Часть IV. Приложения

A Стандартная процедура настройки проекта

Много раз в этой книге вам сначала предлагалось создать новый проект, а затем приводился программный код для опробования. Эта стандартная процедура, которую вы должны были выполнять каждый раз, включает в себя создание нового проекта, настройку сцены, создание нового сценария на C# и подключение этого сценария к главной камере в сцене. Чтобы не повторять инструкции в каждой главе, я собрал их здесь.

Настройка нового проекта

Выполните следующие шаги для настройки нового проекта. Скриншоты демонстрируют процедуру в двух операционных системах: OS X и Windows:

1. После первого запуска Unity выведет начальный экран, изображенный на рис. A.1. Здесь вы можете щелкнуть на кнопке New (Новый), чтобы создать новый проект. Если вы уже запускали Unity прежде, выберите в меню пункт File > New Project... (Файл > Новый проект...).

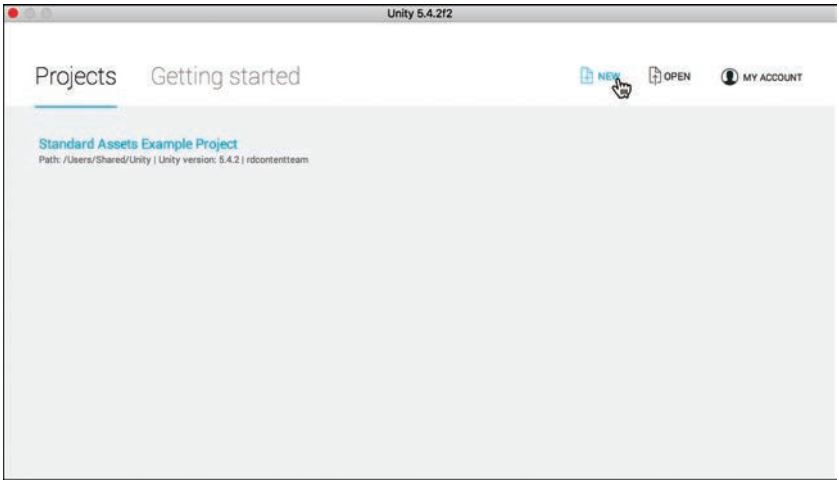

Рис. A.1. Создание нового проекта в начальном экране Unity

2. После этого откроется диалог создания нового проекта, изображенный на рис. А.2. Когда вы заполните форму на рис. А.2, Unity создаст новую папку проекта с именем, указанным в поле Project name* (Имя проекта*) каталога, указанного в поле Location* (Местоположение*). Щелчок на многоточии справа в поле Location* (Местоположение*) позволит вам выбрать каталог для размещения папки проекта, открыв стандартный системный диалог выбора каталога. В большинстве проектов, представленных в этой книге, вы должны выбрать радиокнопку 3D и установить движок Enable Unity Analytics (Служба сбора аналитической информации) в положение Off (Выкл.). Дополнительная информация о вариантах выбора приводится во врезке «Параметры нового проекта».

Например, с настройками на рис. А.2 Unity создаст папку *ProtoTools Project* в каталоге *Desktop* и поместит в нее проект трехмерной игры.

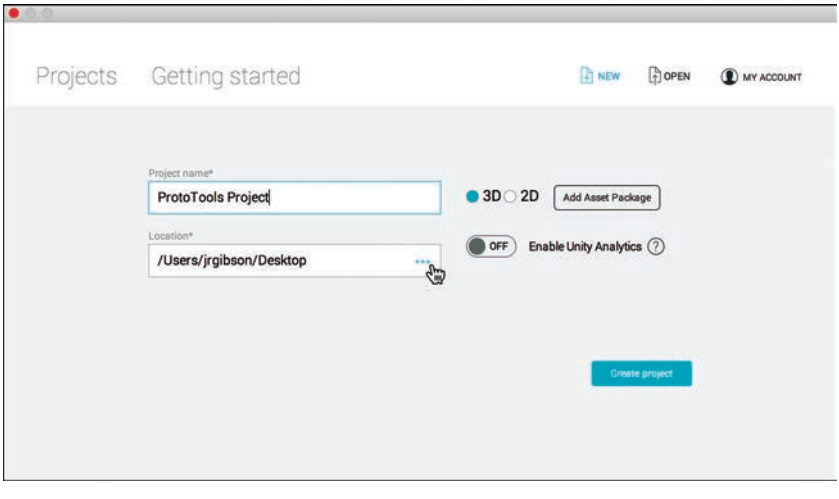

Рис. А.2. Диалог создания нового проекта

ПАРАМЕТРЫ НОВОГО ПРОЕКТА

В диалоге создания нового проекта Unity предлагает несколько параметров настройки.

3D/2D (выбрано значение **3D**) — радиокнопка **3D/2D** настраивает перспективную (**3D**) или ортографическую (**2D**) проекцию для главной камеры и создает сцену по умолчанию. Вот и все.

Enable Unity Analytics (Служба сбора аналитической информации), выбрано значение Off (Выкл.) — служба сбора аналитической информации позволяет получить сведения о том, сколько людей играет в игру, как они играют и т. д. Это фантастический инструмент, но мы не использовали его в проектах в этой книге.

Add Asset Package (Добавить пакет с ресурсами) — в состав Unity входит большое количество пакетов с ресурсами, среди которых можно найти инструменты создания ландшафта, эффекты частиц и т. д. Многие из них позволяют получить потрясающие эффекты, но в этой книге у нас не было оснований добавлять их на этапе создания проектов. Обычно я избегаю их добавления в свои проекты по следующим причинам:

- **Увеличение объема проекта**: если импортировать все доступные пакеты, размер проекта увеличится в 1000 раз по сравнению с исходным (с ≈300 Кбайт до ≈300 Мбайт)!
- **Захламление панели Project (Проект)**: если импортировать все пакеты, в папке Assets и в панели Project (Проект) появится огромное количество дополнительных элементов и папок.
- **Их всегда можно импортировать позднее**: в любой момент можно выбрать в меню пункт Assets > Import Package (Ресурсы > Импортировать пакет) и импортировать любой из потребовавшихся пакетов.

3. В диалоге создания нового проекта щелкните на кнопке Create project (Создать проект). После этого окно Unity закроется и вновь откроется, отобразив чистый холст с вашим новым проектом. На повторный запуск может потребоваться несколько секунд, поэтому будьте готовы подождать немного.

Подготовка сцены к разработке

Вновь созданный проект включает сцену с настройками по умолчанию. Чтобы подготовить ее к дальнейшей разработке, выполните следующие инструкции:

1. **Сохраните сцену.** Первым делом, создав новый проект, обязательно сохраните сцену. Выберите в меню пункт File > Save Scene As... (Файл > Сохранить сцену как...) и укажите имя. (Unity автоматически сохранит сцену в правильной папке.) Я предпочитаю выбирать такие имена, как _Scene_0, которые с легкостью нумеруются, когда в будущем потребуется создать дополнительные сцены. Подчеркивание в начале имени помогает вывести имена сцен в начале списка в панели Project (Проект), по крайней мере в macOS.
2. **Создайте новый сценарий на C# (необязательно).** В некоторых главах предлагается создать один или несколько сценариев на C#, прежде чем начать работу над проектом. Для этого щелкните на кнопке Create (Создать) в панели Project (Проект) и в открывшемся меню выберите пункт C# Script (Сценарий C#). После этого в панель Project (Проект) добавится новый сценарий, имя которого будет выделено и готово к изменению. Проделайте эти операции для каждого сценария, перечисленного в начале главы, и особое внимание уделите регистру символов в их именах. Закончив ввод имени сценария, нажмите

клавишу Return или Enter, чтобы сохранить имя. Например, на рис. А.3 создан сценарий с именем HelloWorld.

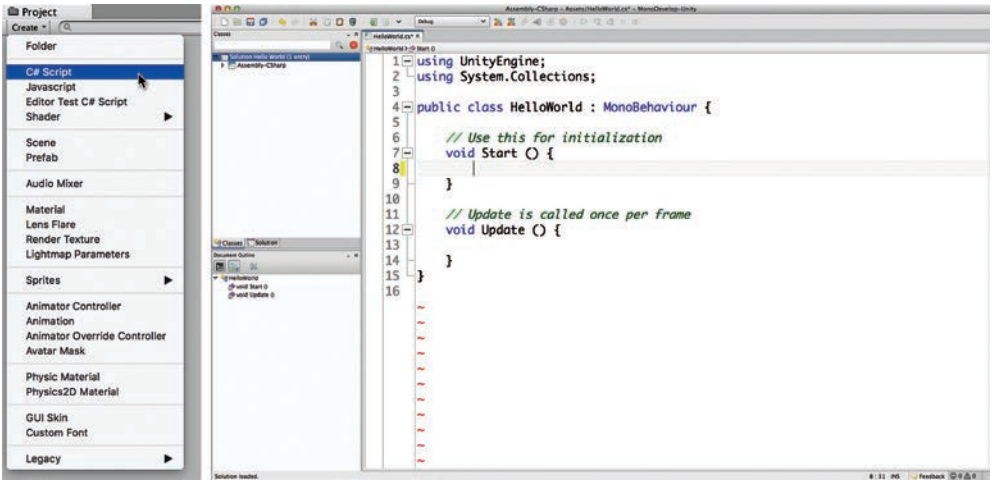

Рис. А.3. Создание нового сценария на C# и его просмотр в MonoDevelop

ИЗМЕНЕНИЕ ИМЕНИ СЦЕНАРИЯ ПОСЛЕ ЕГО СОЗДАНИЯ МОЖЕТ ВЫЗЫВАТЬ ПРОБЛЕМЫ. После выбора имени сценария в процессе его создания Unity автоматически использует это имя в объявлении класса (строка 4 на рис. А.3). Однако если вы решите изменить имя сценария после его создания, вы должны будете не только изменить его имя в панели Project (Проект), но и в строке объявления класса самого сценария. Для случая, изображенного на рис. А.3, вам придется заменить имя `HelloWorld` класса новым именем сценария.

3. **Подключите сценарий на C# к главной камере Main Camera в сцене (необязательно).** В некоторых главах предлагается *подключить* один или несколько новых сценариев к главной камере Main Camera. В результате подключения к игровому объекту, такому как Main Camera, сценарий превращается в компонент этого игрового объекта. По умолчанию все сцены включают объекты Main Camera, поэтому они являются лучшей точкой подключения любых базовых сценариев, которые вы хотели бы выполнять. Обычно если сценарий на C# не подключен ни к какому игровому объекту в сцене, он не выполняется.

   Подключение сценариев к игровым объектам не самая простая процедура, но вы быстро освоите ее, потому что ее часто приходится выполнять в Unity. Наведите указатель мыши на имя нового сценария (в панели Project (Проект)), нажмите левую кнопку и, не отпуская ее, перетащите на объект Main Camera в панели Hierarchy (Иерархия), после этого отпустите кнопку мыши. Как это выглядит, показано на рис. А.4.

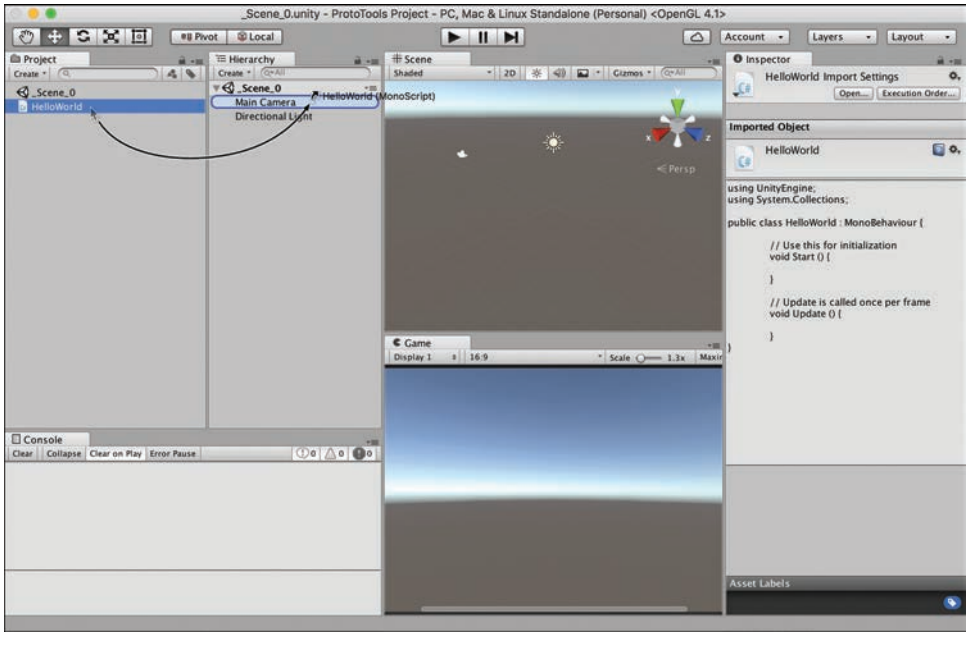

Рис. А.4. Перетаскивание сценария на C# из панели Project (Проект) на объект Main Camera в панели Hierarchy (Иерархия) для подключения сценария HelloWorld к игровому объекту Main Camera

После этого сценарий подключится к Main Camera и появится в инспекторе, если выбрать объект Main Camera. Теперь вы готовы начать работу над проектами в этой книге.

Б Полезные идеи

В этом приложении описывается множество идей, которые помогут вам стать лучшим разработчиком прототипов и программистом. Некоторые из этих идей описывают приемы программирования, другие — методологию. Они собраны здесь, в этом приложении, чтобы вам проще было отыскать нужную информацию потом, когда в ближайшие годы вы снова и снова будете возвращаться к этой книге.

Охватываемые темы

Это приложение охватывает несколько разных тем, организованных в четыре группы. Многие из них включают примеры программного кода, другие ссылаются на конкретные разделы книги, где эти идеи используются.

C# и идеи программирования в Unity

В этом разделе рассматриваются приемы программирования на C#, к изучению которых вы, возможно, пожелаете вернуться, когда закончите чтение книги. Здесь также представлены некоторые идеи, которые хотя и важны, но не вписываются ни в одну из предыдущих глав.

Поразрядные логические операторы и маски слоев

Как вы узнали в главе 21 «Логические операции и условия», один символ — вертикальную черту (`|`) — можно использовать как логический оператор ИЛИ, вычисляющий выражение по полной схеме, а другой символ — амперсанд (`&`) — как логический оператор И, также вычисляющий выражение по полной схеме. Однако операторы `|` и `&` можно также использовать для выполнения *поразрядных операций* с беззнаковыми целыми `unsigned int`, из-за чего их часто называют *поразрядным ИЛИ* и *поразрядным И*.

В поразрядных операциях происходит сравнение отдельных двоичных разрядов (битов) целого числа с применением одного из шести поразрядных операторов, включенных в C#. Все они перечисляются в следующем списке, и демонстрируется результат их применения к 8-битному байту (простейший целочисленный тип данных, способный хранить значения от 0 до 255). Эти операции точно так же воздействуют на 32-битные целочисленные значения, но я не использовал их, потому что 32 бита не умещаются по ширине книжной страницы.

&	И	`00000101 & 01000100`	вернет `00000100`
|	ИЛИ	`00000101 | 01000100`	вернет `01000101`
^	Исключительное ИЛИ	`00000101 ^ 01000100`	вернет `01000001`
~	Дополнение (поразрядное НЕ)	`~00000101`	вернет `11111010`
<<	Сдвиг влево	`00000101 << 1`	вернет `00001010`
>>	Сдвиг вправо	`01000100 >> 2`	вернет `00010001`

В Unity поразрядные операции чаще всего используются для управления маской слоев `LayerMask`. Unity позволяет разработчикам определить до 32 разных слоев и представляет значения типа `LayerMask` как 32-разрядные целые числа без знака, слои в которых анализируются физическим движком или в операциях бросания лучей. Переменные типа `LayerMask` в Unity используются для описания маски слоев, но это всего лишь тонкая обертка вокруг 32-разрядных целых чисел без знака, добавляющая совсем немного дополнительных возможностей. В маске `LayerMask` любой бит, имеющий значение 1, представляет видимый слой, а любой бит, имеющий значение 0, представляет слой, который должен игнорироваться (то есть замаскированный). Это может очень пригодиться, например, когда требуется, чтобы столкновения определялись только с объектами в определенном слое, или когда нужно указать слой, который должен игнорироваться. (Например, встроенный слой 2 с именем Ignore Raycast автоматически маскируется во всех операциях бросания лучей.)

Unity поддерживает восемь зарезервированных «встроенных» слоев, и все игровые объекты по умолчанию помещаются в нулевой (0-й) слой с именем Default. Остальные слои с порядковыми номерами от 8 до 31 называются *пользовательскими слоями*, и присваивание имени любому из них добавляет это имя во все меню, перечисляющие слои (например, меню Layer (Слой) в инспекторе с настройками любого игрового объекта).

Поскольку нумерация слоев начинается с нуля, двоичное представление значения `LayerMask`, не маскирующего нулевой слой, имеет 1 в правом крайнем разряде (как показано в значении переменной `lmZero` в следующем листинге). Это может вызывать путаницу (потому что этому двоичному представлению соответствует целое число 1, а не 0), из-за чего для присваивания значений `LayerMask` многие разработчики используют оператор поразрядного сдвига влево (`<<`). (Например,

выражение `1<<0` вернет значение 1, соответствующее нулевому уровню, а выражение `1<<4` замаскирует все физические слои, кроме четвертого.) Дополнительные примеры приводятся в следующем листинге:

```
LayerMask lmNone = 0;     // 00000000000000000000000000000000              // a
LayerMask lmAll = ~0;     // 11111111111111111111111111111111              // b
LayerMask lmZero = 1;     // 00000000000000000000000000000001
LayerMask lmOne = 2;      // 00000000000000000000000000000010              // c
LayerMask lmTwo = 1<<2;   // 00000000000000000000000000000100              // d
LayerMask lmThree = 1<<3; // 00000000000000000000000000001000

LayerMask lmZeroOrTwo = lmZero | lmTwo;                                    // e
                        // Результат: 00000000000000000000000000000101

LayerMask lmZeroThroughThree = lmZero | lmOne | lmTwo | lmThree;
                        // Результат: 00000000000000000000000000001111

lmZero = 1 << LayerMask.NameToLayer("Default");                            // f
                        // Результат: 00000000000000000000000000000001

LayerMask lmZeroOrOne = LayerMask.GetMask("Default", "TransparentFX");     // g
                        // Результат: 00000000000000000000000000000011
```

a. Когда все биты в `LayerMask` содержат 0, все слои *игнорируются*.

b. Когда все биты в `LayerMask` содержат 1, все слои участвуют в операциях.

c. 2 — целое число в `LayerMask`, соответствующее слою с номером 1. Этот пример демонстрирует, как может возникнуть путаница, если присваивать переменной типа `LayerMask` целые числа. Слой с номером 1 — это предопределенный слой с именем TransparentFX.

d. Использование оператора сдвига влево (`<<`) добавляет ясности, потому что в этом случае 1 сдвигается влево на две позиции, и получается маска `LayerMask`, соответствующая второму слою.

e. Поразрядная операция ИЛИ создает маску `LayerMask`, соответствующую слоям с номерами 0 и 2.

f. Статический метод `LayerMask.NameToLayer()` возвращает номер слоя — целое число, а не `LayerMask`, — соответствующий указанному имени. Например, `LayerMask. NameToLayer("TransparentFX")` вернет целое число 1.

g. Метод `GetMask()` позволяет получить непосредственно маску слоев `LayerMask` по списку их имен.

Сопрограммы

Механизм поддержки сопрограмм в C# позволяет методу приостановиться в середине вычислений, дать возможность выполниться другим процессам и затем продолжить работу с места приостановки. Сопрограммы часто используются

в Unity для выполнения продолжительных вычислений (которые, если их не приостанавливать периодически, могут создать впечатление, что игра зависла). Один из примеров приводится в разделе «Вероятность игровой кости», ниже в этом приложении; для вычисления всех возможных исходов броска множества игровых костей могут потребоваться минуты или даже часы, поэтому возможность периодической приостановки функции для обновления экрана оказывается как нельзя кстати. Сопрограммы можно также использовать в роли таймеров для заданий, которые должны выполняться через определенные интервалы времени (как альтернативу использованию функции `InvokeRepeating`).

Пример в Unity

Следующий пример сопрограммы выводит текущее время один раз в секунду. Если организовать вывод времени в методе `Update()`, он может происходить десятки раз в секунду, что слишком часто.

Создайте новый проект Unity, создайте сценарий на C# с именем Clock, подключите его к Main Camera и введите следующий код:

```
using UnityEngine;
using System.Collections;
public class Clock : MonoBehaviour {

    // Используйте этот метод для инициализации
    void Start () {
        StartCoroutine(Tick());
    }

    // Все сопрограммы должны возвращать значение типа IEnumerator
    IEnumerator Tick() {
        // Этот бесконечный цикл продолжает вывод, пока сопрограмма
        //   не будет прервана или пока программа не остановится
        while (true) {
            print(System.DateTime.Now.ToString());
            // Эта инструкция yield сообщает механизму сопрограмм подождать
            //   1 секунду перед продолжением. Время в сопрограммах
            //   измеряется довольно точно.
            yield return new WaitForSeconds(1);
        }
    }
}
```

В сопрограммах, в отличие от обычных функций, допускается использование бесконечного цикла `while(true)`, при условии, что внутри присутствует инструкция `yield`.

Существует несколько разновидностей инструкций `yield`:

```
yield return null; // Возобновляет выполнение при первой возможности,
                   //   обычно в следующем кадре

yield return new WaitForSeconds(10); // Ждет 10 секунд
```

```
yield return new WaitForEndOfFrame(); // Ждет до следующего кадра

yield return new WaitForFixedUpdate(); // Ждет до следующего вызова FixedUpdate
```

Еще один пример использования сопрограммы для анализа очень большого словаря приводится в главе 34 «Прототип 6: Word Game».

Перечисления

Перечисления дают простую возможность объявить тип переменных, которые могут принимать строго определенные значения, и с успехом используются в этой книге. В ней же перечисления обычно объявлялись за границами определений классов. Имена перечислений принято начинать с буквы `e`.

```
public enum ePetType {
    none,
    dog,
    cat,
    bird,
    fish,
    other
}

public enum eLifeStage {
    baby,
    teen,
    adult,
    senior,
    deceased
}
```

После определения тип перечисления (например, `public ePetType`) можно использовать для объявления переменных. Ссылки на разные варианты записываются как имя перечисления, за которым следует точка и имя варианта (например, `ePetType.dog`):

```
public class Pet {
  public string      name = "Flash";
  public ePetType    pType = ePetType.dog;
  public eLifeStage  age = eLifeStage.baby;
}
```

Фактически варианты перечислений — это целые числа, маскирующиеся под другие значения, поэтому их можно приводить к типу `int` и получать из типа `int` (как показано в строках 7 и 8 в следующем листинге). Кроме того, переменная с типом перечисления по умолчанию получает значение 0-го варианта, если явно не определено иное. Например, с учетом предыдущего определения перечисления `eLifeStage` вновь объявленной переменной `eLifeStage age` (в строке 4 в следующем листинге) автоматически будет присвоено значение `eLifeStage.baby`.

```
 1 public class Pet {
 2     public string name = "Flash";
 3     public ePetType pType = ePetType.dog;
 4     public eLifeStage age; // По умолчанию age получит
                              // значение eLifeStage.baby                    // a
 5
 6     void Awake() {
 7         int i = (int) ePetType.cat; // i получит значение 2               // b
 8         ePetType pt = (ePetType) 4; // pt получит значение ePetType.fish  // c
 9     }
10 }
```

a. По умолчанию `age` получит значение `eLifeStage.baby`.

b. Фрагмент `(int)` в строке 7 — это явное приведение типа, которое заставляет интерпретировать `ePetType.cat` как значение типа `int`.

c. Здесь целочисленный литерал `4` явно приводится к типу `ePetType` с помощью `(ePetType)`.

Перечисления часто используются в инструкциях `switch` (как вы могли видеть в предыдущих главах).

Делегаты функций

Делегат функции проще рассматривать как контейнер для родственных функций (или методов), которые можно вызвать все сразу. Делегаты вы могли видеть в главе 31 «Прототип 3.5: SPACE SHMUP PLUS». Там мы использовали вызов единственного делегата `fireDelegate()` для выстрела сразу из всех видов оружия, подключенного к космическому кораблю игрока. Делегаты часто применяются для реализации искусственного интеллекта с использованием шаблона проектирования «Стратегия». Больше узнать о шаблоне «Стратегия» вы сможете в разделе «Шаблоны проектирования программного обеспечения» в этом приложении.

Первый шаг на пути к использованию делегата функции — определение типа делегата (`FloatOpDelegate` в примере ниже). Это определение объявляет набор параметров и тип значения, возвращаемого любым экземпляром этого делегата (например, поле делегата `fod` далее в этом разделе). Оно также задает параметры и тип возвращаемого значения функции, которую можно присвоить экземпляру этого типа делегата.

```
public delegate float FloatOpDelegate( float f0, float f1 );
```

В предыдущей строке определяется тип делегата `FloatOpDelegate` (Float Operation Delegate — делегат операций с числами типа `float`), который требует, чтобы функция принимала два значения `float` и возвращала одно значение `float`. Объявив тип делегата, мы можем определить соответствующие ему методы (например, `FloatAdd()` и `FloatMultiply()`, как показано ниже):

```
using UnityEngine;
using System.Collections;
public class DelegateExample : MonoBehaviour {
    // Объявление типа делегата с именем FloatOpDelegate
    // В нем определяются типы параметров и возвращаемого
    //   значения для целевых функций
    public delegate float FloatOpDelegate( float f0, float f1 );

    // FloatAdd должен иметь те же типы параметров и возвращаемого значения,
    //   что и тип делегата FloatOpDelegate
    public float FloatAdd( float f0, float f1 ) {
        float result = f0+f1;
        print("The sum of "+f0+" & "+f1+" is "+result+".");
        return( result );
    }

    // FloatMultiply должен иметь те же типы параметров и возвращаемого значения,
    //   что и тип делегата FloatOpDelegate
    public float FloatMultiply( float f0, float f1 ) {
        float result = f0 * f1;
        print("The product of "+f0+" & "+f1+" is "+result+".");
        return( result );
    }
    ...
}
```

Теперь можно объявить переменную типа `FloatOpDelegate` и присвоить ей любую из целевых функций. Затем эту переменную с типом делегата можно вызвать как обычную функцию (см. поле делегата `fod` в следующем листинге).

```
using UnityEngine;
using System.Collections;
public class DelegateExample : MonoBehaviour {
    // Объявление типа делегата с именем FloatOpDelegate
    // В нем определяются типы параметров и возвращаемого
    //   значения для целевых функций
    public delegate float FloatOpDelegate( float f0, float f1 );

    // FloatAdd должен иметь те же типы параметров и возвращаемого значения,
    //   что и тип делегата FloatOpDelegate
    public float FloatAdd( float f0, float f1 ) { ... }

    // FloatMultiply должен иметь те же типы параметров и возвращаемого значения,
    //   что и тип делегата FloatOpDelegate
    public float FloatMultiply( float f0, float f1 ) { ... }

    // Объявление поля "fod" с типом FloatOpDelegate
    public FloatOpDelegate fod; // Поле делегата

    void Awake() {
        // Присвоить метод FloatAdd полю fod
        fod = FloatAdd;

        // Вызвать fod как простой метод; fod вызовет FloatAdd()
```

```
        fod( 2, 3 ); // Выведет: The sum of 2 & 3 is 5.

        // Присвоить метод FloatMultiply полю fod, заменит FloatAdd
        fod = FloatMultiply;

        // Вызов fod(2,3); вызовет FloatMultiply(2,3), вернет 6
        fod( 2, 3 ); // Выведет: The product of 2 & 3 is 6
    }
    ...
}
```

Делегаты также могут быть *групповыми*, то есть одному делегату можно присвоить несколько целевых методов. Именно это свойство позволило нам одним вызовом делегата произвести выстрел сразу из пяти видов оружия в главе 31 «Прототип 3.5: SPACE SHMUP PLUS». Там единственный вызов делегата `fireDelegate()` вызывал все методы `Fire()` разных экземпляров `Weapon`. Если групповой делегат имеет тип возвращаемого значения, отличный от `void` (как в примере с делегатом `FloatOpDelegate`), он вернет значение, полученное в результате вызова последнего целевого метода. Будьте внимательны: если вызвать пустой делегат, к которому не подключено ни одной функции, он возбудит исключение. Чтобы предотвратить эту проблему, можно предварительно сравнить делегат со значением `null`.

```
// Добавьте этот метод Start() в класс DelegateExample
void Start() {
    // Присвоить метод FloatAdd полю fod
    fod = FloatAdd;

    // Добавить метод FloatMultiply(), теперь fod будет вызывать ОБА метода
    fod += FloatMultiply;

    // Проверить fod перед вызовом
    if (fod != null) {
        // Вызов fod(3,4); вызовет FloatAdd(3,4) и потом FloatMultiply(3,4)
        float result = fod( 3, 4 );
        // Выведет: The sum of 3 & 4 is 7.
        // Затем выведет: The product of 3 & 4 is 12.

        print( result );
        // Выведет: 12
        // Результат 12 вернул вызов последнего целевого метода,
        //   именно он возвращается делегатом.
    }
}
```

Интерфейсы

Интерфейс определяет методы и свойства, которые затем будут реализованы в классе. На любой класс, реализующий интерфейс, можно ссылаться как на тип этого интерфейса, а не на тип этого конкретного класса. У такого подхода есть несколько отличий от наследования классов, наиболее интересным из которых является возможность реализовать в одном классе сразу несколько интерфейсов,

тогда унаследовать можно только один класс. Имена интерфейсов принято начинать с заглавной буквы `I`, чтобы их проще было отличать от имен классов. Применение интерфейсов демонстрировалось в главе 35 «Прототип 7: DUNGEON DELVER».

Пример в Unity — интерфейсы

Создайте в Unity проект. В этом проекте создайте сценарий на C# с именем Menagerie и введите следующий код:

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

// Два перечисления для присваивания определенных значений полям в классах
public enum ePetType {
    none,
    dog,
    cat,
    bird,
    fish,
    other
}

public enum eLifeStage {
    baby,
    teen,
    adult,
    senior,
    deceased
}

// Интерфейс IAnimal объявляет три общедоступных свойства и два общедоступных
метода
//   которые должны быть во всех классах, реализующих IAnimals.
public interface IAnimal {
    // Общедоступные свойства
    ePetType pType { get; set; }
    eLifeStage age { get; set; }
    string name { get; set; }

    // Общедоступные методы
    void Move();
    string Speak();
}

// Класс Fish реализует интерфейс IAnimal
public class Fish : IAnimal {
    private ePetType _pType = ePetType.fish;                                    // a
    public ePetType pType {
        get { return( _pType ); }
        set { _pType = value; }
    }

    public eLifeStage      age { get; set; }                                    // b
```

```
    public string          name { get; set; }                              // c

    public void Move() {
        Debug.Log("The fish swims around.");
    }

    public string Speak() {
        return("...!");
    }
}

// Класс Mammal - суперкласс, который будет наследоваться классами Dog и Cat // d
public class Mammal {
    protected eLifeStage       _age;
    public eLifeStage age {
        get { return( _age ); }
        set { _age = value; }
    }

    public string              name { get; set; }                          // c
}

// Dog - подкласс Mammal И реализует интерфейс IAnimal
public class Dog : Mammal, IAnimal {                                       // e
    private ePetType       _pType = ePetType.dog;

    public ePetType pType {
        get { return( _pType ); }
        set { _pType = value; }
    }

    public void Move() {
        Debug.Log("The dog walks around.");
    }

    public string Speak() {
        return("Bark!");
    }
}

// Cat - подкласс Mammal И реализует интерфейс IAnimal
public class Cat : Mammal, IAnimal {
    private ePetType       _pType = ePetType.cat;

    public ePetType pType {
        get { return( _pType ); }
        set { _pType = value; }
    }

    public void Move() {
        Debug.Log("The cat stalks around.");
    }

    public string Speak() {
        return("Meow!");
    }
}
```

```
// Menagerie - подкласс MonoBehaviour
public class Menagerie : MonoBehaviour {
    // В этот список можно добавлять экземпляры ЛЮБЫХ классов,
    //   реализующих интерфейс IAnimal
    public List<IAnimal>    animals;

    void Awake () {
        animals = new List<IAnimal>();

        Dog d = new Dog();
        d.age = eLifeStage.adult;
        // экземпляр d добавляется в список animals
        //   как экземпляр типа IAnimal, а не Dog
        animals.Add( d );
        animals.Add( new Cat() );
        animals.Add( new Fish() );

        animals[0].name = "Wendy";
        animals[1].name = "Caramel";
        animals[2].name = "Nemo";

        string[] types = new string[] {"none", "dog", "cat", "bird",
            ➥ "fish", "other"};                                          // f
        string[] ages = new string[] {"baby", "teen", "adult", "senior",
            ➥ "deceased"};
        // В этом цикле все экземпляры в списке animals интерпретируются
        //   одинаково, даже при том, что они фактически принадлежат разным
        //   классам, наследующим разные суперклассы
        string aName;
        IAnimal animal;
        for (int i=0; i<animals.Count; i++) {
            animal = animals[i];                                         // g
            aName = animal.name;
            print("Animal #" + i + " is a " + types[ (int) animal.pType ]
                ➥ + " named " + aName + ".");                            // h
            animal.Move();
            print(aName + " says: "+animal.Speak());

            switch (animal.age) {
                case eLifeStage.baby:
                case eLifeStage.teen:
                case eLifeStage.senior:
                    print(aName + " is a " + ages[(int) animal.age] + ".");
                    break;
                case eLifeStage.adult:
                    print(aName + " is an adult.");
                    break;
                case eLifeStage.deceased:
                    print(aName + " is deceased.");
                    break;
            }
        }
    }
}
```

a. `_pType` — скрытое поле, на котором основано общедоступное свойство `pType`.

b. Это *автоматическое свойство*. Когда свойство, такое как `age` здесь, имеет в фигурных скобках только ключевые слова `get; set;`, компилятор автоматически создает скрытую переменную, доступную через свойство.

c. `Name` — еще одно автоматическое свойство.

d. Обратите внимание, что класс `Mammal` не реализует интерфейс `IAnimal`. Это можно было бы сделать, но я хотел показать, что подклассы могут реализовать интерфейс, даже если он не реализован в суперклассе.

e. `Dog` — подкласс `Mammal` и реализует интерфейс `IAnimal`. Так как `Dog` является подклассом `Mammal`, он наследует защищенное поле `_age` и общедоступные свойства `age` и `name`. Если бы поле `_age` было скрытым (`private`), класс `Dog` не унаследовал бы его от класса `Mammal` и не имел бы к нему прямого доступа. Поскольку `Dog` имеет доступ к общедоступному свойству `age`, которое определено в классе `Mammal` (не `Dog`), свойство `age` можно использовать для чтения и изменения поля `_age`. В данном случае унаследованное свойство `age` выполняет требование интерфейса `IAnimals` в отношении наличия общедоступного свойства `age`. Более подробную информацию о защищенных полях и наследовании классов вы найдете в разделе «Области видимости переменной».

f. Напомню, что ↪ в листингах — это символ продолжения строки, то есть код `"fish", "other"};` продолжает предыдущую строку. Вы не должны вводить символ ↪.

g. Независимо от фактического типа `i`-й элемент списка `animals` присваивается локальной переменной `animal` и интерпретируется как экземпляр типа `IAnimal`.

h. `animal.pType` вернет тип животного как значение типа `ePetType`. Фрагмент `(int)` приведет это значение к типу `int`, которое затем будет использовано для получения элемента из массива строк `types`.

Как видите, наличие интерфейса `IAnimal` позволяет единообразно интерпретировать экземпляры классов `Cat`, `Dog` и `Fish`, хранить их в общем списке `List<IAnimal>` и присваивать одной и той же локальной переменной `IAnimal animal`.

Соглашения об именах

Впервые я упомянул соглашения об именах в главе 20 «Переменные и компоненты», но они настолько важны, что я решил повторить их здесь. Код в книге следует целому ряду правил, управляющих выбором имен для переменных, функций, классов и т. д. Ни одно из этих правил не является обязательным, но, следуя им, вы сделаете свой код более удобочитаемым не только для тех, кто будет пытаться расшифровать его, но и для себя, если спустя месяцы вам придется вернуться к нему и понять, что вы написали. Каждый программист использует в своей практике немного отличающиеся правила — даже мои личные правила менялись с течением времени, — но

правила, которые я хочу представить здесь, оказались весьма эффективными для меня и моих студентов, и они совместимы с большинством кода на C#, который мне довелось встречать в Unity:

1. Используйте верблюжийРегистр для любых имен. В именах переменных, состоящих из нескольких слов, каждое слово должно начинаться с заглавной буквы (кроме первого — в именах переменных первое слово должно начинаться со строчной буквы).
2. Имена переменных должны начинаться со строчной буквы (например, `someVariableName`).
3. Имена функций должны начинаться с заглавной буквы (например, `Start()`, `FunctionName()`).
4. Имена классов должны начинаться с заглавной буквы (например, `GameObject`, `ScopeExample`).
5. Имена интерфейсов предпочтительнее начинать с заглавной буквы `I` (например, `IAnimal`).
6. Имена скрытых переменных часто принято начинать с символа подчеркивания (например, `_hiddenVariable`).
7. Имена статических переменных часто состоят только из заглавных букв и записываются с применением «змеиного_регистра» (например, `NUM_INSTANCES`). В змеином_регистре слова, составляющие имя, объединяются символом подчеркивания.
8. Имена перечислений предпочтительнее начинать со строчной буквы `e` (например, `ePetType`, `eLifeStage`).

Предшествование операторов и порядок операций

Так же как в математике, некоторые операторы в C# имеют преимущество перед другими. Одним из примеров, с которым вы наверняка знакомы, является преимущество * перед + (например, 1 + 2 × 3 = 7, потому что сначала 2 умножается на 3, а затем к произведению прибавляется 1). Ниже приводится список часто используемых операторов в порядке предшествования. Операторы в начале этого списка выполняются раньше операторов, следующих ниже.

`( )` — Операции, заключенные в круглые скобки, всегда выполняются в первую очередь

`F()` — Вызов функции

`a[]` — Доступ к элементу массива

`i++` — Постинкремент

`i--` — Постдекремент

`!` — НЕ

`~` — Поразрядное НЕ (дополнение)

`++i` — Преинкремент

`--i` — Предекремент

`*` — Умножение

`/` — Деление

`%` — Деление по модулю (остаток от деления нацело)

`+` — Сложение

`-` — Вычитание

`<<` — Поразрядный сдвиг влево

`>>` — Поразрядный сдвиг вправо

`<` — Меньше

`>` — Больше

`<=` — Меньше или равно

`>=` — Больше или равно

`==` — Равно (оператор равенства)

`!=` — Не равно

`&` — Поразрядное И

`^` — Поразрядное Исключительное ИЛИ

`|` — Поразрядное ИЛИ

`&&` —Логическое И, вычисляется по короткой схеме

`||` — Логическое ИЛИ, вычисляется по короткой схеме

`=` — Присваивание

Состояния гонки

В отличие от многих других тем в этом разделе, состояние гонки — это *нежелательное* явление в играх. Состояние гонки возникает, когда какие-то события должны происходить раньше других, но есть вероятность, что этот порядок нарушится и это нарушение приведет к непредсказуемому поведению или даже к аварийному завершению программы. Состояние гонки — серьезный фактор, который должен учитываться при проектировании любого кода для многопроцессорных компьютеров, многопоточных операционных систем или сетевых приложений (когда разные компьютеры, находящиеся в разных уголках света, могут оказаться в состоянии гонки друг с другом). Но это не менее серьезный фактор и для игр Unity, потому что они часто состоят из огромного количества разных игровых объектов, имеющих свои методы `Awake()`, `Start()` и `Update()`, которые вызываются примерно в одно

время. Состоянию гонки мы уделили некоторое внимание в главе 31 «Прототип 3.5: SPACE SHMUP PLUS».

Рассмотрим это состояние на примере.

Пример в Unity — состояние гонки

Выполните следующие шаги:

1. Создайте в Unity новый проект с именем Unity-RaceCondition.
2. Создайте новый сценарий на C# с именем SetValues и введите следующий код:

```
using UnityEngine;
using System.Collections;

public class SetValues : MonoBehaviour {
    static public int[]        VALUES;

    void Start() {
        VALUES = new int[] { 0, 1, 2, 3, 4, 5 };
    }
}
```

3. Создайте второй сценарий на C# с именем ReadValues и введите следующий код:

```
using UnityEngine;
using System.Collections;

public class ReadValues : MonoBehaviour {

    void Start() {
        print( SetValues.VALUES[2] );
    }
}
```

4. Сохраните оба сценария и вернитесь в Unity.
5. Подключите оба сценария к Main Camera и щелкните на кнопке Play (Играть). После этого в консоли появится одно из двух возможных сообщений:
   - **2**
   - **NullReferenceException:** Object reference not set to an instance of an object[1]

   Конкретный исход зависит от того, какая из двух функций `Start()` будет вызвана первой. Если Unity вызовет `SetValues.Start()` перед `ReadValues.Start()`, все закончится благополучно. Но если функция `ReadValues.Start()` будет вызвана перед `SetValues.Start()`, вы увидите сообщение об ошибке использования пустой ссылки, потому что `ReadValues.Start()` попытается прочитать элемент `SetValues.VALUES[2]` в тот момент, когда `SetValues.VALUES` хранит `null`.

[1] Ссылка на объект не указывает на действительный экземпляр объекта. — *Примеч. пер.*

До выхода версии Unity 5 было очень сложно сказать, какой из этих двух методов `Start()` будет вызван первым. К счастью, в последних версиях Unity появилась возможность явно определять порядок выполнения сценариев.

6. В меню Unity выберите пункт Edit > Project Settings > Script Execution Order (Правка > Параметры проекта > Порядок выполнения сценариев). В инспекторе откроется диалог Script Execution Order (Порядок выполнения сценариев), как показано на рис. Б.1.

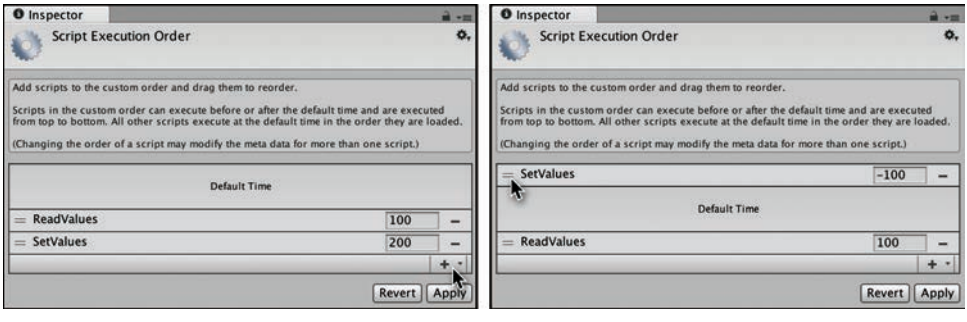

Рис. Б.1. Диалог Script Execution Order (Порядок выполнения сценариев)

7. Добавьте сценарий ReadValues, щелкнув на кнопке + (под указателем мыши на изображении слева на рис. Б.1).
8. Проделайте то же самое со сценарием SetValues.

   По умолчанию сценарии ReadValues и SetValues получают порядковые номера выполнения 100 и 200, как показано на рис. Б.1 слева.
9. Щелкните на кнопке Apply (Применить) и затем на кнопке Play (Играть) в Unity. При таком порядке выполнения вы гарантированно увидите в консоли сообщение NullReferenceException.
10. Остановите игру.
11. Ухватите мышью пиктограмму с двумя горизонтальными линиями, находящуюся рядом с именем SetValues (под указателем мыши на изображении справа на рис. Б.1), и переместите SetValues в таблицу над разделом Default Time. Теперь диалог в инспекторе должен выглядеть так, как показано на рис. Б.1 справа.
12. Щелкните на кнопке Apply (Применить) и затем на кнопке Play (Играть) в Unity.

Теперь `SetValues.Start()` гарантированно будет вызываться раньше, чем `ReadValues.Start()`, и в консоли появится результат «2».

Когда есть два сценария с методами `Start()`, `Awake()` или любыми другими обработчиками класса `MonoBehaviour`, которые автоматически вызываются движком Unity, только инструмент определения порядка выполнения сценариев поможет гарантировать их выполнение в определенном порядке. Все сценарии, для кото-

рых порядок не определен, могут оказаться в состоянии гонки, подобно сценарию ReadValues до того, как мы явно определили очередность его выполнения.

Рекурсивные функции

Иногда бывает необходимо, чтобы функция вызывала саму себя. Такие функции называют *рекурсивными*. Простейшим примером может служить функция вычисления факториала числа.

В математике 5! (5 факториал) — это произведение пятерки и всех предшествующих натуральных чисел:

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Особый случай 0! = 1, и для наших целей будем считать, что факториал любого отрицательного числа равен 0:

$0! = 1$

$-123! = 0$

С учетом всего этого напишем рекурсивную функцию, вычисляющую факториал любого целого числа:

```
 1 using UnityEngine;
 2 using System.Collections;
 3
 4 public class Factorial : MonoBehaviour {
 5
 6     void Awake() {
 7         print( fac (-1) ); // Выведет 0
 8         print( fac (0) );  // Выведет 1
 9         print( fac (5) );  // Выведет 120
10     }
11
12     int fac( int n ) {
13         if (n < 0) { // Предотвратить ошибку, если n<0
14             return( 0 );
15         }
16         if (n == 0) { // Это "терминальный случай" - условие прекращения
                         // рекурсии
17             return( 1 );
18         }
19         int result = n * fac( n-1 ); // Рекурсивный вызов
20         return( result );
21     }
22
23 }
```

Когда предыдущий код произведет вызов `fac(5)` и достигнет строки 19, он вызовет `fac(n-1)`, который соответствует вызову `fac(4)`. Этот процесс продолжится и вызовет `fac(n-1)` еще четыре раза, пока не произойдет вызов `fac(0)`. В строке 16 вызов

`fac(0)` обнаружит соответствие условию `n == 0` и вернет 1. Это «терминальный случай» рекурсии, столкнувшись с которым функция начинает возвращать значения. Эта 1 будет возвращена в строке 19, в рекурсивном вызове `fac(1)`, после этого `fac(1)` вернет 1 (результат умножения n * 1) в строке 20. Далее каждый рекурсивный вызов в цепочке сможет вернуть свое значение, раскручивая рекурсии в обратном направлении. Вот как выглядит порядок вычисления рекурсивных вызовов:

```
fac(5)
fac(5) * fac(4)
fac(5) * fac(4) * fac(3)
fac(5) * fac(4) * fac(3) * fac(2)
fac(5) * fac(4) * fac(3) * fac(2) * fac(1)
fac(5) * fac(4) * fac(3) * fac(2) * fac(1) * fac(0)
fac(5) * fac(4) * fac(3) * fac(2) * fac(1) * 1
fac(5) * fac(4) * fac(3) * fac(2) * 1
fac(5) * fac(4) * fac(3) * 2
fac(5) * fac(4) * 6
fac(5) * 24
120
```

Чтобы лучше понять суть происходящего в рекурсивной функции, установите точку останова в строке 19, подключите отладчик MonoDevelop к процессу Unity и используйте кнопку Step Into (Шаг внутрь), чтобы по шагам пройти весь цикл рекурсии. Чтобы освежить в памяти, как пользоваться отладчиком, прочитайте главу 25 «Отладка».

Рекурсивная функция для интерполяции кривых Безье

Другим фантастическим примером применения рекурсивных функций может служить статический метод интерполяции кривой Безье (с именем `Bezier`), входящий в класс `ProtoTools`, который объявлен в сценарии Utils из пакета, импортированного в начале главы 32 и следующих за ней. Эта функция может интерполировать позицию точки вдоль кривой Безье, заданной произвольным количеством опорных точек. Код функции `Bezier` вы найдете в конце раздела «Интерполяция» в этом приложении.

Шаблоны проектирования программного обеспечения

В 1994 году «банда четырех» (Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес) выпустили книгу «Design Patterns: Elements of Reusable Object-Oriented Software»[1], где описывают разные шаблоны проектирования, используемые в раз-

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vissides, «Design Patterns: Elements of Reusable Object-Oriented Software» (Reading, MA: Addison-Wesley, 1994). Среди многих других в их книге описан шаблон «Фабрика» (Factory). Там же описывается шаблон «Одиночка» (Singleton), который широко используется в учебных примерах в этой книге. (Гамма Эрих, Хелм Ричард, Джонсон Ральф, Влиссидес Джон. Приемы объектно-ориен-

работке программного обеспечения для создания эффективного кода, поддерживающего возможность повторного использования. В этой книге использовались два из этих шаблонов и упоминался третий.

Шаблон проектирования «Одиночка»

Шаблон проектирования «Одиночка» (Singleton) широко использовался в нескольких главах этой книги. Если известно, что в игре всегда будет только один экземпляр некоторого класса, можно создать объект-одиночку этого класса — статическую переменную с типом класса — и использовать его для ссылки на экземпляр из любого места в коде, как показано в следующем листинге:

```
public class Hero : MonoBehaviour {
    static public Hero S;                                          // a

    void Awake() {
        if (S == null) {                                           // c
            S = this;                                              // b
        } else {
            Debug.LogError("The singleton S of Hero has already been set!");
        }
    }
}

public class Enemy : MonoBehaviour {
    void Update() {
        public Vector3 heroLoc = Hero.S.transform.position;        // d
    }
}
```

a. Статическое общедоступное поле `S` — это объект-одиночка, представляющий героя. Для всех своих объектов-одиночек я обычно использую имя `S`.

b. Поскольку в игре может существовать только один экземпляр класса `Hero`, ссылка на него присваивается переменной `S` в методе `Awake()`, сразу после его создания.

c. Инструкция `if (S == null)` защищает от ошибочного создания второго экземпляра `Hero`. Если будет создан второй экземпляр `Hero`, который попытается присвоить ссылку на себя переменной `S`, в консоли появится сообщение об ошибке.

d. Так как поле `S` является общедоступным и статическим, на него можно сослаться из любого места в коде через имя класса, как `Hero.S`.

Поискав в интернете, вы наверняка увидите множество возражений против шаблона проектирования «Одиночка». Часто они обусловлены следующими двумя причинами:

тированного проектирования. Паттерны проектирования. СПб.: Питер, 2016. — *Примеч. пер.*)

- **Объекты-одиночки небезопасны в эксплуатационном окружении:** объекты-одиночки — это статические и общедоступные поля, то есть к ним могут обращаться ЛЮБЫЕ классы или функции. Опасность заключается в том, что некоторый класс, написанный кем-то другим, теоретически может изменить ссылку на экземпляр вашего класса, и вы не узнаете об этом!

  К счастью, эта проблема имеет несколько решений. Одно из них, которое мне нравится больше всего, — объявить поле со ссылкой на объект-одиночку статическим и *скрытым* (`private`), чтобы доступ к нему имел только экземпляр класса (а это может быть только один экземпляр, поскольку речь идет об одиночке). К этому полю можно добавить статическое общедоступное свойство, посредством которого другие классы смогут читать и изменять поле объекта-одиночки. Обнаружив, что какой-то неизвестный вам код изменяет свойство, вы сможете установить в отладчике точку останова в методе записи (`set`) свойства и с помощью панели Call Stack (Стек вызовов) в отладчике увидеть, какой метод изменяет свойство.

- **Шаблон проектирования «Одиночка» настолько прост в реализации, что им начинают злоупотреблять:** благодаря простоте реализации, многие начинают бесконтрольно использовать шаблон проектирования «Одиночка» и вскоре сталкиваются с проблемой, описанной в предыдущем пункте. Это означает, что нередко этот шаблон применяется не к месту.

При создании прототипов часто скорость разработки важнее безопасности, поэтому я советую при прототипировании использовать объекты-одиночки везде, где это покажется вам уместным, но в окончательной версии игры их лучше избегать.

Шаблон проектирования «Компонент»

Шаблон проектирования «Компонент» (Component) впервые упоминался в главе 27 «Объектно-ориентированное мышление». И он широко используется в самом движке Unity. Основная идея шаблона заключается в группировке тесно связанных функций и данных в один класс и, одновременно, в сохранении классов как можно более маленькими и специализированными.[1]

Все *компоненты*, подключаемые к игровым объектам в Unity, написаны с применением этого шаблона проектирования. Каждый игровой объект в Unity — это очень маленький класс, который может служить контейнером для множества компонентов, каждый из которых решает конкретную — независимо от других компонентов — задачу. Например:

- Компонент Transform обслуживает координаты, поворот, масштабирование и местоположение в иерархии.
- Компонент Rigidbody решает задачи, связанные с движением и моделированием законов физики.

[1] Полное определение шаблона проектирования «Компонент» намного сложнее, но для нас достаточно и такого упрощенного описания.

- Компоненты Collider поддерживают возможность обнаружения столкновений и определяют форму объема, в котором обнаруживаются столкновения.

Все эти задачи связаны между собой, и все же они достаточно разные, чтобы их решение реализовать в разных компонентах. Создание отдельных компонентов также упрощает расширение их возможностей в будущем: благодаря отделению компонента Collider от Rigidbody вы легко сможете добавить новый вид коллайдера, например конический коллайдер ConeCollider, и Rigidbody сможет использовать его без каких-либо изменений в коде Rigidbody.

Это особенно важно для разработчиков игровых движков, но что это дает разработчикам игр и прототипов? Прежде всего, следуя компонентно-ориентированному стилю, вы получаете более простые и короткие классы. Короткие сценарии проще писать, ими проще делиться с другими людьми, их проще использовать повторно, и они проще в отладке — все это весьма достойные цели.

Единственный недостаток компонентно-ориентированного подхода — его реализация требует большой предусмотрительности, что несколько противоречит философии прототипирования, согласно которой действующий прототип должен быть создан как можно быстрее. Из-за этой дилеммы в части III книги были представлены оба стиля: традиционный — в первых нескольких главах, где мы писали простые действующие прототипы, и более компонентно-ориентированный в последних главах. Более полно компонентно-ориентированный подход раскрывается в главе 35 «Прототип 7: DUNGEON DELVER» — совершенно новой, написанной с нуля для второго издания книги.

Шаблон проектирования «Стратегия»

Как отмечается в разделе «Делегаты функций» этого приложения, шаблон проектирования «Стратегия» (Strategy) часто используется для реализации искусственного интеллекта и других алгоритмов, поведение которых может изменяться в зависимости от условий, но при этом требуется вызывать один и тот же делегат функции. В шаблоне проектирования «Стратегия» для реализации действия некоторого вида (например, действия в бою) создается делегат функции и затем этому делегату присваиваются разные функции в зависимости от ситуации. Это избавляет от сложных инструкций `switch` в коде, потому что вызов делегата осуществляется всего одной строкой:

```
using UnityEngine;
using System.Collections;

public class Strategy : MonoBehaviour {
    public delegate void ActionDelegate();                              // a

    public ActionDelegate act;                                          // b

    public void Attack() {                                              // c
        // Здесь находится код, реализующий поведение в атаке
```

```
    }

    public void Wait() { ... } // Эти два метода определяют другие действия
    public void Flee() { ... } // Многоточие ( ... ) замещает реализацию метода

    void Awake() {
        act = Wait;                                                    // d
    }

    void Update() {
        Vector3 hPos = Hero.S.transform.position;
        if ( (hPos - transform.position).magnitude < 100 ) {           // e
            act = Attack;
        }
        if (act != null) act();                                        // f
    }
}
```

a. Определение типа делегата `ActionDelegate`. Он не имеет входных параметров и возвращает значение типа `void`.

b. Определение `act` — экземпляра делегата `ActionDelegate`.

c. Реализации функций `Attack()`, `Wait()` и `Flee()` здесь не показаны, они лишь демонстрируют определение разных действий и имеют параметры и тип возвращаемого значения, соответствующие типу делегата `ActionDelegate`.

d. Первоначально персонаж следует стратегии ожидания `Wait`, поэтому переменной `act` присваивается целевой метод `Wait`.

e. Если объект-одиночка `Hero` оказывается от данного персонажа на расстоянии меньше 100 метров, происходит переключение стратегии персонажа на `Attack` заменой целевого метода в `act`.

f. Независимо от выбранной стратегии, вызывается делегат `act()` для ее выполнения. Проверка `act != null` перед вызовом предотвращает вызов пустого делегата (то есть когда ему не присвоена ни одна из функций), который приводит к появлению ошибки во время выполнения.

Дополнительная информация о шаблонах проектирования программного обеспечения

Книга «Game Programming Patterns»[1] Роберта Нистрома — фантастический источник информации по шаблонам проектирования, часто используемым при разработке игр. Вы можете приобрести печатную или электронную копию книги во многих онлайн-магазинах или прочитать бесплатную веб-версию, доступную на сайте http://gameprogrammingpatterns.com. Это отличный ресурс для желающих повысить свое мастерство.

[1] Robert Nystrom, «Game Programming Patterns» (Genever Benning, 2014). ©2014 by Robert Nystrom.

Области видимости переменной

Области видимости переменных — важное понятие в любом языке программирования. Область видимости определяет область кода, в которой существует переменная. Переменная с *глобальной* областью видимости доступна любому коду, тогда как переменная с *локальной* областью видимости существует строго в определенной области и недоступна коду за ее границами. Если переменная локальна для класса, она будет недоступна коду вне этого класса. Если переменная локальна для функции, она будет существовать только в этой функции и исчезать, как только функция завершится.

Следующий пример демонстрирует несколько разных областей видимости переменных в одном классе. Комментарии, следующие за листингом, поясняют наиболее важные аспекты. Выделение переменной красным цветом подсказывает, что она недоступна в этом разделе кода.

Это реализация класса `ScopeExample`, наследующего `MonoBehaviour`:

```
using UnityEngine;
using System.Collections;

public class ScopeExample : MonoBehaviour {

    // общедоступные поля (общедоступные переменные класса)
    public bool                 trueOrFalse = false;                        // a
    public int                  graduationAge = 18;
    public float                goldenRatio = 1.618f;

    // скрытые поля (скрытые переменные класса)
    private bool                _hiddenVariable = true;                     // b
    private float               _anotherHiddenVariable = 0.5f;

    // защищенные поля (защищенные переменные класса)
    protected int               partiallyHiddenInt = 1;                     // c
    float                       anotherProtectedVariable = 1.0f;

    // статические поля (статические переменные класса)
    static public int           NUM_INSTANCES = 0;                          // d
    static private int          NUM_TOO = 0;                                // e

    public bool hiddenVariableAccessor {                                    // f
        get { return _hiddenVariable; }
    }

    void Awake() {
        trueOrFalse = true; // Работает: присвоит true переменной trueOrFalse // g
        print( "tOF: "+trueOrFalse ); // Работает: выведет "tOF: True"

        int ageAtTenthReunion = graduationAge + 10; // Работает              // h
        print( "_aHV: "+_anotherHiddenVariable ); // Работает:
                                                  // выведет "_aHV: 0.5"     // i
        NUM_INSTANCES += 1; // Работает                                      // j
        NUM_TOO++; // Работает                                               // k
```

```
    }

    void Update() {
        print( ageAtTenthReunion ); // ОШИБКА                                  // l
        float ratioed = 1f; // Работает
        for (int i=0; i<10; i++) { // Работает                                  // m
            ratioed *= goldenRatio; // Работает
        }
        print( "ratioed: "+ratioed ); // Работает: выведет "ratioed: 122.9661"
        print( i ); // ОШИБКА                                                  // n
    }
}
```

Реализация класса `ScopeExampleChild`, наследующего `ScopeExample`:

```
using UnityEngine;
using System.Collections;

public class ScopeExampleChild : ScopeExample {                               // o
    void Start() {
        print( "tOF: "+trueOrFalse );     // Работает: выведет "tOF: True" // p
        print( "pHI: "+partiallyHiddenInt ); // Работает: выведет "pHI: 1" // q
        print( "_hV: "+_hiddenVariable );    // ОШИБКА                     // r
        print( "NI: " +NUM_INSTANCES );      // Работает: выведет "NI: 1"  // s
        print( "NT: " +NUM_TOO );            // ОШИБКА                     // t
        print( "hVA: "+hiddenVariableAccessor ); // Работает: выведет
                                                 // "hVA: True"            // u
    }
}
```

a. Общедоступные поля: переменные `trueOrFalse`, `graduationAge` и `goldenRatio` — это **общедоступные** *поля*. *Поля* — это переменные экземпляра класса, они объявляются в определении класса и видимы всем функциям в любом экземпляре класса. Так как эти поля объявлены общедоступными (`public`), они наследуются подклассом `ScopeExampleChild`, то есть `ScopeExampleChild` также имеет, например, логическую переменную `trueOrFalse`. Общедоступные переменные видимы в любом коде, имеющем ссылку на экземпляр класса, то есть функция, имеющая переменную `ScopeExample se`, сможет прочитать и изменить поле `se.trueOrFalse`.

b. Скрытые поля: эти две переменные являются **скрытыми** полями. *Скрытые* (`private`) поля доступны только внутри экземпляра `ScopeExample` (то есть экземпляр `ScopeExample` может читать и изменять свои скрытые поля, но они недоступны другим экземплярам). Подклассы не наследуют скрытые поля, то есть подкласс `ScopeExampleChild` не имеет логической переменной `_hiddenVariable`. Функция, имеющая переменную `ScopeExample se`, не сможет прочитать и изменить скрытое поле `se._hiddenVariable`.

c. Защищенные поля: **защищенные** поля (объявленные как `protected`) занимают промежуточное положение между общедоступными и скрытыми полями. Подклассы наследуют защищенные поля, то есть подкласс `ScopeExampleChild` унаследует целочисленную переменную `partiallyHiddenInt`, объявленную в классе

ScopeExample. Однако защищенные поля недоступны за пределами класса или его подклассов, то есть функция, имеющая переменную ScopeExample se, не сможет прочитать и изменить защищенное поле se.partiallyHiddenVariable. Поля, объявленные без спецификатора области видимости private или public, по умолчанию считаются защищенными.

d. Статические поля: **статическое** поле (объявленные как static) — это поле самого класса, оно не принадлежит какому-то конкретному экземпляру. Это означает, что поле NUM_INSTANCES доступно как ScopeExample.NUM_INSTANCES. Общедоступные статические поля являются наиболее близким аналогом глобальных переменных в C#. Любой сценарий в проекте сможет обратиться к статическому полю ScopeExample.NUM_INSTANCES, а кроме того, NUM_INSTANCES будет хранить одно и то же значение для всех экземпляров ScopeExample. Функция, имеющая переменную ScopeExample se, не сможет прочитать или изменить поле se.NUM_INSTANCES (потому что его просто не существует), но она сможет обратиться к полю ScopeExample.NUM_INSTANCES. Класс ScopeExampleChild, наследующий ScopeExample, также имеет доступ к NUM_INSTANCES. Внутри экземпляра ScopeExample к полю NUM_INSTANCES можно обратиться непосредственно (без префикса ScopeExample.).

e. NUM_TOO — **скрытое статическое** (private static) поле. Все экземпляры ScopeExample совместно используют одно и то же значение NUM_TOO, но другие классы не имеют к нему доступа. Подкласс ScopeExampleChild не имеет доступа к NUM_TOO.

f. hiddenVariableAccessor — **общедоступное** свойство только для чтения, позволяющее другим классам читать значение _hiddenVariable. Свойство не имеет метода записи set, поэтому оно доступно только для чтения.

g. Комментарий // Работает означает, что эта строка выполнится без ошибок. trueOrFalse — это общедоступное поле ScopeExample, поэтому данный метод класса ScopeExample может обратиться к нему.

h. Эта строка объявляет и определяет переменную ageAtTenthReunion с локальной областью видимости, ограниченной границами метода ScopeExample.Awake(). Это означает, что когда функция ScopeExample.Awake() завершится, переменная ageAtTenthReunion перестанет существовать. Никакой другой код за пределами этой функции не сможет прочитать или изменить переменную ageAtTenthReunion.

i. Скрытое поле _anotherHiddenVariable доступно только внутри методов экземпляров этого класса.

j. Внутри класса к статическим общедоступным полям можно обращаться по их именам, то есть в методе ScopeExample.Awake() можно использовать имя NUM_INSTANCES без префикса с именем класса перед ним.

k. Переменная NUM_TOO также доступна из любого места внутри класса ScopeExample.

l. Комментарий `// ERROR` означает, что эта строка выполнится с ошибкой. Эта строка сгенерирует ошибку, потому что `ageAtTenthReunion` — это локальная переменная метода `ScopeExample.Awake()`, она недоступна в методе `ScopeExample.Update()`.

m. Переменная `int i` объявляется и определяется внутри цикла `for`, ее область видимости им ограничена. Это означает, что переменная `i` прекратит существование, как только цикл `for` завершится.

n. Эта строка сгенерирует ошибку, потому что `i` недоступна за пределами предшествующего цикла `for`.

o. Эта строка объявляет и определяет класс `ScopeExampleChild` как подкласс класса `ScopeExample`. Будучи подклассом, `ScopeExampleChild` имеет доступ ко всем общедоступным и защищенным полям и методам класса `ScopeExample`, но не может обратиться к скрытым полям и методам. Так как методы `Awake()` и `Update()` не были объявлены в классе `ScopeExample` общедоступными или скрытыми, они по умолчанию становятся защищенными и потому наследуются классом `ScopeExampleChild`. Так как `ScopeExampleChild` не определяет своих функций `Awake()` и `Update()`, он будет вызывать версии, объявленные в базовом классе `ScopeExample`.

p. `trueOrFalse` — общедоступная переменная, поэтому `ScopeExampleChild` унаследует поле `trueOrFalse`. Кроме того, поскольку метод `Awake()` базового класса (`ScopeExample`) выполнится к моменту вызова метода `Start()` в классе `ScopeExampleChild`, переменная `trueOrFalse` уже получит значение `true` в методе `Awake()` базового класса (`ScopeExample`).

q. `ScopeExampleChild` также наследует от `ScopeExample` защищенное поле `partiallyHiddenInt`.

r. Переменная `_hiddenVariable` не наследуется от класса `ScopeExample`, потому что она скрытая.

s. Переменная `NUM_INSTANCES` доступна в `ScopeExampleChild`; она объявлена как общедоступная, поэтому наследуется от базового класса `ScopeExample`. Два класса совместно используют одно и то же значение `NUM_INSTANCES`, то есть если будет создано по одному экземпляру каждого класса, `NUM_INSTANCES` вернет значение 2 при обращении из любого класса, `ScopeExample` или `ScopeExampleChild`.

t. Как скрытая статическая переменная, `NUM_TOO` не наследуется классом `ScopeExampleChild`. При этом важно отметить следующее: даже при том, что `NUM_TOO` не наследуется, при создании экземпляра `ScopeExampleChild` будет вызвана версия метода `Awake()` в базовом классе, то есть метод `Awake()`, объявленный в базовом классе `ScopeExample`, и этот вызов `Awake()` сможет обратиться к `NUM_TOO`, не генерируя ошибку, потому что эта версия метода выполняется в области видимости базового класса `ScopeExample`, хотя фактически запускается экземпляром класса `ScopeExampleChild`.

u. И самое необычное в нашем примере: `ScopeExampleChild` сможет прочитать общедоступное свойство `hiddenVariableAccessor`. На первый взгляд все кажется легко объяснимым, но стоит копнуть глубже... Метод `get` свойства `hiddenVariableAccessor` читает значение скрытого поля `_hiddenVariable`. Это тонкий, но важный аспект области видимости переменой. Так как `ScopeExampleChild` наследует `ScopeExample`, все скрытые поля в `ScopeExample` все равно будут созданы для экземпляра `ScopeExampleChild`, даже при том, что экземпляр `ScopeExampleChild` не сможет обратиться к ним непосредственно. Но `ScopeExampleChild` может пользоваться общедоступными средствами доступа, такими как свойство `hiddenVariableAccessor`, которое объявлено в области видимости базового класса `ScopeExample`, чтобы обратиться к скрытым полям, таким как `_hiddenVariable`, также находящимся в области видимости базового класса `ScopeExample`. Унаследованные методы, такие как `Awake()`, которые `ScopeExampleChild` наследует от `ScopeExample`, тоже имеют доступ к скрытым полям базового класса.

Эти многочисленные примечания описывают и очень простые, и очень сложные примеры областей видимости переменных. Если что-то вам показалось непонятным, не переживайте. Вы сможете вернуться сюда позже и снова прочитать этот раздел, когда еще немного поработаете с языком C# и у вас появятся более конкретные вопросы.

XML

XML (eXtensible Markup Language — расширяемый язык разметки) — это формат файлов, который специально проектировался, чтобы быть гибким и понятным человеку. Ниже приводится примере разметки XML из главы 32 «Прототип 4: PROSPECTOR SOLITAIRE». Я добавил дополнительные пробелы для удобочитаемости, но это никак не влияет на корректность разметки, потому что XML интерпретирует любое количество пробелов или разрывов строк, следующих подряд, как один пробел.

```
<xml>
    <!-- элементы decorator отображаются в углах каждой карты
         и представляют их масть и достоинство. -->
    <decorator type="letter" x="-1.05" y="1.42" z="0" flip="0" scale="1.25"/>
    <decorator type="suit" x="-1.05" y="1.03" z="0" flip="0" scale="0.4" />
    <decorator type="suit" x="1.05" y="-1.03" z="0" flip="1" scale="0.4" />
    <decorator type="letter" x="1.05" y="-1.42" z="0" flip="1" scale="1.25"/>
    <!-- Список всех карт, определяющий места размещения значков. -->
    <card rank="1">
        <pip x="0" y="0" z="0" flip="0" scale="2"/>
    </card>
    <card rank="2">
        <pip x="0" y="1.1" z="0" flip="0"/>
        <pip x="0" y="-1.1" z="0" flip="1"/>
    </card>
</xml>
```

Даже те, кто почти ничего не знает о XML, сможет извлечь некоторую информацию из этого фрагмента. XML основан на *тегах* (также известных как *разметка* документа) — словах, заключенных в угловые скобки (например, `<xml>`, `<card rank="2">`). Большинство *элементов* XML имеют *открывающий тег* (например, `<card rank="2">`) и *закрывающий тег*, содержащий символ слеша сразу после открывающей угловой скобки (например, `</card>`). Все, что находится между открывающим и закрывающим тегами элемента (например, теги `<pip .../>` между `<card>` и `</card>` в листинге выше), называют *содержимым* этого элемента. Существуют также теги *пустых элементов*, элементов без содержимого, которые одновременно являются открывающими и закрывающими. Например, в листинге выше тег `<pip x="0" y="1.1" z="0" flip="0" />` является тегом пустого элемента, который не требует наличия парного ему закрывающего тега `</pip>`, потому что сам завершается символами `/>`. В общем случае XML-файлы должны начинаться с тега `<xml>` и заканчиваться тегом `</xml>`, то есть все содержимое XML-документа является содержимым элемента `<xml>`.

Теги XML могут иметь *атрибуты*, которые можно сравнить с полями в C#. Пустой элемент `<pip x="0" y="1.1" z="0" flip="0"/>` в листинге выше включает атрибуты `x`, `y`, `z` и `flip`.

Все, что в XML-файле находится между последовательностями символов `<!--` и `-->`, считается *комментарием* и игнорируется программами, которые читают содержимое XML-файла. В предыдущем листинге можно видеть, как я использовал их, чтобы оставить свой комментарий, как я обычно делаю это в коде на C#.

В состав C# .NET входит надежный инструмент чтения XML, но он слишком объемный (увеличивает размер скомпилированной программы примерно на 1 Мбайт, что очень много, особенно для мобильных устройств) и громоздкий (то есть не самый простой в использовании). Поэтому в сценарии ProtoTools, который является частью пакета, импортированного в последних главах с учебными примерами, я подключил меньший по объему (хотя и не такой надежный) интерпретатор XML с именем `PT_XMLReader`. Пример его использования вы найдете в главе 32 «Прототип 4: PROSPECTOR SOLITAIRE».

Математика

Многие испытывают чувство неуверенности, когда слышат слово *математика*, но в действительности в ней нет ничего страшного. Как вы не раз увидите в этой книге, знание математики помогает находить по-настоящему интересные решения. Ниже я опишу несколько интересных математических идей, которые могут вам пригодиться в разработке игр.

Косинус и синус (функции Cos и Sin)

Синус и косинус — это функции, преобразующие величину угла Θ (тета) в точку на кривой волнообразной формы, координата Y которой изменяется в диапазоне от −1 до 1. Они показаны на рис. Б.2.

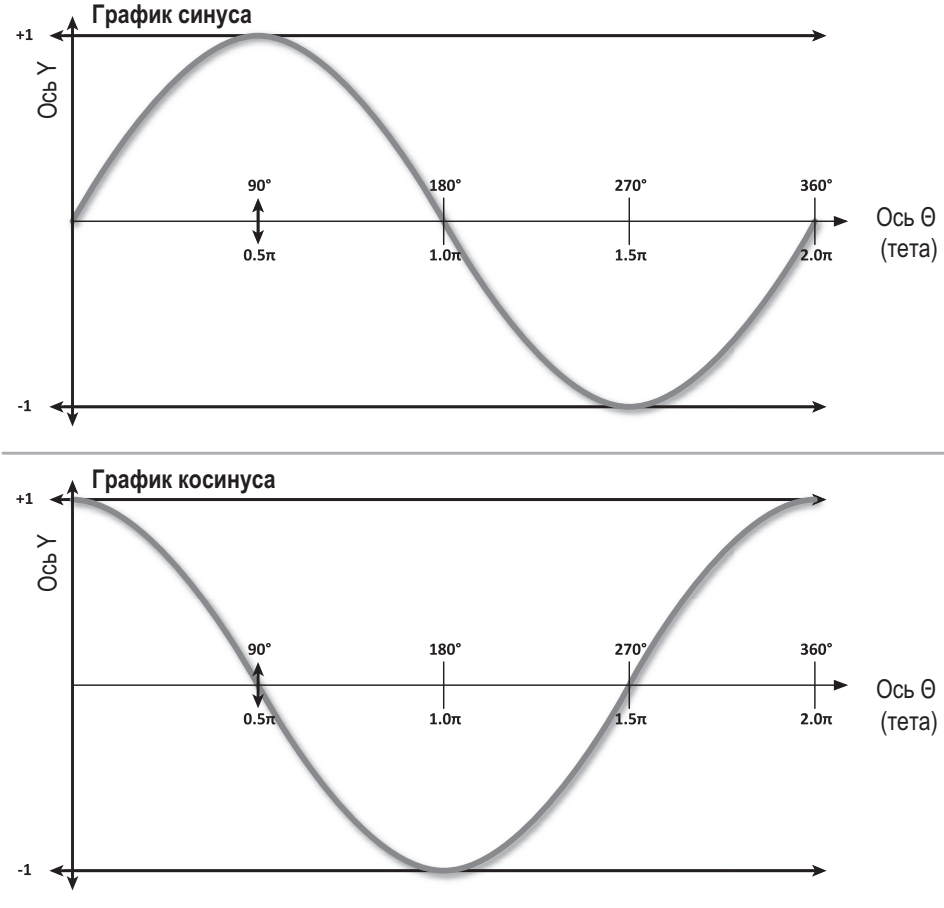

Рис. Б.2. Традиционное представление графиков синуса и косинуса

Однако синус и косинус — это нечто большее, чем просто волнообразные кривые; они описывают отношение между X и Y при вращении по кругу. Я покажу вам смысл сказанного на примере программного кода.

Пример в Unity — синус и косинус

Выполните следующие шаги:

1. Откройте Unity и создайте новую сцену. В верхней части панели Scene (Сцена) найдите кнопку с изображением горных пиков (правее кнопки с изображением динамика). Щелкните на ней; фон с изображением неба в панели Scene (Сцена) заменит темно-серый сплошной фон. Это сделает элементы в панели Scene (Сцена) более заметными (возможно, вам придется щелкнуть на кнопке не один раз).
2. Создайте в сцене новую сферу (GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера)). Настройте компонент Transform сферы: P:[0, 0, 0], R:[0, 0, 0], S:[0.1, 0.1, 0.1].
3. Добавьте в объект Sphere компонент TrailRenderer. (Щелкните на Sphere в иерархии и в главном меню Unity выберите пункт Component > Effects > Trail Renderer (Компонент > Эффекты > Визуализатор следа)). В инспекторе, в разделе TrailRenderer, щелкните на пиктограмме с треугольником рядом с элементом Materials, затем щелкните на кнопке с кружком правее поля Element 0 и выберите текстуру Default-Particle. Установите значения полей Time = 1 и Width = 0.1.
4. Создайте новый сценарий на C# с именем Cyclic. Подключите его к объекту Sphere в иерархии. Откройте сценарий в MonoDevelop и введите следующий код:

```
using UnityEngine;
using System.Collections;

public class Cyclic : MonoBehaviour {
    [Header("Set in Inspector")]
    public float            theta = 0;
    public bool             showCosX = false;
    public bool             showSinY = false;

    [Header("Set Dynamically")]
    public Vector3          pos;

    void Update () {
        // Вычислить значение радианов по текущему времени
        float radians = Time.time * Mathf.PI;
        // Преобразовать радианы в градусы для отображения в инспекторе
        // Операция "% 360" ограничивает значение диапазоном 0 .. 359.9999
        theta = Mathf.Round( radians * Mathf.Rad2Deg ) % 360;
        // Установить исходную позицию
        pos = Vector3.zero;
        // Вычислить x и y как косинус и синус угла соответственно
        pos.x = Mathf.Cos(radians);
        pos.y = Mathf.Sin(radians);

        // Использовать синус и косинус, если
        //   соответствующие флажки установлены в инспекторе
        Vector3 tPos = Vector3.zero;
        if (showCosX) tPos.x = pos.x;
        if (showSinY) tPos.y = pos.y;
```

```
        // Позиционировать this.gameObject (Sphere)
        transform.position = tPos;
    }

    void OnDrawGizmos() {
        if (!Application.isPlaying) return; // Показывать только во время
                                            // проигрывания

        // Нарисовать цветные волнистые линии
        //   (можете просто пропустить этот цикл for)
        int inc = 10;
        for (int i=0; i<360; i+=inc) {
            int i2 = i+inc;
            float c0 = Mathf.Cos(i*Mathf.Deg2Rad);
            float c1 = Mathf.Cos(i2*Mathf.Deg2Rad);
            float s0 = Mathf.Sin(i*Mathf.Deg2Rad);
            float s1 = Mathf.Sin(i2*Mathf.Deg2Rad);
            Vector3 vC0 = new Vector3( c0, -1f-(i/360f), 0 );
            Vector3 vC1 = new Vector3( c1, -1f-(i2/360f), 0 );
            Vector3 vS0 = new Vector3( 1f+(i/360f), s0, 0 );
            Vector3 vS1 = new Vector3( 1f+(i2/360f), s1, 0 );

            Gizmos.color = Color.HSVToRGB( i/360f, 1, 1 );
            Gizmos.DrawLine(vC0, vC1);
            Gizmos.DrawLine(vS0, vS1);
        }

        // Нарисовать линии и окружности относительно игрового объекта Sphere
        Gizmos.color = Color.HSVToRGB( theta/360f, 1, 1 );
        // Показать отдельные аспекты синуса и косинуса с использованием Gizmos
        Vector3 cosPos = new Vector3( pos.x, -1f-(theta/360f), 0 );
        Gizmos.DrawSphere(cosPos, 0.05f);
        if (showCosX) Gizmos.DrawLine(cosPos, transform.position);

        Vector3 sinPos = new Vector3( 1f+(theta/360f), pos.y, 0 );
        Gizmos.DrawSphere(sinPos, 0.05f);
        if (showSinY) Gizmos.DrawLine(sinPos, transform.position);
    }
}
```

5. Перед тем как щелкнуть на кнопке Play (Играть), выберите двумерный режим отображения сцены, щелкнув на кнопке 2D в верхней части панели Scene (Сцена). Щелкните на кнопке Play (Играть).

Вы увидите, что сначала сфера неподвижна, но двигаются цветные точки на графиках ниже и правее сферы. (Возможно, вам потребуется уменьшить или увеличить масштаб, чтобы увидеть их.) Точки на правом графике движутся по кривой, определяемой `Mathf.Sin(theta)`, а точки на нижнем графике — по кривой, определяемой `Mathf.Cos(theta)`.

Если теперь в инспекторе установить флажок showCosX в разделе Sphere:Cyclic (Script), объект Sphere начнет двигаться вдоль оси X, следуя за графиком косинуса. Вы увидите, как координата X объекта Sphere непосредственно связана с графиком

косинуса внизу. Снимите флажок showCosX и установите флажок showSinY. Теперь вы увидите, как координата Y объекта Sphere непосредственно связана с графиком синуса. Если установить оба флажка, showCosX и showSinY, объект Sphere будет двигаться по кругу, определяемому комбинацией X = cos(theta) и Y = sin(theta). Полный оборот составляет 360°, или 2π радиан (то есть `2 * Mathf.PI`).

Эта связь также показана на рис. Б.3, где используются цвета, близкие к выбранным в примере Unity.

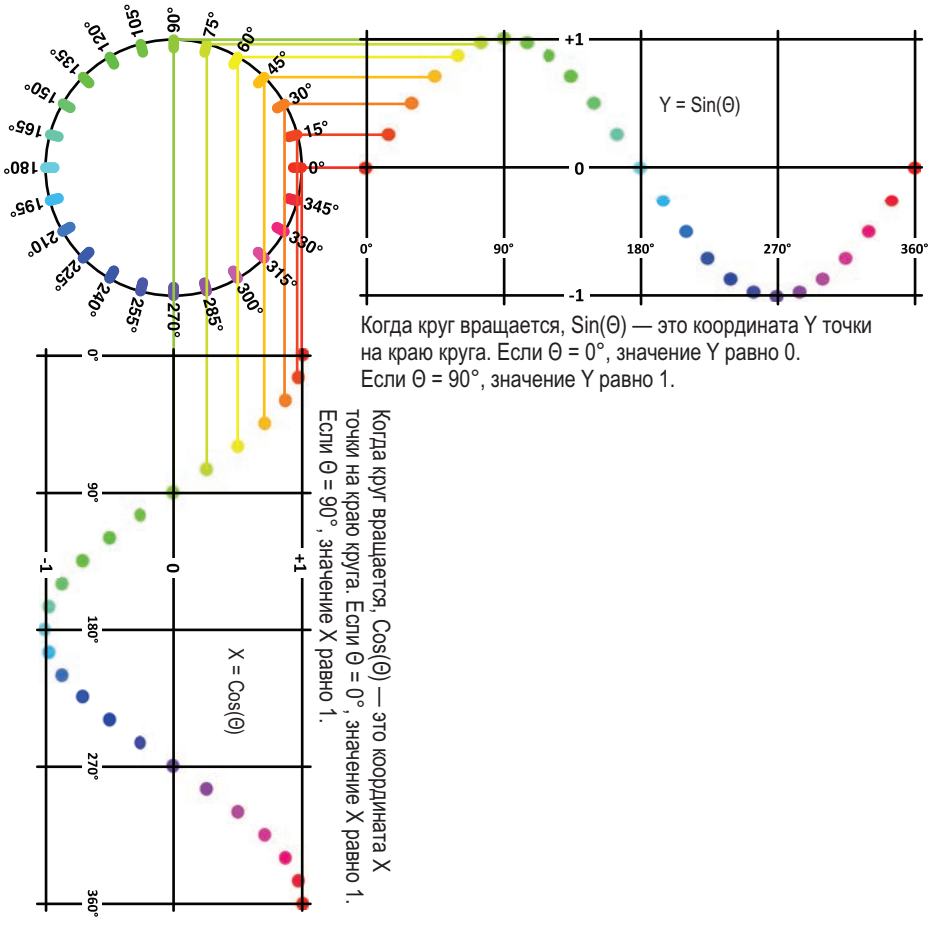

Рис. Б.3. Связь синуса и косинуса с окружностью

Это означает, что синус и косинус можно использовать для реализации всех видов циклического поведения!

Эти свойства синуса и косинуса использовались в главе 31 «Прототип 3.5: SPACE SHMUP PLUS», для реализации волнообразного движения врагов типа Enemy_1

и изменения скорости движения врагов типа Enemy_2 методом линейной интерполяции с функцией сглаживания (подробнее о линейной интерполяции и функциях сглаживания рассказывается в разделе «Интерполяция» в этом приложении).

Вероятность игральной кости

В главе 11 «Математика и баланс игры» приводились десять правил определения вероятности, сформулированные Джесси Шеллом, где правило 4 гласит: «Сложные математические задачи можно решать методом перебора». Ниже приводится короткая программа для Unity, которая может перечислить все возможные исходы для любого количества костей с любым количеством граней. Но будьте осторожны: добавляя новую кость, можно значительно увеличить объем вычислений (например, перебор исходов для кости 5d6 (пять шестигранных кубиков) занимает в шесть раз больше времени, чем перебор исходов для кости 4d6, и в 36 раз больше, чем для кости 3d6.)

Пример в Unity — вероятность игральной кости

Выполните следующие шаги, чтобы создать программу, которая перечислит все возможные исходы броска любого количества костей с любым количеством граней. По умолчанию в коде используется кость 2d6 (два шестигранных кубика). С этими значениями по умолчанию программа переберет все возможные варианты выпадения очков на двух кубиках (например, 1|1, 1|2, 1|3, 1|4, 1|5, 1|6, 2|1, 2|2, ... 6|5, 6|6) и подсчитает вероятность выпадения каждой суммы.

1. Создайте новый проект в Unity. Создайте новый сценарий на C# с именем DiceProbability и подключите его к главной камере Main Camera в панели Scene (Сцена). Откройте сценарий DiceProbability в MonoDevelop и введите следующий код:

```
using UnityEngine;
using System.Collections;

public class DiceProbability : MonoBehaviour {
    [Header("Set in Inspector")]
    public int      numDice = 2;
    public int      numSides = 6;
    public bool     checkToCalculate = false;
    // ^ Вычисления начинаются после установки флажка checkToCalculate
    public int      maxIterations = 10000;
    // ^ Максимальное число итераций для одного цикла вычислений
    //   в сопрограмме CalculateRolls()
    public float    width = 16;
    public float    height = 9;

    [Header("Set Dynamically")]
    public int[]    dice;  // Массив значений для каждой кости
    public int[]    rolls; // Массив выпадений каждого исхода
    // для кости 2d6 массив rolls может, например, содержать
    //   [ 0, 0, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1 ].
```

```
// Число 1 во втором элементе массива означает, что сумма 2 выпала 1 раз.
// Число 6 в седьмом элементе - что сумма 7 выпала 6 раз.

void Awake() {
    // Настройка камеры для отображения графика
    Camera cam = Camera.main;
    cam.backgroundColor = Color.black;
    cam.orthographic = true;
    cam.orthographicSize = 5;
    cam.transform.position = new Vector3(8, 4.5f, -10);
}

void Update() {
    if (checkToCalculate) {
        StartCoroutine( CalculateRolls() );
        checkToCalculate = false;
    }
}

void OnDrawGizmos() {
    float minVal = numDice;
    float maxVal = numDice*numSides;

    // Если массив rolls не готов - выйти
    if (rolls == null || rolls.Length == 0 || rolls.Length != maxVal+1) {
        return;
    }

    // Отобразить массив rolls
    float maxRolls = Mathf.Max(rolls);
    float heightMult = 1f/maxRolls;
    float widthMult = 1f/(maxVal-minVal);

    Gizmos.color = Color.white;
    Vector3 v0, v1 = Vector3.zero;
    for (int i=numDice; i<=maxVal; i++) {
        v0 = v1;
        v1.x = ( (float) i - numDice ) * width * widthMult;
        v1.y = ( (float) rolls[i] ) * height * heightMult;
        if (i != numDice) {
            Gizmos.DrawLine(v0,v1);
        }
    }
}

public IEnumerator CalculateRolls() {
    // Вычислить максимальное значение - максимально возможную сумму
    //   (например, для 2d6 maxValue = 12)
    int maxValue = numDice*numSides;
    // Создать массив достаточного размера для хранения всех возможных сумм
    rolls = new int[maxValue+1];

    // Создать массив, каждый элемент которого соответствует одной кости.
    // Всем элементам присвоить начальное значение 1, кроме нулевого,
    //   которому следует присвоить 0 (чтобы обеспечить правильную
```

```
    //   работу метода RecursivelyAddOne())
    dice = new int[numDice];
    for (int i=0; i<numDice; i++) {
        dice[i] = (i==0) ? 0 : 1;
    }

    // Итерации костей.
    int iterations = 0;
    int sum = 0;

    // Обычно я избегаю циклов while, потому что они могут превращаться
    //   в бесконечные циклы, но так как это сопрограмма с инструкцией
    //   yield в теле цикла while, это перестает быть большой проблемой.
    while (sum != maxValue) {
        // ^ Сумма sum будет == maxValue, когда все кости получат
        //   их максимальные значения

        // Увеличить на 1 значение нулевой кости в массиве dice
        RecursivelyAddOne(0);

        // Суммировать очки
        sum = SumDice();
        // и прибавить 1 к элементу с этим индексом в массиве rolls
        rolls[sum]++;

        // увеличить число итераций iterations и приостановиться
        iterations++;
        if (iterations % maxIterations == 0) {
            yield return null;
        }
    }
    print("Calculation Done");

    string s = "";
    for (int i=numDice; i<=maxValue; i++) {
        s += i.ToString()+" "+rolls[i].ToString("N0")+"\n";            // a
    }

    int totalRolls = 0;
    foreach (int i in rolls) {
        totalRolls += i;
    }
    s += "\nTotal Rolls: "+totalRolls.ToString("N0")+"\n";            // a

    print(s);
}

// Это рекурсивный метод - он вызывает сам себя. Дополнительную информацию
//   о рекурсивных функциях вы найдете в этом приложении.
public void RecursivelyAddOne(int ndx) {
    if (ndx == dice.Length) return; // Все кости в массиве dice просмотрены,
                                    //   поэтому просто выйти

    // Взять следующее число очков на кости в позиции ndx
    dice[ndx]++;
```

```
        // Если оно превысило число граней кости...
        if (dice[ndx] > numSides) {
            dice[ndx] = 1; // записать значение 1...
            RecursivelyAddOne(ndx+1); // и перейти к следующей кости
        }
        return;
    }

    public int SumDice() {
        // Сложить число очков на всех костях в массиве dice
        int sum = 0;
        for (int i=0; i<dice.Length; i++) {
            sum += dice[i];
        }
        return(sum);
    }
}
```

a. Вызов метода `.ToString("N0")` демонстрирует пример использования стандартных строк форматирования чисел в C#. Символ `N` требует от `ToString()` добавить разделитель групп разрядов перед каждой тройкой цифр (например, запятые в числе 123,456,789), а символ нуля `0` означает, что после десятичной точки должно выводиться ноль цифр. Поищите в интернете по фразе «C# Строки стандартных числовых форматов», чтобы найти описание форматов, поддерживаемых методом `ToString`.

2. Чтобы воспользоваться сценарием DiceProbability, щелкните на кнопке Play (Играть) и затем выберите Main Camera в панели Hierarchy (Иерархия).

3. В инспекторе, в разделе Main Camera:Dice Probability (Script) определите число костей в поле numDice и количество граней на каждой кости numSides и затем установите флажок checkToCalculate, чтобы запустить вычисления вероятности выпадения каждой возможной суммы очков.

   Unity проверит все возможные варианты и выведет результаты в виде списка чисел в панели Console (Консоль), а также в виде графика в панели Scene (Сцена). Чтобы график было лучше видно, щелкните на кнопке с изображением горных пиков в верхней части панели Scene (Сцена); чтобы отключить отображение неба, переключитесь в двумерный режим отображения и уменьшите масштаб.

   Сначала опробуйте сценарий со значениями по умолчанию — с двумя шестигранными кубиками (2d6), и вы увидите в консоли следующие результаты (щелкните на сообщении в консоли, чтобы увидеть весь текст, а не только первые две строки):

```
2       1
3       2
4       3
5       4
6       5
7       6
```

```
8       5
9       4
10      3
11      2
12      1

Total Rolls: 36

UnityEngine.MonoBehaviour:print(Object)
<CalculateRolls>c__Iterator0:MoveNext() (at Assets/DiceProbability.cs:110)
UnityEngine.MonoBehaviour:StartCoroutine(IEnumerator)
DiceProbability:Update() (at Assets/DiceProbability.cs:34)
```

4. В инспекторе попробуйте изменить значения полей: numDice=8 и numSides=6. Затем установите флажок checkToCalculate.

Вы увидите, что теперь для вычислений требуется намного больше времени и результаты (а также график) постоянно обновляются, когда сопрограмма приостанавливается (см. раздел «Сопрограммы» в этом приложении). Чтобы увеличить скорость, попробуйте ввести в поле maxIterations число 100 000. maxIterations — это количество итераций, которые код выполнит перед тем, как сопрограмма приостановится и позволит движку Unity вывести результаты. Чем больше значение maxIterations, тем выше общая скорость вычислений, потому что код будет выполнять больше итераций между отображением результатов. Чем меньше maxIterations, тем чаще будут обновляться результаты, но при этом общее время вычислений будет существенно увеличиваться.

Теперь всякий раз, когда вам понадобится определить, например, вероятность выпадения 13 очков при броске кости 8d6, вы сможете сделать это с помощью сценария. Вот некоторые примечательные строки из вывода в консоли для этого случая:

```
8       1
9       8
...
12      330
13      792
14      1,708
...
47      8
48      1

Total Rolls: 1,679,616
```

Полученные числа означают, что вероятность выпадения 13 очков для кости 8D6 составляет 792 / 1,679,616 = 11 / 23,328 ≈ 0,00047 ≈ 0,05 %.

Вы можете изменить этот код, чтобы он «бросал кости» определенное число раз и выбирал случайный результат. Чем больше будет количество бросков, тем ближе результат окажется к практической вероятности, в противовес теоретической, которая определяется сейчас (см. правило 9 Джесси Шелла в главе 11 «Математика и баланс игры»).

Скалярное произведение

Скалярное произведение — еще одна очень полезная идея. Скалярное произведение двух векторов — это сумма попарных произведений компонентов `X`, `Y` и `Z` этих векторов, как показано в следующем листинге:

```
1 Vector3 a = new Vector3( 1, 2, 3 );
2 Vector3 b = new Vector3( 4, 5, 6 );
3 float dotProduct = a.x*b.x + a.y*b.y + a.z*b.z;                    // a
4 // dotProduct = 1*4 + 2*5 + 3*6
5 // dotProduct = 4 + 10 + 18
6 // dotProduct = 32
7 dotProduct = Vector3.Dot(a,b); // Именно так это делается в C#     // b
```

a. В строке 3 показано, как вручную вычислить скалярное произведение переменных `a` и `b` типа `Vector3`.

b. В строке 7 показано, как выполнить те же вычисления с помощью встроенного статического метода `Vector3.Dot()`.

Кому-то эта идея не покажется важной, тем не менее скалярные произведения обладают *чрезвычайно* полезным свойством: значение, получаемое в результате скалярного произведения[1] a•b, эквивалентно *a*.magnitude * *b*.magnitude * Cos(Θ), где Θ — угол между двумя векторами, как показано на рис. Б.4.

На рис. Б.4.A показан стандартный пример скалярного произведения. В этом примере единичный вектор[2] *b* простирается вдоль оси X. *b* имеет координаты [1, 0], а вектор *a* имеет координаты [1, 1]. Вектор *a* можно представить как состоящий из двух частей: параллельной вектору *b* (тонкий зеленый вектор поверх вектора *b*) и перпендикулярной вектору *b* (показан зеленым пунктиром). Длину части *a*, которая параллельна *b*, называют *проекцией a на b* и вычисляют как скалярное произведение *a*•*b*. Скалярное произведение берет всю длину вектора *a* [1, 1], которая равна квадратному корню из 2 (≈1,414), и сообщает, какая ее часть параллельна вектору *b*. Как отмечалось выше, найти скалярное произведение можно двумя способами, оба они показаны на рис. Б.4.A, и оба дают в результате 1. Это означает, что длина проекции вектора *a* на единичный вектор *b* равна 1.

На рис. Б.4.B показано, что когда два вектора полностью перпендикулярны, их скалярное произведение равно 0. То есть проекция *a* на *b* имеет нулевую длину.

На рис. Б.4.C Показана проекция более длинного вектора *a* на *b*. И снова оба способа вычисления скалярного произведения дают один и тот же правильный результат.

---

[1] Здесь (и вообще в математике) для представления скалярного произведения используется символ •, а не привычная звездочка *, представляющая произведение чисел, и не символ ×, представляющий векторное произведение двух векторов.

[2] *Единичный вектор* — это вектор с длиной 1.

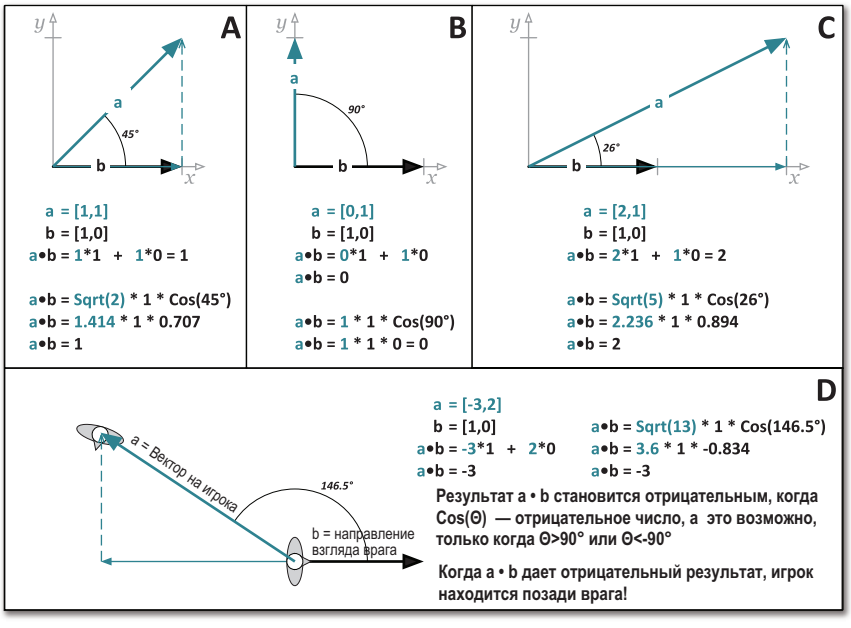

Рис. Б.4. Примеры скалярного произведения (числа представляют приближенные значения)

Как показано на рис. B.4.D, скалярное произведение помогает также определить, повернут ли враг лицом к персонажу игрока (что может пригодиться в стелс-играх). Здесь вектор *a* имеет координаты [−3, 2], а *b* — координаты [1, 0]. Скалярное произведение *a* • *b* дает в результате −3. Если враг смотрит в направлении *b* и скалярное произведение вектора *a*, указывающего на игрока, с единичным вектором *b* возвращает отрицательное значение, это означает, что игрок находится позади врага. Хотя во всех примерах на рис. Б.4 вектор *b* указывает вдоль оси X, скалярное произведение с равным успехом можно использовать, если *b* будет указывать в любом другом направлении, при условии, что *b* — единичный вектор.

Скалярное произведение можно также использовать во многих других случаях, и оно широко применяется в программировании компьютерной графики (например, с помощью скалярного произведения можно определить, повернута ли поверхность к источнику света).

Интерполяция

Под интерполяцией в математике понимается способ определения промежуточных значений по двум величинам. Когда я работал программистом по контракту после окончания колледжа, я получил множество предложений от работодателей, потому что, как мне кажется, элементы в моей графике двигались плавно и сочно

(если использовать термин Кайла Габлера (Kyle Gabler)[1]). Это достигалось за счет использования разных форм интерполяции, сглаживания и применения кривых Безье, о которых я расскажу в этом разделе.

Линейная интерполяция

Линейная интерполяция — это способ математически определить новое значение или позицию, находящиеся между двумя известными значениями. Все виды линейной интерполяции выполняются по одной и той же формуле:

p01 = (1 − u) * p0 + u * p1

В коде она выглядит примерно так:

```
1 Vector3 p0 = new Vector3( 0, 0, 0 );
2 Vector3 p1 = new Vector3( 1, 1, 0 );
3 float   u = 0.5f;
4 Vector3 p01 = (1-u) * p0 + u * p1;
5 print(p01); // выведет: ( 0.5, 0.5, 0 ) точка на середине пути между p0 и p1
```

В предыдущем листинге путем интерполяции определяются координаты новой точки между точками `p0` и `p1`. Значение `u` изменяется в диапазоне между 0 и 1. Интерполяция может выполняться для любого числа измерений, хотя в Unity она обычно применяется к значениям типа `Vector3`.

Линейная интерполяция во времени

Линейная интерполяция во времени гарантирует завершение процесса интерполяции за определенный интервал времени, потому что значение `u` зависит от прошедшего времени, деленного на общую желаемую продолжительность интерполяции.

Пример в Unity — линейная интерполяция во времени

Чтобы создать пример в Unity, выполните следующие шаги:

1. Создайте новый проект с именем Interpolation Project. Сохраните сцену как _Scene_Interp.
2. Создайте в иерархии куб (GameObject > 3D Object > Cube (Игровой объект > 3D объект > Куб)).
   a. Выберите Cube в панели Hierarchy (Иерархия) и подключите к нему компонент TrailRenderer (Component > Effects > Trail Renderer (Компонент > Эффекты > Визуализатор следа)).

[1] «Juice It or Lose It» (сделай сочным или потеряешь) — отличное выступление Мартина Джонассона (Martin Jonasson) и Петри Пурхо (Petri Purho) на тему добавления сочности в игры в 2012 году. Видеозапись выступления доступна по адресу https://www.youtube.com/watch?v=Fy0aCDmgnxg. Если же ссылка окажется недействительной, просто поищите в интернете по фразе: «juice it or lose it».

b. Раскройте массив Materials в компоненте TrailRenderer и в поле Element 0 выберите встроенный материал Default-Particle. (Щелкните на кнопке с кружком правее поля Element 0, чтобы увидеть Default-Particle в списке доступных материалов.)

3. В панели Project (Проект) создайте новый сценарий на C# с именем Interpolator. Подключите его к объекту Cube, откройте в MonoDevelop и введите следующий код:

```
using UnityEngine;
using System.Collections;

public class Interpolator : MonoBehaviour {
    [Header("Set in Inspector")]
    public Vector3      p0 = new Vector3(0,0,0);
    public Vector3      p1 = new Vector3(3,4,5);
    public float        timeDuration = 1;
    // Установите флажок checkToStart, чтобы запустить перемещение
    public bool         checkToStart = false;

    [Header("Set Dynamically")]
    public Vector3      p01;
    public bool         moving = false;
    public float        timeStart;

    // Update вызывается в каждом кадре
    void Update () {
        if (checkToStart) {
            checkToStart = false;

            moving = true;
            timeStart = Time.time;
        }

        if (moving) {
            float u = (Time.time-timeStart)/timeDuration;
            if (u>=1) {
                u=1;
                moving = false;
            }

            // Стандартная формула линейной интерполяции
            p01 = (1-u)*p0 + u*p1;
            transform.position = p01;
        }
    }
}
```

4. Вернитесь в Unity и щелкните на кнопке Play (Играть). В компоненте Cube:Interpolator (Script) установите флажок checkToStart, и объект Cube переместится из точки p0 в точку p1 за 1 секунду. Если ввести в поле timeDuration другое значение и затем снова установить флажок checkToStart, вы увидите, что Cube всегда перемещается из точки p0 в точку p1 за timeDuration секунд. Вы мо-

жете изменить координаты точки `p0` или `p1`, и траектория движения куба Cube изменится соответственно.

Линейная интерполяция с использованием парадокса Зенона

Зенон Элейский (примерно 490–430 гг. до н. э.) — греческий философ, сформулировавший несколько парадоксов относительно движения, противоречащих здравому смыслу.

В парадоксе «Дихотомия» Зенон ставит под вопрос возможность достижения конечной точки движущимся объектом. Вообразите лягушку, прыгающую в направлении к стене. В каждом прыжке она преодолевает половину оставшегося расстояния. Сколько бы прыжков ни сделала лягушка, каждый следующий прыжок все равно будет покрывать половину расстояния, оставшегося до стены после последнего прыжка, поэтому лягушка никогда не достигнет стены.

Игнорируя философский подтекст (и полное отсутствие здравого смысла), похожую идею можно использовать наряду с линейной интерполяцией для создания эффекта движения к некоторой точке с плавно уменьшающейся скоростью. Этот прием, например, используется в этой книге для плавного перемещения камеры к разным точкам.

Пример в Unity — интерполяция парадокса Зенона

Продолжим работу с проектом Interpolation Project, созданным выше:

1. Добавьте в сцену сферу (GameObject > 3D Object > Sphere (Игровой объект > 3D объект > Сфера)) и поместите ее где-нибудь в стороне от объекта Cube.
2. Создайте в панели Project (Проект) новый сценарий на C# с именем ZenosFollower и подключите его к объекту Sphere.
3. Откройте сценарий ZenosFollower в MonoDevelop и введите следующий код:

```
using UnityEngine;
using System.Collections;

public class ZenosFollower : MonoBehaviour {
    [Header("Set in Inspector")]
    public GameObject              poi; // Point Of Interest - точка интереса
    public float                   u = 0.1f;
    public Vector3                 p0, p1, p01;

    // FixedUpdate вызывается при каждом пересчете физики (50 раз в секунду)
    void FixedUpdate () {
        // Получить позицию этого игрового объекта и poi
        p0 = this.transform.position;
        p1 = poi.transform.position;
```

```
        // Выполнить интерполяцию между ними
        p01 = (1-u)*p0 + u*p1;

        // Переместить этот игровой объект в новую позицию
        this.transform.position = p01;
    }
}
```

4. Сохраните сценарий и вернитесь в Unity.
5. В поле poi компонента Sphere:ZenosFollower выберите куб (перетащив Cube из панели Hierarchy (Иерархия) в поле poi компонента Sphere:ZenosFollower (Script) в инспекторе).
6. Сохраните сцену!

Если теперь щелкнуть на кнопке Play (Играть), сфера плавно переместится в точку, где находится куб. Если в настройках куба в инспекторе установить флажок checkToStart, сфера последует за перемещением куба. Можете попробовать вручную перетащить куб мышью в панели Scene (Сцена) и понаблюдать за поведением сферы.

Попробуйте изменить значение u в настройках компонента Sphere:ZenosFollower в инспекторе. С уменьшением значения сфера будет перемещаться медленнее, а с увеличением — быстрее. При значении *0.5* сфера будет преодолевать половину расстояния до куба в каждом кадре, точно имитируя парадокс Зенона «Дихотомия» (в действительности не совсем точно, но очень близко). Это верно, что с таким кодом сфера никогда не достигнет точки, где находится куб, и ее движение почти не контролируется, но цель этого примера — показать простой и короткий сценарий, реализующий следование за точкой интереса.

В сценарии ZenosFollower вместо `Update()` использован метод `FixedUpdate()`, чтобы обеспечить одинаковое поведение на компьютерах всех типов. Если использовать `Update()`, то, в зависимости от нагрузки на процессор в каждый конкретный момент времени, сфера могла бы иногда отставать от куба из-за разного числа вызовов `Update()` в каждую секунду, потому что частота кадров может меняться. По той же причине при использовании `Update()` на быстрых машинах сфера имела бы более высокую скорость движения, чем на медленных. Использование метода `FixedUpdate()` обеспечивает единообразие поведения на всех машинах и все время, потому что он всегда вызывается с частотой 50 раз в секунду.[1]

[1] `FixedUpdate()` вызывается 50 раз в секунду, потому что по умолчанию `Time.fixedDeltaTime` имеет значение 0,02 (1/50 секунды), то есть, изменив величину `Time.fixedDeltaTime`, можно изменить частоту вызова `FixedUpdate()`. Это может пригодиться при изменении `Time.timeScale`, например, до величины 0,1 (уменьшит обычную скорость работы движка Unity в 10 раз). Если свойству `Time.timeScale` присвоить значение 0,1, метод `FixedUpdate()` будет вызываться каждые 0.2 секунды реального времени, что может создать визуальный эффект движения рывками. Всякий раз, изменяя `Time.timeScale`, вы должны также пропорционально изменить `Time.fixedDeltaTime`; то есть для значения 0,1 в `Time.timeScale` следует присвоить свойству `Time.fixedDeltaTime` значение 0,002, чтобы `FixedUpdate()` продолжал вызываться с частотой 50 раз в секунду реального времени.

Интерполировать можно не только координаты

Интерполировать можно практически любые числовые значения. Это означает, что в Unity очень легко можно интерполировать, например, такие значения, как масштаб, поворот и цвет.

Пример в Unity — интерполяция разных атрибутов

Для реализации следующего примера можно использовать проект из предыдущих разделов, посвященных интерполяции, или создать новый:

1. Создайте сцену с именем _Scene_Interp2 и добавьте в иерархию два куба с именами c0 и c1.
2. Создайте для каждого куба новый материал (Assets > Create > Material (Ресурсы > Создать > Материал)) с именами Mat_c0 и Mat_c1.
3. Перетащите каждый материал на соответствующий куб.
4. Выберите c0 и настройте его *координаты*, *поворот* и *масштаб* по своему желанию (главное, чтобы куб оставался видимым на экране и его масштаб по осям X, Y и Z описывался положительными значениями). В инспекторе, в разделе c0:Mat_c0, выберите любой желаемый цвет.
5. Аналогично настройте куб c1 и его материал Mat_c1, но так, чтобы c1 и c0 имели разные координаты, поворот, масштаб и цвет.
6. Добавьте в сцену третий куб с именем Cube01 и настройте его координаты P:[0, 0, 0].
7. Создайте новый сценарий на C# с именем Interpolator2, подключите его к объекту Cube01 и введите следующий код:

```
using UnityEngine;
using System.Collections;

public class Interpolator2 : MonoBehaviour {
    [Header("Set in Inspector")]
    public Transform    c0;
    public Transform    c1;
    public float        timeDuration = 1;
    // Установите флажок checkToStart, чтобы запустить перемещение
    public bool         checkToStart = false;

    [Header("Set Dynamically")]
    public Vector3      p01;
    public Color        c01;
    public Quaternion   r01;
    public Vector3      s01;
    public bool         moving = false;
    public float        timeStart;

    private Material    mat, matC0, matC1;
```

```
    void Awake() {
        mat = GetComponent<Renderer>().material;
        matC1 = c1.GetComponent<Renderer>().material;
        matC0 = c0.GetComponent<Renderer>().material;
    }

    // Update вызывается в каждом кадре
    void Update () {
        if (checkToStart) {
            checkToStart = false;

            moving = true;
            timeStart = Time.time;
        }

        if (moving) {
            float u = (Time.time-timeStart)/timeDuration;
            if (u>=1) {
                u=1;
                moving = false;
            }

            // Стандартная формула линейной интерполяции
            p01 = (1-u)*c0.position + u*c1.position;
            c01 = (1-u)*matC0.color + u*matC1.color;
            s01 = (1-u)*c0.localScale + u*c1.localScale;
            // Углы поворота обрабатываются иначе из-за особенностей Quaternion
            r01 = Quaternion.Slerp(c0.rotation, c1.rotation, u);

            // Применить новые значения к Cube01
            transform.position = p01;
            mat.color = c01;
            transform.localScale = s01;
            transform.rotation = r01;
        }
    }
}
```

8. Сохраните сценарий и вернитесь в Unity.
9. Перетащите `c0` из панели Hierarchy (Иерархия) в поле `c0` компонента Cube01:Interpolator2 (Script) в инспекторе. Также перетащите `c1` из иерархии в поле `c1` компонента Cube01:Interpolator2 (Script).
10. Щелкните на кнопке Play (Играть) и затем установите флажок checkToStart компонента Cube01:Interpolator2 в инспекторе. Вы увидите, что Cube01 теперь интерполируются не только координаты Cube01.

Линейная экстраполяция

Во всех примерах интерполяции, представленных выше, значение `u` изменялось в диапазоне от 0 до 1. Если позволить значению `u` выйти за границы этого диапазона, получится *экстраполяция* (эта процедура получила такое название, потому что

вместо интерполирования между двумя значениями она экстраполирует данные за границы двух исходных точек).

Для двух исходных точек на числовой прямой со значениями 10 и 20 экстраполяция для u=2 даст результат, изображенный на рис. Б.5.

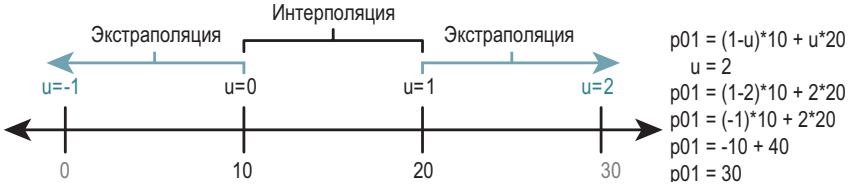

Рис. Б.5. Пример экстраполяции

Пример в Unity — линейная экстраполяция

Чтобы увидеть, как осуществляется экстраполяция в коде, добавьте следующие строки, выделенные жирным, в сценарий Interpolator2. Кроме экстраполяции новый код также позволяет выполнять бесконечное повторение движения.

```
public class Interpolator2 : MonoBehaviour {
    [Header("Set in Inspector")]
    public Transform    c0;
    public Transform    c1;
    public float        uMin = 0;
    public float        uMax = 1;
    public float        timeDuration = 1;
    public bool         loopMove = true; // Включает повторение движения
    ...
    void Update () {
        ...
        if (moving) {
            float u = (Time.time-timeStart)/timeDuration;
            if (u>=1) {
                u=1;
                if (loopMove) {
                    timeStart = Time.time;
                } else {
                    moving = false; // эта строка теперь в ветке else
                }
            }

            // Скорректировать значение u, чтобы уместить его в диапазон от uMin
            // до uMax
            u = (1-u)*uMin + u*uMax;
            // ^ Ничего не напоминает? Здесь мы снова использовали линейную
            // интерполяцию!

            // Стандартная формула линейной интерполяции
```

```
            p01 = (1-u)*c0.position + u*c1.position;
            ...
        }
    }
}
```

Если теперь щелкнуть на кнопке PIay (Играть) в Unity и затем установить флажок checkToStart в настройках Cube01, вы получите то же поведение, что и прежде. В инспекторе попробуйте ввести в поле uMin число –1 и в поле uMax число 2 в разделе Cube01:Interpolator2 (Script). Установите флажок checkToStart, и вы увидите, что цвет, координаты и масштаб экстраполируются за границы начальных значений[1]. После этого можете также установить флажок loopMove и заставить процесс интерполяции повторяться снова и снова, до бесконечности.

Величина угла поворота не экстраполируется за границами `c0` и `c1` из-за ограничений метода `Quaternion.Slerp()` (от англ. *Spherical Linear interpolation* — сферическая линейная интерполяция), который используется в Unity для поворота объекта. Вместо того чтобы независимо интерполировать величины углов поворота относительно осей X, Y и Z, метод `Slerp` пытается выбрать самый прямой путь от одного угла поворота к другому. Однако если методу `Slerp()` передать в аргументе `u` любое число меньше 0, он будет интерпретировать его как 0 (а любое число больше 1 — как 1).

В документации с описанием класса `Vector3` говорится, что он также имеет метод `Lerp()` (от англ. *Linear interpolation* — линейная интерполяция), выполняющий интерполяцию между значениями типа `Vector3`, но я никогда не использую эту функцию, потому что она «втискивает» значения `u` в диапазон от 0 до 1 и не допускает возможность экстраполяции. В Unity 5 был добавлен метод `Vector3.LerpUnclamped()`, который не ограничивает аргумент `u` диапазоном 0...1. Я использую этот метод в своей практике, но все еще считаю, что для вас важнее научиться выполнять линейную интерполяцию вручную, именно поэтому я не использовал `Vector3.LerpUnclamped()` в примерах в этом разделе.

Сглаживание для линейной интерполяции

Виды интерполяции, применявшиеся до сих пор, создают приятные визуальные эффекты, но вызывают ощущение механистичности, потому что движение начинается резко, происходит с постоянной скоростью и затем резко прекращается. К счастью, существуют разнообразные функции сглаживания (easing functions), которые могут сделать эффект движения более интересным. Проще всего это объяснить на примере в Unity.

---

[1] В консоли может появиться предупреждение, что «коллайдеры BoxCollider не поддерживают отрицательный масштаб или размер». Пусть вас это не беспокоит. При экстраполяции масштаб может получиться отрицательным, но в данном примере нас не волнует проблема обнаружения столкновений.

Пример в Unity — интерполяция со сглаживанием

Чтобы создать пример, выполните следующие шаги.

1. Создайте новый сценарий на C# с именем Easing, откройте его в MonoDevelop и введите следующий код. Обратите внимание, что класс `Easing` не наследует `MonoBehaviour`.

```
using UnityEngine;

public class Easing {

    public enum Type {                                                    // a
        linear,
        easeIn,
        easeOut,
        easeInOut,
        sin,
        sinIn,
        sinOut
    }

    static public float Ease (float u, Type eType, float eMod = 2) {      // c
        float u2 = u;

        switch (eType) {                                                  // b

            case Type.linear:
                u2 = u;
                break;

            case Type.easeIn:
                u2 = Mathf.Pow(u, eMod);
                break;

            case Type.easeOut:
                u2 = 1 - Mathf.Pow( 1-u, eMod );
                break;

            case Type.easeInOut:
                if ( u <= 0.5f ) {
                    u2 = 0.5f * Mathf.Pow( u*2, eMod );
                } else {
                    u2 = 0.5f + 0.5f*( 1 - Mathf.Pow( 1-(2*(u-0.5f)), eMod ) );
                }
                break;

            case Type.sin:
                // Попробуйте ввести в поле eMod значения 0.15f и -0.2f
                //   для Easing.Type.sin                                  // c
                u2 = u + eMod * Mathf.Sin( 2*Mathf.PI*u );
                break;

            case Type.sinIn:
```

```
                // Для SinIn значение eMod игнорируется
                u2 = 1 - Mathf.Cos( u * Mathf.PI * 0.5f );
                break;

            case Type.sinOut:
                // Для SinOut значение eMod игнорируется
                u2 = Mathf.Sin( u * Mathf.PI * 0.5f );
                break;
        }

        return( u2 );
    }
}
```

a. Это перечисление определяет тип сглаживания, а переменные за пределами этого класса можно объявлять с типом `Easing.Type`. Внутри класса `Easing` можно просто использовать тип `Type`.

b. Эта инструкция `switch` реализует виды сглаживания, перечисленные в `Type`.

c. `eMod` — необязательный параметр типа `float` статической функции `Ease()`. Он используется как модификатор в некоторых типах сглаживания. Например, для сглаживания вида `easeIn` он используется как показатель степени, в которую возводится число `u`: например, если `eMod` = 2, тогда u2 = u^2; если `eMod` = 3, тогда u2 = u^3. Для сглаживания вида `sin` `eMod` используется как множитель амплитуды синусоиды, которой сглаживается линия (см. примеры на рис. Б.6).

Класс `Easing` хранит все функции сглаживания для `u`, что позволяет легко импортировать их в любые проекты. На рис. Б.6 показаны разные кривые сглаживания, кроме `sinIn` и `sinOut`, которые являются менее гибкими синусоидальными версиями `easeIn` и `easeOut`.

2. Сохраните сценарий Easing, откройте сценарий Interpolator2 и внесите следующие изменения.

```
public class Interpolator2 : MonoBehaviour {
    [Header("Set in Inspector")]
    ...
    public bool         loopMove = true; // Включает повторение движения
    public Easing.Type  easingType = Easing.Type.linear;
    public float        easingMod = 2;

    // Установите флажок checkToStart, чтобы запустить перемещение
    public bool         checkToStart = false;
    ...
    void Update () {
        ...
        if (moving) {
            ...
            // Скорректировать значение u, чтобы уместить его в диапазон от uMin
            // до uMax
            u = (1-u)*uMin + u*uMax;
```

```
            // ^ Ничего не напоминает? Здесь мы снова использовали линейную
            // интерполяцию!

            // Функция Easing.Ease изменяет u и влияет на характер движения
            u = Easing.Ease(u, easingType, easingMod);

            // Стандартная формула линейной интерполяции
            p01 = (1-u)*c0.position + u*c1.position;
            ...
        }
    }
}
```

3. Сохраните сценарий Interpolator2 и вернитесь в Unity.
4. В инспекторе, в разделе Cube01:Interpolator2 (Script), установите значения: uMin = 0 и uMax = 1. Также установите флажок loopMove.
5. Сохраните сцену!
6. Щелкните на кнопке Play (Играть) и установите флажок checkToStart. Теперь, так как флажок loopMove уже установлен, объект Cube01 будет снова и снова интерполироваться между объектами c0 и c1.

Попробуйте поиграть с настройками easingType. Значение в поле easingMod влияет только на типы сглаживания `easeIn`, `easeOut`, `easeInOut` и `sin`. Для типа `sin` попробуйте задать в поле easingMod значение 0.15 а потом –0.3, чтобы увидеть гибкость этого вида сглаживания на основе синусоиды.

На рис. Б.6 показано, как выглядят разные кривые сглаживания. Здесь горизонтальная ось соответствует начальным значениям `u`, а вертикальная — сглаженному значению `u` (`u2`). Как можно заметить в каждом примере, когда u = 0, u2 также равно 0, а когда u = 1, u2 также равно 1. Как результат, если линейная интерполяция осуществляется во времени, интерполяция между p0 и p1 всегда будет заканчиваться в течение одного и того же времени, независимо от настроек сглаживания.

На графике *Linear* показана исходная кривая (то есть прямая) без сглаживания (u2 = u). На остальных графиках изображен результат сглаживания исходной прямой u2 = u, которая показана пунктиром. Если какая-то часть кривой оказывается ниже пунктирной прямой, значит, объект, движущийся по закону, описываемому этой кривой, отстает от объекта, движущегося по линейному закону. Аналогично, если какая-то часть кривой оказывается выше пунктирной прямой, значит, движение по этому закону опережает линейное движение. Наклон кривой представляет скорость интерполяции в этой точке: наклон 45° соответствует линейной интерполяции, меньший угол означает меньшую скорость, а больший — большую.

Объект, движущийся по закону *EaseIn*, начинает движение с маленькой скоростью и затем ускоряется к концу (u2 = u*u). Такой вид сглаживания получил название «easing in» (с замедлением в начале), потому что первая фаза движения — «медленная», а последующие происходят с ускорением.

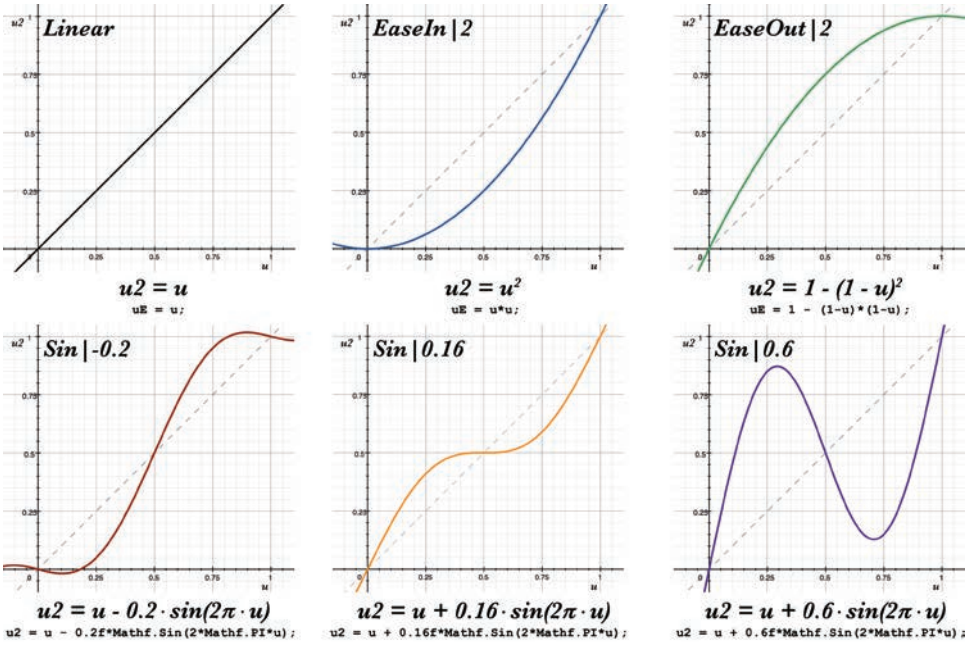

Рис. Б.6. Разные кривые сглаживания и соответствующие им формулы. Во всех случаях число после вертикальной черты (|) представляет значение eMod (то есть easingMod)

Объект, движущийся по закону *EaseOut*, напротив, начинает движение на высокой скорости, а потом замедляется к концу. Такой вид движения часто называют «easing out» (с замедлением в конце).

Три кривые *Sin* на нижних графиках соответствуют одной и той же формуле (u2 = u + eMod * sin(u*2π)), где eMod — вещественное число (переменные `eMod` и `easingMod` в коде). Произведение u * 2π внутри sin() гарантирует отображение диапазона значений 0...1 в u на полный цикл синусоиды (из центра, вверх, к центру, вниз и обратно к центру). При значении eMod=0 сглаживания исходной кривой не происходит (то есть прямая остается прямой). Но чем дальше значение eMod отклоняется от 0 (в положительную или отрицательную сторону), тем более выраженным получается эффект.

Кривая *Sin|-0.2* описывает движение с замедлением в начале и конце, с эффектом «подскока». Значение –0.2 в eMod добавляет в линейное движение отрицательную синусоиду, что заставляет движущийся объект немного отступить назад за p0, быстро переместиться к p1, уйти немного дальше и затем вернуться в p1. Более близкое к нулю значение eMod (например, Sin|–0.1) также обеспечит движение объекта с замедлением в начале и в конце, но сделает эффект «подскока» на концах менее выраженным, практически незаметным.

Кривая *Sin|0.16* добавляет в линейное движение пологую синусоиду. В этом случае движение начинается с высокой скоростью, замедляется до короткой остановки в середине пути и затем скорость снова увеличивается к концу. Объект, движущийся по этому закону, начнет движение сразу на большой скорости, замедлится к середине пути, приостановится «в раздумье» и затем с ускорением отправится к конечной точке.

Кривую *Sin|0.6* мы уже использовали для сглаживания движения Enemy_2 в главе 31 «Прототип 3.5: SPACE SHMUP PLUS». В этом случае в линейное движение добавляется ярко выраженная положительная синусоида, заставляющая объект «выстрелить» за центральную точку на пути к p1 примерно на 80 %, вернуться обратно в точку 20 % пути к p1 и, наконец, переместиться в точку p1.

Кривые Безье

Кривая Безье — это линейная интерполяция более чем по двум точкам. Как и при обычной линейной интерполяции, основой служит формула p01 = (1 − u) * p0 + + u * p1. Кривая Безье лишь добавляет дополнительные точки и вычисления.

Для случая с тремя точками: p0, p1 и p2

p01 = (1 − u) * p0 + u * p1

p12 = (1 − u) * p1 + u * p2

p012 = (1 − u) * p01 + u * p12

Как демонстрируют предыдущие уравнения, для трех исходных точек p0, p1 и p2, точка на кривой Безье вычисляется сначала применением линейной интерполяции между p0 и p1 (полученная точка называется p01), потом между p1 и p2 (полученная точка называется p12) и, наконец, между p01 и p12, в результате получается искомая точка p012. На рис. Б.7 показано графическое представление этой процедуры.

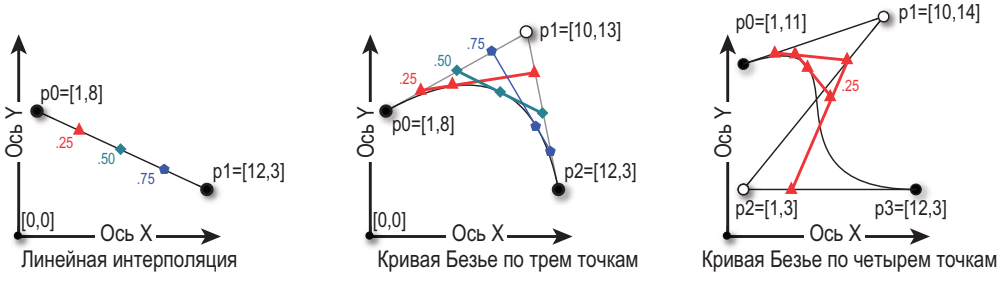

Рис. Б.7. Линейная интерполяция и кривые Безье, построенные по трем и четырем точкам

Для построения кривой Безье по четырем точкам требуется больше вычислений:

$p01 = (1 - u) \times p0 + u \times p1$

$p12 = (1 - u) \times p1 + u \times p2$

$p23 = (1 - u) \times p2 + u \times p3$

$p012 = (1 - u) \times p01 + u \times p12$

$p123 = (1 - u) \times p12 + u \times p23$

$p0123 = (1 - u) \times p012 + u \times p123$

Кривая Безье с четырьмя опорными точками используется во многих графических редакторах для определения управляемых кривых, в том числе в *Adobe Flash*, *Illustrator* и *Photoshop*, *OmniGraffle* компании The Omni Groups и многих других. Фактически редактор кривых в Unity для анимаций и толщины следа, оставляемого компонентом TrailRenderer, использует кривые Безье с четырьмя опорными точками.

Пример в Unity — кривые Безье

Выполните следующие шаги, чтобы создать в Unity пример, демонстрирующий использование кривых Безье. В коде я не использовал акцентированный символ *é* в слове *Bézier* (Безье), потому что такие символы не принято использовать в программном коде.

1. Создайте новую сцену в проекте Unity и сохраните ее с именем _Scene_Bezier.
2. Добавьте четыре куба в панель Hierarchy (Иерархия) с именами c0, c1, c2 и c3.
   a. Для всех четырех кубов настройте масштаб: S:[0.5, 0.5, 0.5].
   b. Разместите кубы в разных местах в сцене и скорректируйте настройки панели Scene (Сцена) так, чтобы все они были видимы.
3. Добавьте в сцену сферу.
   a. Подключите к сцене компонент TrailRenderer.
   b. Раскройте массив Materials в компоненте TrailRenderer и в поле Element 0 выберите встроенный материал Default-Particle.
4. Создайте новый сценарий на C# с именем Bezier и подключите его к объекту Sphere. Откройте сценарий в MonoDevelop и введите следующий код, демонстрирующий создание кривой Безье в Unity:

```
using UnityEngine;
using System.Collections;

public class Bezier : MonoBehaviour {
    [Header("Set in Inspector")]
    public float        timeDuration = 1;
    public Transform    c0, c1, c2, c3;
    // Установите флажок checkToStart, чтобы запустить перемещение
```

```
    public bool         checkToStart = false;

    [Header("Set Dynamically")]
    public float        u;
    public Vector3      p0123;
    public bool         moving = false;
    public float        timeStart;

    void Update () {
        if (checkToStart) {
            checkToStart = false;
            moving = true;
            timeStart = Time.time;
        }

        if (moving) {
            u = (Time.time-timeStart)/timeDuration;
            if (u>=1) {
                u=1;
                moving = false;
            }

            // Вычисление кривой Безье по четырем точкам
            Vector3 p01, p12, p23, p012, p123;

            p01 = (1-u)*c0.position + u*c1.position;
            p12 = (1-u)*c1.position + u*c2.position;
            p23 = (1-u)*c2.position + u*c3.position;

            p012 = (1-u)*p01 + u*p12;
            p123 = (1-u)*p12 + u*p23;

            p0123 = (1-u)*p012 + u*p123;

            transform.position = p0123;
        }
    }
}
```

5. Сохраните сценарий Bezier и вернитесь в Unity.
6. В каждом из четырех полей — c0, c1, c2 и c3 — компонента Sphere:Bezier (Script) выберите соответствующий куб.
7. Щелкните на кнопке Play (Играть) и затем установите флажок checkToStart в инспекторе.

Сфера начнет движение вдоль кривой Безье между четырьмя кубами. Важно отметить, что сфера коснется только кубов c0 и c3. Кубы c1 и c2 оказывают влияние на траекторию сферы, но она их не коснется. Это верно для всех кривых Безье. Концы кривой всегда касаются первой и последней точек, но не касаются промежуточных точек. Если вам нужна кривая, касающаяся промежуточных точек, поищите в интернете примеры «интерполяции сплайнами Эрмита» (а также другими видами сплайнов).

Рекурсивная функция вычисления кривой Безье

Как было показано в предыдущем разделе, дополнительные вычисления для учета большего количества опорных точек кривой Безье реализуются достаточно просто, но требуется некоторое время, чтобы ввести дополнительные строки кода. В примере, следующем ниже, используется рекурсивная функция, обрабатывающая произвольное число точек и избавляющая от необходимости писать дополнительный код. Концептуально рекурсивная реализация немного сложнее, поэтому сначала выясним, как она должна работать.

Для интерполяции стандартной кривой Безье с тремя опорными точками необходимы три точки: [p0, p1, p2]. Прежде всего весь диапазон интерполяции нужно разбить на два поддиапазона: [p0, p1] и [p1, p2]. В результате интерполяции внутри каждого из них получаются еще две точки: p01 и p12. В заключение производится интерполяция между p01 и p12 и получается окончательный результат — точка p012.

Функция `Bezier()` именно так и работает: она рекурсивно разбивает задачу на все меньшие и меньшие списки точек, пока каждая ветвь не получит список, включающий только одну точку, и затем возвращает эти точки вверх по цепочке рекурсивных вызовов, попутно выполняя интерполяцию.

В первом издании книги в каждом рекурсивном вызове функция `Bezier()` создавала новый список `List<Vector3>`, но это крайне неэффективное решение, потому что для создания каждого нового списка требуется много памяти и вычислительных ресурсов. Фактически в процессе интерполяции кривой Безье с четырьмя опорными точками функция `Bezier()` из первого издания создавала 14 дополнительных списков.

Чтобы не создавать массу новых списков, версия `Bezier()` из этого второго издания передает в каждый рекурсивный вызов ссылки на одни и те же списки вместе с двумя целыми числами — `iL` и `iR`, как можно видеть в объявлении функции `Bezier()`.

```
static public Vector3 Bezier(float u, List<Vector3> pts, int iL=0, int iR=-1) {…}
```

Целочисленные параметры `iL` и `iR` определяют индексы в списке `pts`, то есть `iL` и `iR` служат ссылками на элементы в `pts`. Если в параметре `iL` передается число 0, значит, он указывает на 0-й элемент списка `pts`. Если в параметре `iR` передается число 3, значит, он указывает на третий элемент списка `pts`. `iL` и `iR` — необязательные параметры, то есть функцию `Bezier()` можно вызвать только с двумя аргументами, `u` и `pts`, при этом `iL` получит значение 0, а `iR` — значение –1, которое будет заменено индексом последнего элемента в списке `pts`.

`iL` представляет самый левый элемент в списке `pts`, который должен рассматриваться в текущем рекурсивном вызове `Bezier()`, а `iR` представляет самый правый элемент. Таким образом, для списка `pts` с четырьмя элементами параметр `iL` получит начальное значение 0, а `iR` — значение 3 (индекс последней точки в списке `pts`). Каждый раз, выполняя рекурсивный вызов, функция `Bezier()` разбивает поддиапазон и посылает на следующий уровень меньшее количество точек. Вместо создания списков новая версия `Bezier()` подбирает значения `iL` и `iR` для анализа

укороченного фрагмента общего списка `pts`. В конечном счете каждая ветвь достигнет терминального случая, когда оба параметра, `iL` и `iR`, ссылаются на один и тот же элемент списка `pts`, и вернет значение этого элемента вверх по цепочке в виде значения `Vector3`. После этого с началом раскручивания цепочки рекурсивных вызовов начинается этап фактической интерполяции.

На рис. Б.8 показана последовательность рекурсивных вызовов, производимых единственным вызовом функции `Bezier()` для интерполяции кривой Безье по четырем опорным точкам. Зеленые стрелки показывают вызовы, а красные — возврат из вызовов. Вы можете видеть, как в каждом вызове `Bezier()` уменьшается или увеличивается значение `iL` или `iR` и какие элементы в списке `pts` соответствуют тому или другому диапазону от `iL` до `iR`.

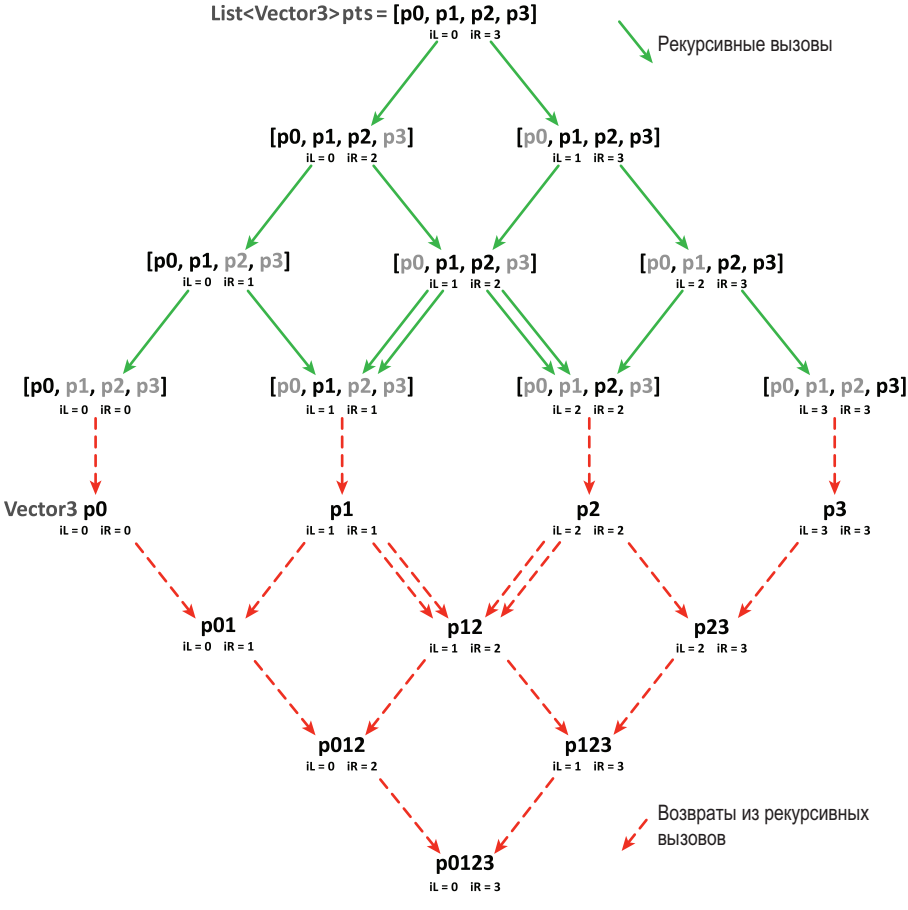

Рис. Б.8. Последовательность вызовов (сплошные зеленые стрелки) и возвратов (пунктирные красные стрелки) в процессе рекурсивной интерполяции кривой Безье по четырем точкам

Функция `Bezier()` производит два отдельных рекурсивных вызова для определения значения p12 (две двойные зеленые стрелки и две двойные красные стрелки, возвращающие его). Это несколько неэффективно, но для устранения такого перекрытия потребовался бы более сложный код.

В следующем листинге приводится реализация рекурсивной функции вычисления кривой Безье с любым количеством опорных точек. Эта версия включена в класс `ProtoTools` в сценарии Utils, который входит в состав импортируемого пакета для глав с 31-й по 35-ю:

```
static public Vector3 Bezier(float u, List<Vector3> pts,                    // a
                             int iL=0, int iR=-1)
{
    if (iR == -1) {                                                         // b
        iR = pts.Count-1;
    }

    if (iL == iR) { // Терминальный случай                                  // c
        return( pts [iL] );
    }

    // Два рекурсивных вызова, в каждый передается
    //   уменьшенное на 1 количество точек
    Vector3 lV3 = Bezier(u, pts, iL, iR-1);                                 // d
    Vector3 rV3 = Bezier(u, pts, iL+1, iR );                                // e

    // Интерполяция по точкам, полученным из вызовов в строках d и e
    Vector3 res = Vector3.LerpUnclamped( lV3, rV3, u );                     // f
    return( res );
}
```

a. Функция `Bezier()` принимает на входе число с плавающей точкой `u` и список точек `pts` для интерполяции. Кроме того, она имеет два необязательных параметра, `iL` и `iR`, которые представляют индексы самого левого (`iL`) и самого правого (`iR`) элементов в списке `pts`, которые должны рассматриваться в этом рекурсивном вызове `Bezier()`. Порядок выполнения рекурсивных вызовов показан на рис. Б.8.

b. Если параметр `iR` имеет значение по умолчанию –1, ему присваивается индекс последнего элемента в `pts`.

c. Терминальный случай рекурсии достигается, когда `iL == iR`. Если `iL == iR`, значит, оба индекса указывают на один и тот же элемент типа `Vector3` в списке `pts`. Когда это происходит, возвращается элемент, на который они указывают.

d. Это один из двух рекурсивных вызовов `Bezier()`. Здесь индекс `iR` уменьшается на 1, и в рекурсивный вызов передается полный список с индексами `iL` и `iR` в виде аргументов. Это равносильно созданию нового списка, содержащего все элементы из `pts`, кроме последнего, но работает намного эффективнее.

e. Это другой рекурсивный вызов `Bezier()`. Здесь индекс `iL` увеличивается на 1. Это равносильно передаче содержимого списка `pts` без первого элемента.

f. Результаты рекурсивных вызовов в строках *d* и *e* сохраняются в двух переменных — `lV3` и `rV3`. Затем по этим двум переменным выполняется интерполяция вызовом метода `Vector3.LerpUnclamped()`, и результат возвращается вверх по цепочке рекурсивных вызовов.

Сценарий Utils включает несколько перегруженных версий функции `Bezier()` для точек разных типов (например, `Vector3`, `Vector2`, `float` и `Quaternion`). В нем также имеются перегруженные версии, использующие ключевое слово `params`, чтобы дать возможность функции `Bezier()` передать опорные точки непосредственно, не в виде списка.

```
// Эта перегруженная версия Bezier() позволяет передать
//   массив или серию значений типа Vector3
static public Vector3 Bezier( float u, params Vector3[] vecs ) {           // g
    return( Bezier( u, new List<Vector3>(vecs) ) ); // Вызов Bezier() в строке a
}
```

g. Как рассказывалось в главе 24 «Функции и параметры», ключевое слово `params` разрешает передать массив параметров `vecs` либо как массив значений `Vector3`, либо как серию отдельных параметров типа `Vector3`, разделенных запятыми (после первого аргумента `float u`).

То есть для интерполяции кривой Безье по пяти точкам эту перегруженную версию можно вызвать двумя способами:

```
float u = 0.1f;
Vector3 p0, p1, p2, p3, p4;

Vector3[] points = new Vector3[] { p0, p1, p2, p3, p4 };

Utils.Bezier( u, points );                                                 // h
Utils.Bezier( u, p0, p1, p2, p3, p4 );                                     // i
```

h. Здесь массив `points` передается в параметре `vecs`, что не является неожиданностью.

i. В этой строке передается последовательность аргументов типа `Vector3` (то есть `p0, p1, p2, p3, p4`), которую ключевое слово `params` автоматически преобразует в массив элементов `Vector3` и присвоит параметру `vecs`.

Обе строки, *h* и *i*, вызовут перегруженную версию `Bezier()` с массивом `vecs` (объявлена с строке *g*). Затем эта перегруженная версия преобразует массив `vecs` в список `List<Vector3>` и вызовет версию `Bezier()` со списком `List<Vector3>` во втором аргументе, то есть оригинальную версию `Bezier()`, объявленную в строке *a*.

Ролевые игры

В мире существует много хороших ролевых игр (Role-Playing Game, RPG). Из них наибольшей, пожалуй, популярностью до сих пор пользуется *Dungeons & Dragons*

компании Wizards of the Coast (D&D), которая пережила уже пятое издание. Начиная с третьего издания, *D&D* основана на системе d20 с единственной игровой костью, имеющей двадцать граней, вместо множества сложных костей, использовавшихся в предыдущих системах. Мне многое нравится в *D&D*, но я заметил, что мои студенты часто испытывают труднопреодолимые сложности, начиная свою первую кампанию в системе *D&D*; она имеет очень специфические правила, особенно в четвертом издании.

Лично я советую в качестве первой системы RPG использовать *FATE* компании Evil Hat Productions, и особенно упрощенной ее версии *FATE Accelerated* (*FAE*). *FAE* — простая система, позволяющая игрокам вносить в сюжет больший вклад, чем другие системы. (Другие системы отдают всю власть над происходящим мастеру игры.) Познакомиться с базовой версией *FATE* можно на веб-сайте http://faterpg.com, бесплатный справочный документ с описанием системы FATE (System Reference Document, SRD) можно найти по адресу http://fate-srd.com. Дополнительную информацию о *FATE Accelerated* и бесплатную 50-страничную электронную книгу со всеми необходимыми начальными сведениями вы найдете по адресу http://www.evilhat.com/home/fae/.

Советы для начала хорошей ролевой кампании

Запуск кампании ролевой игры может поднять до небывалых высот ваши способности как дизайнера игр и рассказчика. Вот несколько советов, которые я даю своим студентам, когда они начинают свои кампании:

- **Начинайте с простого:** существует множество разных ролевых систем, и они сильно различаются сложностью своих правил. Как упоминалось в предыдущем разделе, на начальном этапе я советую использовать простую систему, такую как *FATE Accelerated* компании Evil Hat Productions. Сыграв несколько игр в этой системе, вы сможете перейти к более сложной, такой как *D&D*. В пятом издании *D&D* имеется относительно простой свод основных правил и множество дополнительных наборов правил, которые добавляются по мере погружения в систему.
- **Начинайте с короткого:** не старайтесь начать первый эпизод кампании, который, по вашим ожиданиям, займет целый год, попробуйте начать с простой миссии, которую можно завершить за одну ночь. Это даст вашей группе возможность почувствовать своих персонажей и систему и понять, нравятся ли они им. Если игрокам что-то не понравится, вы легко сможете это изменить, — гораздо важнее, чтобы игроки получили удовольствие от первого опыта ролевой игры, чем пытаться начать эпическую кампанию.
- **Помогайте игрокам на начальном этапе:** если игроки, принявшие участие в вашей кампании, имеют мало опыта в ролевых играх или вообще не имеют его, создайте персонажей для них сами. Это даст вам возможность снабдить их всеми дополнительными особенностями и организовать хорошую команду. Стандартная команда в ролевых играх включает следующих персонажей:

- воин, для противостояния врагам и боя на близкой дистанции (также известный как танк);
- маг, для боя на дальней дистанции и определения магических воздействий (также известный как стеклянная (хрустальная) пушка);
- вор, для обезвреживания ловушек и нападения исподтишка (также известный как бластер);
- священник, для выявления зла и исцеления других членов команды (также известный как контроллер).

Если вы решите сами создать персонажей для своих игроков, вы должны прежде побеседовать с ними и попросить рассказать вам, какой игровой опыт они хотели бы получить и какими способностями хотели бы наделить своего персонажа. Раннее участие и заинтересованность — одно из главных условий, которые помогут вашим игрокам избежать трудностей в начале кампании.

- **Планируйте импровизации:** ваши игроки часто будут делать что-то неожиданное для вас. Единственный способ запланировать это — подготовить себя к гибкости и импровизации. Подготовьте такие реквизиты, как обобщенные карты пространств, список имен для персонажей, не управляемых игроками, с которыми может столкнуться команда, и несколько обобщенных врагов или монстров, которых вы сможете вызвать. Чем тщательнее вы подготовитесь заранее, тем меньше времени будете тратить на исследование своих правил в середине игры.
- **Будьте готовы принимать решения:** если вы не найдете ответ в правилах за пять минут, просто примите решение на основе своих убеждений и договоритесь с игроками, что поищете ответ после окончания сеанса игры. Это предотвратит угасание игры из-за малопонятных правил.
- **Это также повествование игроков:** позволяйте игрокам отклоняться от проторенного пути. Если вы подготовили слишком узкий сценарий, у вас может появиться соблазн запретить им это, но тогда есть риск, что они потеряют интерес к игре.
- **Помните, что постоянное оптимальное напряжение вызывает разочарование:** в обсуждении потокового состояния в главе 8 «Цели проектирования» говорилось, что поддержание постоянного оптимального напряжения истощает игрока. Это также верно для ролевых игр. Бой с боссом всегда должен вызывать оптимальное напряжение. Но также должны быть бои, из которых игроки легко выходят победителями (это поможет им заметить, что их персонажи действительно становятся сильнее), а иногда и бои, когда игроки должны спасать свою жизнь бегством (обычно игроки не думают о таком исходе, и подобная ситуация может оказаться очень драматичной для них). В отличие от большинства систем, *FAE* имеет по-настоящему интригующую игровую механику, которая делает отступление и бегство лучшим выбором в сравнении с гибелью в битве, и это еще одна причина, почему она мне так нравится.

Эти советы помогут вам сделать свои ролевые кампании намного более увлекательными и для вас, и для ваших игроков.

Идеи по организации пользовательского интерфейса

В этом разделе обсуждается привязка кнопок игрового пульта компании Microsoft на машинах с Windows, macOS или Linux и рассказывается, как включить поддержку щелчка правой кнопкой в macOS.

Оси ввода и привязка кнопок для пультов Microsoft

Большинство игр, представленных в этой книге, используют интерфейс мыши или клавиатуры, но я полагаю, что рано или поздно у вас появится желание организовать управление своими играми с использованием игрового пульта. Самым простым, пожалуй, пультом, который способен работать с PC, macOS и Linux, является Microsoft Xbox 360 Controller для Windows, хотя я также видел людей, довольных пультом PS4 или Xbox One.

К сожалению, все платформы (PC, macOS и Linux) обрабатывают пульты поразному, поэтому вы должны настроить диспетчер ввода Unity InputManager, адаптировав его для работы с пультом на каждой платформе.

С другой стороны, можно избежать многих проблем и приобрести диспетчер ввода в онлайн-магазине Unity Asset Store. Некоторые из моих студентов использовали *InControl*, созданный в Gallant Games, который отображает ввод с пультов Microsoft, Sony, Logitech и Ouya в одни и те же коды в Unity. Просто поищите в Unity Asset Store по фразе «InControl» или перейдите по ссылке:

http://www.gallantgames.com/pages/incontrol-introduction

Для желающих самостоятельно настроить Unity InputManager на рис. Б.9 приводится информация со страницы сообщества Unify с описанием игрового пульта Xbox 360[1]. Числа на рисунке соответствуют номерам кнопок джойстика в разделе Axes, в окне InputManager. Оси обозначаются начальной буквой *a* (например, aX, a5). При использовании нескольких джойстиков на одной машине конкретный джойстик в InputManager Axes можно отличить по обозначению *joystick # button #* (например, «joystick 1 button 3»). Там же, на странице сообщества Unify, можно загрузить настройки диспетчера ввода InputManager для работы сразу с четырьмя пультами Microsoft.

[1] Эту страницу можно найти по адресу http://wiki.unity3d.com/index.php?title=Xbox360Controller.

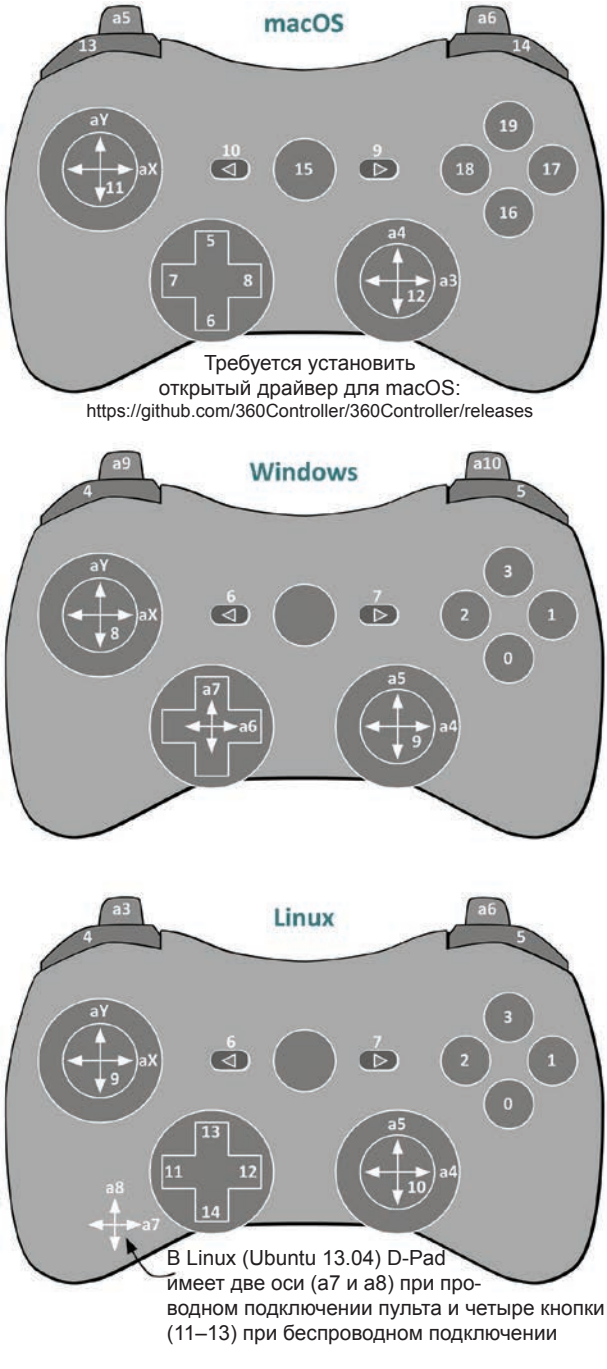

Рис. Б.9. Привязка кнопок на пульте Xbox в PC, macOS и Linux

В Windows драйвер пульта устанавливается автоматически. В Linux (Ubuntu 13.04 и выше) он также должен входить в комплект системы. В macOS вам придется загрузить драйвер из открытого проекта на GitHub: https://github.com/360Controller/360Controller/releases.

Щелчок правой кнопкой в macOS

На протяжении всей книги я предлагал вам время от времени щелкнуть правой кнопкой мыши на чем-нибудь. Однако многие пользователи Macintosh не знают, как выполнить щелчок правой кнопкой, потому что по умолчанию на мышках их трекпадов для macOS эта кнопка отсутствует. Вообще щелчок правой кнопкой можно выполнить несколькими способами, и выбор зависит от степени новизны вашего Mac и ваших предпочтений.

Control-Click = Right-Click

В левом нижнем углу всех современных клавиатур для macOS имеется клавиша Control. Если нажать эту клавишу и, удерживая ее нажатой, выполнить щелчок левой кнопкой мыши (обычный щелчок), macOS будет интерпретировать его как щелчок правой кнопкой.

Используйте любую мышь для PC

В macOS можно использовать любую мышь для PC с двумя или тремя кнопками. Лично я использую модель *MX Anywhere 2* компании Logitech или *Orochi* компании Razer.

Настройте поддержку правого щелчка в своей мыши для macOS

Пользующиеся мышью для macOS, произведенной в 2005 году или позже (Apple Mighty Mouse или Apple Magic Mouse), могут включить поддержку щелчка правой кнопкой так:

1. В меню Apple (пиктограмма с изображением яблока в верхнем левом углу экрана) выберите пункт System Preferences > Mouse (Системные настройки > Мышь).
2. В верхней части открывшегося диалога выберите вкладку Point & Click (Выбор и нажатие).
3. Установите флажок Secondary click (Имитация правой кнопки).
4. В раскрывающемся списке, непосредственно под подписью Secondary click (Имитация правой кнопки), выберите пункт Click on right side (Нажатие справа).

В результате нажатие слева будет интерпретироваться как щелчок левой кнопкой, а справа — правой.

Настройте имитацию щелчка правой кнопкой на трекпаде

По аналогии с Apple Mouse можно настроить трекпад любого ноутбука Apple (или Bluetooth Magic Trackpad) для поддержки щелчка правой кнопкой.

1. В меню Apple (пиктограмма с изображением яблока в верхнем левом углу экрана) выберите пункт System Preferences > Trackpad (Системные настройки > Трекпад).
2. В верхней части открывшегося диалога выберите вкладку Point & Click (Выбор и нажатие).
3. Установите флажок Secondary click (Имитация правой кнопки).
4. Если в раскрывающемся списке непосредственно под подписью Secondary click (Имитация правой кнопки) выбрать пункт Click or tap with two fingers (Нажатие двумя пальцами), касание одним пальцем будет интерпретироваться как щелчок левой кнопкой, а двумя — правой. Также возможны другие варианты имитации щелчка правой кнопки на трекпаде.

B Ссылки на интернет-ресурсы

Во многих онлайн-руководствах просто приводится список веб-сайтов, где можно найти дополнительную информацию, но я решил использовать это приложение, чтобы показать, где и как я сам ищу ответы на вопросы. Соответственно, это приложение включает не только несколько основных ссылок, но и описывает стратегии отслеживания информации и поиска новых ответов на вопросы по проблемам, с которыми вы можете столкнуться.

Я советую прочитать это приложение целиком (оно очень короткое), а потом вернуться к нему, когда вы столкнетесь с проблемой.

Учебные руководства для Unity

С течением времени создателями Unity было написано множество учебных руководств, которые могут вам пригодиться. В этой книге описываются короткие учебные примеры, цель которых — помочь вам понять, как программируется игровая механика, тогда как руководства, написанные создателями Unity, одинаково подробно описывают не только программирование, но и подходы к созданию графических ресурсов и анимаций, конструированию сцен и визуальных эффектов. Эта книга учит проектированию игр и созданию их прототипов, а руководства от создателей Unity описывают все разнообразие возможностей движка Unity.

Однако, просматривая эти руководства, имейте в виду, что многие из них написаны для более старых версий Unity и иногда не соответствуют новым версиям движка (то есть какие-то элементы интерфейса Unity или библиотеки кода могли измениться с того времени). Кроме того, в некоторых старых руководствах сценарии написаны на JavaScript, а не на C#. Это не должно быть большим препятствием для вас, особенно теперь, когда вы видели и разбирали код в этой книге, но вообще я советую искать руководства с примерами на C#, потому что они, как правило, более современные.

На веб-сайте Unity имеется раздел Learn (Обучение), цель которого — познакомить вас с Unity с помощью нескольких учебных руководств. Следующая ссылка приведет вас прямо на эту страницу. Выберите тему для изучения, и вашему вниманию будут предложены видеоуроки, которые помогут вам в этом:

- Раздел Learn — Tutorials (Обучение — Обучающие руководства): https://unity3d.com/ru/learn/tutorials

Архив конференции Unite

Получив некоторый опыт использования Unity, вам, возможно, будет интересно обратиться к другим источникам информации. Одним из великолепных источников являются видеозаписи с докладами на конференции Unite, сделанными за многие годы. Unity хранит множество таких видеозаписей, сделанных на конференциях Unite по всему миру. Вы найдете их по следующему адресу:

- https://unite.unity.com/archive

Помощь в программировании

По мере углубления в программирование для Unity вы заметите, что документация по программированию на C# для Unity сосредоточена в двух основных местах: в справочнике по программированию для Unity и в справочнике по языку C# компании Microsoft. Составители справочника по программированию для Unity проделали фантастическую работу по документированию особенностей, классов и компонентов, характерных для Unity, но в нем не рассматриваются базовые классы языка C# (такие, как `List<>`, `Dictionary<>` и др.). Их вы найдете в документации по C# компании Microsoft. В первую очередь я советую заглянуть в документацию по Unity, которая устанавливается вместе с движком на локальный компьютер, и если там вы не найдете ответа на свой вопрос, обращайтесь к документации Microsoft.

Справочник по программированию для Unity

Справочник по программированию для Unity включает:

- Онлайн-справочник: http://docs.unity3d.com/Documentation/ScriptReference/
- Локальный справочник: устанавливается вместе с Unity, выберите в меню пункт Help > Scripting Reference (Справка > Справочник по программированию). В результате откроется версия справочника, хранящаяся локально на вашем компьютере. Этот справочник будет доступен, даже если у вас нет подключения к интернету. (Я постоянно пользуюсь им в поездках.)

Справочник по C# компании Microsoft

Зайдите на поисковый сайт Bing.com и выполните поиск по фразе «Справочник по C# Microsoft». В первом же результате вы должны получить то, что хотели. На момент написания этих строк прямая ссылка на справочник выглядела так:

- http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx

Stack Overflow

Stack Overflow — это интернет-сообщество, где разработчики помогают разработчикам. Одни члены сообщества посылают вопросы, а другие отвечают на них. Для увеличения вовлеченности тем, кто дает лучшие ответы (по оценке других членов сообщества), присваиваются очки, из которых складывается авторитет на сайте:

- http://ru.stackoverflow.com

Часто, пытаясь понять, как сделать что-нибудь новое и необычное, я нахожу ответ на Stack Overflow. Например, если мне нужно узнать, как отсортировать список `List<>` средствами LINQ, я ввожу в Google фразу «C# LINQ sort list of objects» (C# LINQ сортировка списка объектов), и как раз когда я пишу эти строки, первые восемь ссылок в результатах ведут к ответам на Stack Overflow. Обычно я начинаю поиск ответов на вопросы в Google, а не на самом сайте Stack Overflow, но если в результатах присутствуют ссылки на stackoverflow.com, я первым делом обращаюсь туда, потому что хорошие ответы нахожу там чаще.

Узнайте больше о C#

Я настоятельно рекомендую две дополнительные книги о языке C#:

- **Для начинающих:** Rob Miles, «C# Programming Yellow Book», http://www.csharpcourse.com

  Роб Майлс (Rob Miles), преподаватель из Университета Халла, написал превосходную книгу о программировании на C# и довольно часто обновляет ее. Текущую версию вы найдете на его веб-сайте. Это очень остроумная, ясная и всесторонняя книга.

- **Как справочник:** «C# 4.0 Pocket Reference, 3rd Edition», http://shop.oreilly.com/product/0636920013365.do

  Даже при том, что уже вышли версии этого руководства для C# 5.0, C# 6.0 и C# 7.0, в Unity все еще используется C# 4.0 (или почти C# 4.0, кроме некоторых исключений), поэтому вам нужно именно это руководство. Когда у меня появляются вопросы по языку C#, первым делом я беру в руки этот справочник. Это усеченная версия книги «C# 4.0 in a Nutshell» издательства O'Reilly, но я считаю, что очень удобно иметь справочник карманного формата. Он также включает довольно много информации о LINQ.

Советы по поиску

Когда вам потребуется найти ответ на вопрос, касающийся языка C#, вставьте в начало фразы поиска название языка *C#*. Если выполнить поиск просто по слову *list* (список), вы получите массу результатов, никак не связанных с программированием. Но если добавить в начало поисковой фразы *C#*, вы сразу получите то, что искали.

Аналогично, если вопрос связан с Unity, обязательно добавьте слово *Unity* в начало фразы поиска.

Поиск ресурсов

В следующих разделах приводятся советы по поиску графических и аудиоресурсов.

Магазин Unity Asset Store

Доступ к онлайн-магазину Asset Store можно получить, открыв окно Asset Store в Unity (выберите в меню пункт Window > Asset Store (Окно > Asset Store)) или перейдя на веб-сайт магазина в обычном браузере. В Asset Store собрана гигантская коллекция моделей, анимационных эффектов, звуков, программного кода и даже законченных проектов Unity, которые можно загрузить. Большая часть ресурсов доступна за очень скромную плату, а некоторые даже бесплатно. Есть очень дорогие ресурсы, но часто они стоят таких денег, потому что помогают сэкономить сотни часов разработки:

- https://www.assetstore.unity3d.com/

Модели и анимации

Далее перечислены сайты, где можно найти трехмерные модели. Некоторые распространяются бесплатно, но большинство стоит денег. Также имейте в виду, что многие бесплатные модели разрешено использовать только в некоммерческих целях:

- **TurboSquid:** http://www.turbosquid.com/
- **Google 3D Warehouse:** http://3dwarehouse.sketchup.com/

  Имейте в виду, что почти все модели на сайте Google 3D Warehouse имеют формат SketchUp или Collada. В Unity имеется инструмент импортирования файлов в формате SketchUp, но на момент написания этих строк поддерживался только формат SketchUp 2015. Вам может потребоваться открыть файл формата SketchUp или Collada в программе SketchUp и сохранить его в формате SketchUp 2015 для последующего импортирования в Unity.

Шрифты

Почти все шрифты на следующих сайтах доступны бесплатно для некоммерческого использования, но решив задействовать их в коммерческих проектах, вы должны будете купить выбранные шрифты:

- http://www.1001fonts.com/
- http://www.1001freefonts.com/
- http://www.dafont.com/
- http://www.fontsquirrel.com/
- http://www.fontspace.com/

Другие инструменты и образовательные скидки

Студенты и преподаватели университетов имеют право на многочисленные скидки на программное обеспечение.

- **Adobe:** компания Adobe предлагает студентам годовую подписку на полный комплект своих инструментов Creative Cloud с приличной ежемесячной скидкой. В комплект входят Photoshop, Illustrator, Premier и другие инструменты.

  http://www.adobe.com/creativecloud/buy/students.html

- **Affinity:** компания Affinity предлагает Designer (конкурент Illustrator) и Photo (конкурент Photoshop) — очень добротные программы — всего за 40 долларов США каждую, причем в вечное пользование (в отличие от Adobe, которая требует оплаты за каждый месяц). Скидки студентам не предусмотрены, но эти инструменты и без того стоят относительно недорого и имеют достаточно высокое качество.

  http://affinity.serif.com/

- **AutoDesk:** компания AutoDesk дает студентам и преподавателям бесплатную 36-месячную лицензию практически на все свои инструменты, включая 3ds Max, Maya, Motionbuilder, Mudbox и многие другие.

  http://www.autodesk.com/education/free-software/featured

- **Blender:** Blender — это бесплатный и открытый инструмент для моделирования и создания анимаций. Он поддерживает многие возможности, которые можно найти в Maya и 3ds Max, но полностью бесплатный и может использоваться в коммерческих целях. Однако его интерфейс полностью отличается от интерфейса других программных продуктов для моделирования и анимации.

  http://www.blender.org/

- **SketchUp:** SketchUp — еще один инструмент для моделирования. Имеет очень понятный пользовательский интерфейс и часто обновляется. Базовая версия, SketchUp Make, распространяется бесплатно (правда, только для некоммерческого использования), а на SketchUp Pro учащимся и преподавателям предоставляется скидка. В последних версиях SketchUp появилась возможность экспортировать модели в форматах obj и fbx, которые легко импортируются в Unity, хотя лучшим, пожалуй, форматом для импортирования в Unity попрежнему остается SketchUp 2015. Если вы создали модель в SketchUp и решили экспортировать ее в формат obj или fbx, установите флажки Triangulate all faces (Триангулировать все поверхности), Export two-sided faces (Экспортировать двусторонние поверхности) и Swap YZ coordinates (Замена положения координат YZ) и в раскрывающемся списке Units (Единицы) выберите пункт Meters (Метры) в диалоге экспортирования (эти настройки не требуется выполнять при экспортировании в формат SketchUp 2015).

  http://www.sketchup.com/