C# Game Programming
Cookbook for Unity 3D
C# Game Programming Cookbook for Unity 3D

Second Edition

Jeff W. Murray
This book is dedicated to my amazing wife, Tori, and to my boys, Ethan and William. Boys, be nice to the cat and the cat will be nice to you!
Contents

Acknowledgments xiii
Introduction xv
Prerequisites xvii

I. Making Games in a Modular Way 1

1.1 Important Programming Concepts ......................... 1
  1.1.1 Manager and Controller Scripts ...................... 1
    1.1.1.1 Managers .................................. 1
    1.1.1.2 Controllers ................................. 1
    1.1.1.3 Communication between Classes ................. 2
    1.1.1.4 Public Static ............................... 2
    1.1.1.5 Private Static ................................ 3
  1.1.2 The Singleton ..................................... 3
    1.1.2.1 Using the Singleton in Unity ................ 3
  1.1.3 Inheritance ....................................... 3
  1.1.4 Coroutines ....................................... 5
  1.1.5 Namespaces ..................................... 6
  1.1.6 Finite State Machines (FSM) ....................... 7

1.2 Naming Conventions .................................... 7

1.3 Where to Now? ........................................ 8
## 2. Making a 2D Infinite Runner Game

2.1 Anatomy of a Unity Project .................................. 10
2.2 Open Unity, Then the Example Project for This Chapter .......... 10
   2.2.1 A Few Notes on the Example Project ...................... 12
   2.2.2 Sprites ........................................ 12
   2.2.3 Animation ....................................... 13
2.3 Open the Main Game Scene .................................. 14
2.4 The Game Manager ........................................ 15
2.5 Making Platforms to Run On .................................. 18
2.6 Building the Player, RunMan .................................. 22
   2.6.1 Adding the RunMan Sprite ......................... 22
   2.6.2 Adding Physics and Collisions to RunMan .......... 23
   2.6.3 Player Scripts .................................. 24
   2.6.4 RunMan Sounds .................................. 26
   2.6.5 Animating RunMan ................................ 27
   2.6.6 Scoring .......................................... 28
2.7 The Game Loop and the Game Manager ......................... 28
   2.7.1 Game States ...................................... 28
   2.7.2 The Main Loop of RunMan_GameManager.cs ........ 29
   2.7.3 Setting Up the RunMan_GameManager Component in the Editor ........ 35
2.8 The Demise of RunMan ...................................... 36
2.9 Adding the User Interface .................................... 37
   2.9.1 Main Menu Setup .................................. 47

## 3. Building the Core Game Framework

3.1 Controllers and Managers .................................... 50
   3.1.1 Managers ......................................... 50
   3.1.2 Controllers ....................................... 50
3.2 Core Framework Scripts ..................................... 50
   3.2.1 BaseGameManager ................................ 51
   3.2.2 ExtendedCustomMonoBehaviour ................ 56
   3.2.3 BaseCameraController ......................... 58

## 4. Player Structure

4.1 A Player Controller ......................................... 62
4.2 Dealing with Input ......................................... 63
4.3 User Data ............................................... 65
   4.3.1 The UserData Class ................................ 66
   4.3.2 The BaseUserManager Class ..................... 66
4.4 The BasePlayerStatsController Class ..................... 70
4.5 Managing Players ......................................... 73

## 5. Recipes: Common Components

5.1 Introduction .............................................. 75
5.2 AI ....................................................... 76
5.3 Camera .................................................... 76
  5.3.1 Third-Person Camera .................................. 77
  5.3.2 Top-Down Camera .................................... 80
5.4 Game Control ............................................ 80
  5.4.1 GlobalRaceManager ................................... 80
  5.4.2 RaceController ....................................... 80
5.5 Input ...................................................... 81
  5.5.1 Mouse Input ........................................... 81
5.6 Level Loading ............................................. 82
5.7 ScriptableObjects ......................................... 83
  5.7.1 ProfileScriptableObject ................................. 83
5.8 Spawning .................................................. 83
  5.8.1 A Spawner Component .................................. 84
  5.8.2 Trigger Spawning ..................................... 88
  5.8.3 Timed Spawning ....................................... 89
5.9 User Interface ............................................. 91
  5.9.1 CanvasManager ......................................... 91
  5.9.2 ScreenandAudioFader ................................ 91
  5.9.3 MenuWithProfiles .................................. 91
5.10 Utility ...................................................... 91
  5.10.1 AlignToGround ........................................ 91
  5.10.2 AutomaticDestroyObject .............................. 93
  5.10.3 AutoSpinObject .................................... 93
  5.10.4 FaceCamera .......................................... 94
  5.10.5 LookAtCamera ...................................... 94
  5.10.6 PretendFriction .................................... 94
  5.10.7 TimerClass ........................................ 96
    5.10.7.1 Modifying the Timer to Update automatically ....................................... 100
  5.10.8 WaypointsController ................................ 100
5.11 Weapons .................................................. 100

6. Player Movement Controllers .................................. 101
  6.1 Humanoid Character ...................................... 101
  6.2 Wheeled Vehicle ......................................... 108
  6.3 Wheel Alignment ......................................... 115
  6.4 2D Platform Character Controller ........................ 118

7. Weapon Systems ........................................... 123
  7.1 Building the Scripts ..................................... 123
    7.1.1 The BaseWeaponController Class .................... 124
    7.1.2 BaseWeaponScript .................................. 130
    7.1.3 Switching between Weapons ........................ 133

8. Waypoints Manager ........................................ 137
  8.1 Waypoint System ........................................ 137
9. Recipe: Sound and Audio

9.1 Audio Mixers and Mixing Audio in Unity ...................... 145
9.1.1 Mixing Audio ............................................. 145
9.1.2 Exposing Mixer Properties to Modify Them with Scripts . 146
9.1.3 Audio Effects ............................................. 147
9.2 The BaseSoundManager Class .................................. 147
9.3 Adding Sound to Weapons .................................... 153

10. AI Manager

10.1 AI States ......................................................... 156
10.2 The BaseAIController Class – Base AI Control .......... 156
10.3 Enemy Stats .................................................. 170
10.4 The AIBotController Class – Making an Enemy Bot Controller .... 171
10.5 The AISteeringController Class – An AI Steering Controller .... 177
10.6 Adding Weapon Control to the AI Controller .......... 181
  10.6.1 Attack States .......................................... 181
  10.6.2 Base Enemy Weapon Controller .................... 181

11. Menus and User Interface

11.1 Canvas Manager Class/Canvas Management .................. 187
11.2 The Main Menu .............................................. 193
  11.2.1 Designing the Main Menu ................................ 194
  11.2.2 The BaseMainMenuManager Script ...................... 195
  11.2.3 Combining BaseMainMenuManager to Use the ProfileManager Script for Saving and Loading Audio Settings ............................................. 198
11.3 In-Game User Interface ..................................... 200

12. Saving and Loading Profiles

12.1 The Save Profile ScriptableObject ............................. 201
12.2 The BaseProfileManager Class ............................... 203

13. Loading Level Scenes

13.1 A Script for Loading Levels: LevelLoader.cs .................. 212
13.2 Multiple Level Loading ....................................... 215
13.3 Building a Multi-Scene Level Loader in Unity ............. 216

14. Racing Game

14.1 Ingredients ..................................................... 221
14.2 Main Menu Scene Overview .................................. 223
14.3 Level Select Scene Overview ................................ 223
14.4 The Core Scene Overview ................................... 224
  14.4.1 Game Manager – The RaceGameManager Class ........ 224
  14.4.2 Race Control – The GlobalRaceManager Class ....... 238
  14.4.3 Race Control – The RaceController Class ............. 245
  14.4.4 Race Control – RacePlayerController Class .......... 253
15. Blaster Game Example

15.1 Ingredients .................................................. 264
15.2 Main Menu Scene Overview ................................. 265
15.3 The Core Scene Overview ................................... 265
   15.3.1 The BlasterGameManager Class .................... 265
   15.3.2 The BlasterPlayer Class ............................... 274
   15.3.3 Enemy Bots and the BlasterEnemyBot Class ......... 277
   15.3.4 Weapons and Projectiles ............................. 278
   15.3.5 User Interface ......................................... 278
15.4 Level Scenes .................................................. 280
   15.4.1 Spawning Enemies ..................................... 280

Index .......................... 283
Acknowledgments

Thanks to all of my family, friends and everyone who has helped me with my books, games and projects over the years: Brian Robbins, Chris Hanney, Technicat, Molegato, NoiseCrime, Quantum Sheep, Mickael Laszlo, Juanita Leatham, CoolPowers, Dave Brake, Sigurdur Gunnarsson, Kevin Godoy R., Lister, Wim Wouters, Edy (VehiclePhysics), IdeaFella, Liam Twose, RafaEnd, Lynxie, Alistair Murphy, McFunkypants, Dani Moss, Malcolm Evans, Vinh, Scott Wilson, Jon Keon, Dunky, Huck Z, Paradise Decay, Kryten, Silverware Games, Pete Patterson, Dr. Not Oculus VR, Say Mistage, Andy Hatch, Lydia, RT Warner, GermanRifter VR, Sylvain Demers, Kenneth Seward Jr, Dancer (Space Dad), Steve Fulton, Paul Bettner, Derek Scottishgeeks, Jonny Gorden, Edward Atkin, Ottawa Pete, Sam W., Dylan Stout, Shane McCafferty, Will Goldstone, Sanborn VR, Gary Riches, J. Dakota Powell, Mayonnaise Boy, Stephen Parkes, Ram Kanda, Alex Bethke, Itzik Goldman, Joachim Ante, Robert Scoble, Tony Walsh, Andreas ‘BOLL’ Aronsson, Cat, Darrel Plant, Mike Baker, Rimsy, Cassie, Christopher Brown, Phil Nolan, Pixel Hat Studio, Marcus Valles, Trev, Karyl, Tami Quiring, Nadeem Rasool, Dwayne Dibley, Liz and Pete Smyth, Isaac and Aiden, David Helgason, VR Martin, James Gamble, Vasanth Mohan, Simona Ioffe, Alexander Kondratskiy, Tim and Paul, The Oliver Twins, Jeevan Aurol, Rick King, Aldis Sipolins, Ric Lumb, Craig Taylor, Rob Hewson, Dani Moss, JayenKai (JNK), Matthew Kirubakaran, Elliot Mitchell, Ethan and William, Pablo Rojo, Paul Bettner, AdrellaDev, Gordon Little, Ryan Evans, Sasha Boersma, Matt Browning at Perfect Prototype, Hermit, Dirty Rectangles and the whole Ottawa game dev community.
I would also like to sincerely thank Anya Hastwell, Thivya Vasudevan and the team at SPI Global, the whole team at Routledge/CRC Press/AK Peters, including Randi Cohen, Jessica Vega and Rick Adams, for making this book a reality.

Thank you for buying this book and for wanting to do something as cool as to make games. I wish I could tell you how awesome it feels to know that someone else is reading this right now. I cannot wait to see your games and I sincerely hope this book helps you in your game making adventures. Have fun making games!
The overall goal of this book is to provide a library of C# code with which to jumpstart your projects and to help you with the overall structure of your games. Many development cookbooks focus on only providing snippets of code, but, here, we take a different approach. What you are holding in your hands right now (or on your screen) is a cookbook for game development that has a highly flexible core framework designed to speed up development of just about any type of Unity project.

You might think of the framework as a base soup and the scripting components as ingredients. We can mix and match script components and we can share the same core scripts in many of them. The framework takes care of the essentials and we add a little extra code to pull it all together the way we want it to work.

The framework is optional, however – you can use a lot the components individually. If you intend on using the components in this book for your own games, the framework could either serve as a base to build your games on or simply as a tutorial test bed for you to rip apart and see how it all works. Perhaps you can develop a better framework or maybe you already have a solid framework in place. If you do find a way to develop your own framework, I say do it. The key to game development is to do what works for you and your game projects – whatever it takes to cross the finish line.

I hope it helps you to make your games and tell your stories. I also hope you remember to try to have fun doing it!
Prerequisites

You can get up and running with the required software for the grand total of zero dollars. Everything you need can be downloaded free of charge with no catches. All you need is:

Unity Personal or Unity Pro version

C# programming knowledge.

This is *not* a book about learning how to program. You will need to know some C# and there are several other books out there for that purpose, even if I have tried to make the examples as simple as possible!

What this book doesn’t cover

This is not a book about learning to program from scratch. We assume that the reader has some experience of the C# programming language. Know that I am a self-taught programmer and I understand there may be better or alternative ways to do things. Techniques and concepts offered in this book are meant to provide foundation and ideation, not to be the final word on any subject.
1 Making Games in a Modular Way

1.1 Important Programming Concepts

There are some things you will need to know to utilize this book fully. For a lot of programmers, these concepts may be something you were taught in school. As a self-taught coder who did not find out about a lot of these things until long after I had been struggling on without them, I see this section as a compilation of ‘things I wish I’d known earlier’!

1.1.1 Manager and Controller Scripts

Sometimes, even I get confused about the difference between a manager and a controller script. There is no established protocol for this, and it varies depending on who you ask. Like some readers of the first edition of this book, I find it a little confusing and I will try to clear that up.

1.1.1.1 Managers

Managers deal with overall management, in a similar way to how a Manager would work in a workplace situation.

1.1.1.2 Controllers

Controllers deal with systems that the managers need to do their jobs. For example, in the racing game example game for this book, we have race controller scripts and a global race manager script. The race controller scripts are attached to the players and track their positions on the track, waypoints, and other relevant player-specific race information. The global race manager script talks to all
the race controller scripts attached to the players to determine who is winning and when the race starts or finishes.

1.1.1.3 Communication between Classes

An important part of our modular structure is how scripts are going to communicate with each other. It is inevitable that we will need to access our scripts from different areas of the game code, so you will need to know how to do that in Unity.

1. Direct referencing scripts via variables set in the editor by the Inspector window.

The easiest way to have script Components talk to each other (that is, scripts attached to GameObjects in the Scene as Components) is to have direct references, in the form of public variables within your code. They can then be populated in the Inspector window of the Unity editor with a direct link to another Component on another GameObject.

2. GameObject referencing using SendMessage.

SendMessage sends a message to a GameObject or Transform and calls a function within any of its attached script Components. It only works one way, so it is only useful when we do not need any kind of return result. For example:

```csharp
someGameObject.SendMessage("DoSomething");
```

Above, this would call the function DoSomething() on any of the script Components attached to the GameObject referenced by someGameObject.

3. Static variables.

The static variable type makes a variable accessible to other scripts without a direct reference to the GameObject it is attached to. This is particularly useful behavior where several different scripts may want to manipulate a variable to do things like adding points to a score or set the number of lives of a player, and so on.

An example declaration of a static variable might be:

```csharp
public static GameManager aManager;
```

You can have private or public static variables:

1.1.1.4 Public Static

A public static variable exists everywhere and may be accessed by any other script.

For example, imagine a player script which needs to tell the Game Manager to increase the current score by one:

1. In our GameManager.cs Game Manager script, we set up a variable, public static typed:

```csharp
public static gameScore;
```

2. In any other script, we can now access this static variable and alter the score as needed:

```csharp
GameController.gameScore++;
```
1.1.1.5 Private Static

A private static variable is accessible only to any other instances of the same type of class. Other types of class will not be able to access it.

As an example, imagine all players need to have a unique number when they are first spawned into the game. The player class declares a variable named uniqueNum like this:

```c
private static int uniqueNum;
```

When a player script is first created, it uses the value of uniqueNum for itself and increases uniqueNum by one:

```c
myUniqueNum = uniqueNum;
uniqueNum++;
```

The value of uniqueNum will be shared across all player scripts. The next player to be spawned will run the same start up function, getting uniqueNum again – only this time it has been increased by one, thanks to the player spawned earlier. This player again gets its own unique number and increases the static variable ready for the next one.

1.1.2 The Singleton

A singleton is a commonly used method for allowing access to an instance of a class, accessible to all other scripts. This pattern is ideal for code that needs to communicate with the entire game, such as a Game Manager.

1.1.2.1 Using the Singleton in Unity

The most straightforward way to achieve singleton-like behavior is to use a static variable and, when Unity calls the Awake() function, set the instance variable to become a reference to the script instance it represents. The keyword ‘this’ refers to the instance of the script it is a part of.

For example:

```c
public class MyScript : Monobehaviour
{
    private static MySingleton instance;
    void Awake()
    {
        if (instance != null)
            Destroy(this);
        instance = this;
    }
}
```

The singleton instance of our script can be accessed anywhere, by practically any script, with the following syntax:

```c
MySingleton.Instance.SomeFunctionInTheSingleton();
```

1.1.3 Inheritance

Inheritance is a complex concept, which calls for some explanation here because of its key role within the scripts provided by this book. Have a read through this section but don’t worry if you don’t pick up inheritance right away. Once we get to the programming it will probably become clearer.
The bottom line is that inheritance is used in programming to describe a method of providing template scripts that may be overridden, or added to, by other scripts. As a metaphor, imagine a car. All cars have four wheels and an engine. The types of wheels may vary from car to car, as will the engine, so when we say ‘this is a car’ and try to describe how our car behaves; we may also describe the engine and wheels.

These relationships may be shown in a hierarchical order:

Car–
  –Wheels
  –Engine

Now try to picture this as a C# script:

Car class.

Wheels function

Engine function

If we were building a game with lots of cars in it, having to rewrite the car class for each type of car would be silly. A far more efficient method might be to write a base class and populate it with virtual functions. When we need to create a car, rather than use this base class, we build a new class, which inherits the base class. Because our new class is inherited, it is optional whether we choose to override wheels or engine functions to make them behave in ways specific to our new class. That is, we can build ‘default’ functions into the base class and if we only need to use a default behavior for an engine, our new class doesn’t need to override the engine function.

A base class might look something like this:

```csharp
public class BaseCar : MonoBehaviour {
    public virtual void Engine () {
        Debug.Log("Vroom");
    }
    public virtual void Wheels () {
        Debug.Log("Four wheels");
    }
}
```

There are two key things to notice in the above script. One is the class declaration itself and the fact that this class derives from MonoBehaviour. MonoBehaviour is itself a class – the Unity documentation describes it as “the base class every script derives from” – this MonoBehaviour class contains many engine-specific functions and methods such as Start(), Update(), FixedUpdate(), and more. If our script didn’t derive from MonoBehaviour it would not inherit those functions and the engine wouldn’t automatically call functions like Update() for us to be able to work with. Another point to note is that MonoBehaviour is a class that is built into the engine and not something we can access to edit or change.

The second point to note is that our functions are both declared as virtual functions. Both are public, both are virtual. Making virtual functions means that the behavior in our base class may be overridden by any scripts that derive from it. The behavior we define in this base class could be thought of as its default behavior. We will cover overriding in full a little further on in this section.
Let's take a look at what this script actually does: if we were to call the Engine() function of our BaseCar class, it would write to the console “Vroom.” If we called Wheels, the console would read “Four wheels.”

Now that we have our BaseCar class, we can use this as a template and make new versions of it like this:

```csharp
public class OctoCar : BaseCar {
    public override void Wheels() {
        Debug.Log("Eight wheels");
    }
}
```

The first thing you may notice is that the OctoCar class derives from BaseCar rather than MonoBehaviour. This means that OctoCar inherits functions and methods belonging to our BaseCar script. As the functions described by BaseCar were virtual, they may be overridden. For OctoCar, we override Wheels with the line:

```csharp
public override void Wheels() {
    Debug.Log("Eight wheels");
}
```

Let's take a look at what this script actually does: In this case, if we were to call the Engine() function on OctoCar, it would do the same as the BaseCar class; it would write “Vroom” to the console. It would do this because we have inherited the function but have not overridden it, which means we keep that default behavior. In OctoCar, however, we have overridden the Wheels() function. The BaseCar behavior of Wheels would print “Four wheels” to the console but if we call Wheels() on OctoCar, the overridden behavior will write “Eight wheels,” instead.

Inheritance plays a huge part in how our core game framework is structured. The idea is that we have basic object types and specific elaborated versions of these objects inheriting the base methods, properties, and functions. By building our games in this manner, the communication between the different game components (such as game control scripts, weapon scripts, projectile controllers, etc.) becomes universal without having to write out the same function declarations over and over again for different variations of script. For the core framework, our main goal is to make it as flexible and extensible as possible and this would be much more difficult if we were unable to use inheritance.

### 1.1.4 Coroutines

Unity lets you run a function known as a coroutine, outside of the regular built in functions like Update, FixedUpdate, LateUpdate, and so on. A coroutine is self-contained code that will just go off and do its own thing once you have told it to run. As it is running on its own, you can do some cool stuff like pause it for a set amount of time and then have it start again, and coroutines are ideally suited to time-based actions like fade effects or animations.

Here is an example of a coroutine (this example code comes from the Unity engine documentation):

```csharp
IEnumerator Fade() {
    for (float ft = 1f; ft >= 0; ft -= 0.1f) {
        Color c = renderer.material.color;
        c.a = ft;
        renderer.material.color = c;
        yield return null;
    }
}
```
This code loops the value of ft from 1 to 0 and sets the alpha value of a color on a material to that value. This will fade out whatever renderer this script is attached to, fading more as the alpha channel gets closer to 0. Notice the strange line:

```csharp
yield return null;
```

Do not be alarmed by this odd code above! All it does is tell the engine that, at this point, the coroutine is ready to end this update and go on to the next one. You need this in your code, but don’t worry too much about why at this stage. I will go into some more detail below, but for now let us continue by looking at how you would start this coroutine running:

```csharp
StartCoroutine("Fade");
```

StartCoroutine() takes the name of the coroutine as a string. Once called to start, the coroutine will do its thing and you do not need to do anything else. It will either run until we tell it to stop, or it will run until the object is destroyed.

A key thing to note here is that the function is declared as type IEnumerator. At first, I found this confusing, but it turns out that Unity coroutines use C#’s built-in system for iterator blocks. IEnumerator (Iterator Enumerator) was intended to be a method for iterating through collections of objects. Coroutines instead use IEnumerator to iterate through your coroutine code, starting or stopping it between updates.

Another key point is that statement:

```csharp
yield return null;
```

You can think of IEnumerator like a cursor making its way through your code. When the cursor hits a yield statement, that tells the engine that this update is done and to move the cursor on to the next block.

Another thing you can do with coroutines is to pause the code inline for a certain amount of time, like this:

```csharp
yield return new WaitForSeconds(1.0f);
```

Using WaitForSeconds(), you can quite literally pause the coroutine script execution inline. For example:

```csharp
IEnumerator DebugStuff()
{
    Debug.Log("Start.");
    yield return new WaitForSeconds(1.0f);
    Debug.Log("Now, it's one second later!");
}
```

Earlier in this section, I mentioned that the yield statement is used to tell Unity the current update is done and to move on to the next. In the code above, rather than just telling Unity to move on to the next bit of code, it tells Unity to wait for the specified number of seconds first.

### 1.1.5 Namespaces

Namespaces compartmentalize chunks of code away from each other. You can think of a namespace as a box to put your own scripts in, so that they can stay separated from other code or code libraries. This helps to prevent possible
overlaps between code libraries (APIs and so on) and your own code, such as
duplicate function names or variables.

To use namespaces, you wrap the entire class in a namespace declaration
like this:

```csharp
namespace MyNamespace
{
    public class MyClass() : Monobehaviour
    {
    }
}
```

When you want to refer to the code from another class, you need to be either
wrapped in the same namespace or to have the using keyword at the top of the
script, like:

```csharp
using MyNamespace;
```

### 1.1.6 Finite State Machines (FSM)

A finite state machine is a commonly used system for tracking states. In video
games, they are used to track game states. Example states might be GameLoaded,
GameStarting, LevelStarting, and so on. The term *finite state machine* sounds a
lot more complicated than it usually is. The actual code often takes the form of a
long case statement that will execute code or call functions based on the current
game state. For example:

```csharp
Switch(currentGameState)
{
    Case GameStates.GameLoaded:
        GameLoaded();
        Break;
    Case GameStates.GameStarting:
        GameStart();
        Break;
}
```

Having code in each case statement can get unruly. To counter this, I like to
split all of the code out into individual functions. This keeps the code tidy, easier
to manage, and easier to debug when things do not go to plan.

### 1.2 Naming Conventions

Naming conventions vary from programmer to programmer and even between
companies. Some games companies have strict guidelines on how code is con-
structed and how to name variables, where they can and cannot be used, and so
on, which is mostly in place to make it easier for other programmers to be able to
understand the code and pick it up when they might need to. You may not need
to take it that far, but I do feel that some organization of your naming helps. In
this book, I follow a very loose set of rules for naming variables – just enough to
make it easier for me to know quickly what type of variable I am looking at or
what its purpose might be. Those are:

Object references (Transforms, Components, GameObjects, and so forth)
are prefixed by an underscore (_) symbol. For example, a variable to hold a
reference to a Transform might be named _TR. A variable used to talk to a Game Manager might be named _GameManager.

Most of the objects in this book (players, enemies, and more) derive from the ExtendedMonoBehaviour class, which contains a few variables that are used more commonly than others:

>_RB _ Used to reference a Rigidbody attached to this object.
>_RB2D For a 2D Rigidbody reference.
>_TR The Transform of the object.
>_GO The GameObject.

Where scripts need to be initialized before they can be used, in this book we always use a Boolean variable named didInit which gets set to true after initialization. You can use didInit to make sure initialization has completed.

Many programmers frown on the idea of declaring temporary variables, but I like to have _tempVEC and _tempTR variables available for whenever I need to refer to a quick Vector3 or another Transform inline. Having these variables already declared is just a little quicker than having to declare a new Vector3 each time, or to make a new Transform variable.

1.3 Where to Now?

Think about how your objects work together in the game world and how their scripts need to communicate with each other. I find it helps to make flow diagrams or to use mind-mapping software to work things out beforehand. A little planning early on can reduce a whole lot of code revisions later. That said, it is perfectly O.K. for your structure to grow and evolve as you develop it further. The framework shown in this book went through many iterations before reaching its final form.

Try to break up your scripts into smaller components to help debugging and flexibility. Also try to design for re-useability; hard-coding references to other scripts reduces the portability of the script and increases the amount of work you will need to do in the future when you want to carry it over into another project.
This chapter is different from the rest of the book and a dramatic diversion from what you may have seen in its first edition. Elsewhere in the text, I focus mainly on the code behind the games. This chapter offers up a step by step tutorial to using the framework to make a 2D infinite runner game. The goal of this chapter is to demonstrate two things: 1) How quickly you can turn around a game when you have a framework in place that takes care of a lot of the repetitive tasks. 2) The basics of how this book’s framework fits together to give you some background knowledge before we get down into nitty gritty of the framework code in Chapter 3.

This infinite runner (Figure 2.1) has a character that can move left, right, and jump. Platforms are spawned off-screen and moved to the left to create the illusion of movement.

I have already set up animations and imported the required graphics into the example project to get things going quickly. As the focus of this book is more toward code, project structure, and programming, I want to avoid using up too many pages on making graphics or on Unity editor-specifics. The Unity documentation features plenty of content on this.

Players jump and move to stay on the platforms if possible, with the score incremented at timed intervals. If the player falls off the bottom of the screen, the game ends.

To accomplish this, we will need:

1. The C# Game Programming Cookbook (GPC) framework, which is included as part of this book as a download from Github at https://github.com/psychicparrot/CSharpUnityCookbookExampleFiles
2. A platformer character capable of running left, right, and jumping
3. Platforms (auto moving to the left) and a method to spawn platforms off screen
2. Making a 2D Infinite Runner Game

4. A Game Manager script to keep track of game state and deal with scoring and so forth

5. Sounds for jumping and the game ending

6. An animated character

With just the framework and assets, by the end of this chapter, we will have a working game.

2.1 Anatomy of a Unity Project

Unity projects are made up of lots and lots of files. A game is made up of Scenes full of GameObjects. GameObjects are all the objects that make up a game. GameObjects may take the form of anything from an empty placeholder to a fully animated 3d model. Every GameObject has something called a Transform, which holds information on the rotation, position, and scale of the GameObject.

Scenes are essentially just files that hold information about which objects to load at runtime. They can contain GameObjects, or they can contain Prefabs. Prefabs are essentially files containing information about one or more GameObjects to load as though they are combined. By using Prefabs, you can reduce the time it takes to build multiple Scenes by combining multiple GameObjects into one, single,Prefab. Think of Prefabs as templates you make yourself by arranging GameObjects in the Scene and then creating Prefabs from them as templates to load and use elsewhere. When you drag a Prefab into a Unity Scene, it will appear as one or more GameObjects – replicating the original structure you created when you made the original Prefab. Since game projects are made up of many files, a lot of your time in Unity will be spent manipulating files, turning them into GameObjects, and Prefabs and then organizing and laying them out in Scenes. In the next section, we look at how Unity helps you to do this and how the editor is structured for exactly these kinds of tasks.

2.2 Open Unity, Then the Example Project for This Chapter

Open Unity. At the splash screen, choose to open an existing project and browse to the folder named Chapter2_UnComplete_ProjectFiles, which contains the example project for this chapter. Open the project. If you have any trouble along the way, or you just want to try out the completed game, you can find the

Figure 2.1 In the Infinite Runner example game, players jump and run to try to stay on the platforms for as long as possible.
To keep everything in line with the information in this section, you should change the editor layout. In the top right of the editor, find the Layout dropdown button. Click on it and choose 2 by 3. After a short delay, the editor should arrange its panels into the default 2 by 3 layout.

Now that we have the editor laid out in similar ways, let’s look through each panel to make sure we are using the same terminology. Check out Figure 2.2 – outlined below are the different sections shown in the figure, what those sections are and what they are for:

A. Scene
The Scene panel is your drag and drop visual window into a Unity Scene. You can manipulate GameObjects directly inside the currently loaded Scene.

B. Game
The Game panel shows a preview of the game. When you press Play, the editor runs the game inside the Game panel as if it were running in a standalone build. There are also a few extra features, such as Gizmos, that the Game preview offers to help you build out and debug your games.

C. Project
The Project panel is very similar to a file browser, in that it lists out all the files that go up to make your project. You can click and drag them, right click, and access a menu, use shortcuts, and delete them – almost all the regular functionality of a Windows Explorer window right inside the editor.

D. Hierarchy
The Hierarchy works in a similar way to the Project panel, only instead of being an interface to manipulate project files, it’s there to manipulate GameObjects inside the current Scene.
E. Inspector

Whenever you have a GameObject selected in the Scene panel, or highlighted in the Hierarchy, the Inspector window shows you editable properties of the GameObject and any Components attached to it. Whatever you can’t do with the Scene panel, you can probably do here instead.

Another point of interest in the main Unity editor are along the toolbar across the top of the editor, below the menus.

In the top left, find the Transform toolbar (Figure 2.3). This toolbar is to manipulate objects in the Scene panel. Tools, from left to right:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand Tool</td>
<td>Allows you to pan the view around the Scene.</td>
</tr>
<tr>
<td>Move</td>
<td>Selection and movement of GameObjects.</td>
</tr>
<tr>
<td>Rotate</td>
<td>Selection and rotation of GameObjects.</td>
</tr>
<tr>
<td>Scale</td>
<td>Selection and scaling of GameObjects.</td>
</tr>
<tr>
<td>Rect Transform</td>
<td>Modify a GameObjects boundaries as a Rect (combined scaling and movement).</td>
</tr>
<tr>
<td>Transform</td>
<td>A method to manipulate scale, position, and rotation in a single, combined tool.</td>
</tr>
<tr>
<td>Custom</td>
<td>This tool allows access to custom Editor tools and will not be covered in this book.</td>
</tr>
</tbody>
</table>

At the center of the toolbar, find the Play, Pause and Step buttons (Figure 2.4) which are used to preview your game and to control what happens in the Game panel.

2.2.1 A Few Notes on the Example Project

Unity uses something called packages to extend the functionality of the editor in an officially supported and controlled way. To facilitate editing and set up of the 2D (two dimensional) sprites in this project, a package named 2D Sprite Package had to be imported. Accessing packages is done via the Package Manager – found in the menu Window/Package Manager. Check in Package Manager regularly, for updates to any libraries you download.

2.2.2 Sprites

2D sprites may be found in the projects Assets/Games/RunMan/Sprites folder. The sprites were made with a sprite editing program called Aesprite and exported as sprite sheets. Sprite sheets are groups of sprites which are split up into individual sprites by the engine. We tend to use sprite sheets for animations, exporting multiple frames of animation in a single image to save memory and having...
the engine take care of cutting them up to display. The import settings for all of
the game images are set to their default values, the Sprite Mode is set to Multiple
to accommodate the sprite sheets (Multiple allows you to slice up those images)
and the Filter Mode has been set to Point (no filter) so that Unity does not try to
filter images to make them smooth. For this type of visual style, we want the
pixels to be defined and visible without filtering. The only other deviation from
the defaults is the Compression level, which is set to None just to be sure to avoid
any artifacts or anomalies that might stem from compressing small images.

2.2.3 Animation

Animation is a complicated subject and outside the scope of this book, but this
section acts as a quick guide on how the animations for this game works.

Look out for RunMan_Animator in the Sprites folder. You can open it to view
by double clicking on the RunMan_Animator file in the Project pane. An Animator
contains a state machine graph (Figure 2.5) which controls the flow of animation.
When an animated GameObject is initialized, the state machine graph begins in
its Entry state and will transition to the default state. In Figure 2.5, you can see that
RunMan_Animator has a default state of RunMan_Idle (that is, an arrow goes
from Entry to RunMan_Idle) – this will play an idle animation for whenever
RunMan is standing still. The animation is looped and there is no further flow (no
more arrows pointing to boxes!) out from RunMan_Idle in the Animator graph;
so, until told otherwise, the animation will stay in the idle state.

In the graph (Figure 2.5) look for the Any State state – this literally means any
state in that this state can be triggered from any animation. The Any State has two
arrows coming off it; one pointing to RunMan_Jump state and the other to a
RunMan_Run state. So, how do we get the animation to change to run or jump
when the Animator graph has no link from the endlessly looping RunMan_Idle
state? Parameters are the answer! Parameters are variables that can be set and
modified through code and used as Conditions by the Animator graph to control
flow. In the case of Any State, whenever a Jump parameter is triggered the Animator
will transition from Any State (that is, whatever state the graph is currently in) to
RunMan_Jump. Whenever a Run parameter is triggered, the Animator will transi-
tion from Any State to RunMan_Run. RunMan_Run is a looping animation that,
again, will play until the Animator graph is told to do otherwise.

In conclusion, the animation system for this game is set up to be as straight-
forward as possible. The Animator can be controlled by two parameters: Jump
and Run. In our code, to transition to a run animation we just set the Run param-
eter. To play a jump animation, we set the Jump parameter.

Figure 2.5 The Animator graph for the RunMan character.