Искусство создания сценариев в Unity

Алан Торн

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

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

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.)

Copyright © 2015 Packt Publishing
© Оформление, перевод, ДМК Пресс, 2016

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

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#),

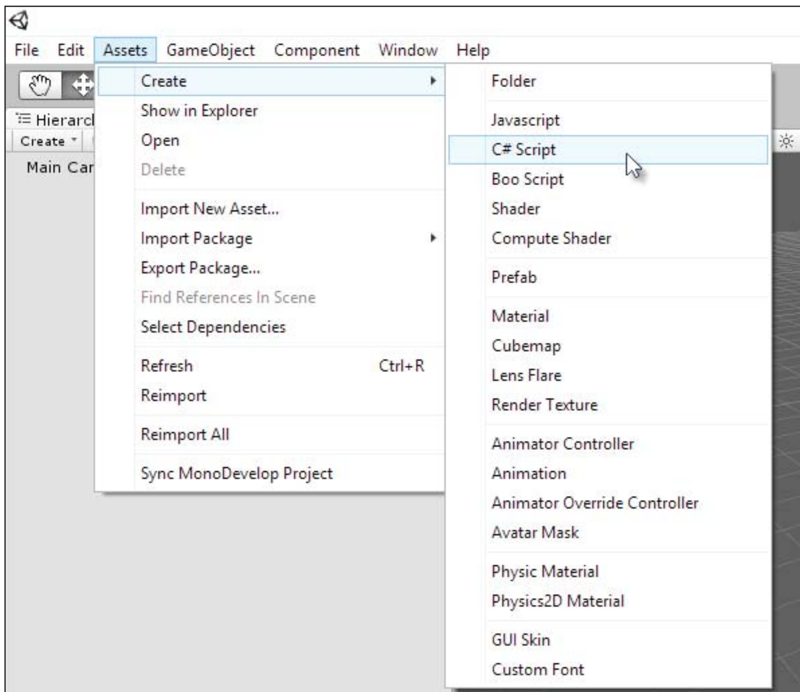

Рис. 1.1. Создание файла сценария с помощью меню приложения

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

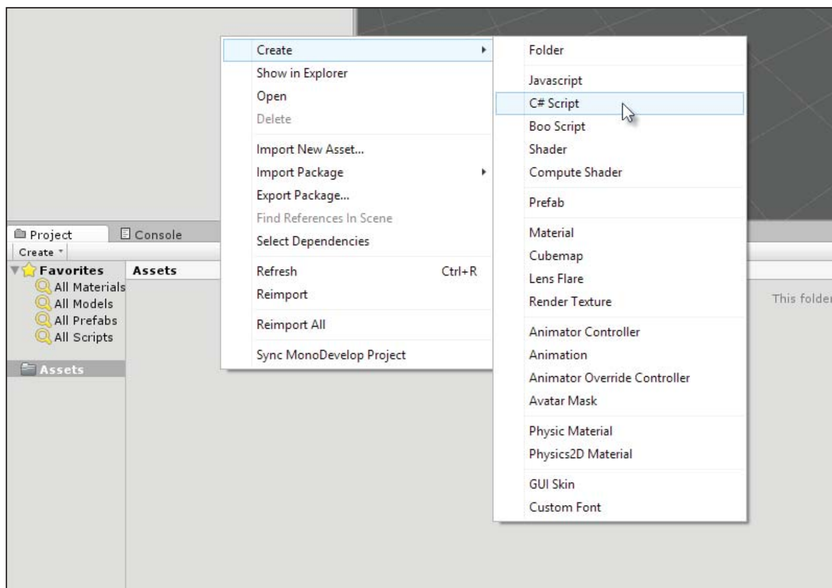

Рис. 1.2. Создание файла сценария
с помощью контекстного меню панели **Project**

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.

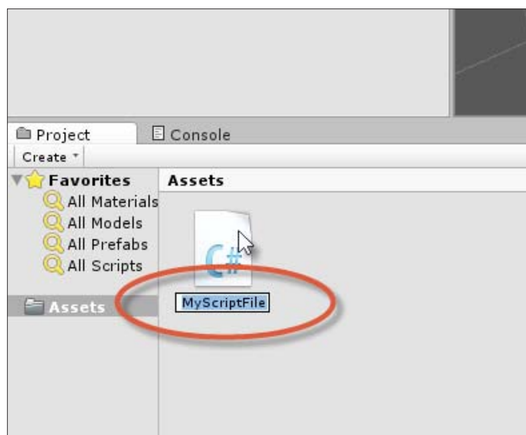

Рис. 1.3. Имена файлов должны быть уникальными и соответствовать принятым в C# соглашениям об именах классов

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-

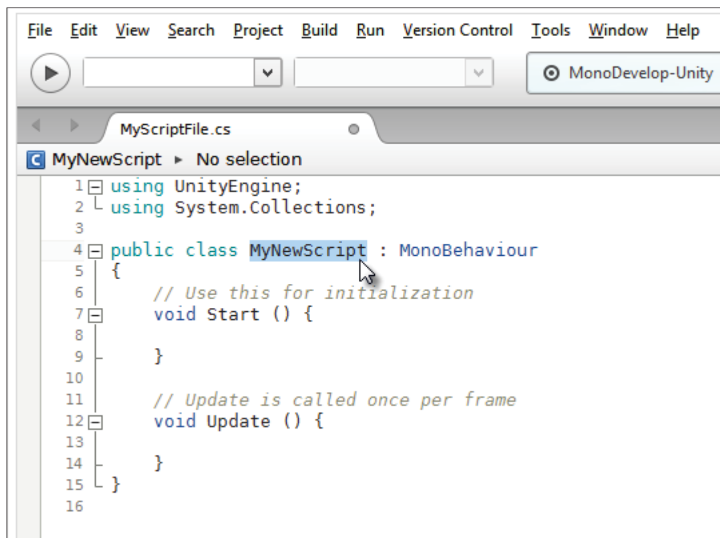

Рис. 1.4. Переименование класса
для соответствия имени файла

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.

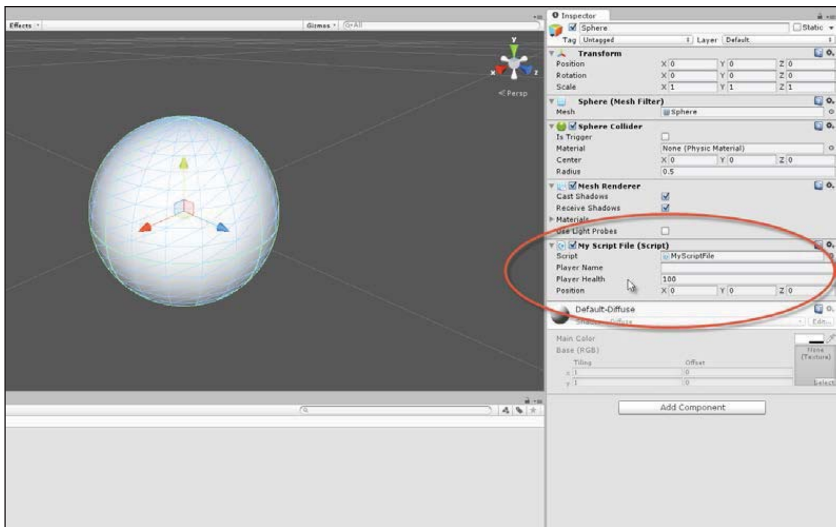

Рис. 1.5. Подключение сценария к объекту игры

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};
```