.9. Реализация состояния преследования

```
01 // Выполняется, когда объект находится в состоянии преследования
02 public IEnumerator State_Chase()
03 {
04   // Установить текущее состояние
05   CurrentState = AI_ENEMY_STATE.CHASE;
06
07   // Активировать состояние преследования
08   ThisAnimator.SetTrigger((int) AI_ENEMY_STATE.CHASE);
09
10   // Бесконечный цикл на время пребывания в состоянии преследования
11   while(CurrentState == AI_ENEMY_STATE.CHASE)
12   {
```

```
13      // Установить точку назначения в позиции игрока
14      ThisAgent.SetDestination(PlayerTransform.position);
15
16      // Если игрок потерялся из виду, продолжить преследование
17      if(!CanSeePlayer)
18      {
19        // Сбросить время преследования невидимого игрока
20        float ElapsedTime = 0f;
21
22        // Продолжать преследование
23        while(true)
24        {
25          // Увеличить время
26          ElapsedTime += Time.deltaTime;
27
28          // Установить точку назначения в позиции игрока
29          ThisAgent.SetDestination(PlayerTransform.position);
30
31          // Ждать следующего кадра
32          yield return null;
33
34          // Время истекло?
35          if(ElapsedTime >= ChaseTimeOut)
36          {
37            // Если игрок невидим, остановиться
38            if(!CanSeePlayer)
39            {
40              // Перейти в состояние ожидания
41              StartCoroutine(State_Idle());
42              yield break;
43            }
44            else
45            break; // игрок снова видим
46          }
47        }
48      }
49
50      // Игрок настигнут, начать атаку
51      if(Vector3.Distance(ThisTransform.position,
        PlayerTransform.position) <= DistEps)
52      {
53        // Перейти в состояние атаки
54        StartCoroutine(State_Attack());
55        yield break;
56      }
57
58      // Ждать следующего кадра
59      yield return null;
60    }
61  }
```

Ниже приводится несколько комментариев к листингу 7.9.

- **Строки 17–48**: в этой фазе цикл в `State_Chase` определяет потерю игрока из видимости. Когда это происходит, враг продолжит преследование в течение времени `ChaseTimeOut`. По истечении этого времени вновь будет проверена видимость игрока. Если игрок снова виден, погоня продолжится. В противном случае врага перейдет в состояние `Idle`, то есть, готовности к началу нового патрулирования в поисках игрока.
- **Строки 51–59**: здесь выполняется проверка достижения дистанции атаки (`DistEps`). В этом случае конечный автомат перейдет в состояние `Attack`.

Создание состояния Attack

В состоянии `Attack` враг атакует игрока, пока он виден. После очередной атаки противник должен остановиться, прежде чем начать новую атаку. Единственной причиной выхода из этого состояния является потеря игрока из видимости. Когда это происходит, враг возвращается в состояние преследования `Chase`, а оттуда вновь переходит в состояние `Attack` или в состояние `Idle`, в зависимости от того, был ли найден игрок, как это показано в листинге 7.10.

Листинг 7.10. Реализация состояния атаки

```
// Выполняется, когда объект находится в состоянии атаки
public IEnumerator State_Attack()
{
  // Установить текущее состояние
  CurrentState = AI_ENEMY_STATE.ATTACK;

  // Активировать состояние атаки
  ThisAnimator.SetTrigger((int) AI_ENEMY_STATE.ATTACK);

  // Остановить перемещение агента навигационного меша
  ThisAgent.Stop();

  // Установить таймер задержки между атаками
  float ElapsedTime = 0f;

  // Бесконечный цикл на время пребывания в состоянии атаки
  while(CurrentState == AI_ENEMY_STATE.ATTACK)
  {
    // Обновить таймер
    ElapsedTime += Time.deltaTime;
```

```
    // Проверить, находится ли игрок на дистанции атаки
    if(!CanSeePlayer || Vector3.Distance(ThisTransform.position,
          PlayerTransform.position) > DistEps)
    {
      // Начать преследование
      StartCoroutine(State_Chase());
      yield break;
    }

    // Проверить продолжительность задержки
    if(ElapsedTime >= AttackDelay)
    {
      // Сбросить таймер
      ElapsedTime = 0f;

      // Начать атаку
      PlayerTransform.SendMessage("ChangeHealth", -AttackDamage,
        SendMessageOptions.DontRequireReceiver);
    }

    // Ждать следующего кадра
    yield return null;
  }
}
```

Создание состояния бегства SeekHealth

Состояние `SeekHealth` наступает, когда уровень здоровья врага падает и он должен попытаться восстановить его, найдя аптечку. Переход в это состояние, в отличие от других, может быть выполнен из любого другого состояния. Необходимость этого состояния определяется только уровнем здоровья врага. В частности, переход в это состояние должен быть выполнен, когда здоровье врага уменьшилось ниже минимального порога. Поэтому состояние `SeekHealth` должно быть связано в графе Mecanim с узлом **Any State** (Любое состояние), что позволит запустить анимацию при срабатывании триггера, независимо от текущего состояния, как показано на рис. 7.24.

Каждый вражеский персонаж имеет переменную `Health`, значение которой увеличивается, когда враг находит аптечку, или уменьшается, когда враг подвергается нападению. Ее изменение происходит в методе `ChangeHealth`, и именно здесь должна определяться необходимость перехода в состояние `SeekHealth`. Функция `ChangeHealth` общедоступная; это позволяет вызывать ее с помощью функций `SendMessage` и `BroadcastMessage`, как показано в листинге 7.11.

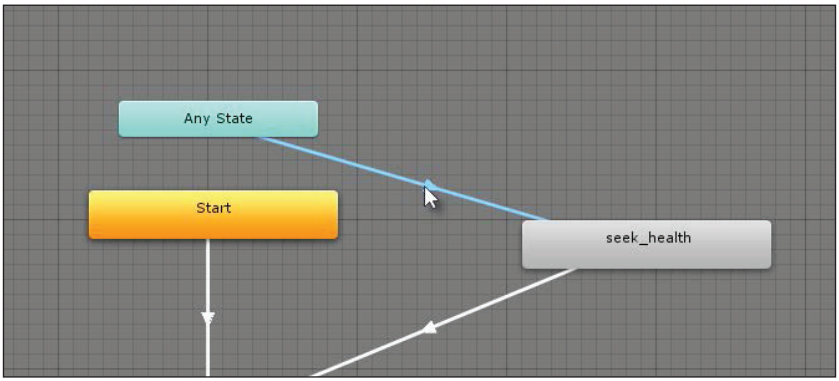


Рис. 7.24. В состояние SeekHealth можно перейти из состояния **Any State**

Листинг 7.11. Реализация метода ChangeHealth

```
// Вызывается при изменении уровня здоровья
public void ChangeHealth(float Amount)
{
  // Изменить уровень здоровья
  Health += Amount;

  // Наступила смерть?
  if(Health <= 0)
  {
    StopAllCoroutines();
    Destroy(gameObject);
    return;
  }

  // Проверить, насколько опасно падение уровня здоровья
  if(Health > HealthDangerLevel) return;

  // Уровень здоровья опасно мал, начать поиск аптечки
  StopAllCoroutines();
  StartCoroutine(State_SeekHealth());
}
```

Метод State_SeekHealth, реализующий состояние поиска аптечки, приводится в листинге 7.12.

Листинг 7.12. Реализация состояния поиска аптечки

```
01 // Выполняется, когда объект находится в состоянии поиска аптечки
02 public IEnumerator State_SeekHealth()
03 {
```

```
04   // Установить текущее состояние
05   CurrentState = AI_ENEMY_STATE.SEEKHEALTH;
06
07   // Активировать состояние поиска аптечки
08   ThisAnimator.SetTrigger((int) AI_ENEMY_STATE.SEEKHEALTH);
09
10   // Ближайшая аптечка
11   HealthRestore HR = null;
12
13   // Бесконечный цикл на время поиска аптечки
14   while(CurrentState == AI_ENEMY_STATE.SEEKHEALTH)
15   {
16     // Если аптечки нет, искать ближайшую
17     if(HR == null) HR = GetNearestHealthRestore(ThisTransform);
18
19     // Аптечка найдена, начать движение к ней
20     ThisAgent.SetDestination(HR.transform.position);
21
22     // Если HR == null, значит аптечек больше нет, бездействовать
23     if(HR == null || Health > HealthDangerLevel)
24     {
25       // Перейти в состояние ожидания
26       StartCoroutine(State_Idle());
27       yield break;
28     }
29
30     // Ждать следующего кадра
31     yield return null;
32   }
33 }
```

Ниже приводится несколько комментариев к листингу 7.12.

- **Строка 17**: обработка состояния `HealthSeek` начинается с поиска ближайшей аптечки и использования ее позиции в качестве пункта назначения для агента. Это в определенном смысле мошенничество, потому что (конечно) без возможности удаленного обзора враг не должен знать, где находится ближайшая аптечка. Однако помните, что значение имеет не то, что враг знает или не знает, а то, как это воспримет игрок. Если игрок не знает об этой логике и не может об этом догадаться, значит это не имеет никакого значения. Также обратите внимание, что, возможно, тот же игрок или другие враги уже подберут аптечку, прежде чем враг придет в пункт назначения. По этой причине в каждом кадре враг должен проверять, доступна ли еще назначенная аптечка, и если нет – выбрать другую ближайшую аптечку.

- **Строка 23**: если не осталось доступных аптечек или уровень здоровья был восстановлен, враг вернется в состояние `Idle`.

Для обработки состояния `SeekHealth` необходимо найти в сцене ближайшую аптечку и вернуть ссылку на нее. Этот поиск реализуется методом `GetNearestHealthRestore`, представлены в листинге 7.13.

Листинг 7.13. Реализация метода GetNearestHealthRestore

```
01 // Ищет аптечку, ближайшую к объекту Target
02 private HealthRestore GetNearestHealthRestore(Transform Target)
03 {
04   // Получить список всех аптечек
05   HealthRestore[] Restores = Object.FindObjectsOfType<HealthRestore>();
06
07   // Расстояние до ближайшей
08   float DistanceToNearest = Mathf.Infinity;
09
10   // Выбранная аптечка
11   HealthRestore Nearest = null;
12
13   // Перебрать все аптечки
14   foreach(HealthRestore HR in Restores)
15   {
16     // Определить расстояние до данной аптечки
17     float CurrentDistance = Vector3.Distance(
         Target.position, HR.transform.position);
18
19     // Если данная аптечка ближе предыдущей, запомнить ее
20     if(CurrentDistance <= DistanceToNearest)
21     {
22       Nearest = HR;
23       DistanceToNearest = CurrentDistance;
24     }
25   }
26
27   // Вернуть ближайшую аптечку или null
28   return Nearest;
29 }
```

Итоги

Весь проект реализации искусственного интеллекта, представленный в этой главе, можно найти в пакете с примерами для данной главы, в папке `ai`. Я рекомендую открыть его и протестировать. Использование контроллера игры от первого лица позволяет игроку перемещаться по уровню, уходить от врагов, а также нападать, когда враги находятся достаточно близко, нажимая на клавишу пробела, как показано на рис. 7.25.

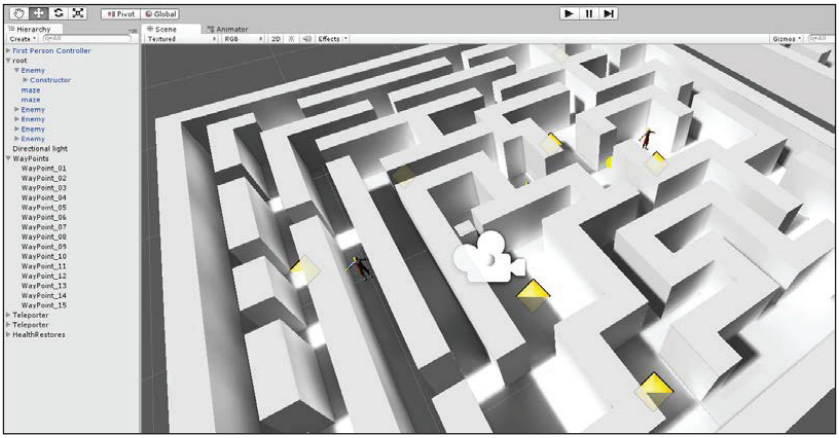


Рис. 7.25. Тестирование класса `AI_Enemy`

Есть много возможностей дальнейшего улучшения проекта. Например, можно добавить несколько типов врагов, создать разные стратегии для каждого типа, от нападения из засады до умения притворяться мертвым, и т. д. Однако, мы прошли долгий путь и создали искусственный интеллект, основанный на конечном автомате C#, а также на конечном автомате Mecanim для воспроизведения анимаций. В следующей главе мы покинем мир искусственного интеллекта и перейдем к рассмотрению настроек редактора, чтобы сделать разработку игр более удобной!

Глава 8

Настройка редактора Unity

Редактор Unity – это мощный, универсальный инструмент для разработки игр. Но иногда при разработке возникают ситуации, когда необходимы некоторые особые функции редактора, которые он не может вам предоставить, связанные с вашими предпочтениями или с особенностями разрабатываемой игры. Например, вам могут понадобиться функции редактирования путей, возможность пакетного переименования, инструменты создания мешей, или что-то еще. В таких случаях вы можете посетить магазин ресурсов Unity для поиска подходящих расширений. Но и там вы можете не найти того, что вам нужно. Тогда можно попытаться настроить или адаптировать редактор, чтобы лучше приспособить его для ваших целей. К счастью, существует много способов настройки Unity как инструмента, и в этой главе мы основное внимание уделим знакомству с конкретными методами такой настройки. Во-первых, мы узнаем, как создать инструмент пакетного переименования **Batch Rename** для переименования нескольких выбранных объектов в одной операции. Во-вторых, как создать поле цветовой гаммы в инспекторе объектов для смешивания двух цветов при помощи ползунка. В-третьих, как отобразить глобальные свойства C# в инспекторе объектов и сделать их доступными для установки и получения значений. И наконец, как использовать атрибуты C# для создания инструмента локализации, который позволит автоматически менять язык (английский, французский и т. д.) всех текстов в игре нажатием одной кнопки.

Пакетное переименование

При создании сцены с многочисленными врагами, бонусами, реквизитами или экземплярами других объектов, как правило, используется функция клонирования объектов (**Ctrl**+**D**). Это приводит к появлению множества объектов с одинаковыми именами. Само по себе

дублирование имен не будет ошибкой, но это неудобно, потому что приводит к отображению в панели иерархии многочисленных объектов с одинаковыми именами, что не позволяет различать объекты по именам. Кроме того, при поиске объекта в сценарии с помощью функции `GameObject.Find` будет получен не нужный вам конкретный объект, а один из объектов, имеющих одинаковое имя. Решить проблему можно, если придать каждому объекту уникальное имя. Но это может занять много времени, особенно если таких объектов очень много. То есть, вам нужен инструмент пакетного переименования.

Теоретически такой инструмент должен позволять выбирать несколько объектов в панели иерархии, а затем автоматически переименовывать их, в соответствии с правилом нумерации. Проблема в том, что Unity изначально не поддерживает такой функции. Но мы можем написать ее код сами, как показано на рис. 8.1.

Для начала создайте папку `Editor` внутри проекта. Это важно. Папка `Editor` является специальной папкой, предназначенной для размещения всех сценариев настройки редактора. То есть, если вы планируете настраивать редактор Unity, убедитесь, что все сценарии с настройками находятся в папке `Editor`. Не имеет значения, сколько в вашем проекте папок с именем `Editor`, имеет значение только наличие хотя бы одной папки `Editor` и сценария настройки в ней, как показано на рис. 8.2.

Далее создадим утилиту пакетного переименования `BatchRename` в виде потомка класса `ScriptableWizard`. Этот класс считается предком для всех классов настроек редактора. Все порожденные от него классы действуют как диалоги, которые можно вызывать из главного меню Unity. Окно любого такого диалога будет содержать набор вариантов, выбираемых пользователем перед нажатием кнопки подтверждения, которая запускает выполнение операции. Другими словами, классы, наследующие `ScriptableWizard`, идеально подходят для выполнения разовых операций над одним или несколькими объектами.

Более подробную информацию о классе `ScriptableWizard` можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/ScriptableWizard.html.

В листинге 8.1 приводится полная реализация утилиты пакетного переименования.

Листинг 8.1. Утилита пакетного переименования

```
01 //-----------------------------------
02 using UnityEngine;
03 using UnityEditor;
04 using System.Collections;
```

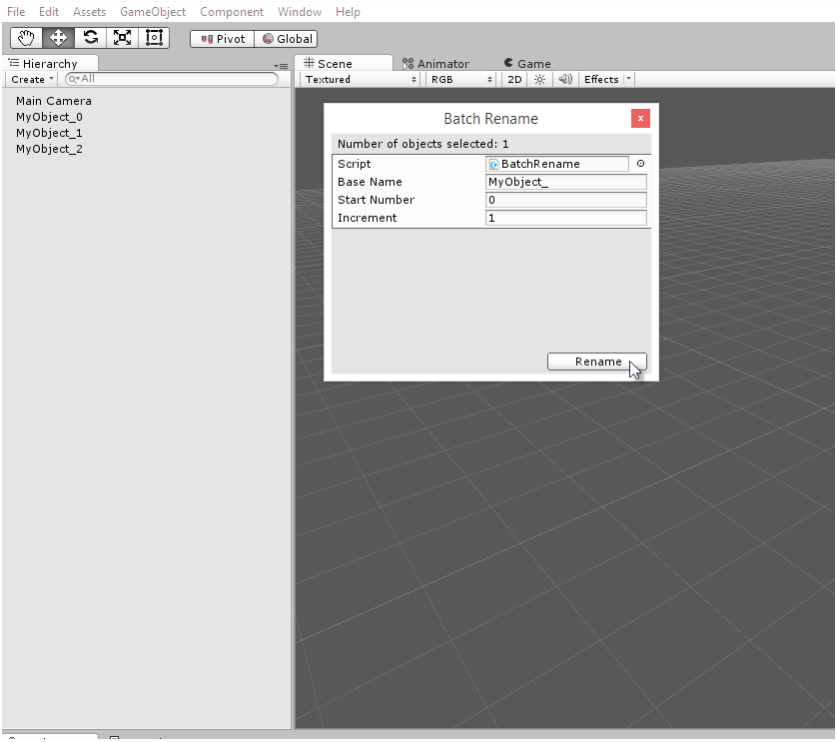


Рис. 8.1. Создание расширения пакетного переименования

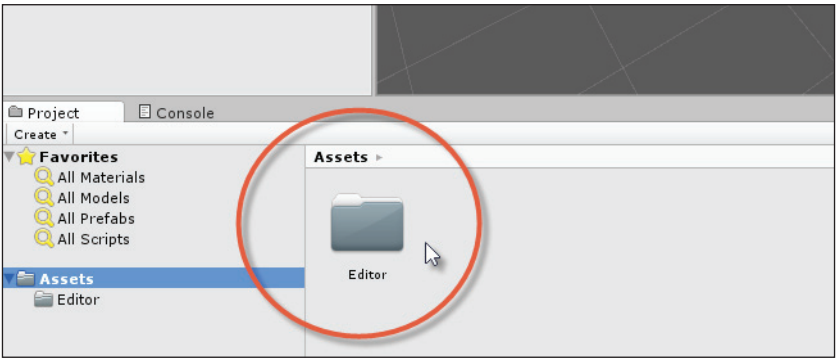


Рис. 8.2. Создание папки `Editor`
для всех сценариев настройки редактора

```
05 //------------------------------------
06 public class BatchRename : ScriptableWizard
07 {
08   // Базовое имя
09   public string BaseName = "MyObject_";
10
11   // Начальный номер
12   public int StartNumber = 0;
13
14   // Шаг
15   public int Increment = 1;
16
17   [MenuItem("Edit/Batch Rename...")]
18   static void CreateWizard()
19   {
20     ScriptableWizard.DisplayWizard(
       "Batch Rename",typeof(BatchRename),"Rename");
21   }
22   //------------------------------------
23   // Вызывается при первом появлении окна
24   void OnEnable()
25   {
26     UpdateSelectionHelper();
27   }
28   //------------------------------------
29   // Вызывается, когда изменяется область выбора в сцене
30   void OnSelectionChange()
31   {
32     UpdateSelectionHelper();
33   }
34   //------------------------------------
35   // Изменяет счетчик выбранных объектов
36   void UpdateSelectionHelper()
37   {
38     helpString = "";
39
40     if (Selection.objects != null)
41       helpString = "Number of objects selected: " +
         Selection.objects.Length;
42   }
43   //------------------------------------
44   // Переименование
45   void OnWizardCreate()
46   {
47     // Если ничего не выбрано, выйти
48     if (Selection.objects == null)
49       return;
50
51     // Текущий шаг
```

```
52      int PostFix = StartNumber;
53
54      // Цикл переименования
55      foreach(Object O in Selection.objects)
56      {
57        O.name = BaseName + PostFix;
58        PostFix += Increment;
59      }
60    }
61    //-------------------------------------
62 }
63 //-------------------------------------
```

Ниже приводится несколько комментариев к листингу 8.1:

- **Строка 03**: расширение редактора должно подключать пространство имен `UnityEditor`, чтобы получить доступ к классам и объектам редактора.
- **Строка 06**: класс `BatchRename` наследует не класс `MonoBehaviour`, как большинство классов в файлах сценариев, а класс `ScriptableWizard`. Наследники класса `ScriptableWizard` представляют собой независимые утилиты Unity, которые могут быть вызываться из меню приложения.
- **Строки 17–21**: атрибут `MenuItem` служит префиксом функции `CreateWizard`. Он определяет пункт в меню приложения и вызывает функцию `CreateWizard` для отображения окна **Batch Rename** (Пакетное переименование).
- **Строки 8–16**: вызов `CreateWizard` выведет окно `BatchRename`. В нем будут содержаться редактируемые поля для всех общедоступных переменных класса (в нашем случае это поля **Base Name** (Базовое имя), **Start Number** (Начальный номер) и **Increment** (Шаг)).
- **Строки 45–60**: функция `OnWizardCreate` вызывается как событие, когда пользователь щелкает на кнопке **Rename** (Переименовать) в окне **Batch Rename** (Пакетное переименование). Имя кнопки **Rename** в данном случае задано в строке 20. Функция `OnWizardCreate` перебирает все выбранные объекты, если таковые имеются, и последовательно переименовывает их в соответствии со значениями полей **Base Name** (Базовое имя), **Start Number** (Начальный номер) и **Increment** (Шаг), как показано на рис. 8.3.

Чтобы воспользоваться инструментом пакетного переименования, просто выберите группу объектов в сцене, а затем выберите пункт **Edit** ⇒ **Batch Rename** (Правка ⇒ Пакетное переименование) в меню

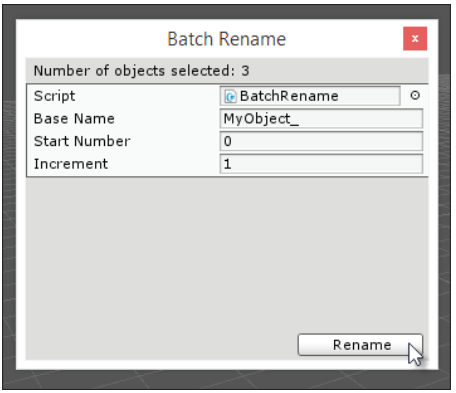


Рис. 8.3. Инструмент **Batch Rename** (Пакетное переименование)

приложения. Значение поля **Base Name** (Базовое имя) определяет строку, которая должна стать префиксом для всех имен объектов, а значение поля **Increment** (Шаг) определяет величину изменения счетчика перед добавлением к базовому имени. Значение поля **Start Number** (Начальный номер) служит начальным значением счетчика, как показано на рис. 8.4.

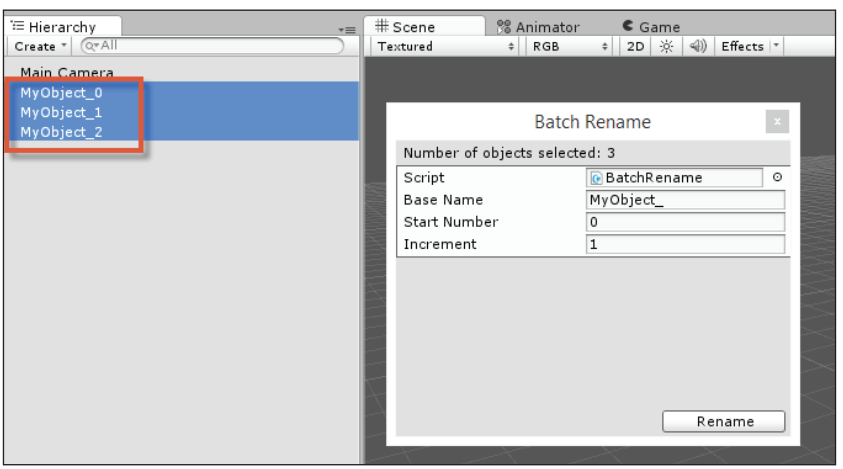


Рис. 8.4. Переименование объектов с помощью инструмента **Batch Rename** (Пакетное переименование)

Атрибуты C# и рефлексия

С этого момента в данной главе все расширения редактора будут основываться на понятиях атрибутов и рефлексии. Эти понятия не являются специальными понятиями для Unity, это более общие понятия в информатике, программировании, и применяются в таких языках, как C#, а также в фреймворке .NET. Прежде чем приступить к следующему расширению, рассмотрим атрибуты и связанное с ними понятие рефлексии на примере атрибута `Range`, встроенного в Unity. Взгляните на следующую строку кода:

```
public float MyNumber = 0;
```

Эта общедоступная переменная будет отображаться в инспекторе объектов в виде поля редактирования, что позволит пользователю ввести в него любое действительное число, установив, таким образом, значение переменной `MyNumber`, как показано на рис. 8.5.

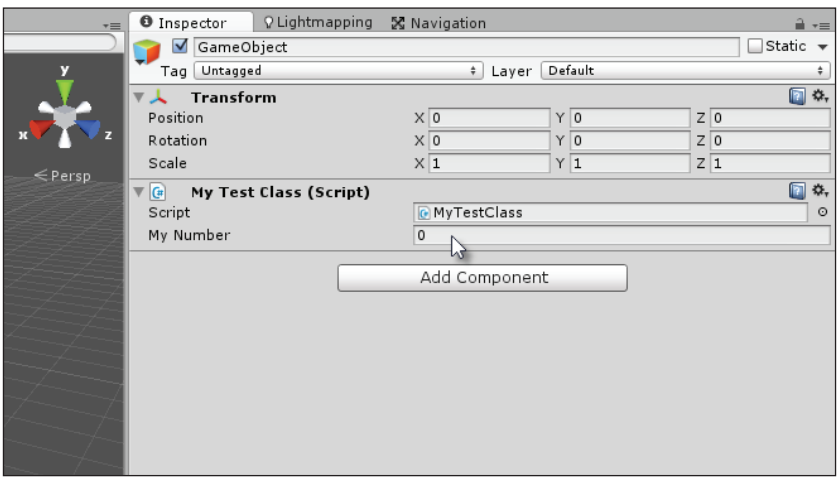


Рис. 8.5. Ввод действительного числа в инспекторе объектов

Этого достаточно для многих ситуаций, но иногда желательно иметь возможность проверять ввод на принадлежность числа некоторому диапазону значений. Вы можете сделать это в коде, используя функцию `Mathf.Clamp`, но есть возможность проверить ввод, используя атрибут. Вы можете прикрепить атрибут `Range` к вещественной пере-

менной (MyNumber), чтобы отобразить ползунок в окне редактирования, как показано ниже:

```
[Range(0f,1f)]
public float MyNumber = 0;
```

Более подробную информацию об атрибутах можно найти в электронной документации Unity по адресу http://unity3d.com/learn/tutorials/modules/intermediate/scripting/attributes.

После того как этот код будет скомпилирован, поле переменной MyNumber в инспекторе объектов будет выглядеть иначе – появится диапазон чисел между 0 и 1, как показано на рис. 8.6. Обратите внимание, что все числа, задаваемые в атрибуте **Range** в качестве аргументов, должны иметь явные значения, известные на момент компиляции, и не могут быть выражениями, зависящими от переменных, значения которых могут измениться во время выполнения. Все значения атрибутов должны быть известны на момент компиляции.

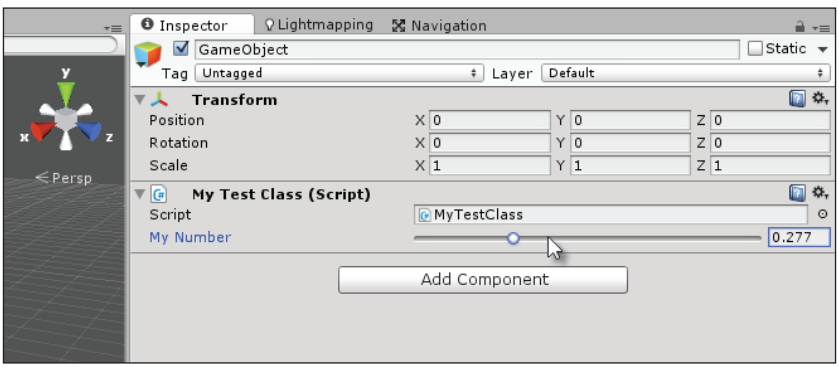


Рис. 8.6. Использование атрибутов для настройки отображения в инспекторе объектов

Так как же действуют атрибуты? Если коротко, атрибуты являются разновидностью метаданных, они действуют как метки. Программисты могут прикрепить атрибут к классу, переменной или методу, чтобы связать с ними данные, которые станут известны компилятору. Сам атрибут имеет только описательный характер, он ничего не делает, это просто данные. Польза атрибутов в том, что код, построенный на основе фреймворка .NET (или Mono), имеет возможность «взглянуть на себя со стороны», то есть просмотреть все классы, типы

данных и экземпляры, имеющиеся в программе. Для каждого объекта в программе могут быть запрошены и просмотрены его метаданные (атрибуты). Эта способность программы «взглянуть на себя со стороны» называется рефлексией (или отражением), это то же самое, что посмотреть на себя в зеркало. Конечно, программа видит себя не искаженно, в обратном отражении, а правильно, в том числе и все свои метаданные. Для быстрого знакомства с рефлексией рассмотрим следующий пример в листинге 8.2. Этот код перечисляет в цикле все классы проекта, во всех исходных файлах. Обратите внимание, что он перечисляет не только все экземпляры классов в сцене, но и сами классы (то есть, образно выражаясь, делает ксерокопию).

Листинг 8.2. Обход всех классов в проекте

```
01 using UnityEngine;
02 using System.Collections;
03 using System.Reflection;
04 using System;
05
06 public class MyTestScript : MonoBehaviour
07 {
08   // Этот метод выполняет инициализацию
09   void Start ()
10   {
11     // Обойти все классы в сборке
12     foreach(Type t in Assembly.GetExecutingAssembly().GetTypes())
13     {
14       Debug.log (t.Name);
15     }
16   }
17 }
```

Ниже приводится несколько комментариев к листингу 8.2:

- **Строки 03–04**: необходимо подключить пространства имен `System` и `System.Reflection`, потому что они содержат классы и объекты, необходимые для доступа к механизму рефлексии в .NET.
- **Строка 12**: этот цикл `foreach` выполняет обход всех классов (типов) в активной сборке (то есть в скомпилированном коде, в том числе и во всех файлах сценариев, созданных пользователем).

Вы можете сделать еще шаг вперед и, например, вместо перечисления всех типов, как показано в листинге 8.2, перечислить методы, свойства и переменные (поля) определенного типа. Рассмотрим следующий пример в листинге 8.3, который перечисляет все общедоступные переменные указанного типа.

Листинг 8.3. Обход всех общедоступных переменных

```
// Перечисляет все общедоступные переменные класса t
public void ListAllPublicVariables(Type t)
{
  // Обойти все общедоступные переменные
  foreach(FieldInfo FI in t.GetFields(
          BindingFlags.Public | BindingFlags.Instance)
  {
    // Вывести имя переменной
    Debug.log (FI.Name);
  }
}
```

Более подробную информацию о поразрядных операциях, используемых в этом примере, можно найти по адресу http://www.blackwasp.co.uk/CSharpLogicalBitwiseOps.aspx.

И, самое важное, можно перечислить атрибуты, назначенные типу. Это позволит получить метаданные типа и просмотреть его свойства во время выполнения, как показано в листинге 8.4.

Листинг 8.4. Обход атрибутов типа

```
01 public void ListAllAttributes(Type t)
02 {
03   foreach(Attribute attr in t.GetCustomAttributes(true))
04   {
05     // Обойти все найденные атрибуты
06     Debug.log (attr.GetType());
07   }
08 }
```

Листинг 8.4 демонстрирует возможность получения всех атрибутов для заданного типа данных во время выполнения. Это значит, что типы данных и переменные могут иметь метаданные, связанные с ними, которые можно извлечь и использовать для принятия решений, касающихся обработки этих объектов. Это мощное средство для создания расширений редактора, потому что, создавая собственные атрибуты, которые можно присоединять к типам данных и переменным, мы можем интегрировать наш код с редактором Unity без изменения его логической структуры и не нарушая его работы. То есть мы сможем, пометив переменные атрибутами, настроить их отображение в редакторе Unity без нарушения его логики и структуры. Далее мы увидим, как создать пользовательские атрибуты для настройки редактора.

Смешивание цветов

Атрибут `Range`, рассматривавшийся выше, можно присоединить к целому или действительному числу при его объявлении, чтобы ограничить диапазон значений, доступных для ввода в редакторе Unity. При этом редактируемое поле в редакторе Unity будет заменено ползунком, ограничивающим диапазон допустимых значений переменной. Это, конечно, никак не повлияет на присваивание переменным значений в программном коде. Во время выполнения атрибут `Range` не имеет никакой силы. Атрибут `Range` определяет представление общедоступных переменных в инспекторе объектов и управляет вводом их значений пользователем. За кулисами класс `Editor` получает объект данных `Attribute` с помощью механизма рефлексии, чтобы определить порядок отображения в инспекторе объектов.

Атрибут `Range` хорошо подходит для чисел. Но было бы здорово распространить его действие и на другие типы данных. Например, часто применяемый эффект смены цветов, такой как выцветание от черного до полной прозрачности, с целью создания эффектов плавного появления и постепенного растворения при смене сцен. Это называется линейной интерполяцией цветов. Промежуточный цвет генерируется как смесь двух цветов с использованием действительного значения нормализации, задаваемого значением ползунка (от 0 до 1).

Соответствующее свойство `Inspector` для этого типа данных могло бы управляться ползунком, по аналогии с атрибутом `Range`, и осуществлять интерполяцию цвета в диапазоне от `0` до `1`, как показано на рис. 8.7.

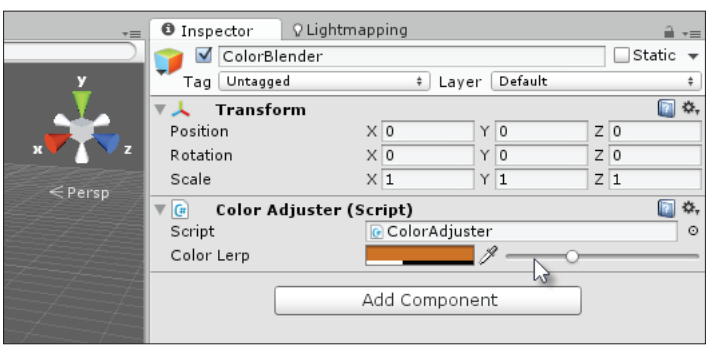


Рис. 8.7. Линейная интерполяция двух цветов

В сущности, нам нужно настроить редактор так, чтобы всякий раз, когда в сцене выбирается объект, содержащий общедоступную переменную определенного типа, он нужным образом отображал эту переменную в инспекторе объектов. Это позволит добавлять свои элементы управления в инспектор объектов и проверять вводимые в нем данные. Чтобы начать реализацию этой задачи, создадим пользовательский класс и определим все данные для смешивания цветов. Нам потребуются четыре переменные. Две переменные для граничных цветов `SourceColor` и `DestColor`. Переменная `BlendFactor` для действительного числа в диапазоне между `0` и `1` (начало и конец), определяющего промежуточный цвет. И, наконец, переменная для результата `BlendedColor`. Полное определение класса приводится в листинге 8.5.

Листинг 8.5. Класс смешивания цветов

```
[System.Serializable]
public class ColorBlend : System.Object
{
  public Color SourceColor = Color.white;
  public Color DestColor = Color.white;
  public Color BlendedColor = Color.white;
  public float BlendFactor = 0f;
}
```

Так как класс `ColorBlend` имеет атрибут `[System.Serializable]`, Unity автоматически отображает класс и его общедоступные члены в инспекторе объектов. По умолчанию будут отображены все общедоступные члены класса `ColorBlend`, а поле `BlendFactor` будет представлено в виде поля ввода чисел без проверки принадлежности диапазону от 0 до 1, как показано на рис. 8.8.

Давайте теперь начнем настройку отображения этого класса в инспекторе объектов. Создадим новый класс атрибута `ColorRangeAttribute`, как показано в листинге 8.6.

Листинг 8.6. Класс атрибута ColorRangeAttribute

```
01 public class ColorRangeAttribute : PropertyAttribute
02 {
03   //-------------------------------------------------------------
04   public Color Min;
05   public Color Max;
06   //-------------------------------------------------------------
07   public ColorRangeAttribute(float r1, float g1, float b1, float a1,
08                              float r2, float g2, float b2, float a2)
09   {
```

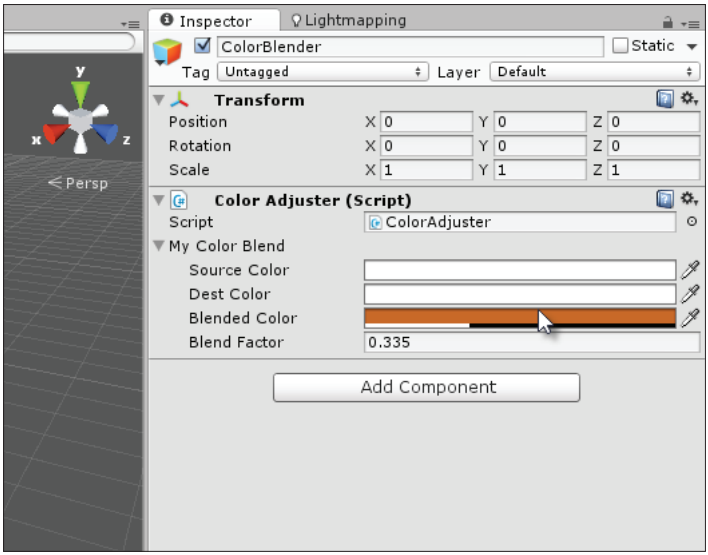


Рис. 8.8. Отображение класса `Color Adjuster` по умолчанию и изменение его свойств

```
10      this.Min = new Color(r1, g1, b1, a1);
11      this.Max = new Color(r2, g2, b2, a2);
12    }
13    //-------------------------------------------------------------
14 }
```

Ниже приводится несколько комментариев к листингу 8.6:

- **Строка 01**: класс `ColorRangeAttribute` определяет структуру метаданных, которой можно отмечать другие типы данных. Обратите внимание, что он наследует класс `PropertyAttribute`. Это значит, кроме всего остального, что `ColorRangeAttribute` является, прежде всего, структурой атрибутов и метаданных, а не обычным классом. Он не предназначен для создания экземпляров, как стандартный класс.
- **Строка 07**: атрибут имеет функцию-конструктор, который принимает восемь вещественных чисел, определяющих каналы RGBA исходного и конечного цветов. Они будут использоваться только при присоединении атрибута к переменной.

Теперь мы напишем класс, объявив экземпляр `ColorBlend` с атрибутом `ColorRangeAttribute`. Даже сейчас добавление атрибута `ColorRange-`

Attribute, по сути, ничего не меняет, потому что класс редактора для его обработки еще не написан. Мы можем увидеть это в следующем коде:

```
public class ColorAdjuster : MonoBehaviour
{
  [ColorRangeAttribute(1f,0f,0f,0f, 0f,1f,0f,1f)]
  public ColorBlend MyColorBlend;
}
```

Создание класса Editor для отображения ColorBlend в инспекторе объектов в виде ползунка включает обработку класса ColorRangeAttribute. В частности, Unity предлагает расширяемый базовый класс PropertyDrawer, от которого можно наследовать новые классы и переопределять в них способ отображения любых атрибутов в инспекторе объектов. Проще говоря, класс PropertyDrawer позволяет настроить прорисовку инспектора для всех переменных, отмеченных общим атрибутом. Итак, создайте в папке Editor проекта новый класс ColorRangeDrawer, как показано в листинге 8.7.

Листинг 8.7. Класс ColorRangeDrawer

```
01 using UnityEngine;
02 using UnityEditor; //Be sure to include UnityEditor for all
   extension classes
03 using System.Collections;
04 //-------------------------------------------------------------
05 // Атрибут CustomPropertyDrawer, переопределяющий
      отображение всех членов ColorRangeAttribute
06 [CustomPropertyDrawer(typeof(ColorRangeAttribute))]
07 public class ColorRangeDrawer : PropertyDrawer
08 {
09   //-------------------------------------------------------------
10   // Событие вызывается редактором Unity для
        отображения элементов пользовательского интерфейса
11   public override void OnGUI (Rect position,
     SerializedProperty property, GUIContent label)
12   {
13     // Получить атрибут диапазона цветов
14     ColorRangeAttribute range = attribute as ColorRangeAttribute;
15
16     // Добавить подпись в инспектор объектов
17     position = EditorGUI.PrefixLabel (position,
         new GUIContent ("Color Lerp"));
18
19     // Определить размеры цветных прямоугольников и ползунков
20     Rect ColorSamplerRect = new Rect(position.x, position.y,
       100, position.height);
```

```
21      Rect SliderRect = new Rect(position.x+105, position.y,
        200, position.height);
22
23      // Вывести цветные прямоугольники
24      EditorGUI.ColorField(ColorSamplerRect,
        property.FindPropertyRelative("BlendedColor").colorValue);
25
26      // Вывести ползунок
27      property.FindPropertyRelative("BlendFactor").floatValue =
        EditorGUI.Slider(SliderRect,
        property.FindPropertyRelative("BlendFactor").floatValue, 0f, 1f);
28
29      // Обновить результат смешивания в зависимости от положения ползунка
30      property.FindPropertyRelative("BlendedColor").colorValue =
        Color.Lerp(range.Min, range.Max,
        property.FindPropertyRelative("BlendFactor").floatValue);
31    }
32    //---------------------------------------------------------------
33  }
34  //---------------------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 8.7:

- **Строка 01**: атрибут `CustomPropertyDrawer` используется, чтобы связать класс `PropertyDrawer` с атрибутом `ColorRangeAttribute`. Редактор Unity применяет эти метаданные для определения типов, требующих особого отображения в инспекторе объектов. В данном случае все члены `ColorRangeAttribute` будут прорисованы вручную, функцией `OnGUI` класса `PropertyDrawer`.
- **Строка 11**: переопределяется функция `OnGUI` базового класса, реализующая отображение всех полей атрибута `ColorRangeAttribute` в инспекторе объектов. Класс `EditorGUI` – это вспомогательный класс редактора Unity, предназначенный для прорисовки элементов интерфейса, таких как кнопки, текстовые поля и ползунки. Более подробную информацию о классе `EditorGUI` можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/EditorGUI.html.
- **Строка 14**: функция `OnGUI` вызывается один раз для каждого уникального члена, отображаемого в инспекторе объектов, возможно много раз в секунду. Здесь извлекаются данные атрибута `ColorRangeAttribute`, что дает нам доступ непосредственно ко всем членам текущего отображаемого объекта. Чтобы получить доступ к переменным самого объекта (для чтения и записи), необходимо использовать аргумент `SerializedProperty`, как, например, в методе `FindPropertyRelative`. Более подробную

информацию можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/SerializedProperty.html.

- **Строка 24**: здесь и далее с помощью функции `FindPropertyRelative` извлекаются общедоступные переменные `SourceColor`, `DestColor` и `BlendedColor` выбранного объекта, и устанавливаются их значения, в зависимости от позиции ползунка.

Более подробную информацию о классе `PropertyDrawer` можно найти в электронной документации Unity по адресу http://docs.unity3d.com/Manual/editor-PropertyDrawers.html.

В листинге 8.7 переопределяется прорисовка в инспекторе объектов любых экземпляров `ColorBlend`, отмеченных атрибутом `ColorRangeAttribute`. Это обеспечивает простой и удобный в использовании способ создания смешанных цветов. Помните, что начальный и конечный цвета можете сделать общедоступными, для отображения их в инспекторе объектов, как показано на рис. 8.9.

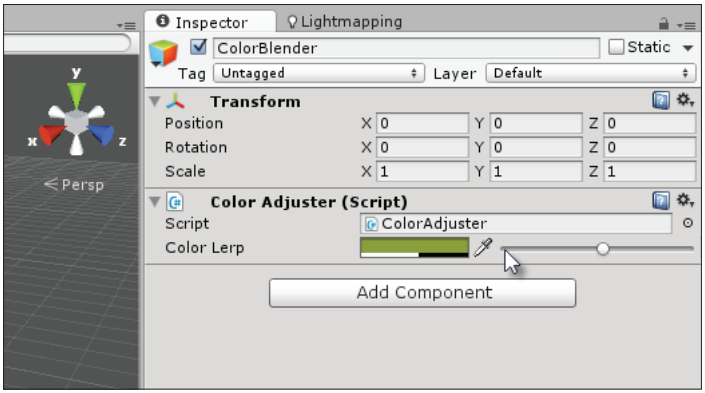


Рис. 8.9. Создание отображения `ColorBlender` для класса `ColorBlend`

Отображение свойств

По умолчанию инспектор объектов отображает все общедоступные переменные класса, но в режиме **Debug** будут также отображаться локальные переменные. Кроме того, локальные переменные отображаются, если отметить их атрибутом `SerializeField` (см. рис. 8.10).

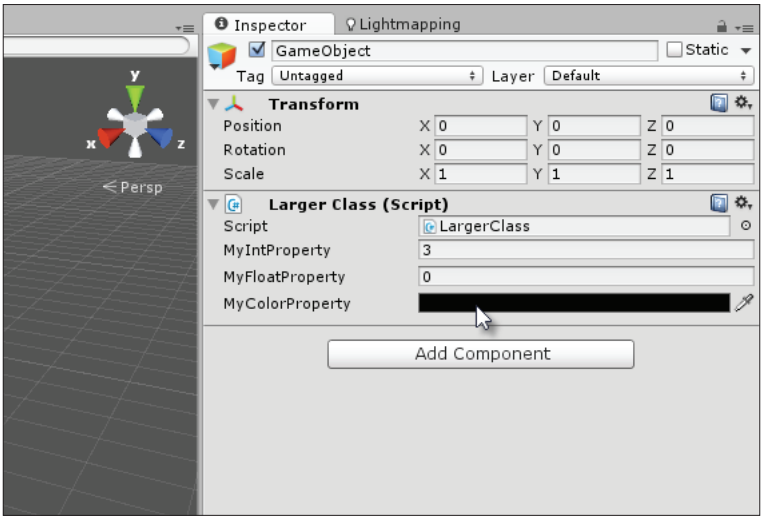


Рис. 8.10. Доступ к свойствам из инспектора объектов

Однако свойства C# никогда не отображаются в инспекторе объектов ни по умолчанию, ни в режимах **Release** или **Debug**. Как уже говорилось в главе 1 «Основы C# Unity», свойства C# реализованы как функции доступа к переменным. Они могут выполнять проверку в каждой операции `get` и `set`, потому что каждая операция `get` и `set` влечет за собой вызов внутренней функции. Тем не менее, независимо от ограничений инспектора объектов Unity, можно написать расширение редактора, которое обеспечит отображение всех свойств класса в инспекторе объектов и позволит получать и устанавливать значения непосредственно. Об этом подробно рассказывается в данном разделе. И снова мы будем опираться на механизм рефлексии.

Более подробную информацию о классе `SerializeField` можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/SerializeField.html.

Взгляните на пример в листинге 8.8, который содержит несколько свойств.

Листинг 8.8. Класс с несколькими свойствами

```
//-----------------------------------------------
using UnityEngine;
using System.Collections;
//-----------------------------------------------
```

```
[System.Serializable]
public class ClassWithProperties : System.Object
{
  // Класс с несколькими свойствами
  //-----------------------------------------------
  public int MyIntProperty
  {
    get{return _myIntProperty;}

    // Выполнить проверку присваиваемых значений
    set{if(value <= 10)_myIntProperty = value;else _myIntProperty=0;}
  }
  //-----------------------------------------------
  public float MyFloatProperty
  {
    get{return _myFloatProperty;}
    set{_myFloatProperty = value;}
  }
  //-----------------------------------------------
  public Color MyColorProperty
  {
    get{return _myColorProperty;}
    set{_myColorProperty = value;}
  }
  //-----------------------------------------------
  // Локальные члены
  private int _myIntProperty;
  private float _myFloatProperty;
  private Color _myColorProperty;
  //-----------------------------------------------
}
//-------------------------------------------------
```

Этот класс будет использоваться внутри другого класса в качестве общедоступной переменной, как показано в листинге 8.9.

Листинг 8.9. Использование экземпляра класса в качестве общедоступной переменной

```
using UnityEngine;
using System.Collections;

public class LargerClass : MonoBehaviour
{
  public ClassWithProperties MyPropClass;
}
```

По умолчанию общедоступный член `MyPropClass` (хотя и отмеченный как `System.Serializable`) не отображает своих членов в инспекторе объектов. Это связано с тем, что для свойств C# не предусмотрена встроенная поддержка (см. рис. 8.11).

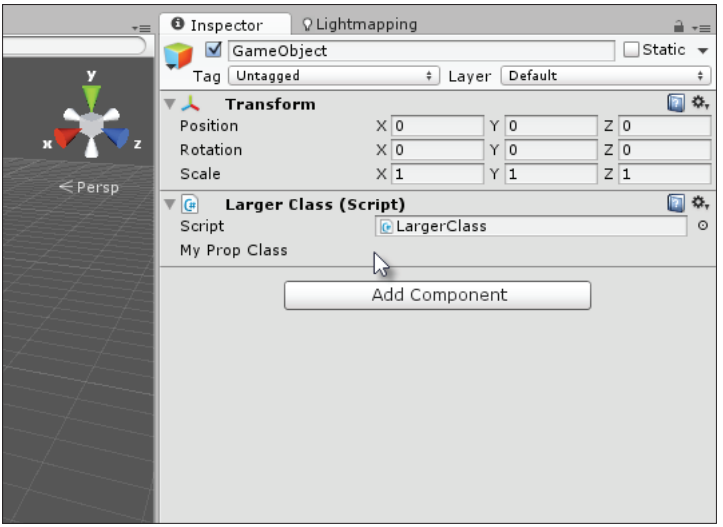


Рис. 8.11. По умолчанию инспектор объектов не отображает свойства C#

Чтобы решить эту проблему, можно вернуться к классу PropertyDrawer, связав его на этот раз с определенным классом, а не с атрибутом, как показано в листинге 8.10.

Листинг 8.10. Вспомогательный класс для отображения общедоступных свойств

```
// Вспомогательный класс для отображения общедоступных свойств
//-----------------------------------------------
using UnityEngine;
using UnityEditor;
using System.Collections;
using System.Reflection;
//-----------------------------------------------
[CustomPropertyDrawer(typeof(ClassWithProperties))]
public class PropertyLister : PropertyDrawer
{
  // Высота панели инспектора объектов
  float InspectorHeight = 0;

  // Высота одной строки в пикселях
  float RowHeight = 15;

  // Промежуток между строками
```

```
18  float RowSpacing = 5;
19
20  // Отобразить свойство в пределах заданного прямоугольника
21  public override void OnGUI(Rect position,
    SerializedProperty property, GUIContent label)
22  {
23    EditorGUI.BeginProperty(position, label, property);
24
25    // Получить объект по ссылке
26    object o = property.serializedObject.targetObject;
27    ClassWithProperties CP =
      o.GetType().GetField(property.name).GetValue(o) as
      ClassWithProperties;
28
29    int indent = EditorGUI.indentLevel;
30    EditorGUI.indentLevel = 0;
31
32    // Разметка
33    Rect LayoutRect = new Rect(position.x, position.y,
      position.width, RowHeight);
34
35    // Найти все свойства объекта
36    foreach(var prop in
      typeof(ClassWithProperties).GetProperties(
      BindingFlags.Public | BindingFlags.Instance))
37    {
38      // Если целочисленное свойство
39      if(prop.PropertyType.Equals(typeof(int)))
40      {
41        prop.SetValue(CP, EditorGUI.IntField(
          LayoutRect, prop.Name, (int)prop.GetValue(CP,null)), null);
42        LayoutRect = new Rect(LayoutRect.x,
          LayoutRect.y + RowHeight+RowSpacing,
          LayoutRect.width, RowHeight);
43      }
44
45      // Если вещественное свойство
46      if(prop.PropertyType.Equals(typeof(float)))
47      {
48        prop.SetValue(CP, EditorGUI.FloatField(LayoutRect,
          prop.Name, (float)prop.GetValue(CP,null)), null);
49        LayoutRect = new Rect(LayoutRect.x,
          LayoutRect.y + RowHeight+RowSpacing, LayoutRect.width, RowHeight);
50      }
51
52      // Если свойство хранит значение цвета
53      if(prop.PropertyType.Equals(typeof(Color)))
54      {
55        prop.SetValue(CP, EditorGUI.ColorField(LayoutRect,
          prop.Name, (Color)prop.GetValue(CP,null)), null);
```

```
56          LayoutRect = new Rect(LayoutRect.x,
            LayoutRect.y + RowHeight+RowSpacing, LayoutRect.width, RowHeight);
57        }
58      }
59
60      // Обновить высоту инспектора
61      InspectorHeight = LayoutRect.y-position.y;
62
63      EditorGUI.indentLevel = indent;
64      EditorGUI.EndProperty();
65    }
66    //-----------------------------------------------
67    // Возвращает высоту поля в пикселях
68    // Это необходимо, чтобы избежать наложения полей друг на друга
69    public override float GetPropertyHeight (
      SerializedProperty property, GUIContent label)
70    {
71      return InspectorHeight;
72    }
73    //-----------------------------------------------
74 }
75 //-----------------------------------------------
```

Ниже приводится несколько комментариев к листингу 8.10:

- **Строка 08**: обратите внимание, что атрибут `CustomPropertyDrawer` теперь связан с обычным классом, а не с атрибутом. В этом случае настраивается отображение конкретного класса, а не всех свойств разных типов, имеющих общий атрибут.
- **Строки 12–18**: объявляются некоторые общедоступные члены, необходимые для вычисления высоты (в пикселях) одной строки в инспекторе объектов. По умолчанию инспектор объектов выделяет одну строку для отображения и все отображаемые поля должны вписаться в эту область. Если общая высота отображения превышает высоту одной строки, все дополнительные элементы управления и данные будут перекрываться и смешиваться с органами управления и виджетами, расположенными ниже. Чтобы решить эту проблему, необходимо использовать функцию `GetPropertyHeight` (строка 69) для возврата высоты отображаемой области в пикселях.
- **Строки 26–27**: эти строки имеют особое значение. Они используют механизм рефлексии для получения ссылки правильного типа на экземпляр `ClassWithProperties`, подготавливаемый для текущего вызова `OnGUI`. В частности, извлекается ссылка на `targetObject` (выбранный объект), а затем из него извлекается экземпляр `ClassWithProperties`. В результате мы получаем прямой доступ к объекту `ClassWithProperties`.

- **Строки 37–58**: цикл по всем общедоступным свойствам объекта и отображение в инспекторе объектов свойств поддерживаемых типов, при этом обеспечивается их чтение и запись, если само свойство поддерживает оба метода.

На рис. 8.12 показано, как выглядят свойства C# в инспекторе объектов.

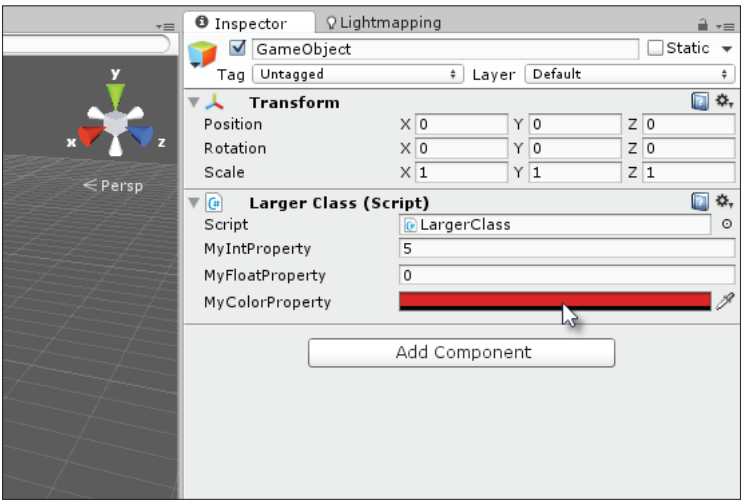


Рис. 8.12. Доступ к свойствам C#

Локализация

Возможно, одним из самых недооцененных и слабо документированных аспектов разработки игр является локализация. К ней относится широкий спектр технических, экономических и лингвистических мер, которые разработчик предпринимает для поддержки нескольких языков в играх, например английского, французского, немецкого, испанского, эсперанто и т. д. Технически цель состоит не столько в поддержке конкретного языка, сколько в создании инфраструктуры, которая может поддержать любой язык, добавленный в любое время, сейчас или позже. Вся область локализации и ее роль в разработке выходят за рамки этой книги, но здесь мы рассмотрим один из способов настройки редактора Unity для ускорения и облегчения процесса локализации. Например, рассмотрим следующий XML-файл, в котором надписи для кнопок главного меню игры определены на

английском и несуществующем языке Йода:

```
<?xml version="1.0"?>
<text>
  <language id="english">
    <text_entry id="text_01"><![CDATA[new game]]></text_entry>
    <text_entry id="text_02"><![CDATA[load game]]></text_entry>
    <text_entry id="text_03"><![CDATA[save game]]></text_entry>
    <text_entry id="text_04"><![CDATA[exit game]]></text_entry>
  </language>
  <language id="yoda">
    <text_entry id="text_01"><![CDATA[new game, you start]]></text_entry>
    <text_entry id="text_02"><![CDATA[load game, you will]]></text_entry>
    <text_entry id="text_03"><![CDATA[game save, you have]]></text_entry>
    <text_entry id="text_04"><![CDATA[leave now, you must]]></text_entry>
  </language>
</text>
```

Обратите внимание, что все пользовательские текстовые узлы заключены в элемент `CDATA`, чтобы можно было использовать любые знаки и символы. Более подробную информацию о `CDATA` можно найти по адресу http://www.w3schools.com/xml/xml_cdata.asp.

В этом файле определяются четыре текстовых элемента, по одному для каждой кнопки меню пользовательского интерфейса. Каждому текстовому элементу присваивается уникальный идентификатор: `text_01`, `text_02`, `text_03` и `text_04`. Эти идентификаторы однозначно определяют каждый элемент текста и будут одинаковыми для всех поддерживаемых языков. Цель состоит в том, чтобы импортировать текстовый файл в Unity, что позволит разработчику переключаться между языками нажатием кнопки, при этом все соответствующие текстовые элементы игры будут меняться автоматически, в соответствии с выбранным языком. Давайте посмотрим, как это работает.

Сначала импортируйте локализованный текст в папку `Resources` проекта. Создайте папку с именем `Resources` и импортируйте локализованный текстовый файл в нее, как показано на рис. 8.13. Теперь любой объект или класс сможет загрузить или открыть текстовый файл с помощью функции `Resources.Load`, как будет показано ниже.

Более подробную информацию о ресурсах можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/Resources.html.

Импортированный текстовый файл содержит текстовые данные, каждый элемент которых связан с определенным идентификатором. То есть, каждое строковое значение привязано к идентификатору,

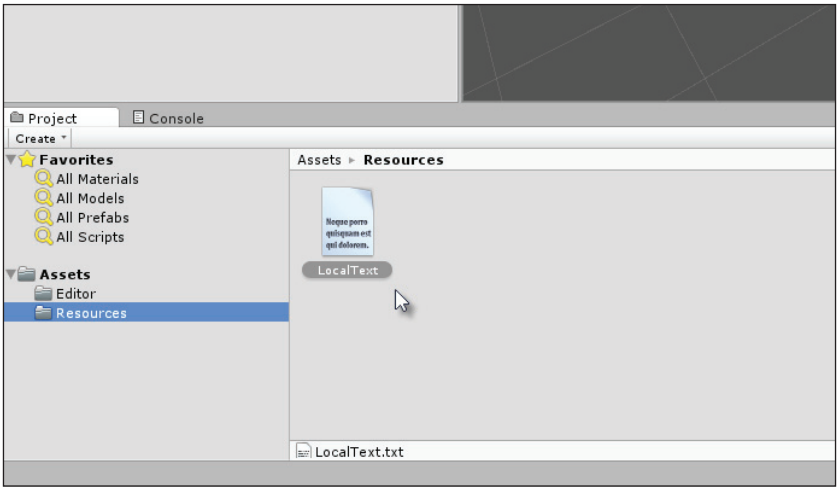


Рис. 8.13. Импорт локализованного текста в проект

а для каждого идентификатора имеется соответствие в языковой схеме, что позволяет произвести смену языков без лишних хлопот. Идентификатор является связующим звеном, которое делает возможным автоматизированную локализацию. Для реализации системы локализации мы сначала создадим атрибут, который должен быть применен ко всем локализованным строкам. Атрибут определяет идентификатор, присоединенный к конкретной строковой переменной, как показано в листинге 8.11.

Листинг 8.11. Атрибут для присоединения к строковым значениям

```
using UnityEngine;
using System.Collections;

// Атрибут для присоединения к строковым значениям
public class LocalizationTextAttribute : System.Attribute
{
  // Присваиваемый идентификатор
  public string LocalizationID = string.Empty;

  // Конструктор
  public LocalizationTextAttribute(string ID)
  {
    LocalizationID = ID;
  }
}
```

Создав атрибут LocalizationTextAttribute, мы можем применить его в сценарии к строковым членам, связывая их с конкретными идентификаторами, как показано в листинге 8.12.

Листинг 8.12. Использование атрибута LocalizationTextAttribute

```
//-------------------------------------------
using UnityEngine;
using System.Collections;
//-------------------------------------------
public class SampleGameMenu : MonoBehaviour
{
  [LocalizationTextAttribute("text_01")]
  public string NewGameText = string.Empty;

  [LocalizationTextAttribute("text_02")]
  public string LoadGameText = string.Empty;

  [LocalizationTextAttribute("text_03")]
  public string SaveGameText = string.Empty;

  [LocalizationTextAttribute("text_04")]
  public string ExitGameText = string.Empty;
}
//-------------------------------------------
```

Класс SampleGameMenu отображается в инспекторе объектов как обычный класс, как видно на рис. 8.14. Позже с помощью класса Editor мы реализуем автоматическую смену языка для всех его строковых членов.

Теперь реализуем класс Editor для переключения между языками. Этот класс будет добавлять элементы в меню приложения, щелкая на которых можно менять активный язык, как показано в листинге 8.13. Этот пример опирается на ряд взаимосвязанных понятий, часть из которых нам уже знакома. В частности, здесь используются классы Reflection, Linq и Editor, а также классы для работы с форматом XML из фреймворка Mono:

Листинг 8.13. Реализация класса смены языка

```
01 //-------------------------------------------
02 using UnityEngine;
03 using UnityEditor;
04 using System.Collections;
05 using System.Xml;
06 using System.Linq;
07 using System.Reflection;
08 //-------------------------------------------
```

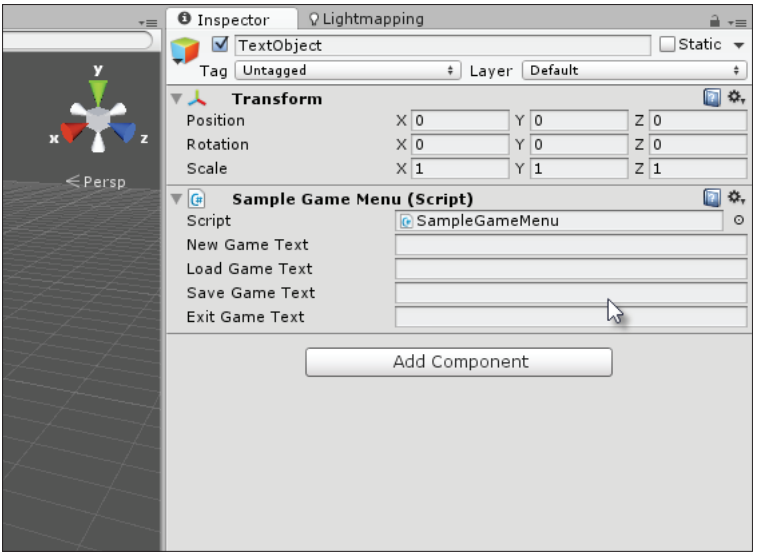


Рис. 8.14. Класс `SampleGameMenu` включает все текстовые элементы для примера экранного меню

```
09 public class LanguageSelector
10 {
11 [MenuItem ("Localization/English")]
12 public static void SelectEnglish()
13 {
14   LanguageSelector.SelectLanguage("english");
15 }
16
17 [MenuItem ("Localization/French")]
18 public static void SelectFrench()
19 {
20   LanguageSelector.SelectLanguage("french");
21 }
22
23 [MenuItem ("Localization/Yoda")]
24 public static void SelectYoda()
25 {
26   LanguageSelector.SelectLanguage("yoda");
27 }
28
29 public static void SelectLanguage(string LanguageName)
30 {
31   // Открыть XML-файл
32   TextAsset textAsset = Resources.Load("LocalText") as TextAsset;
```

```
33
34  // Загрузить текст в объект чтения XML
35  XmlDocument xmlDoc = new XmlDocument();
36  xmlDoc.LoadXml(textAsset.text);
37
38  // Получить узлы, определяющие язык
39  XmlNode[] LanguageNodes = (from XmlNode Node in
    xmlDoc.GetElementsByTagName("language")
40  where Node.Attributes["id"].Value.ToString().Equals(
    LanguageName.ToLower ())
41  select Node).ToArray();
42
43  // Если ни один узел не найден, выйти
44  if(LanguageNodes.Length <= 0)
45    return;
46
47  // Взять первый узел
48  XmlNode LanguageNode = LanguageNodes[0];
49
50  // Получить текстовый объект
51  SampleGameMenu GM = Object.FindObjectOfType<SampleGameMenu>()
    as SampleGameMenu;
52
53  // Выполнить обход дочерних узлов
54  foreach (XmlNode Child in LanguageNode.ChildNodes)
55  {
56    // Получить идентификатор текстового элемента
57    string TextID = Child.Attributes["id"].Value;
58    string LocalText = Child.InnerText;
59
60    // Выполнить обход всех полей
61    foreach(var field in GM.GetType().GetFields(
      BindingFlags.Instance | BindingFlags.Public |
      BindingFlags.NonPublic | BindingFlags.FlattenHierarchy))
62    {
63      // Если это строковое поле
64      if(field.FieldType == typeof(System.String))
65      {
66        // Получить его атрибуты
67        System.Attribute[] attrs =
          field.GetCustomAttributes(true) as System.Attribute[];
68
69        foreach (System.Attribute attr in attrs)
70        {
71          if(attr is LocalizationTextAttribute)
72            {
73              // Мы нашли текст
74              LocalizationTextAttribute LocalAttr =
                attr as LocalizationTextAttribute;
75
```

```
76                if(LocalAttr.LocalizationID.Equals( TextID ))
77                {
78                  // Идентификатор совпадает, установить значение
79                  field.SetValue(GM, LocalText);
80                }
81              }
82            }
83          }
84        }
85      }
86    }
87 }
88 //-----------------------------------------
```

Ниже приводится несколько комментариев к листингу 8.13:

- **Строки 02–07**: не забываем подключить длинный список пространств имен, как показано здесь. Наш код будет использовать каждое из них.
- **Строки 11–23**: в этом примере в меню приложения можно выбрать один из трех языков: английский, французский и йода. Для ваших проектов список языков может быть другим. Но суть в том, что, на основе приведенной здесь системы локализации можно с легкостью добавлять языки когда угодно.
- **Строка 32**: функция `Resources.Load` вызывается для загрузки текстового XML-файла из папки `Resources` проекта: его текстовое содержимое помещается в единственную строковую переменную.
- **Строки 35–36**: XML-строка загружается в объект `XmlDocument`, входящего в состав Mono, для инкапсуляции полного XML-файла, находящегося на диске или в памяти. Класс также проверяет загруженный документ, генерируя исключение, если XML-файл содержит синтаксические ошибки.
- **Строка 53**: после выборки языка из XML-файла выполняется обход всех дочерних узлов (каждый узел является уникальной строкой) в поисках соответствующего идентификатора.
- **Строка 61**: для каждого строкового элемента просматриваются все общедоступные строковые члены класса, для выбора переменных с атрибутом `LocalizationTextAttribute`. После извлечения строка сравнивается с текущим идентификатором. Если они совпали, строковой переменной присваивается соответствующая локализованная строка.

Чтобы использовать приведенную здесь систему локализации, добавьте объект `SampleGameMenu` в сцену, как показано на рис. 8.15.

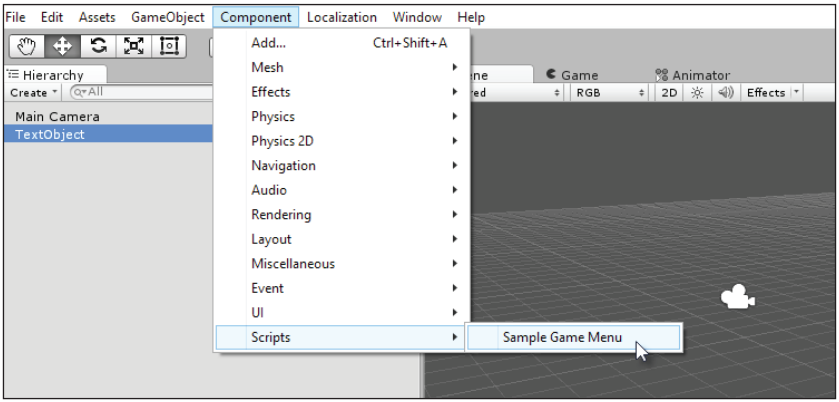


Рис. 8.15. Добавление объекта `SampleGameMenu` в сцену для локализации

Затем выберите в главном меню приложения пункт **Localization ⇒ English** (Локализация ⇒ Английский) или **Localization ⇒ Yoda** (Локализация ⇒ Йода), как показано на рис. 8.16.

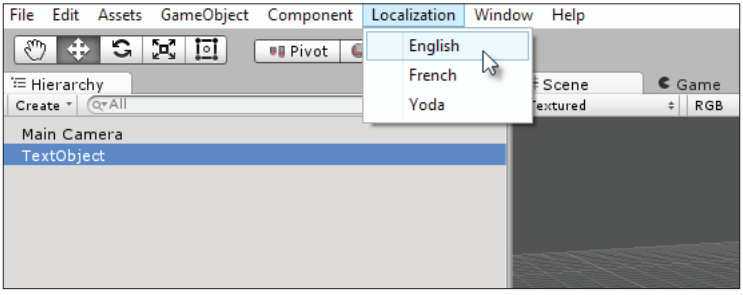


Рис. 8.16. Выбор языка для игры

После выбора языка будут обновлены все строки с атрибутом `LocalizationTextAttribute`, как показано на рис. 8.17.

Итоги

В этой главе был подробно рассмотрен механизм рефлексии и его практическое применение для создания классов, добавляющих новые возможности в редактор. Эти возможности не всегда необходимы собственно для разработки игр в Unity, но они могут облегчить вам рабо-

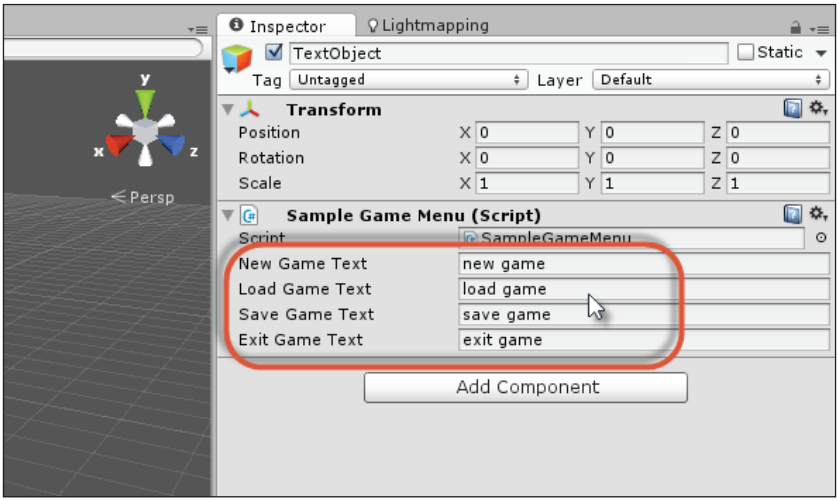


Рис. 8.17. Текстовые значения
изменились после выбора активного языка

ту. Кроме того, разрабатывая расширения, которые помогут в работе другим разработчикам, вы сможете заработать на этом, разместив их магазине ресурсов Unity. Здесь вы узнали, как создать инструмент пакетного переименования с помощью класса `ScriptableWizard` и отображение в инспекторе объектов элементов управления для смешивания цветов. Затем мы использовали механизм рефлексии для отображения в инспекторе объектов общедоступных свойств C#, чтобы получить прямой доступ к методам `set` и `get` во время выполнения. Далее мы рассмотрели способ локализации с помощью обработки XML-файлов в классе `Editor`, который позволяет автоматически изменять строковые переменные для соответствия их выбранному языку. За более подробной информацией обращайтесь к следующим страницам: http://catlikecoding.com/unity/tutorials/editor/custom-data/ и http://catlikecoding.com/unity/tutorials/editor/custom-list/. В следующей главе мы пополним наши теоретические и практические познания, рассмотрев слабо документированные вопросы двухмерного мира.

Глава 9

Работа с текстурами, моделями и двухмерными изображениями

Сегодня большинство игровых движков ориентировано на создание трех-, а не двухмерных игр. Как это ни парадоксально, это делает процесс разработки трехмерных игр более простым, чем двухмерных, по крайней мере на начальном этапе. В этой главе мы рассмотрим некоторые вопросы разработки двухмерных игр с некоторыми оговорками. С момента выхода Unity 4.3 в редактор был добавлен широкий спектр функций для работы с двухмерной графикой: встроенная поддержка спрайтов и новая система отображения пользовательского интерфейса. Хотя они и полезны во многих ситуациях, основное внимание в этой главе будет сосредоточено не на этих функциях. Первая причина в том, что во многих учебных пособиях они уже довольно подробно описаны, а вторая и наиболее важная – даже после добавления эти функций остались нерешенными фундаментальные вопросы, которые возникают при работе с двухмерной графикой: манипулирование геометрией – вершинами и ребрами двухмерных плоскостей, настройка и анимация текстурных координат, редактирование текстур, а также прорисовка текстур в режиме реального времени, с использованием системы кистей, для наложения изображений, таких как пятна крови и т. п. Эти вопросы больше относятся к двухмерной графике, чем к трехмерной, поскольку ка-

саются геометрии и текстур двухмерных плоскостей, но в целом они актуальны для любых игр, двух- и трехмерных. Сегодня эти вопросы имеют большое значение, хотя и недостаточно освещены в учебных пособиях, поэтому я и рассмотрю их здесь. Однако, я ограничусь в этой главе в основном слабо документированными сторонами работы с двухмерной графикой.

Скайбокс

Может показаться странным начинать рассмотрение приемов работы с двухмерной графикой со скайбокса, но все дело в том, что работа с ним демонстрирует возможности настройки камер и использование слоев. Скайбокс, по сути, представляет собой фон на кубической основе для отображения облаков, неба и других отдаленных деталей, которые всегда должны быть в сцене только отдаленным фоном, к которому игрок никогда не сможет приблизиться. Он всегда в отдалении, как показано на рис. 9.1.

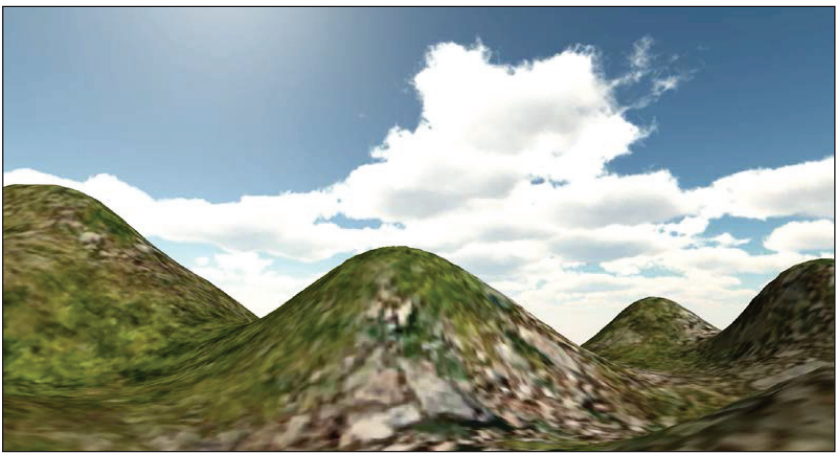


Рис. 9.1. Скайбокс создает фон неба для камеры

Основная проблема скайбоксов, встроенных в Unity, в том, что они по умолчанию остаются неизменными и неподвижными. Большинство разработчиков, однако, хотят, чтобы небо и облака медленно поворачивались, даже если камера неподвижна, чтобы отобразить смену дня и ночи или течение времени. Давайте создадим улучшенный

вариант скайбокса с помощью ресурсов Unity, камеры с двумя слоями и файла сценария на C#.

Окончательный проект вращающегося скайбокса можно найти в пакете примеров для книги.

Для нужд проекта, созданного здесь, импортируем пакет **Character Controllers** с контроллером игры от первого лица, пакет **Terrain Assets** с текстурами ландшафтов, которые можно наносить на местность, и пакет **Skyboxes** с текстурами скайбоксов, как показано на рис. 9.2. Все это пригодится при разработке примера проекта с вращающимся скайбоксом.

Давайте начнем с создания вращающегося скайбокса в виде объекта. Этот объект будет состоять из трех основных частей, или подобъектов: контроллера игры от первого лица, чтобы дать игроку

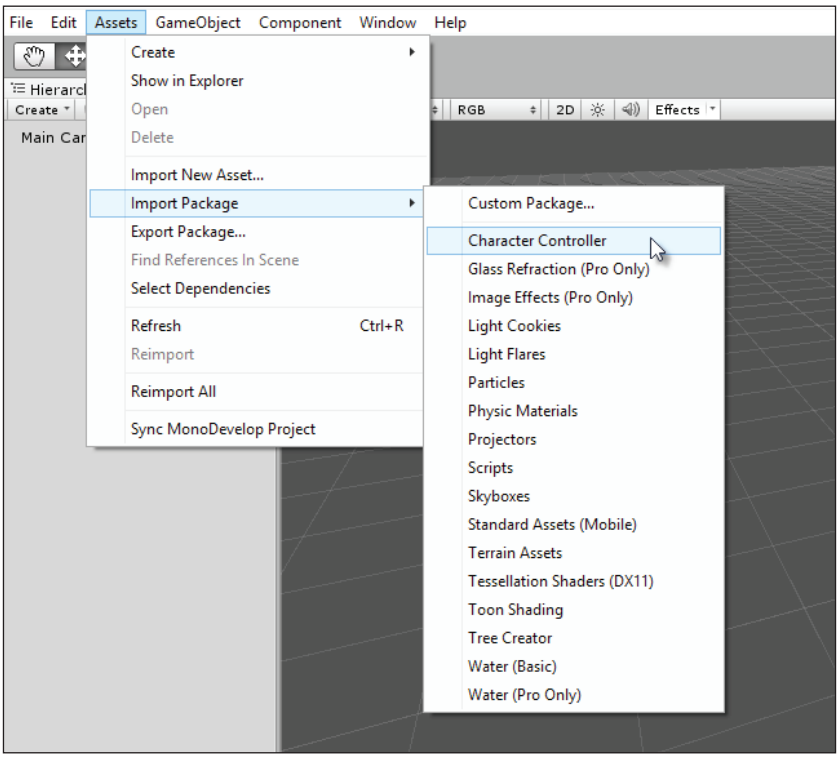


Рис. 9.2. Импорт пакетов **Character Controllers**, **Skyboxes** и **Terrain Assets**

возможность двигаться и отображать объекты сцены, второй камеры (камеры скайбокса), расположенной ниже камеры от первого лица и показывающей только скайбокс, и кубического объекта с перевернутыми нормалями, окружающего камеру скайбокса и отображающего каждую текстуру скайбокса на каждой стороне.

Для начала создайте новый пустой объект в исходной точке сцены (назовем его `SkyBoxCamera`) и добавьте к нему контроллер игры от первого лица. Затем создайте шесть объектов `Quad` (выбрав пункт меню **GameObject ⇒ 3D Object ⇒ Quad** (Игровой объект ⇒ Трехмерные объекты ⇒ Quad)), совместив объекты по углам и расположив вершины так, чтобы сформировать инвертированный куб, то есть куб, грани которого вогнуты внутрь, как показано на рис. 9.3. Он будет служить мешем для управляемого скайбокса.

Измените размеры объектов `Quad`, если это потребуется, и убедитесь, что они охватывают контроллер игры от первого лица, который должен быть в центре скайбокса.

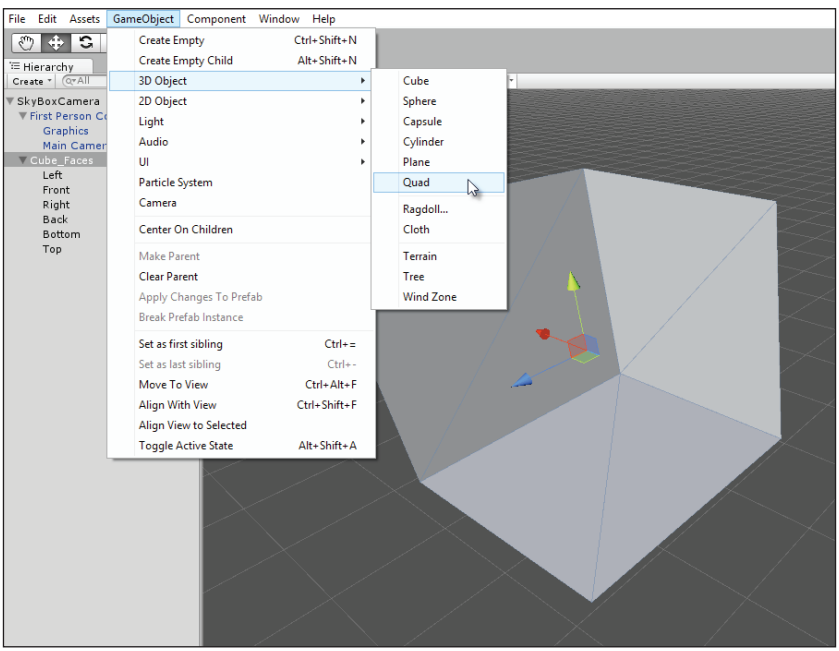


Рис. 9.3. Создание управляемого скайбокса из шести объектов Quad

Назначьте граням скайбокса новый слой `SkyBoxLayer`, выберите камеру контроллера игры от первого лица, а затем измените поле **Culling Mask** (Маска отбраковки) для исключения слоя `SkyBoxLayer`. Камера от первого лица должна отображать только объекты переднего плана, игнорируя фоновые. Для достижения этой цели измените значение поля **Clear Flags** (Флаги очистки) на **Depth only** (Только окружение) в инспекторе объектов, как показано на рис. 9.4. Это сделает фон прозрачным для камеры, что позволит камерам более низкого порядка видеть сквозь него, если они есть.

Теперь создайте новый дополнительный объект `Camera`, как дочерний объект камеры от первого лица, с совпадающими с ней положением, направлением и масштабом. Это позволит вновь созданной камере повторять все перемещения камеры от первого лица. Цель второй камеры – отображать только объект скайбокса как слой, совместно с камерой от первого лица, повторяя позицию и направление камеры от первого лица.

Для этого измените значение поля **Depth** (Глубина) новой камеры на любое значение, меньшее, чем значение поля **Depth** (Глубина) камеры от первого лица, например -1. Удалите все компоненты слушателей аудио там, где это потребуется.

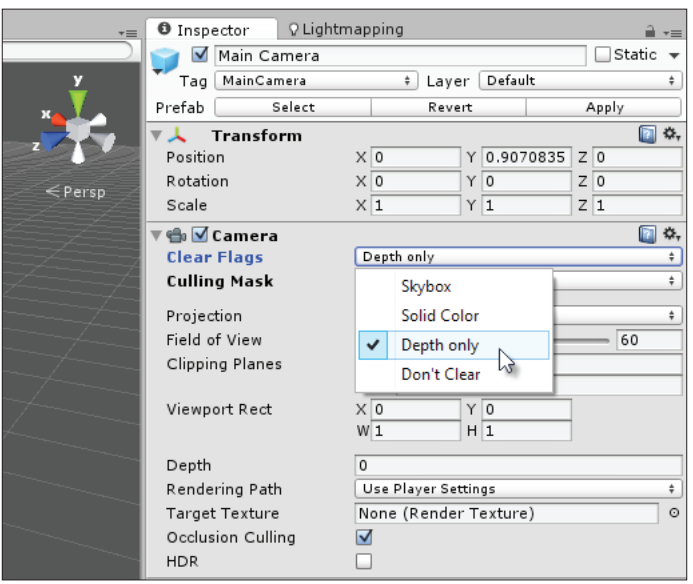


Рис. 9.4. Настройка камеры для прозрачности фона

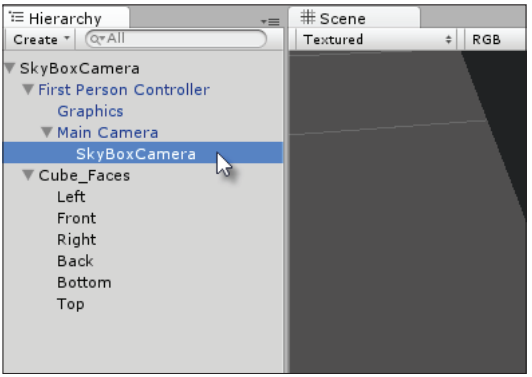


Рис. 9.5. Создание второй камеры для отображения скайбокса

Назначьте уникальную текстуру скайбокса каждой грани куба и позаботьтесь об их выравнивании с помощью небольших поворотов и смещений объектов `Quad`. Затем измените тип материала **Shader** (Шейдер) для текстур скайбокса на **Unlit/Texture** (Неосвещенный/текстура), что сделает скайбокс индифферентным к освещению сцены. Меш скайбокса должен выглядеть, как показано на рис. 9.6.

И, наконец, подключите сценарий из листинга 9.1 к родительскому объекту скайбокса, реализующий его вращение и постоянное соответствие положению камеры. Это гарантирует то, что скайбокс всегда будет располагаться так, что камера будет находиться в его центре, куда бы она не переместилась в сцене.

Листинг 9.1. Сценарий, реализующий вращение скайбокса и его соответствие положению камеры

```
01 //---------------------------------------------------
02 using UnityEngine;
03 using System.Collections;
04 //---------------------------------------------------
05 public class SkyBox : MonoBehaviour
06 {
07   //---------------------------------------------------
08   // Камера для отображения
09   public Camera FollowCam = null;
10
11   // Скорость вращения (градусов в секунду)
12   public float RotateSpeed = 10.0f;
13
14   // Трансформация
```

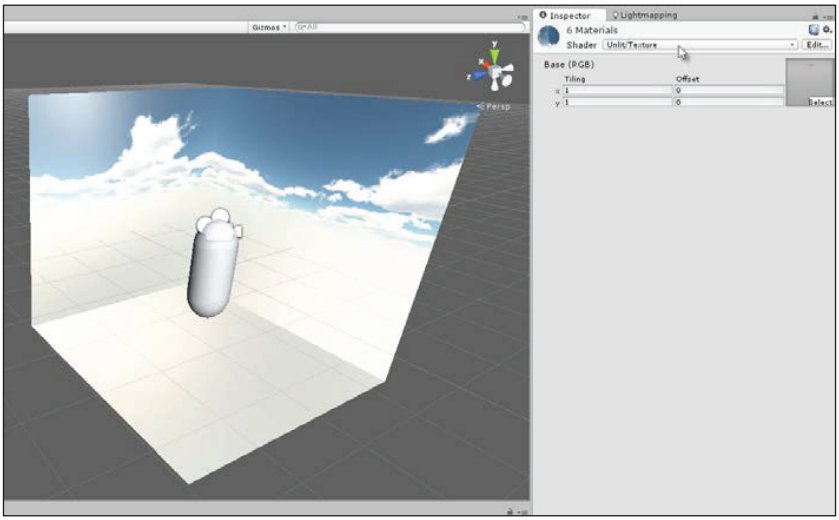


Рис. 9.6. Наложение текстур на объекты Quad

```
15   private Transform ThisTransform = null;
16   //-------------------------------------------------
17   // Этот метод используется для инициализации
18   void Awake () {
19     ThisTransform = transform;
20   }
21   //-------------------------------------------------
22   // Вызывается при отображении каждого кадра
23   void Update () {
24     // Изменить положение
25     ThisTransform.position = FollowCam.transform.position;
26
27     // Изменить угол поворота
28     ThisTransform.Rotate(new Vector3(0,RotateSpeed * Time.deltaTime,0));
29   }
30   //-------------------------------------------------
31 }
32 //-------------------------------------------------
```

Итак, теперь у нас есть усовершенствованный скайбокс, всегда окружающий камеру и вращающийся для придания сценам большей реалистичности. Можно пойти дальше, добавив несколько вложенных друг в друга скайбоксов, каждый с определенной степенью прозрачности, для создания дополнительных эффектов, таких как туман, дымка и т. д.

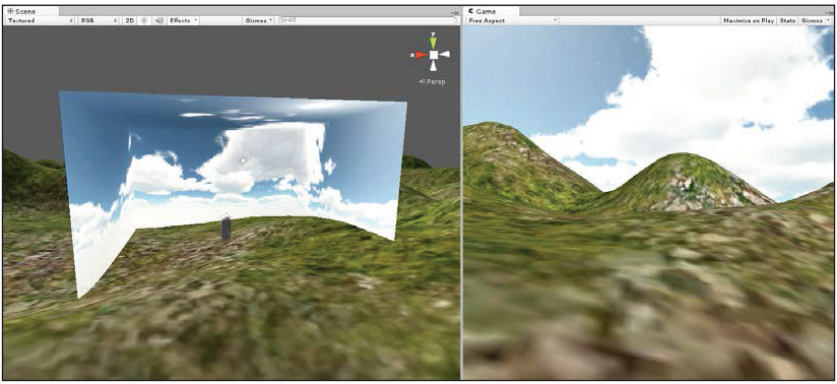


Рис. 9.7. Законченный управляемый скайбокс

Процедурные меши

Хотя Unity и предлагает примитив четырехугольника `Quad`, доступный в виде пункта **GameObject ⇒ 3D Object ⇒ Quad** (Игровой объект ⇒ Трехмерные объекты ⇒ Quad) в меню приложения, но полезно также знать, как создавать подобные геометрические фигуры вручную. И на это есть несколько причин. Во-первых, часто бывает нужно скорректировать координаты вершин из сценария для создания анимационного эффекта или искажения меша с целью имитации разных визуальных эффектов, например желеобразного пола, который изгибается и качается, когда персонажи наступают на него. Во-вторых, вы должны будете отредактировать UV-координаты меша для создания анимированных текстур или эффектов прокрутки текстур, как показано на рис. 9.8.

Рассмотрим следующий пример в листинге 9.2, который следует поместить в папку `Editor` проекта. Он реализует расширение редактора для создания примитивов `Quad` из сценария с возможностью определения позиции центра вращения меша. Как мы увидим в комментариях, этот пример содержит много полезных приемов.

Листинг 9.2. Расширение редактора для создания примитивов Quad

```
001 // Класс расширения редактора для создания примитивов QUAD
       с заданной точкой вращения
002 //-----------------------------------------------
003 using UnityEngine;
004 using UnityEditor;
```

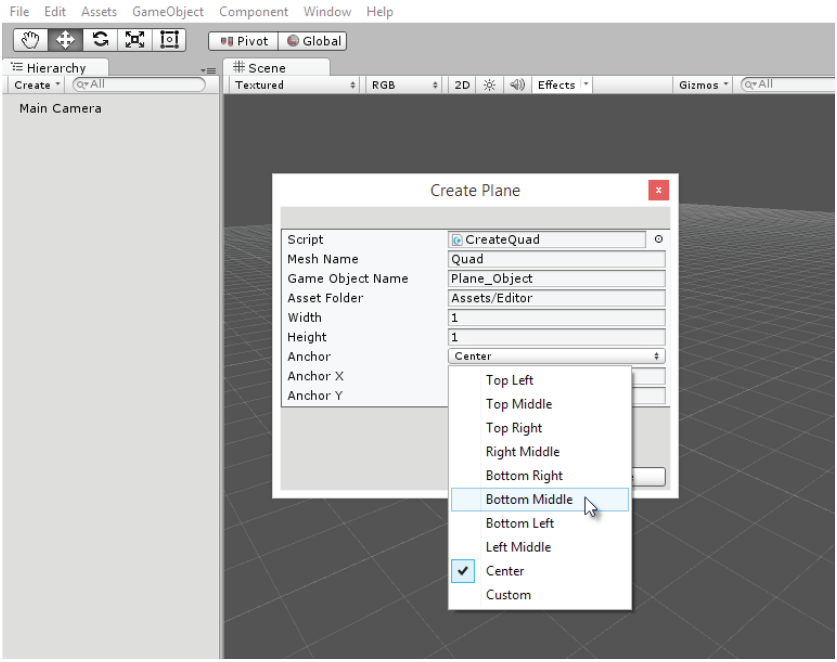


Рис. 9.8. Динамическое создание мешей в сценарии

```
005 using System.IO;
006 //-----------------------------------------------
007 // Используется редактором Unity
008 public class CreateQuad : ScriptableWizard
009 {
010   // Точка вращения для меша
011   public enum AnchorPoint
012   {
013     TopLeft,
014     TopMiddle,
015     TopRight,
016     RightMiddle,
017     BottomRight,
018     BottomMiddle,
019     BottomLeft,
020     LeftMiddle,
021     Center,
022     Custom
023   }
024
025   // Имя ресурса Quad
```

```
026   public string MeshName = "Quad";
027
028   // Имя игрового объекта
029   public string GameObjectName = "Plane_Object";
030
031   // Имя папки с ресурсами
032   public string AssetFolder = "Assets";
033
034   // Ширина примитива в мировых единицах (пикселях)
035   public float Width = 1.0f;
036
037   // Высота примитива в мировых единицах (пикселях)
038   public float Height = 1.0f;
039
040   // Позиция точки вращения
041   public AnchorPoint Anchor = AnchorPoint.Center;
042
043   // Горизонтальная позиция точки вращения на плоскости
044   public float AnchorX = 0.5f;
045
046   // Вертикальная позиция точки вращения на плоскости
047   public float AnchorY = 0.5f;
048   //-------------------------------------------------
049   [MenuItem("GameObject/Create Other/Custom Plane")]
050   static void CreateWizard()
051   {
052     ScriptableWizard.DisplayWizard("Create Plane",typeof(CreateQuad));
053   }
054
055   //-------------------------------------------------
056   // Вызывается в момент создания окна
057   void OnEnable()
058   {
059     // Вызвать событие изменения выбора
060     OnSelectionChange();
061   }
062   //-------------------------------------------------
063   // Вызывается 10 раз в секунду
064   void OnInspectorUpdate()
065   {
066     switch(Anchor)
067     {
068       // Точка вращения установлена в левый верхний угол
069       case AnchorPoint.TopLeft:
070         AnchorX = 0.0f * Width;
071         AnchorY = 1.0f * Height;
072         break;
073
074       // Точка вращения установлена вверху в центре
075       case AnchorPoint.TopMiddle:
```

```
076         AnchorX = 0.5f * Width;
077        AnchorY = 1.0f * Height;
078         break;
079
080       // Точка вращения установлена в правый верхний угол
081       case AnchorPoint.TopRight:
082         AnchorX = 1.0f * Width;
083         AnchorY = 1.0f * Height;
084         break;
085
086       // Точка вращения установлена справа в центре
087       case AnchorPoint.RightMiddle:
088         AnchorX = 1.0f * Width;
089         AnchorY = 0.5f * Height;
090         break;
091
092       // Точка вращения установлена в правый нижний угол
093       case AnchorPoint.BottomRight:
094         AnchorX = 1.0f * Width;
095         AnchorY = 0.0f * Height;
096         break;
097
098       // Точка вращения установлена внизу в центре
099       case AnchorPoint.BottomMiddle:
100         AnchorX = 0.5f * Width;
101         AnchorY = 0.0f * Height;
102         break;
103
104       // Точка вращения установлена в левый нижний угол
105       case AnchorPoint.BottomLeft:
106         AnchorX = 0.0f * Width;
107         AnchorY = 0.0f * Height;
108         break;
109
110       // Точка вращения установлена слева в центре
111       case AnchorPoint.LeftMiddle:
112         AnchorX = 0.0f * Width;
113         AnchorY = 0.5f * Height;
114         break;
115
116       // Точка вращения установлена в центре
117       case AnchorPoint.Center:
118         AnchorX = 0.5f * Width;
119         AnchorY = 0.5f * Height;
120         break;
121
122       case AnchorPoint.Custom:
123         default:
124         break;
125     }
```

```
126   }
127   //-------------------------------------------------
128   // Вызывается, когда выполняется обновление окна
129   void OnSelectionChange()
130   {
131     // Проверить выбор пользователя в редакторе
132     if (Selection.objects != null && Selection.objects.Length == 1)
133     {
134       // Получить путь из выбранного ресурса
135       AssetFolder = Path.GetDirectoryName(AssetDatabase.
          GetAssetPath(Selection.objects[0]));
136     }
137   }
138   //-------------------------------------------------
139   // Функция для создания меша четырехугольника
140   void OnWizardCreate()
141   {
142     // Создать вершины
143     Vector3[] Vertices = new Vector3[4];
144
145     // Создать координаты UV
146     Vector2[] UVs = new Vector2[4];
147
148     // Два треугольника в четырехугольнике
149     int[] Triangles = new int[6];
150
151     // Связать вершины, опираясь на точку вращения
152
153     // Слева внизу
154     Vertices[0].x = -AnchorX;
155     Vertices[0].y = -AnchorY;
156
157     // Справа внизу
158     Vertices[1].x = Vertices[0].x+Width;
159     Vertices[1].y = Vertices[0].y;
160
161     // Слева вверху
162     Vertices[2].x = Vertices[0].x;
163     Vertices[2].y = Vertices[0].y+Height;
164
165     // Справа вверху
166     Vertices[3].x = Vertices[0].x+Width;
167     Vertices[3].y = Vertices[0].y+Height;
168
169     // Присвоить координаты UV
170     // Слева внизу
171     UVs[0].x=0.0f;
172     UVs[0].y=0.0f;
173
174     // Справа внизу
```

```
175     UVs[1].x=1.0f;
176     UVs[1].y=0.0f;
177
178     // Слева вверху
179     UVs[2].x=0.0f;
180     UVs[2].y=1.0f;
181
182     // Справа вверху
183     UVs[3].x=1.0f;
184     UVs[3].y=1.0f;
185
186     // Присвоить треугольники
187     Triangles[0]=3;
188     Triangles[1]=1;
189     Triangles[2]=2;
190
191     Triangles[3]=2;
192     Triangles[4]=1;
193     Triangles[5]=0;
194
195     // Создать меш
196     Mesh mesh = new Mesh();
197     mesh.name = MeshName;
198     mesh.vertices = Vertices;
199     mesh.uv = UVs;
200     mesh.triangles = Triangles;
201     mesh.RecalculateNormals();
202
203     // Создать ресурс в базе данных
204     AssetDatabase.CreateAsset(mesh,
        AssetDatabase.GenerateUniqueAssetPath(
        AssetFolder + "/" + MeshName) + ".asset");
205     AssetDatabase.SaveAssets();
206
207     // Создать плоский игровой объект
208     GameObject plane = new GameObject(GameObjectName);
209     MeshFilter meshFilter =
        (MeshFilter)plane.AddComponent(typeof(MeshFilter);
210     plane.AddComponent(typeof(MeshRenderer));
211
212     // Связать меш с фильтром
213     meshFilter.sharedMesh = mesh;
214     mesh.RecalculateBounds();
215
216     // Добавить компонент коллайдера кубической формы
217     plane.AddComponent(typeof(BoxCollider));
218   }
219
220 //-------------------------------------------------
221 }
```

Ниже приводится несколько комментариев к листингу 9.2:

- **Строка 004**: этот пример представляет расширение редактора. Именно поэтому он подключает пространство имен `UnityEditor`. За более подробной информацией о создании расширений редактора обращайтесь к главе 8 «Настройка редактора Unity».
- **Строка 135**: событие `OnSelectionChanged` вызывается, когда пользователь производит выбор в редакторе Unity с помощью мыши или клавиатуры. Здесь метод `GetAssetPath` вызывается для восстановления открытой в данный момент папки в панели **Project** (Проект).
- **Строка 140**: функция `OnWizardCreate` вызывается для создания меша `Quad`. Она заполняет массивы вершин и координат UV, а затем наполняет содержанием объект `Mesh`, созданный в строке 196.
- **Строка 204**: важно отметить, что меш сохраняется не как объект в определенной сцене, а как ресурс, из которого можно создать несколько экземпляров. Для этого используется класс `AssetDatabase`. Это позволит, при необходимости, использовать меш в нескольких сценах, а также переносить его изменения из сцены в сцену.

Более подробную информацию о классе `AssetDatabase` можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/AssetDatabase.html.

Анимация UV-координат – прокручивание текстур

Прием прокручивания текстур очень часто применяется в играх, и все же он не имеет встроенной поддержки в Unity, поэтому есть смысл написать сценарий для его реализации. Прокручивание текстур используется для создания эффекта смещения: движения облаков, перемещения поверхностей, текучей воды и подчеркивания движений в игре. Как правило, такие текстуры представляют собой бесшовные изображения по вертикали и по горизонтали. Это позволяет выполнять бесконечное их прокручивание, как показано на рис. 9.9.

При подключении к примитиву `Quad`, следующий пример в листинге 9.3 оживит его текстуру с соответствующими скоростями по горизонтали и вертикали.

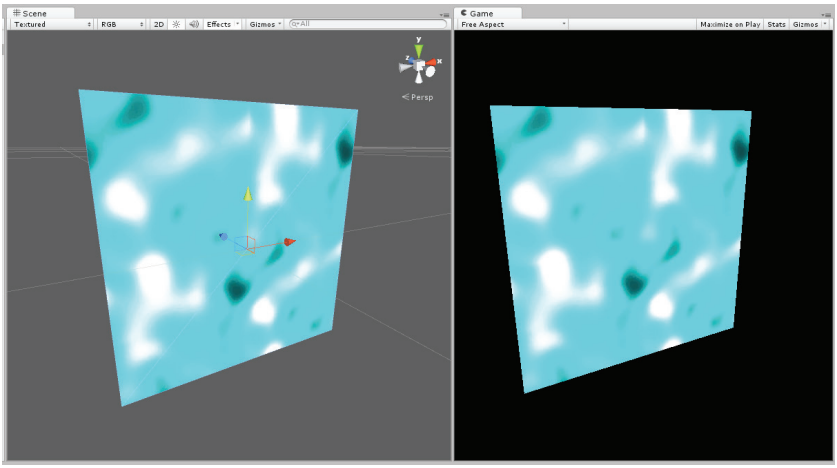


Рис. 9.9. Прокручивание текстуры на четырехугольнике Quad

Листинг 9.3. Класс для прокручивания плоских текстур

```
01 // Класс для прокручивания плоских текстур.
      Может использоваться для создания эффекта движения облаков
02 //-----------------------------------------------
03 using UnityEngine;
04 using System.Collections;
05 //-----------------------------------------------
06 [RequireComponent (typeof (MeshRenderer))]
07 public class MatScroller : MonoBehaviour
08 {
09   // Общедоступные переменные
10   //-----------------------------------------------
11   //  Горизонтальная скорость прокрутки
12   public float HorizSpeed = 1.0f;
13
14   // Вертикальная скорость прокрутки
15   public float VertSpeed = 1.0f;
16
17   // Минимальная и максимальная координаты UV по гризонтали и вертикали
18   public float HorizUVMin = 1.0f;
19   public float HorizUVMax = 2.0f;
20
21   public float VertUVMin = 1.0f;
22   public float VertUVMax = 2.0f;
23
24   // Локальные переменные
```

```
25   //-----------------------------------------------
26   // Ссылка на компонент отображения меша
27   private MeshRenderer MeshR = null;
28
29   // Методы
30   //-----------------------------------------------
31   // Этот метод выполняет инициализацию
32   void Awake ()
33   {
34     // Получить компонент отображения меша
35     MeshR = GetComponent<MeshRenderer>();
36   }
37   //-----------------------------------------------
38   // Вызывается при отображении каждого кадра
39   void Update ()
40   {
41     // Прокручивать текстуру между минимальной и максимальной координатами
42     Vector2 Offset = new Vector2((
       MeshR.material.mainTextureOffset.x > HorizUVMax) ?
       HorizUVMin : MeshR.material.mainTextureOffset.x +
       Time.deltaTime * HorizSpeed,
43     (MeshR.material.mainTextureOffset.y > VertUVMax) ?
       VertUVMin : MeshR.material.mainTextureOffset.y +
       Time.deltaTime * VertSpeed);
44
45     // Обновить UV-координаты
46     MeshR.material.mainTextureOffset = Offset;
47   }
48   //-----------------------------------------------
49 }
50 //-----------------------------------------------
```

Класс `MatScroller` работает с любым компонентом `MeshRenderer` и объектом `Quad`. Полный проект прокручивания текстуры можно найти в пакете примеров для книги.

Подключите этот сценарий к объекту `Quad` и настройте его скорость прокрутки для получения необходимого эффекта, как показано на рис. 9.10. Он пригодится для создания анимированного фона неба и фона для игр-стрелялок с боковой прокруткой. Он также может быть полезен при создании плавно текущей воды и объемного освещения, при применении в сочетании с прозрачностью!

Рисование на текстуре

На практике часто возникают ситуации, когда необходимо рисовать пиксели на текстуре прямо во время игры. Иногда такая задача решается тривиально, например отображение полупрозрачных вставок на

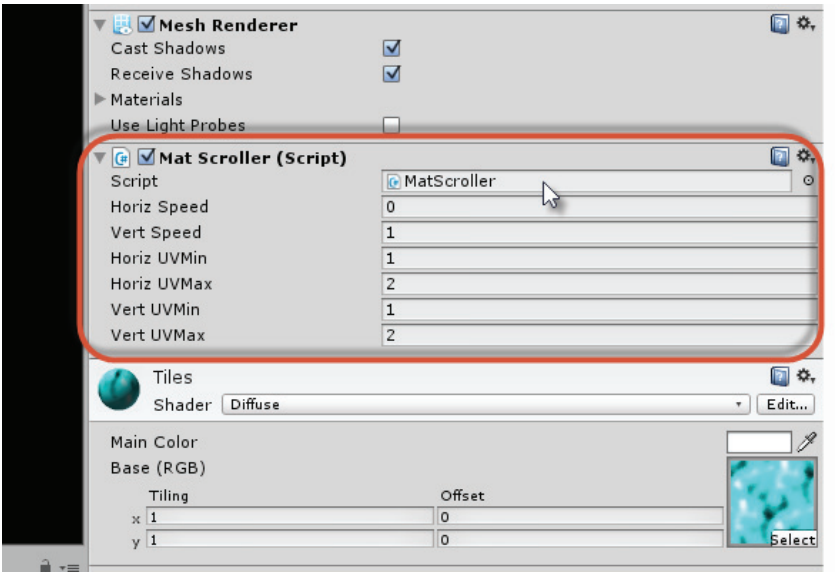


Рис. 9.10. Настройка прокрутки текстуры в инспекторе объектов

текстуре (как следы ног или надписи). В этих случаях можно просто наложить полупрозрачные фрагменты изображений на фоновую поверхность. Но иногда требуется нечто более сложное, когда поможет только полноценное рисование на текстуре. Например, игры с уличными драками, где желательно изобразить брызги крови от ударов, падающие на землю и окружающие предметы так, чтобы они стали частью окружающих текстур. Другим примером может послужить игра в визажиста, где игрок должен наложить макияж на изображение лица.

Здесь нужно не просто натянуть плоскую текстуру на меш, как отдельный объект, чтобы создать декоративный эффект, а прорисовать текстуру (например, с помощью кисти) на фоновой текстуре меша. Здесь картинка и фон не должны быть двумя независимыми текстурами, и при их соединении должны учитываться меш и его UV-координаты. Другими словами, исходная текстура должны быть спроецирована на поверхность меша, а затем прорисованные пиксели возвращены на текстуру с помощью UV-отображения меша. Это гарантирует, что пиксели будут прорисованы в правильном месте на

текстуре назначения, как показано на рис 9.11. Этот метод позволяет прорисовать любую исходную текстуру любого размера на любой трехмерной поверхности с наложенной текстурой любого размера, через UV-отображение.

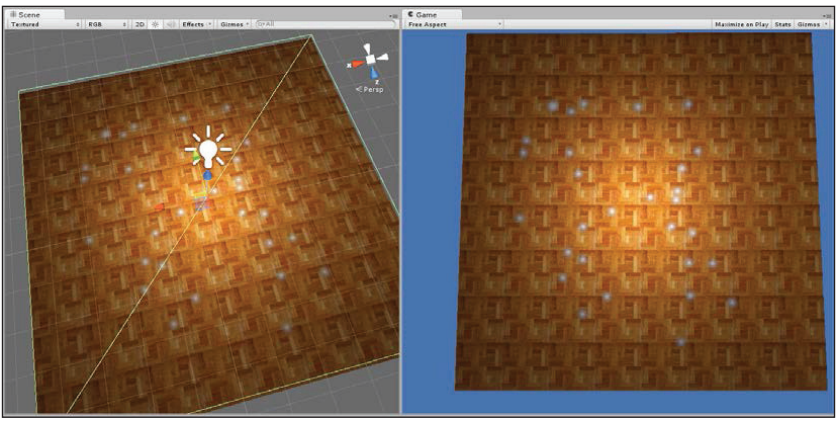


Рис. 9.11. Рисование на текстуре текстурированной кистью во время выполнения с использованием меша и его UV-отображения

В этом разделе мы рассмотрим, как эффективно реализовать это на практике. Перед тем как начать, следует отметить, что рисование на текстуре должно рассматриваться как последнее средство, когда альтернативные методы (например, врезка плоских кусочков) не подходят. Это связано с тем, что рисование на текстуре предъявляет повышенные требования к вычислительным ресурсам.

Законченный проект рисования на текстуре можно найти в пакете с примерами для книги.

Шаг 1 – создание шейдера смешивания текстур

Давайте познакомимся с приемом наложения двух слоев, который идеально подходит в данном случае. Во-первых, у нас есть текстура кисти, которая будут нарисована на фоновой текстуре, когда пользователь щелкнет на меше в сцене, она показана на рис. 9.12.

У нас есть также фоновая текстура, натянутая на меш, которая при рисовании должна перекрываться текстурой кисти, она показана на рис. 9.13.

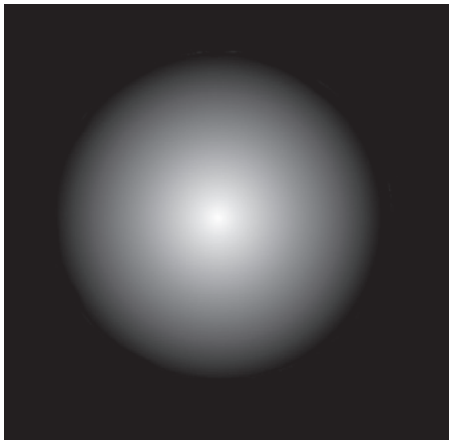


Рис. 9.12. Текстура кисти черного цвета с прозрачностью

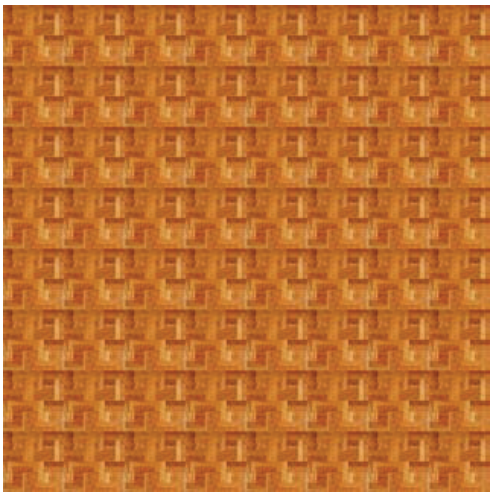


Рис. 9.13. Фоновая текстура, на которой будет прорисована текстура кисти

Однако, обычно не требуется, чтобы при рисовании текстура кисти перекрывала или изменяла пиксели фоновой текстуры. Это связано с тем, что фоновая текстура может быть наложена на несколько объ-

ектов в сцене (по крайней мере, теоретически), и перекрытие или изменение пикселей исходной фоновой текстуры приведет к распространению изменений на все объекты с этой же текстурой.

Вместо этого лучше выделить результат рисования в отдельную текстуру с прозрачным фоном, которую можно наложить вторым слоем на фоновую текстуру. Это позволит разделить фоновую текстуру и текстуру с результатами рисования кистью, хотя по внешнему виду они будут выглядеть как одна сводная текстура. Для достижения этого эффекта необходимо написать свой шейдер, приведенный в листинге 9.4. Этот шейдер накладывает текстуру кисти (с альфа-прозрачностью) поверх текстуры фона.

Листинг 9.4. Шейдер наложения одной текстуры на другую

```
Shader "TextureBlender"
{
  Properties
  {
    _Color ("Main Color", Color) = (1,1,1,1)
    _MainTex ("Base (RGB) Trans (A)", 2D) = "white" {}
    _BlendTex ("Blend (RGB)", 2D) = "white"
  }

  SubShader
  {
    Tags { "Queue"="Geometry-9" "IgnoreProjector"="True"
    "RenderType"="Transparent" }
    Lighting Off
    LOD 200
    Blend SrcAlpha OneMinusSrcAlpha

    CGPROGRAM
    #pragma surface surf Lambert
    uniform fixed4 _Color;
    uniform sampler2D _MainTex;
    uniform sampler2D _BlendTex;

    struct Input
    {
      float2 uv_MainTex;
    };

    void surf (Input IN, inout SurfaceOutput o)
    {
      fixed4 c1 = tex2D( _MainTex, IN.uv_MainTex );
      fixed4 c2 = tex2D( _BlendTex, IN.uv_MainTex );

      fixed4 main = c1.rgba * (1.0 - c2.a);
```

```
34        fixed4 blendedoutput = c2.rgba * c2.a;
35
36        o.Albedo = (main.rgb + blendedoutput.rgb) * _Color;
37        o.Alpha = main.a + blendedoutput.a;
38      }
39      ENDCG
40    }
41    Fallback "Transparent/VertexLit"
42 }
```

После сохранения шейдера, он становится доступен для выбора в качестве типа шейдера для любого материала, создаваемого в панели **Material** (Материал) в инспекторе объектов. Этот шейдер должен использоваться для любых объектов, которые вы собираетесь рисовать, как показано на рис. 9.14. Слот `_MainTex` определяет фоновую текстуру, на которую будет наложено изображение и которая

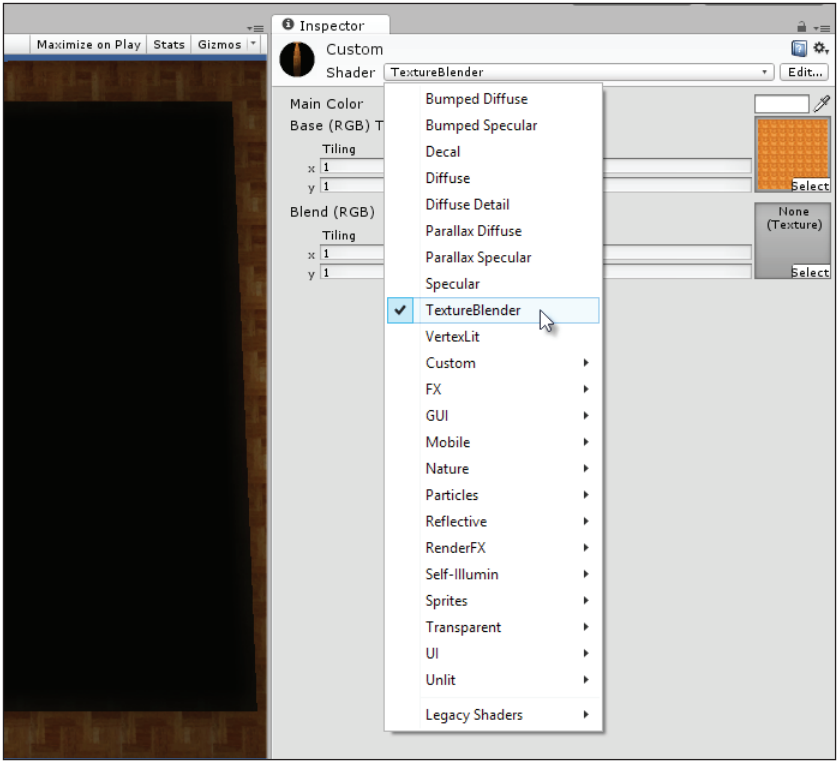


Рис. 9.14. Пользовательский шейдер для смешивания текстур

не должна редактироваться. Слот `_BlendTex` определяет прозрачную текстуру, накладываемую на фоновую текстуру `_MainTex`. Обычно этот слот заполняется программно из сценария, генерирующего прозрачную текстуру кисти, как мы вскоре увидим.

Шаг 2 – создание сценария рисования текстуры

Мы создали шейдер, принимающий две текстуры (верхнюю и нижнюю) и накладывающий верхнюю текстуру на нижнюю. В результате возникает эффект, похожий на использование слоев в Photoshop. Это позволяет поместить нарисованную текстуру в верхний слой, сохранив неизменными пиксели фоновой текстуры под ней, как показано на рис. 9.15.

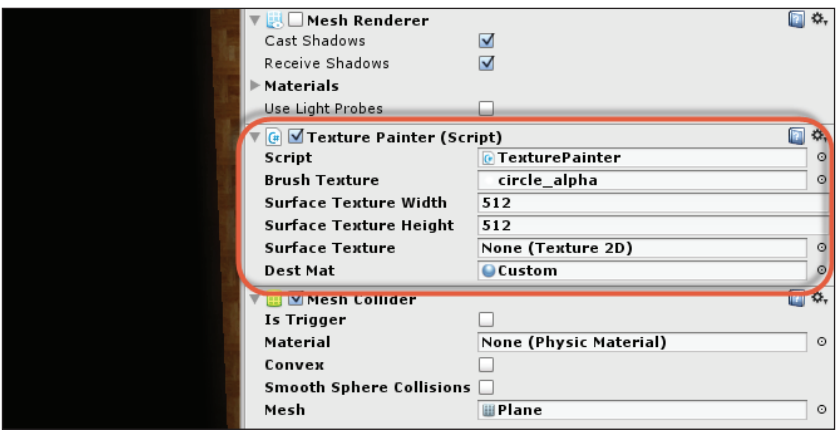


Рис. 9.15. Создание сценария рисования текстуры

Прежде чем двигаться дальше, мы должны сначала отредактировать ресурс текстуры кисти, что мы и сделаем с помощью инспектора объектов. В частности, выберите текстуру кисти в панели **Project** (Проект) редактора Unity и измените **Texture Type** (Тип текстуры) на **Advanced** (Улучшенный). Установите флажок **Read/Write Enabled** (Разрешить чтение/запись), это позволит получить доступ к функциям редактирования текстуры.

Кроме того, установите флажок **Alpha is Transparency** (Прозрачный альфа-канал) и сбросьте флажок **Generate Mip Maps** (Генерировать mip-текстуру), как показано на рис. 9.16.

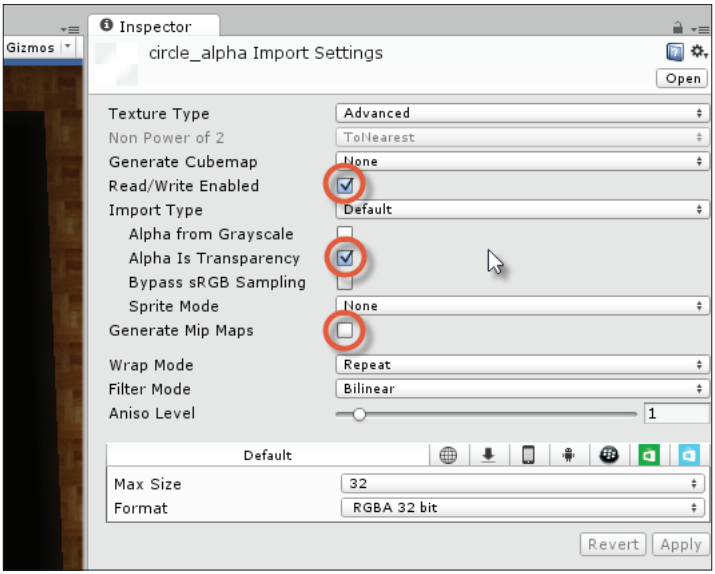


Рис. 9.16. Подготовка текстуры для рисования на другой текстуре

Теперь напишем сценарий рисования на текстуре, который позволит прорисовать текстуру кисти с помощью мыши на трехмерном объекте с использованием его UV-координат. Сценарий представлен в листинге 9.5.

Листинг. 9.5. Сценарий рисования на текстуре

```
001 //-------------------------------------------------------------
002 using UnityEngine;
003 using System.Collections;
004 //-------------------------------------------------------------
005 public class TexturePainter : MonoBehaviour
006 {
007   // Квадратная текстура с прозрачностью
008   public Texture2D BrushTexture = null;
009
010   // Ширина и высота текстуры для рисования
011   public int SurfaceTextureWidth = 512;
012   public int SurfaceTextureHeight = 512;
013
014   // Ссылка на текстуру для рисования
015   public Texture2D SurfaceTexture = null;
016
```

```
017  // Ссылка на материал текстуры для рисования
018  public Material DestMat = null;
019  //-------------------------------------------------
020  // Этот метод выполняет инициализацию
021  void Start ()
022  {
023    // Создать текстуру для рисования
024    SurfaceTexture = new Texture2D(SurfaceTextureWidth,
       SurfaceTextureHeight, TextureFormat.RGBA32, false);
025
026    // Заполнить черными пикселями (прозрачные; alpha=0)
027    Color[] Pixels = SurfaceTexture.GetPixels();
028    for(int i=0; i<Pixels.Length; i++)
029      Pixels[i] = new Color(0,0,0,0);
030    SurfaceTexture.SetPixels(Pixels);
031    SurfaceTexture.Apply();
032
033    // Установить как основную текстуру
034    renderer.material.mainTexture = SurfaceTexture;
035
036    // Если материал текстуры для рисования, установить текстуру кисти
037    // Используется с нестандартным шейдером
038    if(DestMat)
039      DestMat.SetTexture("_BlendTex", SurfaceTexture);
040  }
041  //-------------------------------------------------
042  // Вызывается при отображении каждого кадра
043  void Update ()
044  {
045    // Если кнопка мыши нажата, начать рисование
046    if(Input.GetMouseButtonDown(0))
047    {
048      // Точка под указателем мыши
049      RaycastHit hit;
050
051      // Преобразовать точку на экране в луч в сцене
052      if (!Physics.Raycast(Camera.main.ScreenPointToRay(
         Input.mousePosition), out hit))
053        return;
054
055      // Получить коллайдер
056      Renderer renderer = hit.collider.renderer;
057      MeshCollider Collide = hit.collider as MeshCollider;
058      if (renderer == null || renderer.sharedMaterial == null ||
         renderer.sharedMaterial.mainTexture == null || Collide == null)
059        return;
060
061      // Получить UV-координаты поверхности
062      Vector2 pixelUV = hit.textureCoord;
063      pixelUV.x *= renderer.material.mainTexture.width;
```

```
064       pixelUV.y *= renderer.material.mainTexture.height;
065
066       // Установить центр текстуры кисти в координаты курсора
067       pixelUV.x -= BrushTexture.width/2;
068       pixelUV.y -= BrushTexture.height/2;
069
070       // Привести значения пикселей к диапазону 0-width
071       pixelUV.x = Mathf.Clamp(pixelUV.x, 0,
          renderer.material.mainTexture.width);

072       pixelUV.y = Mathf.Clamp(pixelUV.y, 0,
          renderer.material.mainTexture.height);
073
074       // нарисовать на текстуре для рисования
075       PaintSourceToDestTexture(BrushTexture,
          renderer.material.mainTexture as Texture2D,
          (int)pixelUV.x, (int)pixelUV.y);
076     }
077   }
078   //-------------------------------------------------
079   // Рисует исходную текстуру на текстуре для рисования
080   // Нарисует текстуру кисти на текстуре для рисования
081   public static void PaintSourceToDestTexture(
      Texture2D Source, Texture2D Dest, int Left, int Top)
082   {
083     // Получить пиксели текстуры кисти
084     Color[] SourcePixels = Source.GetPixels();
085
086     // Получить пиксели текстуры для рисования
087     Color[] DestPixels = Dest.GetPixels();
088
089     for(int x=0; x<Source.width; x++)
090     {
091       for(int y=0; y<Source.height; y++)
092       {
093         // Получить исходный пиксель
094         Color Pixel = GetPixelFromArray(SourcePixels, x, y,
            Source.width);
095
096         // Получить смещение в текстуре для рисования
097         int DestOffsetX = Left + x;
098         int DestOffsetY = Top + y;
099
100        if(DestOffsetX < Dest.width && DestOffsetY < Dest.height)
101          SetPixelInArray(DestPixels, DestOffsetX, DestOffsetY,
             Dest.width, Pixel, true);
102       }
103     }
104
105     // Изменить текстуру для рисования
```

```
106     Dest.SetPixels(DestPixels);
107     Dest.Apply();
108   }
109   //-------------------------------------------------------
110   // Читает цвет из массива пикселей
111   public static Color GetPixelFromArray(Color[] Pixels, int X, int Y,
      int Width)
112   {
113     return Pixels[X+Y*Width];
114   }
115   //-----------------------------------------------
116   // Записывает цвет в массив пикселей
117   public static void SetPixelInArray(Color[] Pixels, int X, int Y,
      int Width, Color NewColor, bool Blending=false)
118   {
119     if(!Blending)
120       Pixels[X+Y*Width] = NewColor;
121     else
122     {
123       // Здесь смешивается цвет текстуры с цветом поверхности,
          С сохранением значения прозрачности
124       Color C = Pixels[X+Y*Width] * (1.0f - NewColor.a);
125       Color Blend = NewColor * NewColor.a;
126
127       Color Result = C + Blend;
128       float Alpha = C.a + Blend.a;
129
130       Pixels[X+Y*Width] = new Color(Result.r, Result.g, Result.b, Alpha);
131     }
132   }
133   //-----------------------------------------------
134 }
135 //-------------------------------------------------------------
```

Ниже приводится несколько комментариев к листингу 9.5:

- **Строка 008**: общедоступная переменная в этой строке хранит ссылку на ресурс текстуры, используемой в качестве рисунка кисти. По каждому щелчку мышью эта текстура будет накладываться на текстуру из переменной `SurfaceTexture`.
- **Строка 015**: переменная `SurfaceTexture` хранит ссылку на динамически сгенерированную текстуру с прозрачными пикселями, которая сыграет роль нижнего слоя. Эта текстура будет принимать все мазки кистью при рисовании. Проще говоря, эта текстура будет передана в шейдер `TextureBlender` в переменной `_BlendTex`.
- **Строки 026–031**: новая текстура генерируется во время выполнения функции `Start`. Текстура имеет формат RGBA32, поддер-

живающий альфа-канал. Функция `SetPixels` используется для пакетного заполнения (заливки) текстуры пикселями одного цвета. Более подробно функции `GetPixels` и `SetPixels` рассматриваются ниже.

- **Строка 046**: в функции `Update` отслеживаются щелчки мышью для запуска процедуры рисования на текстуре.
- **Строки 048–059**: при нажатии кнопки мыши необходимо прорисовать текстуру кисти в назначенном месте. Функция `Physics.Raycast` вызывается в строке 52 для выяснения нескольких вопросов, один из них: является ли мешем объект в сцене, на который указывает луч. Такой объект должен иметь компонент `Collider`.
- **Строки 062–072**: если объект обнаружен, UV-координаты места падения луча можно получить из переменной `textureCoord` структуры `RaycastHit`. Более подробную информацию о данной переменной можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/RaycastHit-textureCoord.html. Эта переменная хранит нужные значения, только если объект, на меш которого попал луч, владеет коллайдером типа `MeshCollider`, а не каким-то другим, таким как `BoxCollider` или `CapsuleCollider`. То есть, любой объект, предназначенный для прорисовки на нем текстуры, должен иметь компонент `MeshCollider`, так как только он имеет данные о UV-координатах. В строках 63–72 выполняется преобразование UV-координат в абсолютную позицию в пикселях центра текстуры кисти при заданном положении курсора. Результатом является точно определенная позиция на исходной текстуре, соответствующая точке с координатами x, y, в которой текстура должна быть прорисована на текстуре для рисования.
- **Строка 075**: функция `PaintSourceToDestTexture` служит для самой прорисовки.
- **Строка 081**: функция `PaintSourceToDestTexture` принимает четыре аргумента: `Source`, `Dest`, `Left` и `Top`. На основании их значений текстура `Source` прорисовывается на текстуре `Dest` в месте с координатами `Left` и `Top`. Эта функция объявлена как статическая, то есть для ее вызова нет необходимости создавать экземпляр класса.
- **Строки 084–087**: на первым шаге процесса прорисовки текстуры извлекаются все пиксели текстур `Source` и `Dest`. Для этого используется функция `GetPixels`. Более подробную информа-

цию о функции GetPixels можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/Texture2D.GetPixels.html. Далее, несмотря на то, что каждое изображение внешне выглядит как двумерный массив пикселей, функция GetPixels возвращает линейный (одномерный) массив. Поэтому для преобразования координат x и y пикселей в индексы линейного массива используются две функции: GetPixelFromArray и SetPixelFromArray.

- **Строки 089–101**: здесь каждый пиксель извлекается из текстуры кисти и рисуется в нужном месте. Дополнительно проверяется – находится ли прорисовываемая часть текстуры кисти в пределах текстуры для рисования, и при необходимости производится отсечение. Это связано с тем, что кисть, в принципе, может приблизиться к краю текстуры. В этом случае фактически будет прорисована только часть кисти, а некоторые пиксели будут «отрезаны». Пиксели читаются из текстуры кисти с помощью функции GetPixelFromArray и записываются в текстуру для рисования с помощью функции SetPixelInArray.
- **Строки 106–107**: вызов функции Apply после записи пикселей в текстуру для рисования служит для подтверждения операции. Кроме функции SetPixels (множественное число) Unity поддерживает также функцию SetPixel (единственное число). Однако функция SetPixels показывает лучшую производительность, чем многократный вызов функции SetPixel.
- **Строки 111–114**: функция GetPixelFromArray принимает массив пикселей, координаты x и y пикселя и ширину текстуры в пикселях. На основании этих параметров она возвращает линейный индекс в массиве пикселей, где находится искомый пиксель.
- **Строки 117–131**: функция SetPixelInArray изменяет цвет пикселя в линейном массиве. Способ изменения определяется аргументом Blending. Если аргумент Blending имеет значение false, пиксель в текстуре просто замещается пикселем, переданным функции. Если аргумент Blending имеет значение true, будет выполнено смешивание цвета пикселя в аргументе с цветом пикселя в текстуре, с сохранением альфа-прозрачности. Аргумент Blending следует установить в true для рисования полупрозрачной кистью на текстуре для сложения и смешивания значений цвета.

Шаг 3 – настройка текстуры рисования

Теперь у нас есть работоспособный шейдер, сценарий рисования на текстуре и подготовленные текстуры, осталось пройти шаг за шагом настройку рисования на текстуре в Unity. Начнем с пустого проекта, содержащего шейдер, сценарий рисования на текстуре и две подготовленные текстуры: текстуру фона и текстуру кисти, как показано на рис. 9.17.

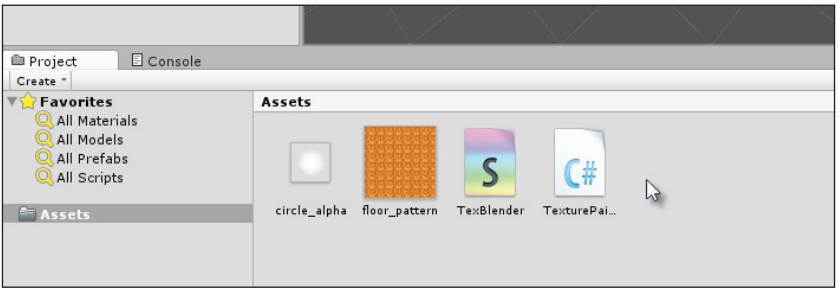


Рис. 9.17. Начальный проект рисования на текстуре

Настройте текстуру кисти в панели **Project** (Проект), указав небольшой размер (например, **32×32**) и выберите в ее поле **Format** (Формат) в значение **RGBA 32 bit** для использования альфа-прозрачности, как показано на рис. 9.18.

Создайте новый материал, используя шейдер `TextureBlender`, и выберите фоновую текстуру в слоте **MainTexture**, как показано на рис. 9.19.

Добавьте в сцену новый объект `Quad`, `Plane` или `Mesh` и удалите у него коллайдер, если имеется. На этом объекте будет выполняться рисование, хотя обнаружение щелчков мышью будет происходить на его дубликате. Я разделил меш для рисования и меш для обнаружения щелчков, чтобы меш для рисования мог иметь другой коллайдер или другие компоненты, если это потребуется.

После добавления четырехугольника `Quad` назначьте ему пользовательский материал с шейдером `TextureBlender`, как показано на рис. 9.20.

Скопируйте объект `Quad`, добавьте **Mesh Collider**, отключите **Mesh Renderer** и назначьте ему пустой диффузный материал. Этот – невидимый меш, и он будет служить для обнаружения щелчков мышью и выполнения операции рисования.

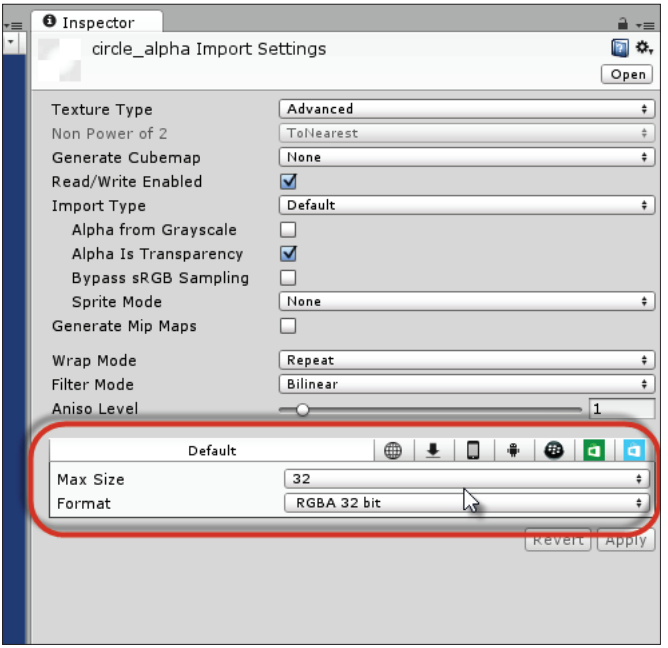


Рис. 9.18. Настройка текстуры кисти

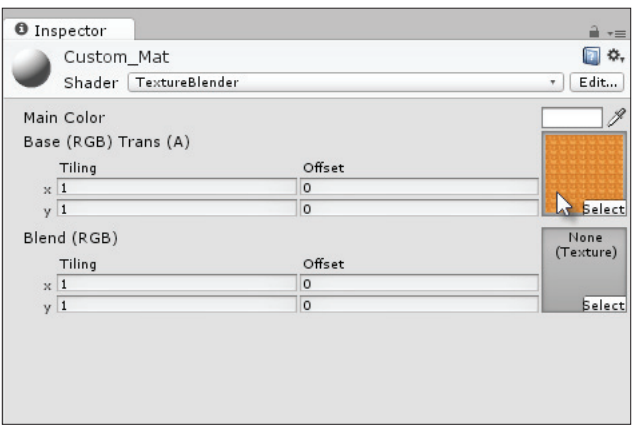


Рис. 9.19. Создание нового материала
с шейдером `TextureBlender`

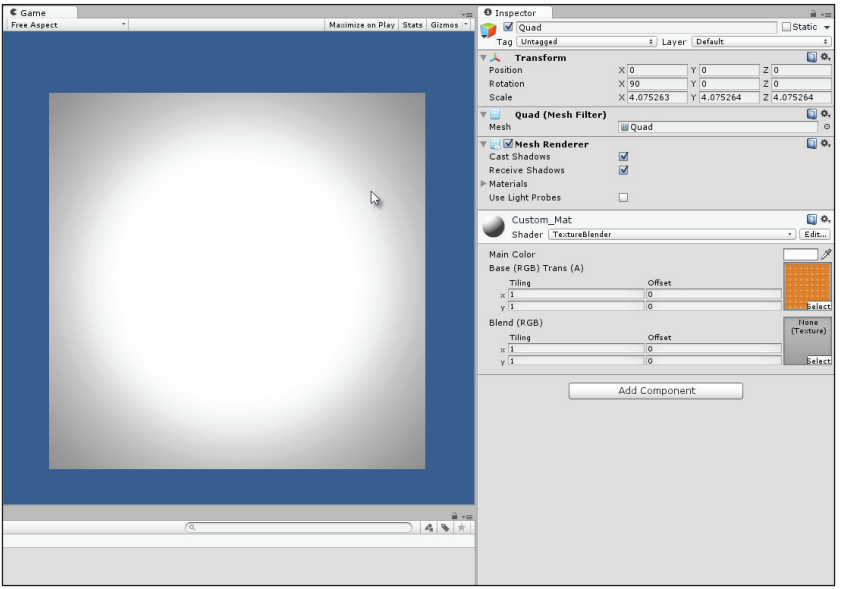


Рис. 9.20. Добавление нового объекта `Quad`

Кроме того, подключите сценарий `TexturePainter` к объекту и выберите в поле **Brush Texture** (Текстура кисти) текстуру кисти, в поле **Dest Mat** (Материал назначения) выберите `Custom_Mat`, как показано на рис. 9.21.

Теперь запустите приложение и начните щелкать мышью на меше. При этом на фоновой текстуре будут прорисовываться мазки кистью, как показано на рис. 9.22.

Итоги

В этой главе были рассмотрены множество деталей работы с двухмерной графикой. Под двухмерной графикой обычно подразумевают только двухмерные игры, но здесь основное внимание было уделено текстурам, что важно не только для двухмерных, но и для трехмерных игр. В главу включены приемы и идеи, которые можно отнести и к двухмерной плоскости, и к двухмерному пространству. В частности, был создан вращающийся скайбокс, как соединение управляемого скайбокса с настройками глубины камеры для визуализации по слоям. Затем был проработан вопрос программного создания геомет-

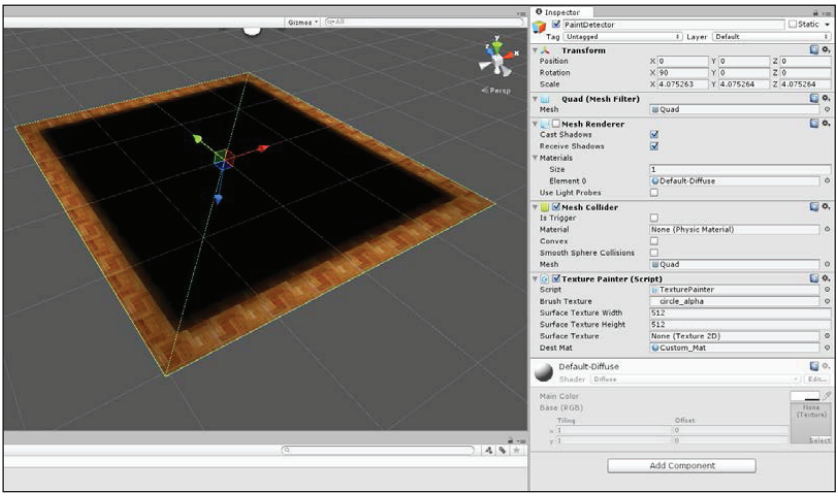


Рис. 9.21. Создания четырехугольника для определения щелчков

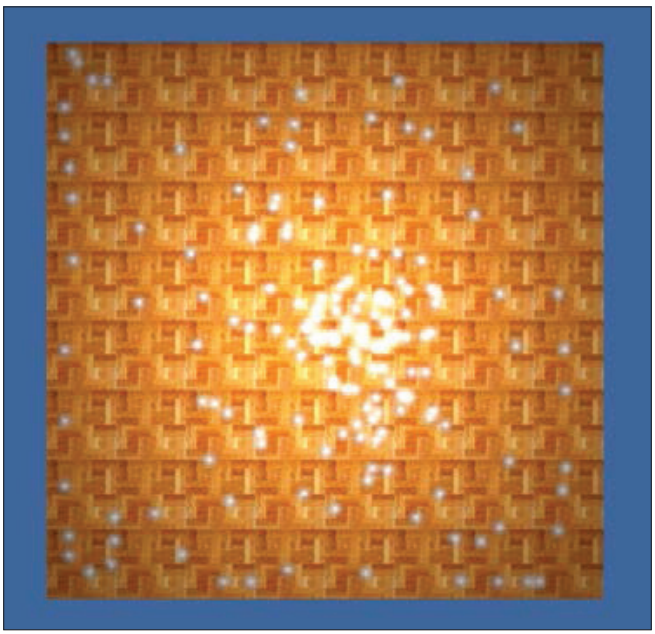


Рис. 9.22. Результат рисования на текстуре

рических фигур на примере плоскости. После добавления в Unity возможности создания примитивов четырехугольников `Quad`, тема создания плоскостей несколько растеряла свою актуальность, но методы и понятия создания объектов `Quad` сохраняют свою значимость, так как позволяют редактировать и настраивать любые меши, а не только `Quad`. Редактирование мешей в режиме реального времени позволяет создать широкий спектр эффектов, от ударной волны взрывов до желеподобных батутов. Мы рассмотрели анимацию UV-мешей. С ее помощью можно реализовать бесконечную прокрутку бесшовной текстуры двухмерного фона на объектах `Quad`, чтобы создать эффект перемещения, а также для имитации текучей воды и других движущихся поверхностей. Наконец, мы рассмотрели динамическое рисование на текстуре меша, запускаемое щелчками мышью, с помощью UV-координат и альфа-прозрачности для смешивания текстур. Этот прием часто применяется для наложения изображений на текстуры, таких как пулевые отверстия, брызги крови или рисование на холсте. В следующей главе мы рассмотрим ряд советов и приемов по повышению эффективности работы с проектами в Unity.

Глава 10

Управление исходными текстами и другие подсказки

В этой главе будут рассмотрены три рекомендации по разработке программного кода на C# и работе со сценариями. Эти рекомендации, несомненно, важны, но их тематически нельзя отнести ни к одной из предыдущих глав. Рекомендации приведены здесь в произвольном порядке, и главное, что служит оправданием включения их в книгу, – это их полезность, а также их отсутствие или лишь поверхностное упоминание в других источниках. Следовательно, в этой главе дан набор советов и подсказок, которые, собранные вместе, предлагают применимые на практике знания. Эти три рекомендации касаются следующих тем:

- управление исходными текстами с помощью Git;
- папки ресурсов и внешние файлы;
- загрузка и сохранение игр.

Git – управление исходными текстами

Под термином **управление исходными текстами** (source control) или **управление версиями** (revision control) понимают любое программное обеспечение, имеющее цель сделать практическую разработку приложений проще и безопаснее, насколько это возможно. Если коротко, такое программное обеспечение позволяет легко и быстро отслеживать изменения в файлах и при необходимости отменять их, а также обмениваться изменениями с другими разработчиками. Как правило, разработка программного обеспечения (в том числе разработка игр) основывается на двух важных аспектах, или ингредиентах. Во-первых, это коллективные усилия, где несколько разработчиков

работают вместе как часть команды в одном месте (например, в офисе) или в удаленных друг от друга местах, а обмен ведется через виртуальное пространство (например, виртуальный офис, форум или даже электронная почта). Во-вторых, в процессе разработки программисты настраивают, редактируют и вносят улучшения в исходный код. Из этих двух, казалось бы, простых аспектов вытекает ряд важных потребностей, которые и должно удовлетворить программное обеспечение управления версиями.

Эти потребности заключаются в следующем.

- **Сотрудничество**: когда несколько разработчиков работают над общим проектом, им обычно нужно использовать общие исходные файлы. Их можно пересылать по электронной почте или какими-то другими способами, но при работе над большими и долгосрочными проектами поддерживать такой обмен становится сложно. С течением времени очень быстро нарастает сложность мониторинга изменений и интеграции двух версий файла в единый файл.
- **Откат**: иногда внесенные изменения или улучшения оказываются ошибочными. Редактирование или исправления не всегда приводят к ожидаемому результату и должны быть отменены, код следует вернуть в прежнее состояние. Можно, конечно, сохранять копии предыдущих файлов вручную, но работа с такими копиями в течение длительного времени может стать утомительной и чересчур запутанной.
- **Отслеживание истории изменений**: часто необходимо отслеживать внесение изменений в код, особенно при его отладке. Если кто-то внес правки, вы наверняка захотите знать, кто изменил код, почему и когда. Опять же, вы можете вручную вести журналы изменений, писать комментарии или как-то иначе документировать изменения, но это будет достаточно трудоемко.

Управление версиями предназначено для решения всех трех основных проблем: сотрудничества, отката и отслеживания изменений. Существует множество приложений управления версиями, это Git, Perforce, Microsoft Team Foundation Server и др. В этой главе будет рассмотрен пакет Git, он пользуется популярностью, бесплатен, кросс-платформенный и с открытым исходным кодом. Использование Git начинается с настройки специальной базы данных, называемой **хранилищем**, или **репозиторием** (repository), которая может быть локальной (на компьютере) или удаленной (в сети). После на-

стройки вы будете иметь возможность отслеживать любые изменения в ваших проектах Unity, откатывать их при необходимости и обмениваться, или сотрудничать с другими разработчиками. Давайте посмотрим, как настроить Git, используя графический интерфейс пользователя.

Шаг № 1 – загрузка

Есть много способов установки Git для проектов Unity. В этой главе рассматривается соединение официального пакета Git с внешним интерфейсом TortoiseGit. С помощью этих двух пакетов разработчики могут отслеживать и управлять всеми изменениями в своих проектах, независимо от того, работают они одни или в команде.

Для начала загрузите и установите программное обеспечение с официального сайта Git http://git-scm.com/.

Более подробную информацию об использовании Git можно найти в бес-

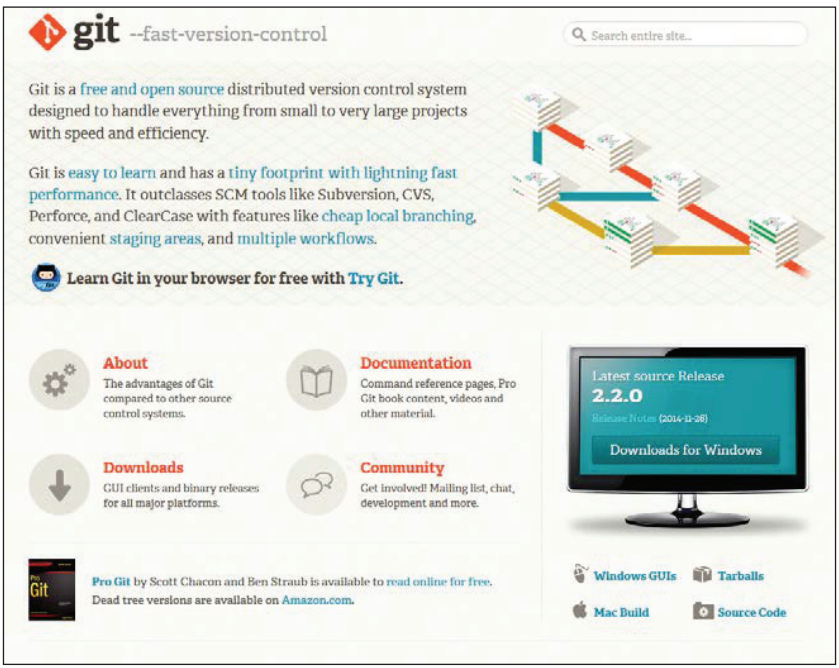


Рис. 10.1. Загрузка и установка Git

платной электронной книге «Pro Git», Скотта Чакона (Scott Chacon) и Бена Штрауба (Ben Straub), выпущенной издательством Apress и доступной по адресу http://git-scm.com/book/en/v2[1].

После загрузки и установки Git имеет смысл установить и пакет TortoiseGit. Он не является частью оригинального пакета Git, это дополнительный графический интерфейс для Windows, который позволит вам интегрировать Git с оболочкой Windows, а также взаимодействовать с Git через графический интерфейс, а не с помощью командной строки.

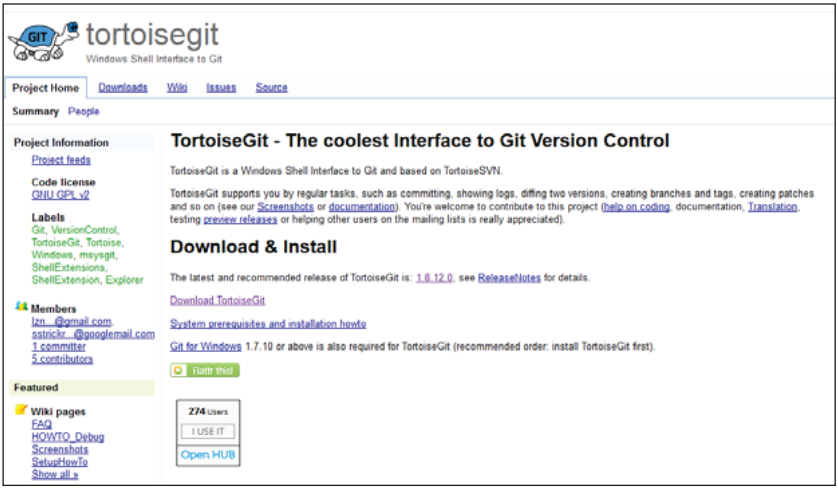


Рис. 10.2. Загрузка и установка TortoiseGit

Для загрузки и установки TortoiseGit перейдите на страницу https://code.google.com/p/ tortoisegit/.

Шаг № 2 – добавление проекта в репозиторий

Основная цель установки Git состоит в возможности отслеживать изменения в проекте, откатывать их при необходимости, использовать в качестве резервной копии при потере исходных файлов и совместном с другими разработчиками внесении изменений. Далее надо определиться, какой существующий проект вы будете поддерживать.

---

[1] Эта версия (v2) книги переведена на русский язык лишь частично и доступна там же, по адресу http://git-scm.com/book/ru/v2, однако версия (v1) переведена полностью: http://git-scm.com/book/ru/v1. – *Прим. ред.*

То есть, после установки и Git, и TortoiseGit надо создать новый проект Unity или выбрать уже существующий. Ниже показано типичное содержимое папки проекта Unity (рис. 10.3).

Выбрав проект, откройте папку проекта в проводнике Windows, чтобы увидеть файлы проекта. Если вы не знаете или не помните где находится папка, ее можно открыть в проводнике прямо из редактора Unity. Для этого щелкните правой кнопкой мыши внутри панели **Project** (Проект) и выберите в контекстном меню пункт **Show in Explorer** (Показать в проводнике), как показано на рис. 10.4.

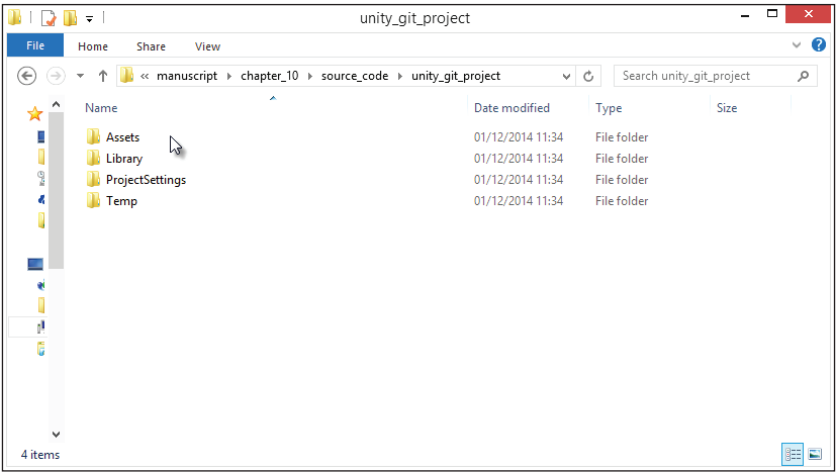


Рис. 10.3. Типичное содержимое папки проекта Unity

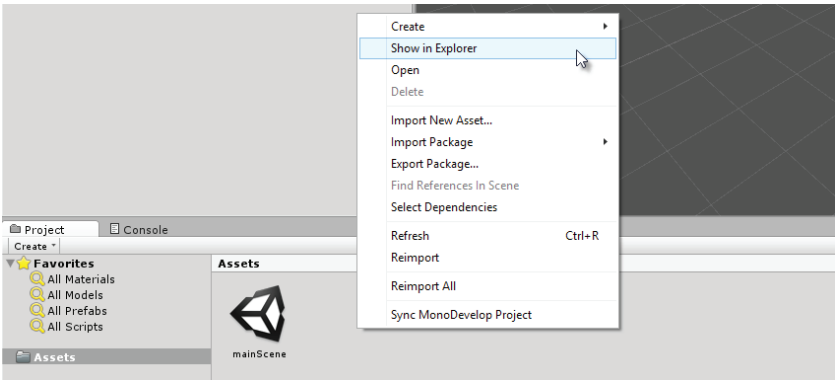


Рис. 10.4. Доступ к папке проекта из редактора Unity

Шаг № 3 – настройка Unity для управления исходными текстами

Git работает и с двоичными, и с текстовыми файлами, но отдает предпочтение текстовым файлам. В процессе работы редактор Unity генерирует множество файлов метаданных для проекта и файлов импортированных ресурсов. Эти файлы находятся в папке проекта Unity. По умолчанию они скрыты и имеют двоичной формат. Некоторые из метафайлов зависят от установленной версии Unity, например настройки интерфейса, тогда как другие относятся к ресурсам и данным, являющимся частью проекта, таким как меши, текстуры и файлы сценариев. Чтобы получить наилучшие результаты при работе с Git, нужно изменить поведение Unity по умолчанию, сделав метафайлы видимыми в панели проекта, и настроить замену двоичных файлов текстовыми. Для этого цели выберите пункт меню **Edit** ⇒ **Project Settings** ⇒ **Editor** (Правка ⇒ Настройки проекта ⇒ Редактор).

Далее настройте в инспекторе объектов поле **Version Control** (Управление версиями), выбрав значение **Visible Meta Files** (Показывать мета-файлы), и поля **Asset Serialization** (Сериализация ресурсов), выбрав значение **Force Text** (Принудительно преобразовывать в текст), как показано на рис. 10.5.

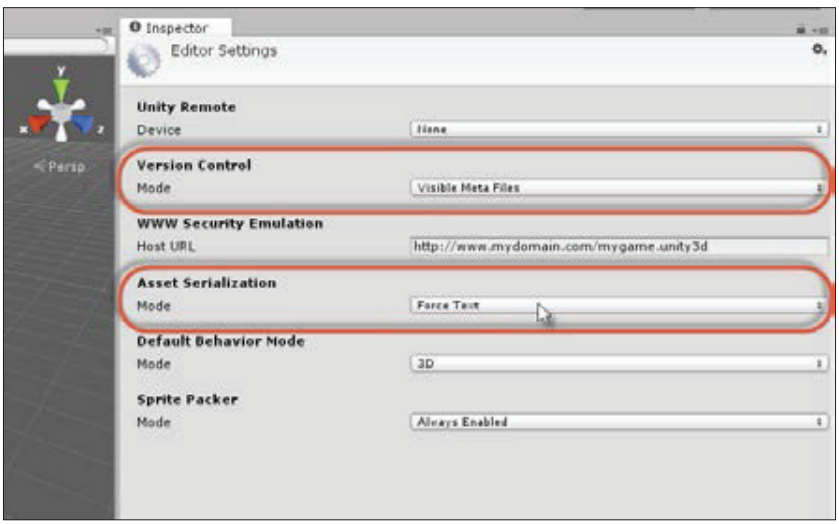


Рис. 10.5. Настройка Unity для управления версиями

После смены значений этих параметров станут видимыми файлы с расширением `.meta`, соответствующие каждому ресурсу проекта, в том числе и сцене. Кроме того, метафайлы будут преобразованы в читаемый текстовый формат, что сделает возможным их редактирование (хотя ручное редактирование не рекомендуется). Взгляните на рис. 10.6.

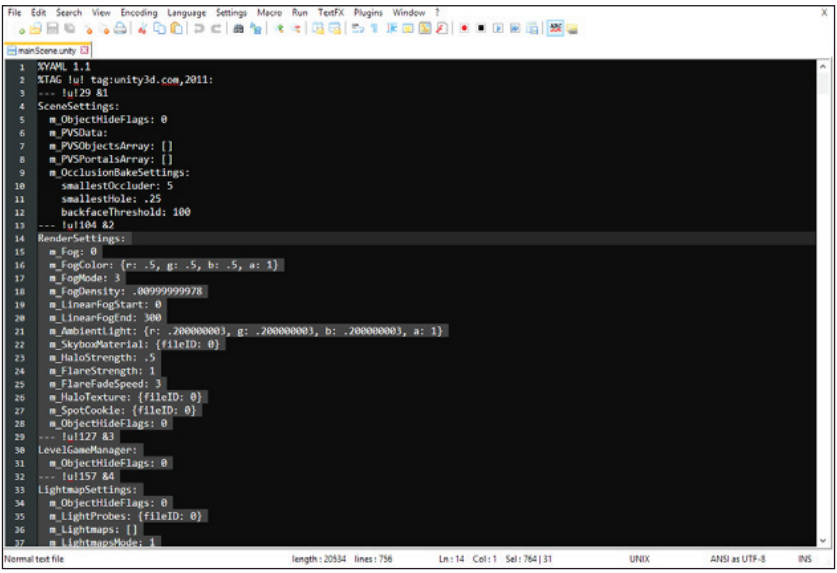


Рис. 10.6. Содержимое ресурса сцены (текстовый формат) в текстовом редакторе

Шаг № 4 – создание репозитория

Следующий этап после создания и настройки проекта – создание базы данных Git, или репозитория, для отслеживания и поддержки всех изменений в файлах. Репозиторий может быть удаленным (в сети или на внешнем компьютере) или локальным (на том же компьютере). В хранилище будут храниться оригинальные файлы и все изменения, сделанные в течение длительного времени, что позволит вернуться к более ранним версиям файлов, если это потребуется. Хранилище также может быть сделано общим и объединено с другими хранилищами для совместного использования файлов. В этой главе рассматриваются только локальные хранилища, давайте создадим одно из них. Для этого откройте папку проекта Unity (корне-

вую папку), а затем щелкните правой кнопкой мыши, чтобы вызвать контекстное меню Windows. Выберите пункт **Git Init Here** (Создать репозиторий здесь).

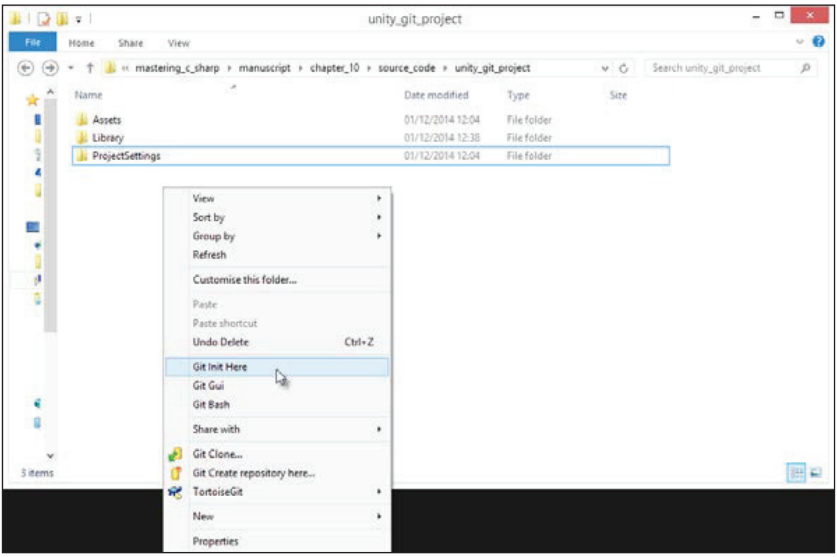


Рис. 10.7. Создание репозитория Git

После этого будет создана новая скрытая папка `.git`. Она содержит все файлы хранилища проекта. Ярлыки файлов и папок будут отмечены красными значками, указывающими, что файлы в папке проекта еще не добавлены в репозиторий и Git не может отслеживать вносимые в них изменения (этим мы займемся в ближайшее время). Это показано на рис. 10.8.

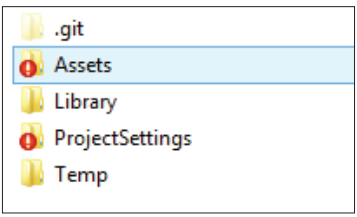


Рис. 10.8. Папки с красными значками содержат файлы, не включенные в репозиторий Git

Шаг № 5 – игнорируемые файлы

Итак, репозиторий Git создан и готов к приему первого набора файлов (*commit*). Однако перед их добавлением заметим, что есть определенные файлы, которые имеет смысл игнорировать. Приложение

Unity использует специальные системные файлы, относящиеся к проекту или ко всей системе в целом, которые в данном контексте не важны. Это файлы настроек пользовательского интерфейса, файлы только для чтения, временные файлы и содержащие некоторые другие данные, которые не должны добавляться в хранилище. Чтобы игнорировать их, нужно создать текстовый файл `.gitignore` в корневой папке проекта и перечислить нем все файлы и папки, которыми можно пренебречь, как показано на рис. 10.9.

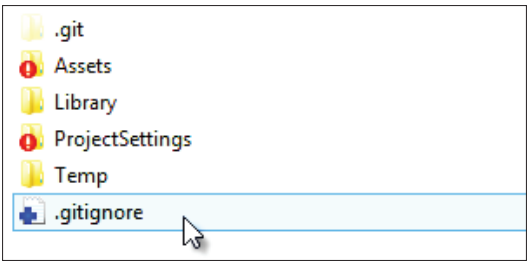


Рис. 10.9. Создание файла для исключения определенных типов файлов из хранилища

Для Unity файл (`.gitignore`) должен выглядеть, как показано ниже. Удостоверьтесь, что файл помещен в корневую папку:

```
[Ll]ibrary/
[Tt]emp/
[Oo]bj/
[Bb]uild/
/*.csproj
/*.unityproj
/*.sln
/*.suo
/*.user
/*.userprefs
/*.pidb
/*.booproj
sysinfo.txt
```

Шаг № 6 – первая фиксация изменений

Теперь репозиторий настроен для получения первого набора файлов проекта Unity. Чтобы добавить их, щелкните правой кнопкой мыши в окне корневой папки и в контекстном меню выберите **Git Commit** ⇒

Master (Передать в Git ⇒ Главная ветвь). В Git файлы обычно представлены не по одному, а в виде партий. Окно **Commit** (Зафиксировать) позволяет выбрать файлы для добавления в хранилище.

Нажмите кнопку **All** (Все), чтобы выбрать все файлы в папке, а затем введите описание текущей версии в поле **Message** (Сообщение). Цель сообщения – дать понять любому пользователю, что содержат новые версии файлов. Когда все будет готово, щелкните на кнопке **OK**, чтобы передать файлы в репозиторий (см. рис. 10.10).

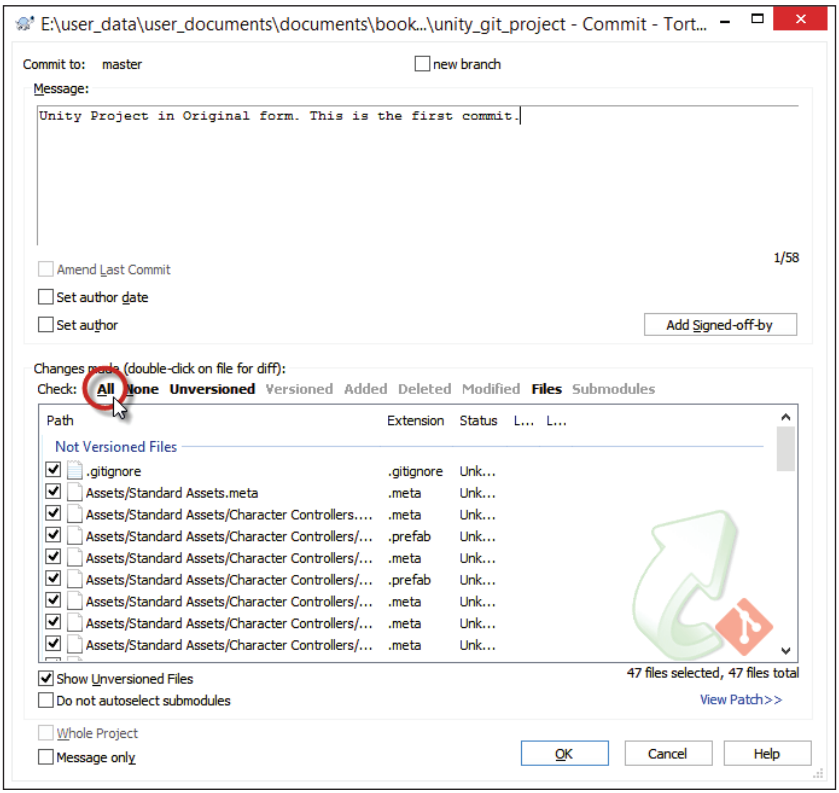


Рис. 10.10. Передача оригинальных файлов проекта

После завершения передачи значки, отмечающие файлы, станут зелеными, что указывает на идентичность файлов в папке проекта и файлов в репозитории (рис. 10.11).

Шаг № 7 – изменение файлов

Git должен полностью отслеживать все изменения файлов, а значит он должен хранить не только исходные файлы, но и все последующие изменения и правки, позволяя вернуться к предыдущим версиям.

Если теперь вернуться в Unity и изменить файлы, добавив новые ресурсы или отредактировав существующие, значки на измененных файлах в проводнике Windows вновь станут красными, указывая на несоответствие между локальными файлами и файлами в репозитории.

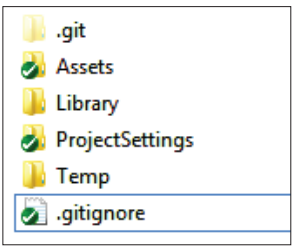


Рис. 10.11. Файлы в папке проекта соответствуют файлам в хранилище

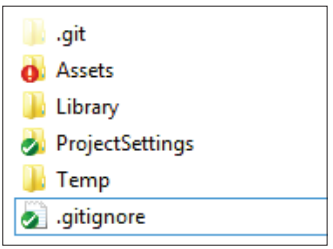


Рис. 10.12. Изменение файлов

Если вы решите, что последние изменения были ошибкой, и захотите вернуться к предыдущему состоянию, это можно сделать, щелкнув правой кнопкой мыши в окне папки проекта и выбрав в контекстном меню пункт **TortoiseGit ⇒ Revert...** (TortoiseGit ⇒ Вернуть...), как показано на рис. 10.13.

В появившемся диалоге **Revert** (Вернуть) можно выбрать восстанавливаемые файлы. Выберите все необходимые файлы, а затем щелкните на кнопке **ОК**. Git восстановит выбранные файлы, перезаписав их локальные версии последними версиями из хранилища.

Но чаще не требуется откатывать или отменять последние изменения. Вы могли внести действительно нужные изменения и теперь их нужно сохранить в репозитории Git, как последнюю версию файлов. Если это так, просто снова отправьте файлы в репозиторий: щелкните правой кнопкой мыши в окне папки проекта и в контекстном меню выберите пункт **Git Commit ⇒ Master** (Передать в Git ⇒ Главная ветвь). Внесите новое значимое сообщение в поле **Message** (Сообщение) и щелкните на кнопке **ОК**.

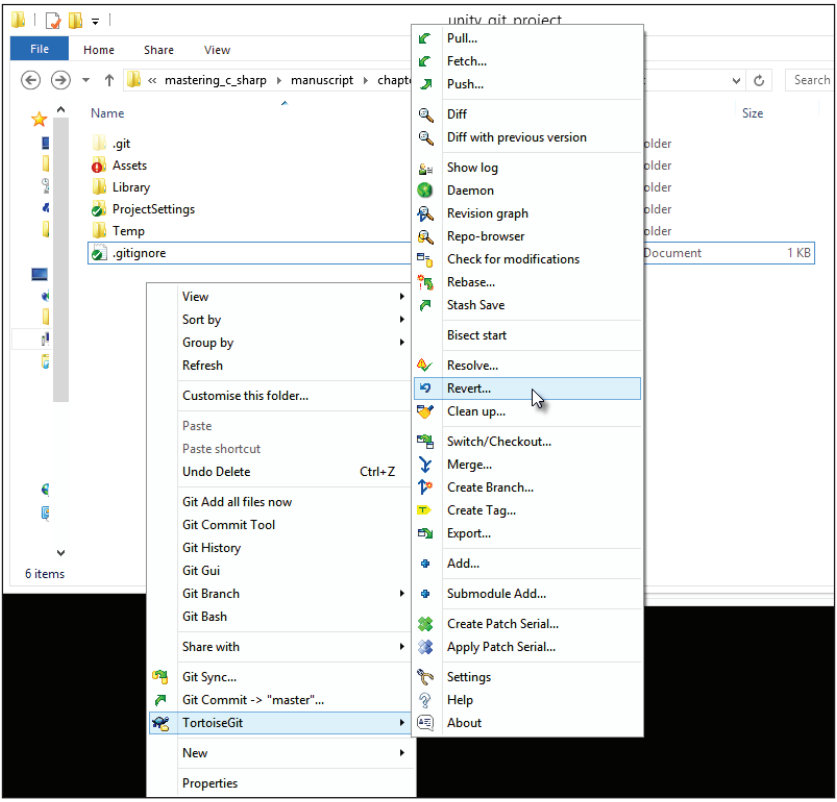


Рис. 10.13. Откат (отмена) последних изменений

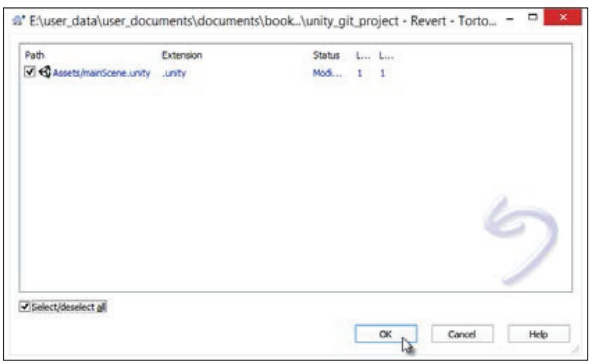


Рис. 10.14. Выбор файлов для восстановления

Шаг № 8 – получение файлов из хранилища

После начальной отправки всех файлов, если вы намеренно или случайно удалите все файлы в папке проекта, за исключением папки `.git` и файла `.gitignore`, вы сможете восстановить последние версии файлов, так как они имеются в хранилище Git.

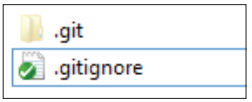


Рис. 10.15. Удаленные файлы можно восстановить из хранилища Git

Конечно, если при чтении книги вы повторяете описываемые действия и удаляете свои файлы, не забудьте вручную сохранить их резервные копии на случай, если что-то пойдет не так во время тестирования!

Для восстановления щелкните правой кнопкой мыши в окне папки проекта и выберите в контекстном пункт **TortoiseGit ⇒ Switch/Checkout** (TortoiseGit ⇒ Выбрать/Извлечь), как показано на рис. 10.16.

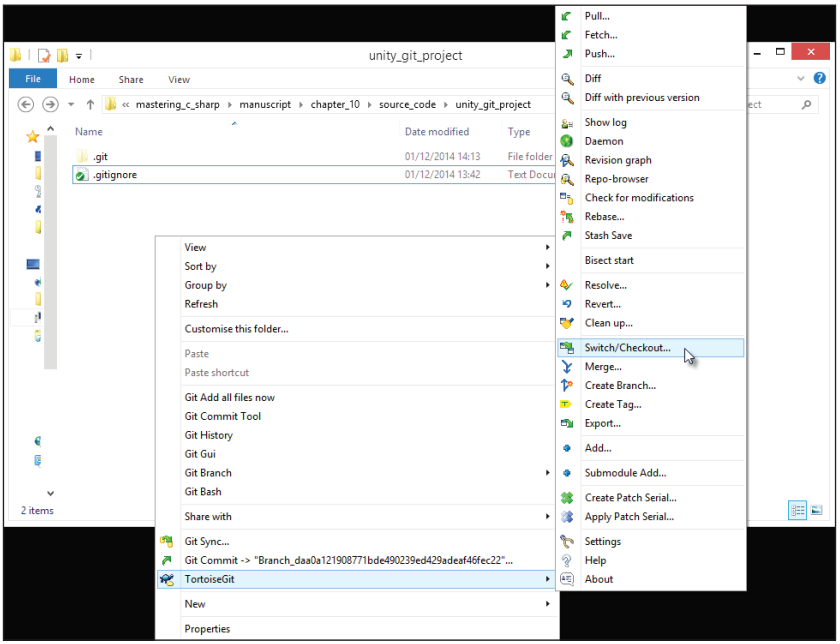


Рис. 10.16. Восстановление последних версий файлов из репозитория

В диалоге **Switch/Checkout** (Выбрать/Извлечь) выберите значение **Master** (Главная ветвь) в поле **Switch To** (Выбрать).

Вам также может потребоваться установить флажок **Force** (Принудительно) (подробности ищите в документации). Затем щелкните на кнопке **OK** для восстановления файлов. Сразу после извлечения файлов вы увидите картину, изображенную на рис. 10.17.

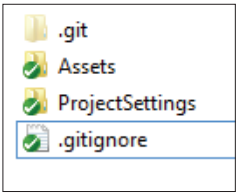


Рис. 10.17. Восстановление последних версий файлов с помощью опции Checkout

Иногда бывает желательно вернуть не самое последнее зафиксированное состояние проекта, а получить более ранние версии файлов. Для этого сначала выберите в контекстном пункт **TortoiseGit** ⇒ **Switch/Checkout** (TortoiseGit ⇒ Выбрать/Извлечь), чтобы открыть диалог **Switch/Checkout** (Выбрать/Извлечь). А затем выберите переключатель **Commit** (Версия) в группе **Switch To** (Выбрать).

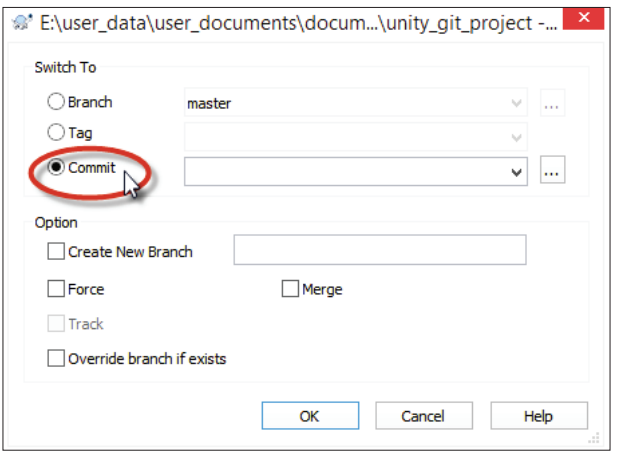


Рис. 10.18. Выбор переключателя **Commit** (Версия) для восстановления зафиксированного ранее состояния

Нажмите кнопку обзора (**...**) рядом с полем **Commit** (Версия) для отображения списка доступных версий и выберите нужную версию. Затем щелкните на кнопке **OK**, чтобы выйти из диалога **Repo Commits** (Версии в репозитории), и щелкните на кнопке **OK** для подтверждения извлечения выбранной версии. Файлы выбранной версии будут

восстановлены в папку проекта. Помните, что каждая версия имеет автора (для тех, кто работает в команде), и это позволяет получить сведения о том, кто и какие внес изменения.

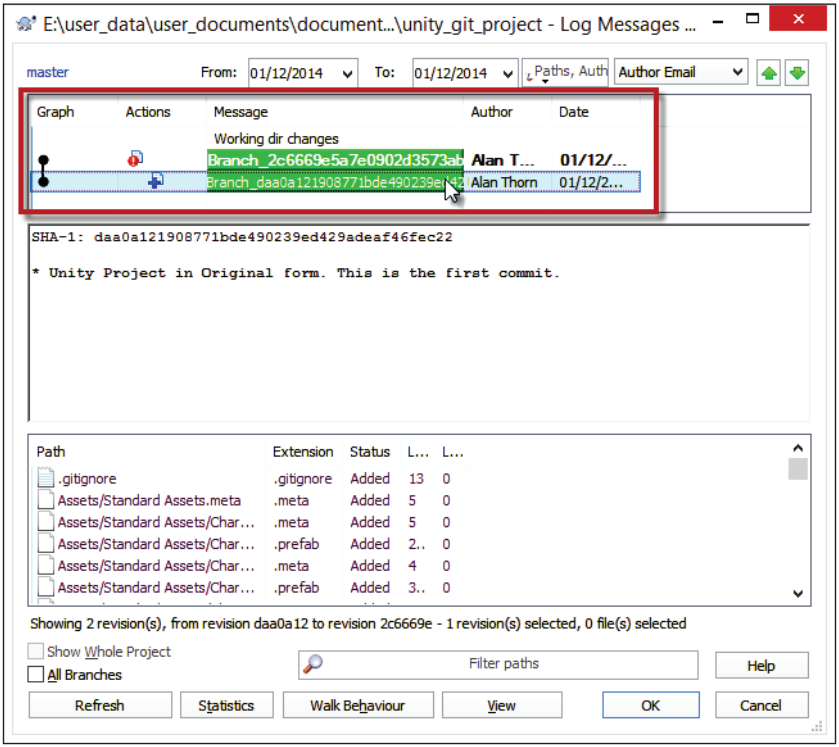


Рис. 10.19. Выбор более ранней версии для восстановления из репозитория

Шаг № 9 – просмотр репозитория

Иногда не нужно ни добавлять файлы в репозиторий, ни извлекать их оттуда, все, что нужно, – просто просмотреть, что содержит репозиторий. Это можно сделать легко и быстро с помощью браузера репозитория, который является частью TortoiseGit. Чтобы получить доступ к браузеру, выберите в контекстном меню пункт **TortoiseGit ⇒ Repo Browser** (TortoiseGit ⇒ Браузер репозитория).

Браузер репозитория (Repo Browser) позволяет просматривать файлы и иерархии в иерархическом виде (см. рис. 10.21).

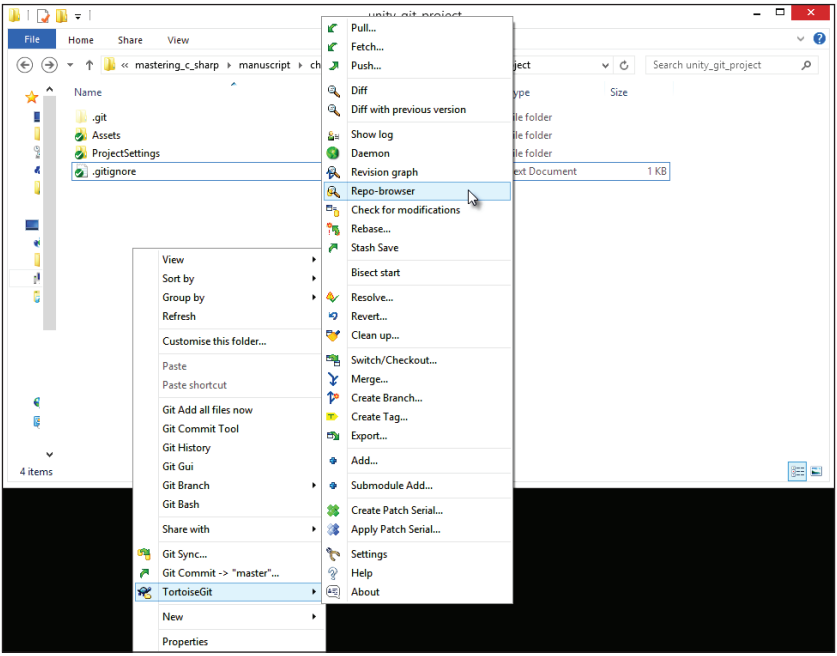


Рис. 10.20. Открытие браузера репозитория

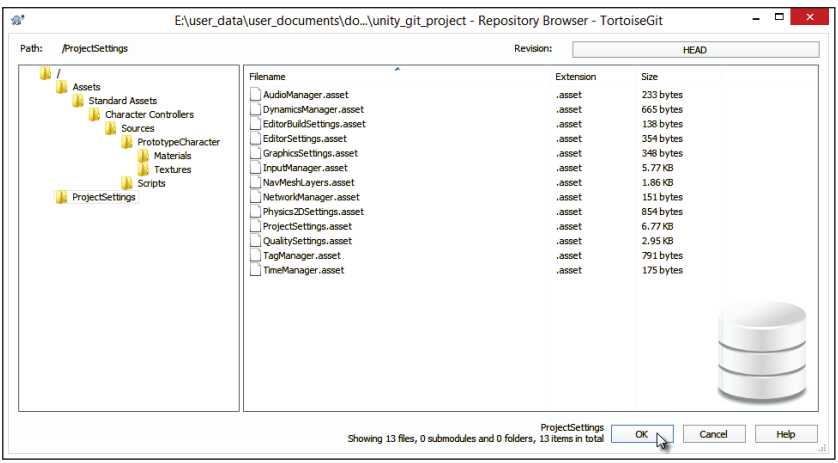


Рис. 10.21. Просмотр файлов в репозитории

Папка ресурсов и внешние файлы

Ваши игры часто будут нуждаться в данных, загруженные из внешних файлов, таких как XML-файлы с субтитрами и локализованными текстами, или сериализованные уровни. Взгляните на рис. 10.22.

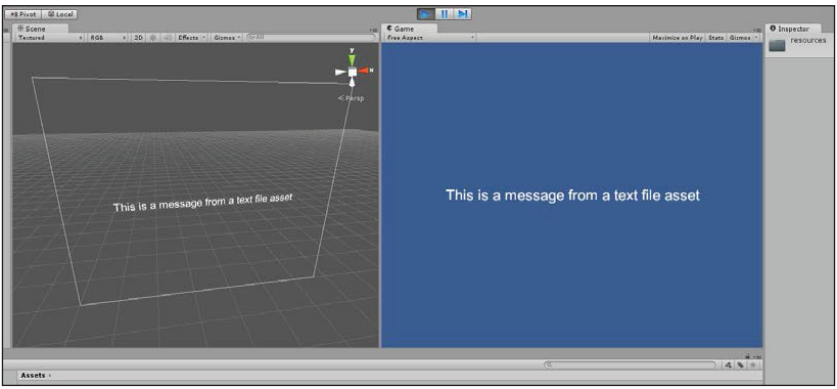


Рис. 10.22. Вывод сообщения, загруженного из внешнего текстового файла, скомпилированного вместе с проектом

В этих случаях вам понадобится реализация определенного набора возможностей. Первой из них является возможность динамически загружать данные из файлов в память, чтобы среда Unity смогла их разобрать и использовать. Во-вторых, возможность редактировать содержимое файлов после импорта в Unity, с последующим обновлением игры, учитывающим внесенные изменения, без правки кода. В-третьих, возможность создания автономного дистрибутива игры в виде одного выполняемого файла, где внешние файлы будут включены в файл сборки Unity, а не входить в дистрибутив в вид отдельных, доступных для редактирования файлов. Многие предпочитают не распространять игру в виде дистрибутива, содержащего внешние файлы, такие как XML-файлы, открытые для редактирования. Вместо этого многие редактируют и изменяют файлы в редакторе Unity, а затем компилируют и встраивают их в окончательную автономную сборку, вместе с другими ресурсами. Все это можно реализовать с помощью папок ресурсов.

Чтобы использовать папки ресурсов, создайте папку с именем `resources` в проекте Unity. Проект может иметь ни одной, одну или

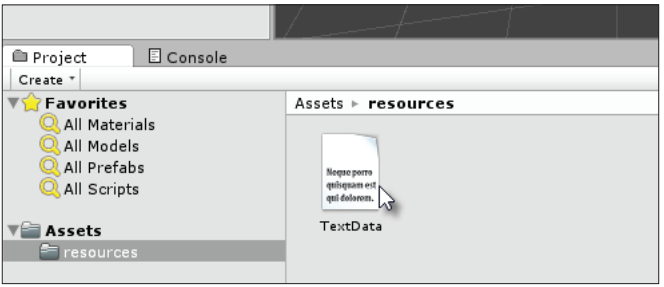


Рис. 10.23. Добавление внешних файлов в папку resources

несколько папок resources. В эту папку добавьте все текстовые файлы, которые загружаются во время выполнения.

После добавления файла в папку resources, его можно загрузить в память с помощью функции Resources.Load. В листинге 10.1 показано, как загрузить текстовый ресурс в компонент интерфейса.

Листинг 10.1. Загрузка текстового ресурса

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
//-------------------------------------------
public class LoadTextData : MonoBehaviour
{
  // Ссылка на компонент интерфейса
  private Text MyText = null;

  // Ссылка на текстовый ресурс в папке resources
  private TextAsset TextData = null;

  //-------------------------------------------
  // Этот метод выполняет инициализацию
  void Awake () {
    // Получить ссылку на компонент интерфейса
    MyText = GetComponent<Text>();

    // Загрузить текст из папки resources
    TextData = Resources.Load("TextData") as TextAsset;
  }
  //-------------------------------------------
  // Вызывается при отображении каждого кадра
  void Update () {
    // Изменить текст в компоненте интерфейса
    MyText.text = TextData.text;
```

```
    }
    //-------------------------------------------
}
//-------------------------------------------
```

Более подробную информацию о папке `resources` и классе `Resources` можно найти в электронной документации Unity по адресу http://docs.unity3d.com/ScriptReference/Resources.html.

Пакеты ресурсов и внешние файлы

Если вы используете версию Unity Pro и хотите предоставить пользователям возможность модификации содержимого игры, то есть возможность добавлять ресурсы и расширения, вам понадобятся пакеты ресурсов. Пакеты ресурсов позволяют упаковать вместе множество разнородных ресурсов в один внешний файл, отделенный от основного проекта, который можно загрузить в любой проект Unity динамически – из локального файла на диске или из Интернета (см. рис. 10.24).

Для начала импортируйте сценарий редактирования пакетов ресурсов, который позволяет создавать пакеты ресурсов из панели **Project** (Проект). Для этого создайте файл сценария на C# в папке

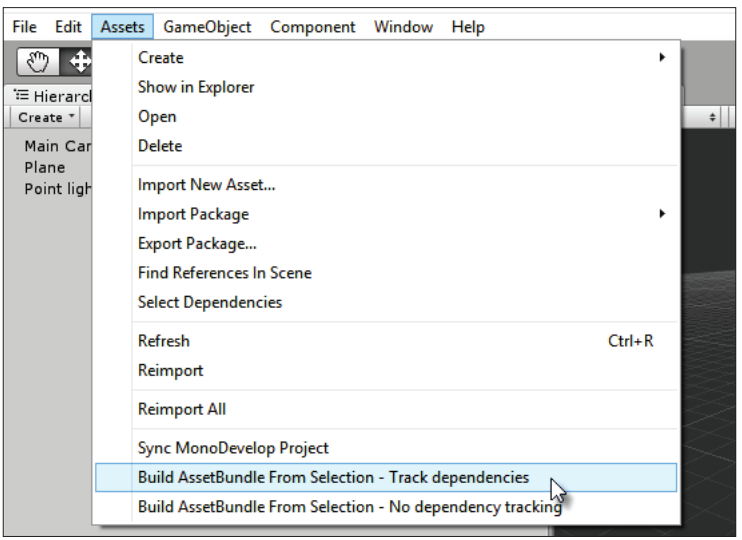


Рис. 10.24. Сборка пакета ресурсов из выбранных активов

Editor проекта и поместите в него содержимое листинга 10.2 или же загрузите готовый сценарий со страницы http://docs.unity3d.com/ScriptReference/BuildPipeline.BuildAssetBundle.html.

```
// Пример на C#
// Создает пакет ресурсов из объектов
// выбранных в представлении проекта.
// После компиляции выберите пункт меню "Menu" -> "Assets" и укажите
// один из вариантов сборки пакета ресурсов
using UnityEngine;
using UnityEditor;

public class ExportAssetBundles {
  [MenuItem("Assets/Build AssetBundle From Selection - Track dependencies")]
  static void ExportResource () {
    // Вывести диалог сохранения
    string path = EditorUtility.SaveFilePanel ("Save Resource", "",
      "New Resource", "unity3d");
    if (path.Length != 0) {
      // Создать файл пакета ресурсов из выбранных файлов.
      Object[] selection = Selection.GetFiltered(typeof(Object),
        SelectionMode.DeepAssets);
      BuildPipeline.BuildAssetBundle (Selection.activeObject, selection,
        path,
        BuildAssetBundleOptions.CollectDependencies |
        BuildAssetBundleOptions.CompleteAssets);
      Selection.objects = selection;
    }
  }
  [MenuItem("Assets/Build AssetBundle From Selection - No dependency tracking")]
  static void ExportResourceNoTrack () {
    // Вывести диалог сохранения
    string path = EditorUtility.SaveFilePanel ("Save Resource", "",
      "New Resource", "unity3d");
    if (path.Length != 0) {
      // Создать файл пакета ресурсов из выбранных файлов.
      BuildPipeline.BuildAssetBundle (Selection.activeObject,
        Selection.objects, path);
    }
  }
}
```

Для создания пакета ресурсов выберите все ресурсы в панели **Project** (Проект) для включения в пакет, а затем выберите в меню пункт **Assets** ⇒ **Build AssetBundle from Selection** (Ресурсы ⇒ Создать пакет ресурсов из выбранных). После этого выберите папку для сохранения пакета.

Чтобы проверить пакет ресурсов, создайте новый проект или откройте существующий, без ресурсов, и загрузите пакет в свой проект

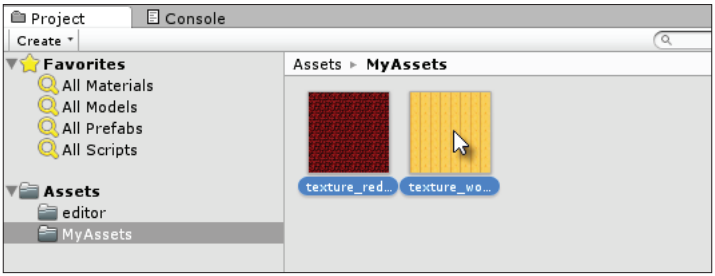


Рис. 10.25. Выбор ресурсов для включения в пакет

во время выполнения с помощью класса WWW. Рассмотрим следующий пример в листинге 10.3, который загружает пакет ресурсов из локального файла, извлекает текстуру и накладывает ее на меш.

Листинг 10.3. Проверка пакета ресурсов

```
using UnityEngine;
using System.Collections;
public class LoadAssetBundle : MonoBehaviour
{
  // Ссылка на меш
  private MeshRenderer MR = null;

  // Этот метод выполняет инициализацию
  IEnumerator Start ()
  {
    // Получить локальный файл пакета ресурсов
    WWW www = new WWW (@"file:///c:\asset_textures.unity3d");

    // Ждать завершения загрузки
    yield return www;

    // Извлечь текстуру из пакета
    Texture2D Tex = www.assetBundle.Load
      ("texture_wood",typeof(Texture2D)) as Texture2D;

    // Наложить текстуру на меш
    MR = GetComponent<MeshRenderer>();
    MR.material.mainTexture = Tex;
  }
}
```

Ниже показано, как выглядит текстура из пакета ресурсов.

Более подробную информацию о пакетах ресурсов можно найти в электронной документации Unity по адресу http://docs.unity3d.com/Manual/AssetBundlesIntro.html.

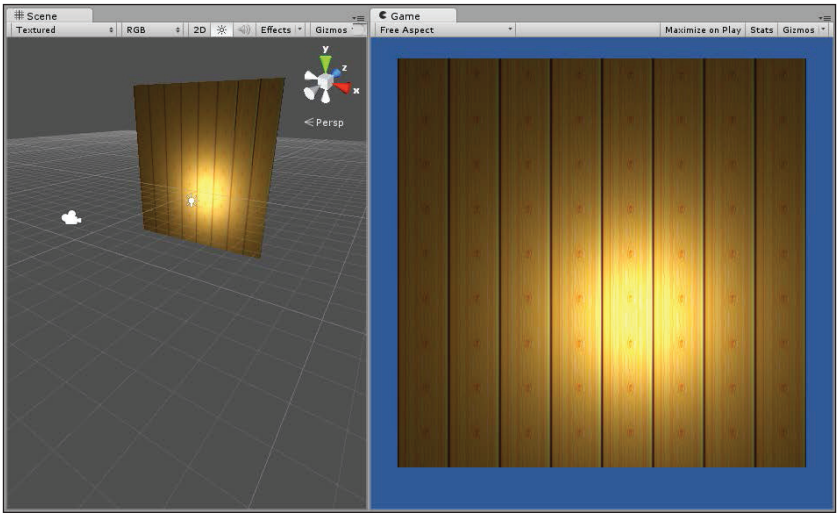


Рис. 10.26. Загрузка текстуры из пакета ресурсов

Хранимые данные и сохранение игры

Предоставление возможности сохранять и восстанавливать состояние игры необходимо во многих играх, особенно с длительным игровым процессом, таких как приключенческие игры, стратегии и ролевые игры. Для реализации подобной возможности надо позволить пользователю сохранять состояние игры во внешний файл и загружать его из внешнего файла.

Это реализуется в Unity с помощью сериализации данных в XML-файлы или в двоичные файлы (см. рис. 10.27).

Сериализация (Serialization) – это процесс преобразования данных, хранящихся в памяти (например, состояние компонента игрового объекта) в поток, который можно записать в файл, а затем вновь загрузить из файла для восстановления состояния компонента в памяти на момент сохранения. Как результат, процесс сохранения игры заключается в выборе данных для сохранения и загрузки (для каждой конкретной игры), и создании нового класса для работы с этими данными. Рассмотрим следующий пример в листинге 10.4 (`ObjSerializer.cs`). Этот файл можно подключить к любому игровому объекту для сериализации свойств его компонента `Transform` и записи во внешний файл в формате XML, например, или в двоичном формате. Для пре-

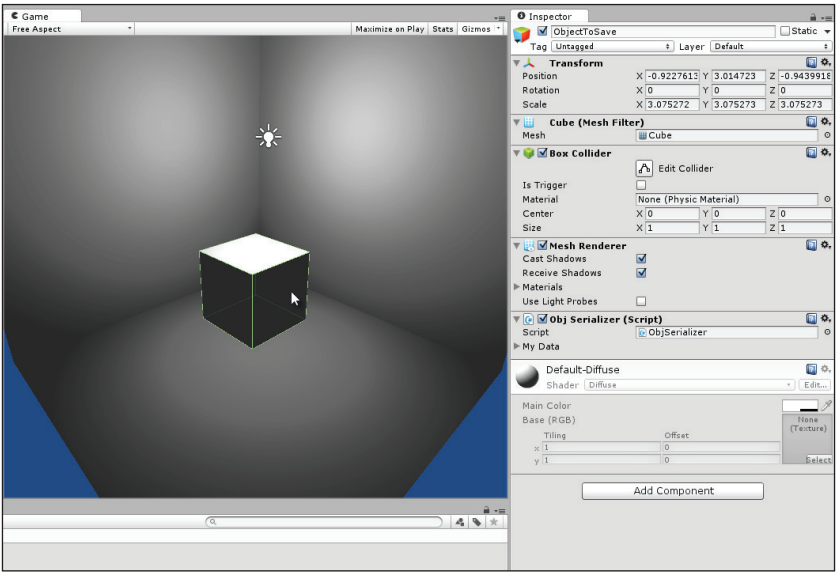


Рис. 10.27. Сохранение свойств компонента `Transform` в XML-файл

образования объекта в памяти в формат XML класс использует объект `XmlSerializer` , а для преобразования в двоичный формат – объект `BinaryFormatter`. XML-файлы – это текстовые файлы, доступные для чтения, тогда как двоичные файлы не могут быть прочитаны людьми.

Листинг 10.4. Реализация сохранения и восстановления состояния игры

```
001 //-----------------------------------------------
002 using UnityEngine;
003 using System.Collections;
004 using System.Collections.Generic;
005 using System.Xml;
006 using System.Xml.Serialization;
007 using System.Runtime.Serialization.Formatters.Binary;
008 using System.IO;
009 //-----------------------------------------------
010 public class ObjSerializer : MonoBehaviour
011 {
012   // Данные для сохранения
013   [System.Serializable]
014   [XmlRoot("GameData")]
015   public class MySaveData
016   {
```

```
017     // Преобразует данные для загрузки из файла или сохранения в файл,
018     // представляющие объект трансформации,
019     // в простые значения
020     [System.Serializable]
021     public struct DataTransform
022     {
023       public float X;
024       public float Y;
025       public float Z;
026       public float RotX;
027       public float RotY;
028       public float RotZ;
029       public float ScaleX;
030       public float ScaleY;
031       public float ScaleZ;
032     }
033
034     // Компонент трансформации для сохранения
035     public DataTransform MyTransform = new DataTransform();
036   }
037
038   // Объект с сохраняемыми данными
039   public MySaveData MyData = new MySaveData();
040   //-----------------------------------------------
041   // Заполняет структуру MyData данными
042   // Эти данные будут сохранены в файл
043   private void GetTransform()
044   {
045     // Получить компонент трансформации для данного объекта
046     Transform ThisTransform = transform;
047
048     // Заполнить структуру данных
049     MyData.MyTransform.X = ThisTransform.position.x;
050     MyData.MyTransform.Y = ThisTransform.position.y;
051     MyData.MyTransform.Z = ThisTransform.position.z;
052     MyData.MyTransform.RotX = ThisTransform.localRotation.eulerAngles.x;
053     MyData.MyTransform.RotY = ThisTransform.localRotation.eulerAngles.y;
054     MyData.MyTransform.RotZ = ThisTransform.localRotation.eulerAngles.z;
055     MyData.MyTransform.ScaleX = ThisTransform.localScale.x;
056     MyData.MyTransform.ScaleY = ThisTransform.localScale.y;
057     MyData.MyTransform.ScaleZ = ThisTransform.localScale.z;
058   }
059   //-----------------------------------------------
060   // Восстанавливает компонент трансформации
061   // Вызывается после загрузки данных из файла для восстановления
062   private void SetTransform()
063   {
064     // Получить компонент трансформации для данного объекта
065     Transform ThisTransform = transform;
066
```

```
067     // Восстановить данные
068     ThisTransform.position = new Vector3
          (MyData.MyTransform.X,     MyData.MyTransform.Y,
          MyData.MyTransform.Z);
069     ThisTransform.rotation = Quaternion.Euler(
          MyData.MyTransform.RotX, MyData.MyTransform.RotY,
          MyData.MyTransform.RotZ);
070     ThisTransform.localScale = new Vector3(
          MyData.MyTransform.ScaleX, MyData.MyTransform.ScaleY,
          MyData.MyTransform.ScaleZ);
071   }
072   //-----------------------------------------------
073   // Сохраняет состояние игры в XML-файл
074   // Вызывается для сохранения данных в XML-файл
075   // Вызывается как Save
076   public void SaveXML(string FileName = "GameData.xml")
077   {
078     // Получить данные для компонента трансформации
079     GetTransform();
080
081     // Сохранить данные
082     XmlSerializer Serializer = new XmlSerializer(typeof(MySaveData));
083     FileStream Stream = new FileStream(FileName, FileMode.Create);
084     Serializer.Serialize(Stream, MyData);
085     Stream.Close();
086   }
087   //-----------------------------------------------
088   // Загружает данные из XML-файла
089   // Вызывается для загрузки данных из XML-файла
090   // Вызывается как Load
091   public void LoadXML(string FileName = "GameData.xml")
092   {
093     // Если файл не найден, выйти
094     if(!File.Exists(FileName)) return;
095
096     XmlSerializer Serializer = new XmlSerializer(typeof(MySaveData));
097     FileStream Stream = new FileStream(FileName, FileMode.Open);
098     MyData = Serializer.Deserialize(Stream) as MySaveData;
099     Stream.Close();
100
101     // Восстановить компонент трансформации
102     SetTransform();
103   }
104   //-----------------------------------------------
105   public void SaveBinary(string FileName = "GameData.sav")
106   {
107     // Получить данные из компонента трансформации
108     GetTransform();
109
110     BinaryFormatter bf = new BinaryFormatter();
```

```
111     FileStream Stream = File.Create(FileName);
112     bf.Serialize(Stream, MyData);
113     Stream.Close();
114   }
115   //-----------------------------------------------
116   public void LoadBinary(string FileName = "GameData.sav")
117   {
118     // Если файл не найден, выйти
119     if(!File.Exists(FileName)) return;
120
121     BinaryFormatter bf = new BinaryFormatter();
122     FileStream Stream = File.Open(FileName, FileMode.Open);
123     MyData = bf.Deserialize(Stream) as MySaveData;
124     Stream.Close();
125
126     // Восстановить компонент трансформации
127     SetTransform();
128   }
129   //-----------------------------------------------
130 }
131 //-------------------------------------------------
```

Полный пример загрузки и сохранения игры можно найти в пакете примеров для книги, в папке `Chapter10/XML_and_Binary`.

Итоги

В этой последней главе были рассмотрены три рекомендации, главная из которых была связана с управлением исходными текстами. Первая рекомендация касалась пакета управления версиями Git, бесплатного программного обеспечения с открытым исходным кодом для управления версиями исходных текстов, позволяющего отслеживать изменения в проекте, а также упрощающего сотрудничество с другими разработчиками. Вторая рекомендация связана с динамической загрузкой файлов данных, с помощью, во-первых, внутренних файлов проекта из папки `resources`, и во-вторых, пакетов ресурсов. Последний вариант особенно полезен для создания внешних ресурсов, которые могут быть отредактированы как разработчиками, так и самими игроками. Третья и последняя рекомендация демонстрирует использование сериализации для сохранения состояния игры в файл и его последующей загрузки. С помощью сериализации пользователям дается возможность сохранять и восстанавливать игру для возобновления ее с того же места, на котором она была закончена.

Предметный указатель

Animator.StringToHash, 218
AssetDatabase, 305
Axially Aligned Bounding Box, 175

Blender, URL, 236
BroadcastMessage, 62
BuildPipeline.BuildAssetBundle, 350

C#
  выбор, 20
  конечный автомат, 251
  свойства, 55
CDATA, 290

EventListener, 144
EventManager, 145
EventPoster, 145

INI-файлы, 230
iTween, описание, 197
ITween, URL, 198

LIFO, 208
Linq, 226
localPosition, 117

Mecanim
  граф анимаций, 245
  конечные автоматы, 244

SendMessage, 62

Анимационные кривые, 195
Анимация камері, 193
Атрибут CustomProperty-Drawer, 288
Атрибуты C#, 61

Бесконечные циклы, 42

Внешние файлы
  и пакеты ресурсов, 349
  и папка ресурсов, 347

Делегирование, 81
Директива #endregion, 153
Директива #region, 153

Инспектор объектов, 60
Интерфейс IEnumerable, 210
Интерфейс IEnumerator, 210
Интерфейс TortoiseGit, URL, 333
Искусственный интеллект, 233
  в играх, 234

Камеры
  анимация, 192
  сопровождающие, 193
  траектория, 197
  шейдер, 315
Класс
  ColorBlend, 279
  ColorRangeAttribute, 279
  EditorGUI, 282
  PropertyDrawer, 281
  ScriptableWizard, 297
  StreamWriter, URL, 82
  Time, 128
  WizardEnumerator, 214
  WWW, 232
Классы
  и интерфейсы, 145
  и наследование, 49
  и объектно-ориентированное программирование, 46

и полиморфизм, 51
Компонент GetComponent, 116
Компонент Transform, 113
Компонет, получение нескольких, 117
Контроль версий, 332

Массивы, 34
Межмешевые ссылки, 239
Метод
FindPropertyRelative, 282
GetEnumerator, 215
GetNearestHealthRestore, 266
Invoke, 120
IsNullOrWhiteSpace, 216
Linecast, 260
OnSave, 119
String.indexOf, 220
ToString, переопределение, 73

Окклюзия, 172
Окно Immediate, 105
Оператор
?, 62
if, 28
switch, 31
Ортографические камеры, 179
Отладка
Debug.log, 70
MonoDevelop, 92
визуальная, 76
описание, 66
профилирование, 87

Пакетное переименование, 268
Пакеты активов, 349
Переменная DeltaTime, 132
Полиморфизм, 51

Синглтон, 134
Скайбокс, 299
Событие
FixedUpdate, 128
LateUpdate, 129
OnApplicationFocus, 164
OnApplicationPause, 164
OnGUI, 159
Update, 128
События, 45
класса MonoBehaviour, 159
мыши, 160
сенсорного ввода, 160
Создание Quad, 301
Состояние
Attack, 262
Chase, 260
Idle, 252
Patrol, 256
Список вызовов, 105
Строки
описание, 216
поиск, 220
создание, 220
сравнение, 217
форматирование, 219
цикл, 219

Текстовые активы
загрузка из CSV-файлов, 231
загрузка из INI-файлов, 230
загрузка из Интернета, 232
загрузка из локальных файлов, 228
статическая загрузка, 227
Текстуры рисование
описание, 313
скрипт, 319
шейдер смешивания, 315
Точки трассировки, 108

Управление исходными текстами Git, 331

Формат CSV, 226
Функции, 42
Функция
ChangeHealth, 263

DontDestroyOnLoad, 132
FindObjectsOfType, 124
GameObject.Find, 37
GameObject.
FindGameObjectWithTag, 122
GetGameViewSize, 171
GetPixelFromArray, 325
GetPixels, URL, 325
Gizmos.DrawFrustum, 170
Mathf.Clamp, 274
OnBecameInvisible, 173
OnBecameVisible, 173
OnWizardCreate, 272
PaintSourceToDestTexture, 324
PostNotification, 152
Print, 69
Quaternion.Slerp, 193
Random.Range, 258
Resources.Load, 290
SetPixelFromArray, 325
SetPixelInArray, 325
SetTrigger, 254
State_Idle, 254
String.Empty, 220
UpdateScor, 44
Vector3.SmoothDamp, 193

Ханойская башня
URL, 208
Хранилище
описание, 332
создание, 337
создание первой фиксации, 339

Цикл for, 39
Цикл foreach, 38
Цикл while, 40

Частота кадров в секунду, 46

Язык интегрированных запросов, 202

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Алан Торн

Искусство создания сценариев в Unity

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Рагимов Р. Н.*
Научный редактор *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 23. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф