SAMPLE CHAPTER

Multiplatform game development in C#

Unity IN ACTION

Covers Unity 5

Joseph Hocking

FOREWORD BY Jesse Schell

MANNING

Unity in Action

Multiplatform game development in C#

by Joseph Hocking

Chapter 1

brief contents

Part 1

First steps

It's time to take your first steps in using Unity. If you don't know anything about Unity, that's okay! I'm going to start by explaining what Unity *is*, including fundamentals of how to program games in it. Then we'll walk through a tutorial about developing a simple game in Unity. This first project will teach you a number of specific game development techniques as well as give you a good overview of how the process works.

Onward to chapter 1!

1 Getting to know Unity

This chapter covers

- What makes Unity a great choice
- Operating the Unity editor
- Programming in Unity
- Comparing C# and JavaScript

If you're anything like me, you've had developing a video game on your mind for a long time. But it's a big jump from simply playing games to actually making them. Numerous game development tools have appeared over the years, and we're going to discuss one of the most recent and most powerful of these tools. Unity is a professional-quality game engine used to create video games targeting a variety of platforms. Not only is it a professional development tool used daily by thousands of seasoned game developers, it's also one of the most accessible modern tools for novice game developers. Until recently, a newcomer to game development (especially 3D games) would face lots of imposing barriers right from the start, but Unity makes it easy to start learning these skills.

Because you're reading this book, chances are you're curious about computer technology and you've either developed games with other tools or built other kinds of software, like desktop applications or websites. Creating a video game isn't fundamentally different from writing any other kind of software; it's mostly a difference of degree. For example, a video game is a lot more interactive than most websites and thus involves very different sorts of code, but the skills and processes involved in creating both are similar. If you've already cleared the first hurdle on your path to learning game development, having learned the fundamentals of programming software, then your next step is to pick up some game development tools and translate that programming knowledge into the realm of gaming. Unity is a great choice of game development environment to work with.

A warning about terminology

This book is about programming in Unity and is therefore primarily of interest to coders. Although many other resources discuss other aspects of game development and Unity, this is a book where programming takes front and center.

Incidentally, note that the word *developer* has a possibly unfamiliar meaning in the context of game development: *developer* is a synonym for *programmer* in disciplines like web development, but in game development the word *developer* refers to anyone who works on a game, with *programmer* being a specific role within that. Other kinds of game developers are artists and designers, but this book will focus on programming.

To start, go to the website www.unity3d.com to download the software. This book uses Unity 5.0, which is the latest version as of this writing. The URL is a leftover from Unity's original focus on 3D games; support for 3D games remains strong, but Unity works great for 2D games as well. Meanwhile, although advanced features are available in paid versions, the base version is completely free. Everything in this book works in the free version and doesn't require Unity Pro; the differences between those versions are in advanced features (that are beyond the scope of this book) and commercial licensing terms.

1.1 Why is Unity so great?

Let's take a closer look at that description from the beginning of the chapter: Unity is a professional-quality game engine used to create video games targeting a variety of platforms. That is a fairly straightforward answer to the straightforward question "What is Unity?" However, what exactly does that answer mean, and why is Unity so great?

1.1.1 Unity's strengths and advantages

A game engine provides a plethora of features that are useful across many different games, so a game implemented using that engine gets all those features while adding custom art assets and gameplay code specific to that game. Unity has physics simulation, normal maps, screen space ambient occlusion (SSAO), dynamic shadows…and the list goes on. Many game engines boast such features, but Unity has two main advantages over other similarly cutting-edge game development tools: an extremely productive visual workflow, and a high degree of cross-platform support.

The visual workflow is a fairly unique design, different from most other game development environments. Whereas other game development tools are often a complicated mishmash of disparate parts that must be wrangled, or perhaps a programming library that requires you to set up your own integrated development environment (IDE), build-chain and whatnot, the development workflow in Unity is anchored by a sophisticated visual editor. The editor is used to lay out the scenes in your game and to tie together art assets and code into interactive objects. The beauty of this editor is that it enables professional-quality games to be built quickly and efficiently, giving developers tools to be incredibly productive while still using an extensive list of the latest technologies in video gaming.

> **NOTE** Most other game development tools that have a central visual editor are also saddled with limited and inflexible scripting support, but Unity doesn't suffer from that disadvantage. Although everything created for Unity ultimately goes through the visual editor, this core interface involves a lot of linking projects to custom code that runs in Unity's game engine. That's not unlike linking in classes in the project settings for an IDE like Visual Studio or Eclipse. Experienced programmers shouldn't dismiss this development environment, mistaking it for some click-together game creator with limited programming capability!

The editor is especially helpful for doing rapid iteration, honing the game through cycles of prototyping and testing. You can adjust objects in the editor and move things around even while the game is running. Plus, Unity allows you to customize the editor itself by writing scripts that add new features and menus to the interface.

Besides the editor's significant productivity advantages, the other main strength of Unity's toolset is a high degree of cross-platform support. Not only is Unity multiplatform in terms of the deployment targets (you can deploy to the PC, web, mobile, or consoles), but it's multiplatform in terms of the development tools (you can develop the game on Windows or Mac OS). This platform-agnostic nature is largely because Unity started as Mac-only software and was later ported to Windows. The first version launched in 2005, but now Unity is up to its fifth major version (with lots of minor updates released frequently). Initially, Unity supported only Mac for both developing and deployment, but within a few months Unity had been updated to work on Windows as well. Successive versions gradually added more deployment platforms, such as a cross-platform web player in 2006, iPhone in 2008, Android in 2010, and even game consoles like Xbox and PlayStation. Most recently they've added deployment to WebGL, the new framework for 3D graphics in web browsers. Few game engines support as many deployment targets as Unity, and none make deploying to multiple platforms so simple.

Meanwhile, in addition to these main strengths, a third and subtler benefit comes from the modular component system used to construct game objects. In a component

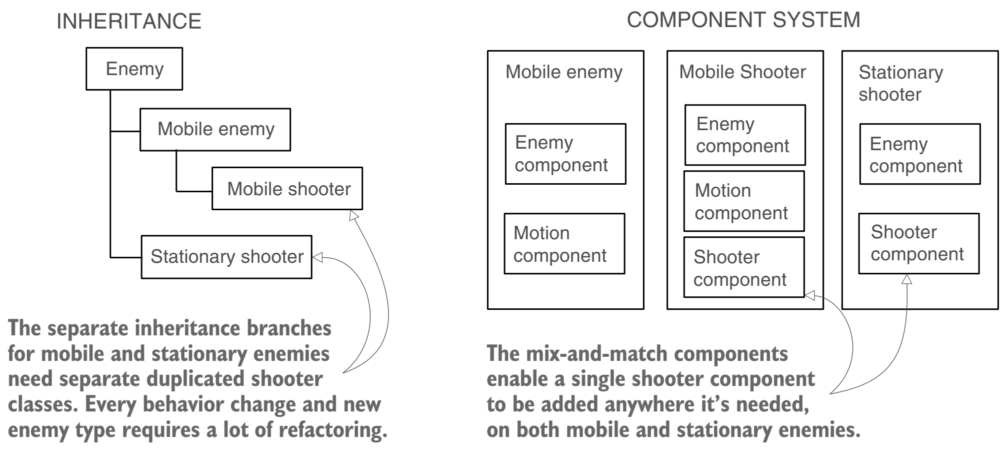

INHERITANCE
Enemy
Mobile enemy
Mobile shooter
Stationary shooter
The separate inheritance branches for mobile and stationary enemies need separate duplicated shooter classes. Every behavior change and new enemy type requires a lot of refactoring.
COMPONENT SYSTEM
Mobile enemy
Enemy component
Motion component
Mobile Shooter
Enemy component
Motion component
Shooter component
Stationary shooter
Enemy component
Shooter component
The mix-and-match components enable a single shooter component to be added anywhere it's needed, on both mobile and stationary enemies.

Figure 1.1 Inheritance vs. components

system, "components" are mix-and-match packets of functionality, and objects are built up as a collection of components, rather than as a strict hierarchy of classes. In other words, a component system is a different (and usually more flexible) approach to doing object-oriented programming, where game objects are constructed through composition rather than inheritance. Figure 1.1 diagrams an example comparison.

In a component system, objects exist on a flat hierarchy and different objects have different collections of components, rather than an inheritance structure where different objects are on completely different branches of the tree. This arrangement facilitates rapid prototyping, because you can quickly mix-and-match different components rather than having to refactor the inheritance chain when the objects change.

Although you could write code to implement a custom component system if one didn't exist, Unity already has a robust component system, and this system is even integrated seamlessly with the visual editor. Rather than only being able to manipulate components in code, you can attach and detach components within the visual editor. Meanwhile, you aren't limited to only building objects through composition; you still have the option of using inheritance in your code, including all the best-practice design patterns that have emerged based on inheritance.

1.1.2 Downsides to be aware of

Unity has many advantages that make it a great choice for developing games and I highly recommend it, but I'd be remiss if I didn't mention its weaknesses. In particular, the combination of the visual editor and sophisticated coding, though very effective with Unity's component system, is unusual and can create difficulties. In complex scenes, you can lose track of which objects in the scene have specific components attached. Unity does provide search functionality for finding attached scripts, but that search could be more robust; sometimes you still encounter situations where you need to manually inspect everything in the scene in order to find script linkages. This doesn't happen often, but when it does happen it can be tedious.

Another disadvantage that can be surprising and frustrating for experienced programmers is that Unity doesn't support linking in external code libraries. The many libraries available must be manually copied into every project where they'll be used, as opposed to referencing one central shared location. The lack of a central location for libraries can make it awkward to share functionality between multiple projects. This disadvantage can be worked around through clever use of version control systems, but Unity doesn't support this functionality out of the box.

NOTE Difficulty working with version control systems (such as Subversion, Git, and Mercurial) used to be a significant weakness, but more recent versions of Unity work just fine. You may find out-of-date resources telling you that Unity doesn't work with version control, but newer resources will describe.meta files (the mechanism Unity introduced for working with version-control systems) and which folders in the project do or don't need to be put in the repository. To start out with, read this page in the documentation: http://docs.unity3d.com/Manual/ExternalVersionControlSystemSupport.html

A third weakness has to do with working with prefabs. Prefabs are a concept specific to Unity and are explained in chapter 3; for now, all you need to know is that prefabs are a flexible approach to visually defining interactive objects. The concept of prefabs is both powerful and unique to Unity (and yes, it's tied into Unity's component system), but it can be surprisingly awkward to edit prefabs. Considering prefabs are such a useful and central part of working with Unity, I hope that future versions improve the workflow for editing prefabs.

1.1.3 Example games built with Unity

You've heard about the pros and cons of Unity, but you might still need convincing that the development tools in Unity can give first-rate results. Visit the Unity gallery at http://unity3d.com/showcase/gallery to see a constantly updated list of hundreds of games and simulations developed using Unity. This section explores just a handful of games showcasing a number of genres and deployment platforms.

DESKTOP (WINDOWS, MAC, LINUX)

Because the editor runs on the same platform, deployment to Windows or Mac is often the most straightforward target platform. Here are a couple of examples of desktop games in different genres:

- Guns of Icarus Online (figure 1.2), a first-person shooter developed by Muse Games

Figure 1.2 Guns of Icarus Online

- Gone Home (figure 1.3), an exploration adventure developed by The Fullbright Company

MOBILE (IOS, ANDROID)

Unity can also deploy games to mobile platforms like iOS (iPhones and iPads) and Android (phones and tablets). Here are a few examples of mobile games in different genres:

- Dead Trigger (figure 1.4), a first-person shooter developed by Madfinger Games
- Bad Piggies (figure 1.5), a physics puzzle game developed by Rovio
- Tyrant Unleashed (figure 1.6), a collectible card game developed by Synapse Games

CONSOLE (PLAYSTATION, XBOX, WII)

Unity can even deploy to game consoles, although the developer must obtain licensing from Sony, Microsoft, or Nintendo. Because of this requirement and Unity's easy cross-platform deployment, console games are often available on desktop computers as well. Here are a couple examples of console games in different genres:

- Assault Android Cactus (figure 1.7), an arcade shooter developed by Witch Beam
- The Golf Club (figure 1.8), a sports simulation developed by HB Studios

As you can see from these examples, Unity's strengths definitely can translate into commercial-quality games. But even with Unity's significant advantages over other game development tools, newcomers may have a misunderstanding about the involvement of programming in the development process. Unity

Figure 1.3 Gone Home

Figure 1.4 Dead Trigger

Figure 1.5 Bad Piggies

Figure 1.6 Tyrant Unleashed

is often portrayed as simply a list of features with no programming required, which is a misleading view that won't teach people what they need to know in order to produce commercial titles. Though it's true that you can click together a fairly elaborate prototype using preexisting components even without a programmer involved (which is itself a pretty big feat), rigorous programming is required to move beyond an interesting prototype to a polished game for release.

Figure 1.7 Assault Android Cactus

Figure 1.8 The Golf Club

1.2 How to use Unity

The previous section talked a lot about the productivity benefits from Unity's visual editor, so let's go over what the interface looks like and how it operates. If you haven't done so already, download the program from www.unity3d.com and install it on your computer (be sure to include "Example Project" if that's unchecked in the installer). After you install it, launch Unity to start exploring the interface.

You probably want an example to look at, so open the included example project; a new installation should open the example project automatically, but you can also select File > Open Project to open it manually. The example project is installed in the shared user directory, which is something like C:\Users\Public\Documents\Unity Projects\ on Windows, or Users/Shared/Unity/ on Mac OS. You may also need to open the example scene, so double-click the Car scene file (highlighted in figure 1.9; scene files have the Unity cube icon) that's found by going to SampleScenes/Scenes/ in the file browser at the bottom of the editor. You should be looking at a screen similar to figure 1.9.

The interface in Unity is split up into different sections: the Scene tab, the Game tab, the Toolbar, the Hierarchy tab, the Inspector, the Project tab, and the Console tab. Each section has a different purpose but all are crucial for the game-building lifecycle:

- You can browse through all the files in the Project tab.
- You can place objects in the 3D scene being viewed using the Scene tab.
- The Toolbar has controls for working with the scene.
- You can drag and drop object relationships in the Hierarchy tab.
- The Inspector lists information about selected objects, including linked code.
- You can test playing in Game view while watching error output in the Console tab.

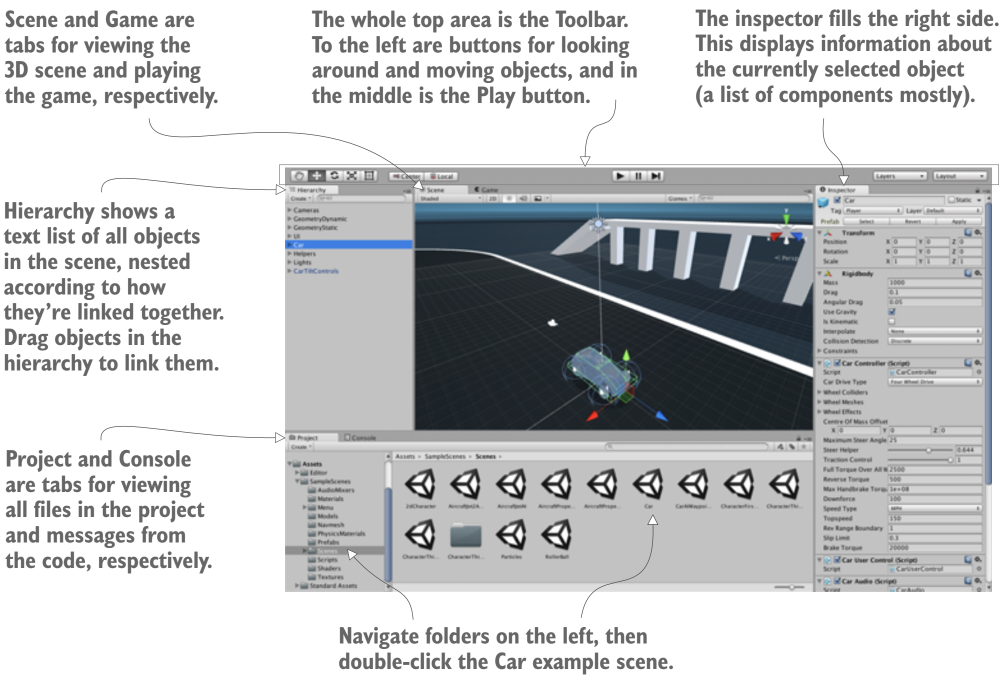

Scene and Game are tabs for viewing the 3D scene and playing the game, respectively.
The whole top area is the Toolbar. To the left are buttons for looking around and moving objects, and in the middle is the Play button.
The inspector fills the right side. This displays information about the currently selected object (a list of components mostly).
Hierarchy shows a text list of all objects in the scene, nested according to how they're linked together. Drag objects in the hierarchy to link them.
Project and Console are tabs for viewing all files in the project and messages from the code, respectively.
Navigate folders on the left, then double-click the Car example scene.

Figure 1.9 Parts of the interface in Unity

This is just the default layout in Unity; all of the various views are in tabs and can be moved around or resized, docking in different places on the screen. Later you can play around with customizing the layout, but for now the default layout is the best way to understand what all the views do.

1.2.1 Scene view, Game view, and the Toolbar

The most prominent part of the interface is the Scene view in the middle. This is where you can see what the game world looks like and move objects around. Mesh objects in the scene appear as, well, the mesh object (defined in a moment). You can also see a number of other objects in the scene, represented by various icons and colored lines: cameras, lights, audio sources, collision regions, and so forth. Note that the view you're seeing here isn't the same as the view in the running game—you're able to look around the scene at will without being constrained to the game's view.

DEFINITION A *mesh object* is a visual object in 3D space. Visuals in 3D are constructed out of lots of connected lines and shapes; hence the word *mesh*.

The Game view isn't a separate part of the screen but rather another tab located right next to Scene (look for tabs at the top left of views). A couple of places in the interface have multiple tabs like this; if you click a different tab, the view is replaced by the new

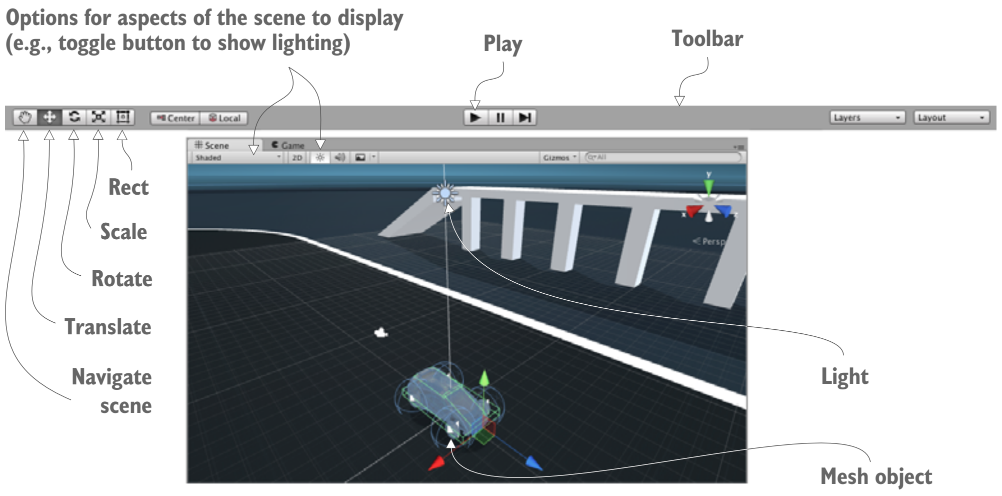


Figure 1.10 Editor screenshot cropped to show Toolbar, Scene, and Game

active tab. When the game is running, what you see in this view is the game. It isn't necessary to manually switch tabs every time you run the game, because the view automatically switches to Game when the game starts.

> **TIP** While the game is running, you can switch back to the Scene view, allowing you to inspect objects in the running scene. This capability is hugely useful for seeing what's going on while the game is running and is a helpful debugging tool that isn't available in most game engines.

Speaking of running the game, that's as simple as hitting the Play button just above the Scene view. That whole top section of the interface is referred to as the Toolbar, and Play is located right in the middle. Figure 1.10 breaks apart the full editor interface to show only the Toolbar at the top, as well as the Scene/Game tabs right underneath.

At the left side of the Toolbar are buttons for scene navigation and transforming objects—how to look around the scene and how to move objects. I suggest you spend some time practicing looking around the scene and moving objects, because these are two of the most important activities you'll do in Unity's visual editor (they're so important that they get their own section following this one). The right side of the Toolbar is where you'll find drop-down menus for layouts and layers. As mentioned earlier, the layout of Unity's interface is flexible, so the Layouts menu allows you to switch between layouts. As for the Layers menu, that's advanced functionality that you can ignore for now (layers will be mentioned in future chapters).

1.2.2 Using the mouse and keyboard

Scene navigation is primarily done using the mouse, along with a few modifier keys used to modify what the mouse is doing. The three main navigation maneuvers are

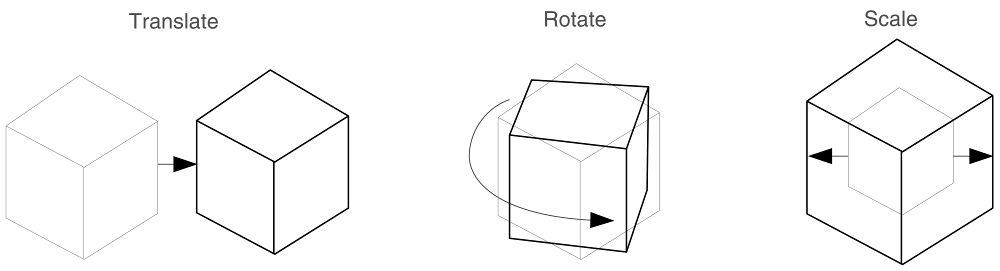


Figure 1.11 Applying the three transforms: Translate, Rotate, and Scale. (The lighter lines are the previous state of the object before it was transformed.)

Move, Orbit, and Zoom. The specific mouse movements for each are described in appendix A at the end of this book, because they vary depending on what mouse you're using. Basically, the three different movements involve clicking-and-dragging while holding down some combination of Alt (or Option on Mac) and Ctrl. Spend a few minutes moving around in the scene to understand what Move, Orbit, and Zoom do.

TIP Although Unity can be used with one- or two-button mice, I highly recommend getting a three-button mouse (and yes, a three-button mouse works fine on Mac OS X).

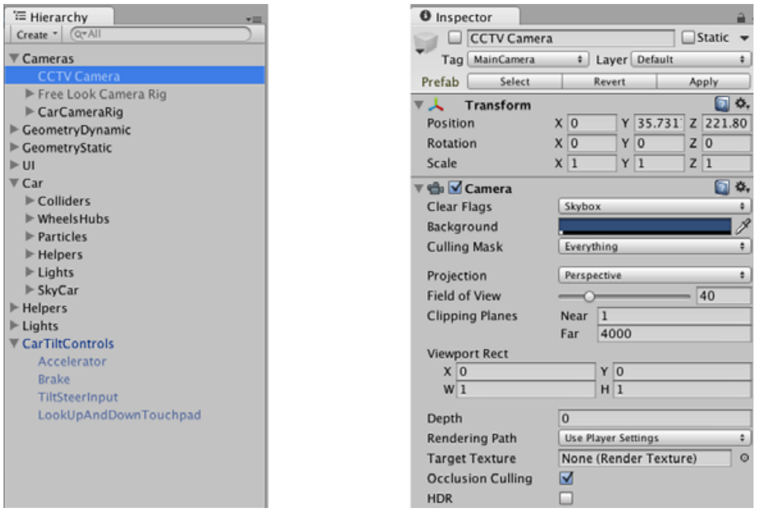
Transforming objects is also done through three main maneuvers, and the three scene navigation moves are analogous to the three transforms: Translate, Rotate, and Scale (figure 1.11 demonstrates the transforms on a cube).

When you select an object in the scene, you can then move it around (the mathematically accurate technical term is *translate*), rotate the object, or scale how big it is. Relating back to scene navigation, Move is when you Translate the camera, Orbit is when you Rotate the camera, and Zoom is when you Scale the camera. Besides the buttons on the Toolbar, you can switch between these functions by pressing W, E, or R on the keyboard. When you activate a transform, you'll notice a set of color-coded arrows or circles appears over the object in the scene; this is the Transform gizmo, and you can click-and-drag this gizmo to apply the transformation.

There's also a fourth tool next to the transform buttons. Called the Rect tool, it's designed for use with 2D graphics. This one tool combines movement, rotation, and scaling. These operations have to be separate tools in 3D but are combined in 2D because there's one less dimension to worry about. Unity has a host of other keyboard shortcuts for speeding up a variety of tasks. Refer to appendix A to learn about them. And with that, on to the remaining sections of the interface!

1.2.3 The Hierarchy tab and the Inspector

Looking at the sides of the screen, you'll see the Hierarchy tab on the left and the Inspector on the right (see figure 1.12). Hierarchy is a list view with the name of every



```
CCTV Camera                      Static
Tag  MainCamera        Layer Default
Prefab
Transform
Position   X 0   Y 35.731   Z 221.80
Rotation   X 0   Y 0        Z 0
Scale      X 1   Y 1        Z 1
Camera
Clear Flags       Skybox
Background
Culling Mask      Everything
Projection        Perspective
Field of View     40
Clipping Planes   Near 1
                  Far  4000
Viewport Rect
  X 0          Y 0
  W 1          H 1
Depth             0
Rendering Path    Use Player Settings
Target Texture    None (Render Texture)
Occlusion Culling ✓
HDR
```

Figure 1.12 **Editor screenshot cropped to show the Hierarchy and Inspector tabs**

object in the scene listed, with the names nested together according to their hierarchy linkages in the scene. Basically, it's a way of selecting objects by name instead of hunting them down and clicking them within Scene. The Hierarchy linkages group objects together, visually grouping them like folders and allowing you to move the entire group together.

The Inspector shows you information about the currently selected object. Select an object and the Inspector is then filled with information about that object. The information shown is pretty much a list of components, and you can even attach or remove components from objects. All game objects have at least one component, Transform, so you'll always at least see information about positioning and rotation in the Inspector. Many times objects will have several components listed here, including scripts attached to that object.

1.2.4 *The Project and Console tabs*

At the bottom of the screen you'll see Project and Console (see figure 1.13). As with Scene and View, these aren't two separate portions of the screen but rather tabs that you can switch between. Project shows all the assets (art, code, and so on) in the

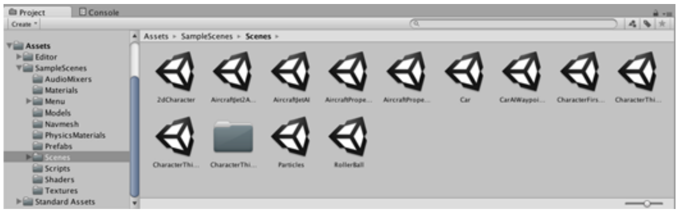


Figure 1.13 **Editor screenshot cropped to show the Project and Console tabs**

project. Specifically, on the left side of the view is a listing of the directories in the project; when you select a directory, the right side of the view shows the individual files in that directory. The directory listing in Project is similar to the list view in Hierarchy, but whereas Hierarchy shows objects in the scene, Project shows files that aren't contained within any specific scene (including scene files—when you save a scene, it shows up in Project!).

TIP Project view mirrors the Assets directory on disk, but you generally shouldn't move or delete files directly by going to the Assets folder. If you do those things within the Project view, Unity will keep in sync with that folder.

The Console is the place where messages from the code show up. Some of these messages will be debug output that you placed deliberately, but Unity also emits error messages if it encounters problems in the script you wrote.

1.3 Getting up and running with Unity programming

Now let's look at how the process of programming works in Unity. Although art assets can be laid out in the visual editor, you need to write code to control them and make the game interactive. Unity supports a few programming languages, in particular JavaScript and C#. There are pros and cons to both choices, but you'll be using C# throughout this book.

Why choose C# over JavaScript?

All of the code listings in this book use C# because it has a number of advantages over JavaScript and fewer disadvantages, especially for professional developers (it's certainly the language I use at work).

One benefit is that C# is strongly typed, whereas JavaScript is not. Now, there are lots of arguments among experienced programmers about whether or not dynamic typing is a better approach for, say, web development, but programming for certain gaming platforms (such as iOS) often benefits from or even requires static typing. Unity has even added the directive `#pragma strict` to force static typing within JavaScript. Although technically this works, it breaks one of the bedrock principles of how JavaScript operates, and if you're going to do that, then you're better off using a language that's intrinsically strongly typed.

This is just one example of how JavaScript within Unity isn't quite the same as JavaScript elsewhere. JavaScript in Unity is certainly similar to JavaScript in web browsers, but there are lots of differences in how the language works in each context. Many developers refer to the language in Unity as UnityScript, a name that indicates similarity to but separateness from JavaScript. This "similar but different" state can create issues for programmers, both in terms of bringing in knowledge about JavaScript from outside Unity, and in terms of applying programming knowledge gained by working in Unity.

Let's walk through an example of writing and running some code. Launch Unity and create a new project; choose File > New Project to open the New Project window. Type a name for the project, and then choose where you want to save it. Realize that a Unity project is simply a directory full of various asset and settings files, so save the project anywhere on your computer. Click Create Project and then Unity will briefly disappear while it sets up the project directory.

WARNING Unity projects remember which version of Unity they were created in and will issue a warning if you attempt to open them in a different version. Sometimes it doesn't matter (for example, just ignore the warning if it appears while opening this book's sample downloads), but sometimes you will want to back up your project before opening it.

When Unity reappears you'll be looking at a blank project. Next, let's discuss how your programs get executed in Unity.

1.3.1 How code runs in Unity: script components

All code execution in Unity starts from code files linked to an object in the scene. Ultimately it's all part of the component system described earlier; game objects are built up as a collection of components, and that collection can include scripts to execute.

NOTE Unity refers to the code files as scripts, using a definition of "script" that's most commonly encountered with JavaScript running in a browser: the code is executed within the Unity game engine, versus compiled code that runs as its own executable. But don't get confused because many people define the word differently; for example, "scripts" often refer to short, self-contained utility programs. Scripts in Unity are more akin to individual OOP classes, and scripts attached to objects in the scene are the object instances.

As you've probably surmised from this description, in Unity, scripts *are* components—not all scripts, mind you, only scripts that inherit from `MonoBehaviour`, the base class for script components. `MonoBehaviour` defines the invisible groundwork for how components attach to game objects, and (as shown in listing 1.1) inheriting from it provides a couple of automatically run methods that you can override. Those methods include `Start()`, which is called once when the object becomes active (which is generally as soon as the level with that object has loaded), and `Update()`, which is called every frame. Thus your code is run when you put it inside these predefined methods.

DEFINITION A *frame* is a single cycle of the looping game code. Nearly all video games (not just in Unity, but video games in general) are built around a core game loop, where the code executes in a cycle while the game is running. Each cycle includes drawing the screen; hence the name *frame* (just like the series of still frames of a movie).

Listing 1.1 Code template for a basic script component

```
using UnityEngine;                          <-- Include namespaces for
using System.Collections;                       Unity and Mono classes.

public class HelloWorld : MonoBehaviour {   <-- The syntax for inheritance

    void Start() {
        // do something once              <-- Put code in here that runs once.
    }

    void Update() {
        // do something every frame        <-- Put code in here that
    }                                           runs every frame.
}
```

This is what the file contains when you create a new C# script: the minimal boilerplate code that defines a valid Unity component. Unity has a script template tucked away in the bowels of the application, and when you create a new script it copies that template and renames the class to match the name of the file (which is HelloWorld.cs in my case). There are also empty shells for `Start()` and `Update()` because those are the two most common places to call your custom code from (although I tend to adjust the whitespace around those functions a tad, because the template isn't quite how I like the whitespace and I'm finicky about that).

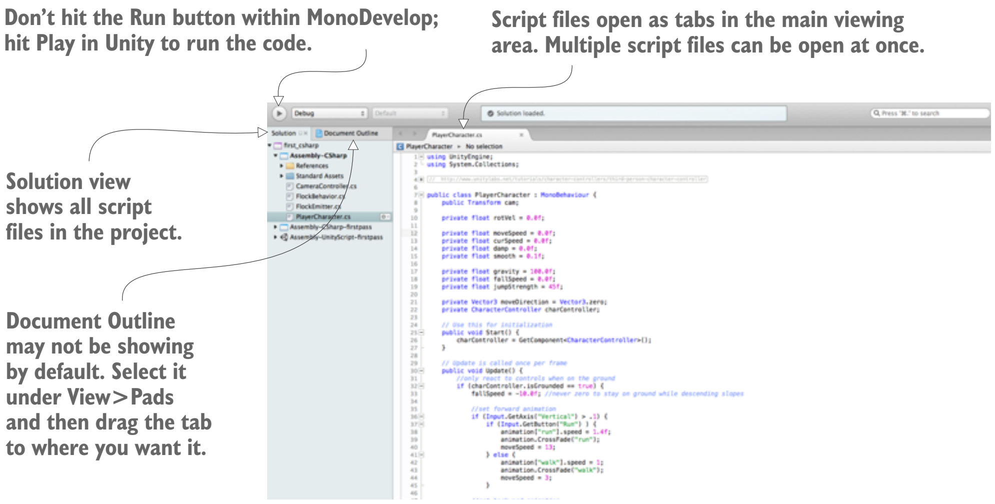
To create a script, select C# Script from the Create menu that you access either under the Assets menu (note that Assets and GameObjects both have listings for Create but they're different menus) or by right-clicking in the Project view. Type in a name for the new script, such as HelloWorld. As explained later in the chapter (see figure 1.15), you'll click-and-drag this script file onto an object in the scene. Double-click the script and it'll automatically be opened in another program called MonoDevelop, discussed next.

1.3.2 *Using MonoDevelop, the cross-platform IDE*

Programming isn't done within Unity exactly, but rather code exists as separate files that you point Unity to. Script files can be created within Unity, but you still need to use some text editor or IDE to write all the code within those initially empty files. Unity comes bundled with MonoDevelop, an open source, cross-platform IDE for C# (figure 1.14 shows what it looks like). You can visit www.monodevelop.com to learn more about this software, but the version to use is the version bundled along with Unity, rather than a version downloaded from their website, because some modifications were made to the base software in order to better integrate it with Unity.

NOTE MonoDevelop organizes files into groupings called a *solution*. Unity automatically generates a solution that has all the script files, so you usually don't need to worry about that.

Because C# originated as a Microsoft product, you may be wondering if you can use Visual Studio to do programming for Unity. The short answer is yes, you can. Support


Don’t hit the Run button within MonoDevelop; hit Play in Unity to run the code.
Script files open as tabs in the main viewing area. Multiple script files can be open at once.
Solution view shows all script files in the project.
Document Outline may not be showing by default. Select it under View>Pads and then drag the tab to where you want it.
Debug
Default
Solution loaded.
Solution
Document Outline
PlayerCharacter.cs
PlayerCharacter ▸ No selection
first_csharp
Assembly-CSharp
References
Standard Assets
CameraController.cs
FlockBehavior.cs
FlockEmitter.cs
PlayerCharacter.cs
Assembly-CSharp-firstpass
Assembly-UnityScript-firstpass

```
using UnityEngine;
using System.Collections;

// http://www.unitylabs.net/tutorials/character-controllers/third-person-character-controller

public class PlayerCharacter : MonoBehaviour {
    public Transform cam;

    private float rotVel = 0.0f;

    private float moveSpeed = 0.0f;
    private float curSpeed = 0.0f;
    private float damp = 0.0f;
    private float smooth = 0.1f;

    private float gravity = 100.0f;
    private float fallSpeed = 0.0f;
    private float jumpStrength = 45f;

    private Vector3 moveDirection = Vector3.zero;
    private CharacterController charController;

    // Use this for initialization
    public void Start() {
        charController = GetComponent<CharacterController>();
    }

    // Update is called once per frame
    public void Update() {
        //only react to controls when on the ground
        if (charController.isGrounded == true) {
            fallSpeed = -10.0f; //never zero to stay on ground while descending slopes

            //set forward animation
            if (Input.GetAxis("Vertical") > .1) {
                if (Input.GetButton("Run") ) {
                    animation["run"].speed = 1.4f;
                    animation.CrossFade("run");
                    moveSpeed = 13;
                } else {
                    animation["walk"].speed = 1;
                    animation.CrossFade("walk");
                    moveSpeed = 3;
                }
```

Figure 1.14 Parts of the interface in MonoDevelop

tools are available from www.unityvs.com but I generally prefer MonoDevelop, mostly because Visual Studio only runs on Windows and using that IDE would tie your workflow to Windows. That’s not necessarily a bad thing, and if you’re already using Visual Studio to do programming then you could keep using it and not have any problems following along with this book (beyond this introductory chapter, I’m not going to talk about the IDE). Tying your workflow to Windows, though, would run counter to one of the biggest advantages of using Unity, and doing so could prove problematic if you need to work with Mac-based developers on your team and/or if you want to deploy your game to iOS. Although C# originated as a Microsoft product and thus only worked on Windows with the .NET Framework, C# has now become an open language standard and there’s a significant cross-platform framework: Mono. Unity uses Mono for its programming backbone, and using MonoDevelop allows you to keep the entire development workflow cross-platform.

Always keep in mind that although the code is written in MonoDevelop, the code isn’t actually run there. The IDE is pretty much a fancy text editor, and the code is run when you hit Play within Unity.

1.3.3 *Printing to the console: Hello World!*

All right, you already have an empty script in the project, but you also need an object in the scene to attach the script to. Recall figure 1.1 depicting how a component system works; a script is a component, so it needs to be set as one of the components on an object.

Select GameObject > Create Empty, and a blank GameObject will appear in the Hierarchy list. Now drag the script from the Project view over to the Hierarchy view and drop it on the empty GameObject. As shown in figure 1.15, Unity will highlight

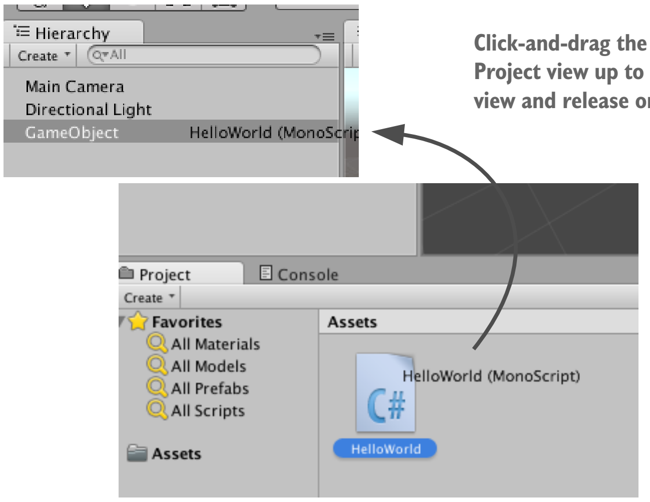


Figure 1.15 **How to link a script to a GameObject**

valid places to drop the script, and dropping it on the GameObject will attach the script to that object. To verify that the script is attached to the object, select the object and look at the Inspector view. You should see two components listed: the Transform component that's the basic position/rotation/scale component all objects have and that can't be removed, and below that, your script.

> **NOTE** Eventually this action of dragging objects from one place and dropping them on other objects will feel routine. A lot of different linkages in Unity are created by dragging things on top of each other, not just attaching scripts to objects.

When a script is linked to an object, you'll see something like figure 1.16, with the script showing up as a component in the Inspector. Now the script will execute when you play the scene, although nothing is going to happen yet because you haven't written any code. Let's do that next!

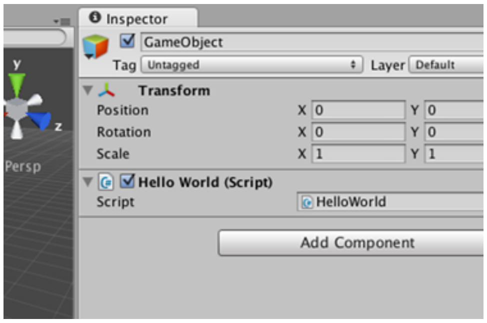


Figure 1.16 **Linked script being displayed in the Inspector**

Open the script in MonoDevelop to get back to listing 1.1. The classic place to start when learning a new programming environment is having it print the text "Hello World!" so add this line inside the `Start()` method, as shown in the following listing.

Listing 1.2 Adding a console message

```
void Start() {
    Debug.Log("Hello World!");
}
```

Add the logging command here.

What the `Debug.Log()` command does is print a message to the Console view in Unity. Meanwhile that line goes in the `Start()` method because, as was explained earlier, that method is called as soon as the object becomes active. In other words, `Start()` will be called once as soon as you hit Play in the editor. Once you've added the log command to your script (be sure to save the script), hit Play in Unity and switch to the Console view. You'll see the message "Hello World!" appear. Congratulations, you've written your first Unity script! In later chapters the code will be more elaborate, of course, but this is an important first step.

"Hello World!" steps in brief

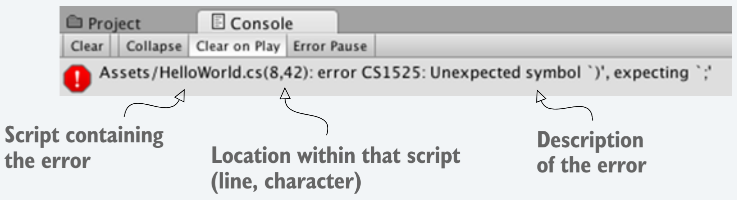
Let's reiterate and summarize the steps from the last several pages:

1. Create a new project.
2. Create a new C# script.
3. Create an empty GameObject.
4. Drag the script onto the object.
5. Add the log command to the script.
6. Press Play!

You could now save the scene; that would create a .unity file with the Unity icon. The scene file is a snapshot of everything currently loaded in the game so that you can reload this scene later. It's hardly worth saving this scene because it's so simple (just a single empty GameObject), but if you don't save the scene then you'll find it empty again when you come back to the project after quitting Unity.

Errors in the script

To see how Unity indicates errors, purposely put a typo in the HelloWorld script. For example, if you type an extra parenthesis symbol, this error message will appear in the Console with a red error icon:



```
Assets/HelloWorld.cs(8,42): error CS1525: Unexpected symbol `)', expecting `;'
```

1.4 Summary

In this chapter you've learned that

- Unity is a multiplatform development tool.
- Unity's visual editor has several sections that work in concert.
- Scripts are attached to objects as components.
- Code is written inside scripts using MonoDevelop.

GAME PROGRAMMING

Unity IN ACTION

Joseph Hocking

This book helps readers build successful games with the Unity game development platform. You will use the powerful C# language, Unity's intuitive workflow tools, and a state-of-the-art rendering engine to build and deploy mobile, desktop, and console games. Unity's single code-base approach minimizes inefficient switching among development tools and concentrates your attention on making great interactive experiences.

Unity in Action teaches you how to write and deploy games. You'll master the Unity toolset from the ground up, adding the skills you need to go from application coder to game developer. Each sample project illuminates specific Unity features and game development strategies. As you read and practice, you'll build up a well-rounded skill set for creating graphically driven 2D and 3D game applications.

What's Inside

- Program characters that run, jump, and interact
- Build code architectures that manage the game's state
- Connect your games to the internet to download live data
- Deploy games to platforms including web and mobile
- Covers Unity version 5

You'll need to know how to program, in C# or a similar OO language. No previous Unity experience or game development knowledge is assumed.

Joe Hocking is a software engineer specializing in interactive media development. He works for Synapse Games and teaches classes in game development at Columbia College Chicago.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/UnityinAction

Free eBook
SEE INSERT

"Joe Hocking wastes none of your time and gets you coding fast."
—From the Foreword by Jesse Schell, author of *The Art of Game Design*

"Gets you up and running in no time."
—Sergio Arbeo, codecantor

"The text is clear and concise, and the examples are outstanding."
—Dan Kacenjar, Sr. Wolters Kluwer

"All the roadblocks evaporated, and I took my game from concept to build in short order."
—Philip Taffet, SOHOsoft LLC

ISBN 13: 978-1-617292-32-3
ISBN 10: 1-617292-32-X
54499
9 781617 292323

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]