Multiplatform game development in C#

Unity IN ACTION

Covers Unity 5.0

Joseph Hocking

Foreword by Jesse Schell

MANNING

Unity in Action: Multiplatform game development in C# with Unity 5

Joseph Hocking

MANNING PUBLICATIONS

Copyright

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: *orders@manning.com*

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Dan Maharry
Technical development editor: Scott Chaussee
Copyeditor: Elizabeth Welch
Proofreader: Melody Dolab
Technical proofreader: Christopher Haupt
Typesetter: Marija Tudor
Cover designer: Marija Tudor

ISBN: 9781617292323

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – EBM – 20 19 18 17 16 15

Brief Table of Contents

Table of Contents

Foreword

I started programming games in 1982. It wasn’t easy. We had no internet. Resources were limited to a handful of mostly terrible books and magazines that offered fascinating but confusing code fragments, and as for game engines—well, there weren’t any! Coding games was a massive uphill battle.

How I envy you, reader, holding the power of this book in your hands. The Unity engine has done so much to open game programming to so many people. Unity has managed to strike an excellent balance by being a powerful, professional game engine that’s still affordable and approachable for someone just getting started.

Approachable, that is, with the right guidance. I once spent time in a circus troupe run by a magician. He was kind enough to take me in and help guide me toward becoming a good performer. “When you stand on a stage,” he pronounced, “you make a promise. And that promise is ‘I will not waste your time.’”

What I love most about *Unity in Action* is the “action” part. Joe Hocking wastes none of your time and gets you coding fast—and not just nonsense code, but interesting code that you can understand and build from, because he knows you don’t just want to read his book, and you don’t just want to program his examples—you want to be coding *your own game*.

And with his guidance, you’ll be able to do that sooner than you might expect. Follow Joe’s steps, but when you feel ready, don’t be shy about diverging from his path and breaking out on your own. Skip around to what interests you most—try experiments, be bold and brave! You can always return to the text if you get too lost.

But let’s not dally in this foreword—the entire future of game development is impatiently waiting for you to begin! Mark this day on your calendar, for today is the day that everything changed. It will be forever remembered as the day you started making games.

JESSE SCHELL

CEO of SCHELL GAMES

AUTHOR OF *THE ART OF GAME DESIGN*

Preface

I’ve been programming games for quite some time, but only started using Unity relatively recently. Unity didn’t exist when I first started developing games; the first version was released in 2005. Right from the start, it had a lot of promise as a game development tool, but it didn’t come into its own until several versions later. In particular, platforms like iOS and Android (collectively referred to as “mobile”) didn’t emerge until later, and those platforms factor heavily into Unity’s growing prominence.

Initially, I viewed Unity as a curiosity, an interesting development tool to keep an eye on but not actually use. During this time, I was programming games for both desktop computers and websites and doing projects for a range of clients. I was using tools like Blitz3D and Flash, which were great to program in but were limiting in a lot of ways. As those tools started to show their age, I kept looking for better ways to develop games.

I started experimenting with Unity around version 3, and then completely switched to it when Synapse Games (the company I work for now) started developing mobile games. At first, I worked for Synapse on web games, but we eventually moved over to mobile games. And then we came full circle because Unity enabled us to deploy to the web in addition to mobile, all from one codebase!

I’ve always seen sharing knowledge as important, and I’ve taught game development for the last several years. In large part I do this because of the example set for me by the many mentors and teachers I’ve had. (Incidentally, you may even have heard of one of my teachers because he was such an inspiring person: Randy Pausch delivered the Last Lecture shortly before he passed away in 2008.) I’ve taught classes at several schools, and I’ve always wanted to write a book about game development.

In many ways, what I’ve written here is the book I wish had existed back when I was first learning Unity. Among Unity’s many virtues is the availability of a huge treasure trove of learning resources, but those resources tend to take the form of unfocused fragments (like the script reference or isolated tutorials) and require a great deal of digging to find what you need. Ideally, I’d have a book that wrapped up everything I needed to know in one place and presented it in a clear and logically constructed manner, so now I’m writing such a book for you. I’m targeting people who already know how to program, but who are newcomers to Unity, and possibly new to game development in general. The choice of projects reflects my experience of gaining skills and confidence by doing a variety of freelance projects in rapid succession.

In learning to develop games using Unity, you’re setting out on an exciting adventure. For me, learning how to develop games meant putting up with a lot of hassles. You, on the other hand, have the advantage of a single coherent resource to learn from: this book!

Acknowledgments

I would like to thank Manning Publications for giving me the opportunity to write this book. The editors I worked with, including Robin de Jongh and especially Dan Maharry, helped me throughout this undertaking, and the book is much stronger for their feedback. My sincere thanks also to the many others who worked with me during the development and production of the book.

My writing benefited from the scrutiny of reviewers every step of the way. Thanks to Alex Lucas, Craig Hoffman, Dan Kacenjar, Joshua Frederick, Luca Campobasso, Mark Elston, Philip Taffet, René van den Berg, Sergio Arbeo Rodríguez, Shiloh Morris, and Victor M. Perez. Special thanks to the notable review work by technical development editor Scott Chaussee and by technical proofreader Christopher Haupt. And I also want to thank Jesse Schell for writing the foreword to my book.

Next, I'd like to recognize the people who've made my experience with Unity a fruitful one. That, of course, starts with Unity Technologies, the company that makes Unity (the game engine). I owe a debt to the community at gamedev.stackexchange.com. I visit that QA site almost daily to learn from others and to answer questions. And the biggest push for me to use Unity came from Alex Reeve, my boss at Synapse Games. Similarly, I've picked up tricks and techniques from my coworkers, and they all show up in the code I write.

Finally, I want to thank my wife Virginia for her support during the time I was writing the book. Until I started working on it, I never really understood how much a book project takes over your life and affects everyone around you. Thank you so much for your love and encouragement.

About this Book

This is a book about programming games in Unity. Think of it as an intro to Unity for experienced programmers. The goal of this book is straightforward: to take people who have some programming experience but no experience with Unity and teach them how to develop a game using Unity.

The best way of teaching development is through example projects, with students learning by doing, and that's the approach this book takes. I'll present topics as steps toward building sample games, and you'll be encouraged to build these games in Unity while exploring the book. We'll go through a selection of different projects every few chapters, rather than one monolithic project developed over the entire book; sometimes other books take the "one monolithic project" approach, but that can make it hard to jump into the middle if the early chapters aren't relevant to you.

This book will have more rigorous programming content than most Unity books (especially beginners' books). Unity is often portrayed as 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. If you don't already know how to program a computer, I suggest going to a resource like Codecademy first (the computer programming lessons at Khan Academy work well, too) and then come back to this book after learning how to program.

Don't worry about the exact programming language; C# is used throughout this book, but skills from other languages will transfer quite well. Although the first half of the book will take its time introducing new concepts and will carefully and deliberately step you through developing your first game in Unity, the remaining chapters will move a lot faster in order to take readers through projects in multiple game genres. The book will end with a chapter describing deployment to various platforms like the web and mobile, but the main thrust of the book won't make any reference to the ultimate deployment target because Unity is wonderfully platform-agnostic.

As for other aspects of game development, extensive coverage of art disciplines would water down how much the book can cover and would be largely about software external to Unity (for example, the animation software used).

Discussion of art tasks will be limited to aspects specific to Unity or that all game developers should know. (Note, though, that there is an appendix about modeling custom objects.)

Roadmap

Chapter 1 introduces you to Unity, the cross-platform game development environment. You'll learn about the fundamental component system underlying everything in Unity, as well as how to write and execute basic scripts.

Chapter 2 progresses to writing a demo of movement in 3D, covering topics like mouse and keyboard input. Defining and manipulating both 3D positions and rotations are thoroughly explained.

Chapter 3 turns the movement demo into a first-person shooter, teaching you raycasting and basic AI. Raycasting (shooting a line into the scene and seeing what intersects) is a useful operation for all sorts of games.

Chapter 4 covers art asset importing and creation. This is the one chapter of the book that does not focus on code, because every project needs (basic) models and textures.

Chapter 5 teaches you how to create a 2D game in Unity. Although Unity started exclusively for 3D graphics, there's now excellent support for 2D graphics.

Chapter 6 introduces you to the latest GUI functionality in Unity. Every game needs a UI, and the latest versions of Unity feature an improved system for creating user interfaces.

Chapter 7 shows how to create another movement demo in 3D, only seen from the third person this time. Implementing third-person controls will demonstrate a number of key 3D math operations, and you'll learn how to work with an animated character.

Chapter 8 goes over how to implement interactive devices and items within your game. The player will have a number of ways of operating these devices, including touching them directly, touching triggers within the game, or pressing a button on the controller.

Chapter 9 covers how to communicate with the internet. You’ll learn how to send and receive data using standard internet technologies, like HTTP requests to get XML data from a server.

Chapter 10 teaches how to program audio functionality. Unity has great support for both short sound effects and long music tracks; both sorts of audio are crucial for almost all video games.

Chapter 11 walks you through bringing together pieces from different chapters into a single game. In addition, you’ll learn how to program point-and-click controls and how to save the player’s game.

Chapter 12 goes over building the final app, with deployment to multiple platforms like desktop, web, and mobile. Unity is wonderfully platform-agnostic, enabling you to create games for every major gaming platform!

There are also four appendixes with additional information about scene navigation, external tools, Blender, and learning resources.

Code conventions, requirements, and downloads

All the source code in the book, whether in code listings or snippets, is in a `fixed-width font like this`, which sets it off from the surrounding text. In most listings, the code is annotated to point out key concepts, and numbered bullets are sometimes used in the text to provide additional information about the code. The code is formatted so that it fits within the available page space in the book by adding line breaks and using indentation carefully.

The only software required is Unity; this book uses Unity 5.0, which is the latest version as I write this. Certain chapters do occasionally discuss other pieces of software, but those are treated as optional extras and not core to what you’re learning.

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. If you see that warning while opening this book's sample downloads, click Continue and ignore it.

The code listings sprinkled throughout the book generally show what to add or change in existing code files; unless it's the first appearance of a given code file, don't replace the entire file with subsequent listings. Although you can download complete working sample projects to refer to, you'll learn best by typing out the code listings and only looking at the working samples for reference. Those downloads are available from the publisher's website at www.manning.com/UnityinAction.

Author Online

The purchase of *Unity in Action* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/UnityinAction. This page provides information on how to get on the forum once you are registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the author

Joseph Hocking is a software engineer living in Chicago, specializing in interactive media development. He works for Synapse Games as a developer of web and mobile games, such as the recently released *Tyrant Unleashed*. He also teaches classes in game development at Columbia College Chicago, and his website is www.newarteest.com.

About the cover illustration

The figure on the cover of *Unity in Action* is captioned "Habit of the Master of Ceremonies of the Grand Signior." The Grand Signior was another name for a sultan of the Ottoman Empire. The illustration is taken from Thomas Jefferys' *A Collection of the Dresses of Different Nations, Ancient and Modern* (4 volumes), London, published between 1757 and 1772. The title page states that these are hand-colored copperplate engravings, heightened with gum arabic. Thomas Jefferys (1719–1771), was called "Geographer to King George III." An English cartographer who was the leading map supplier of his day, Jeffreys engraved and printed maps for government and other official bodies and produced a wide range of commercial maps and atlases, especially of North America. His work as a mapmaker sparked an interest in local dress customs of the lands he surveyed, which are brilliantly displayed in this four-volume collection.

Fascination with faraway lands and travel for pleasure were relatively new phenomena in the late eighteenth century and collections such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other countries. The diversity of the drawings in Jeffreys' volumes speaks vividly of the uniqueness and individuality of the world's nations some 200 years ago. Dress codes have changed since then and the diversity by region and country, so rich at the time, has faded away. It is now hard to tell the inhabitant of one continent apart from another. Perhaps, trying to view it optimistically, we have traded a cultural and visual diversity for a more varied personal life, or a more varied and interesting intellectual and technical life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Jeffreys' pictures.

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!

Chapter 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

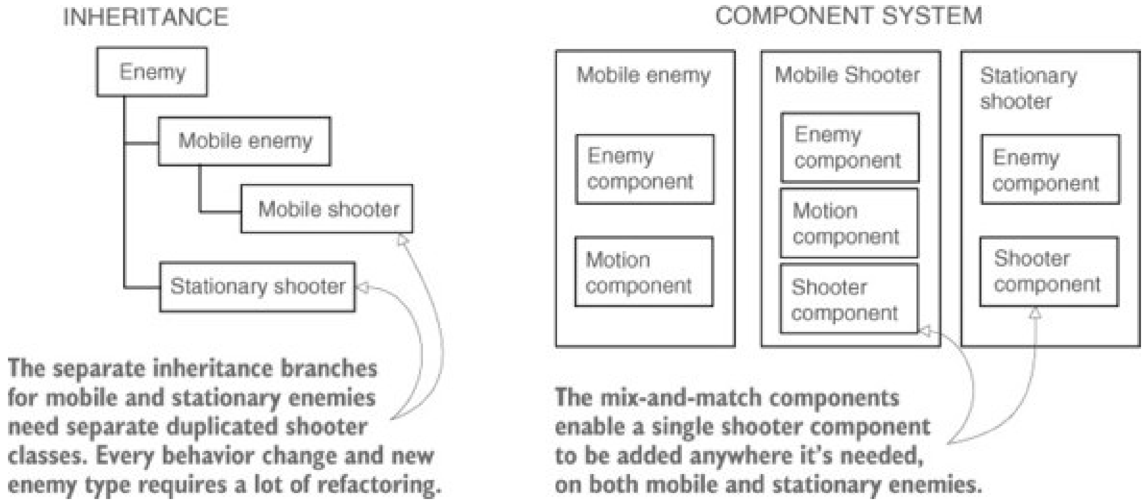
Most other game development tools that have a central visual editor are also saddled with limited and inflexible scripting support, but Unity doesn’t suffer from that disadvantage. Although everything created for Unity ultimately goes through the visual editor, this core interface involves a lot of linking 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 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.

Figure 1.1. Inheritance vs. components



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.

The mix-and-match components enable a single shooter component to be added anywhere it's needed, on both mobile and stationary enemies.

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

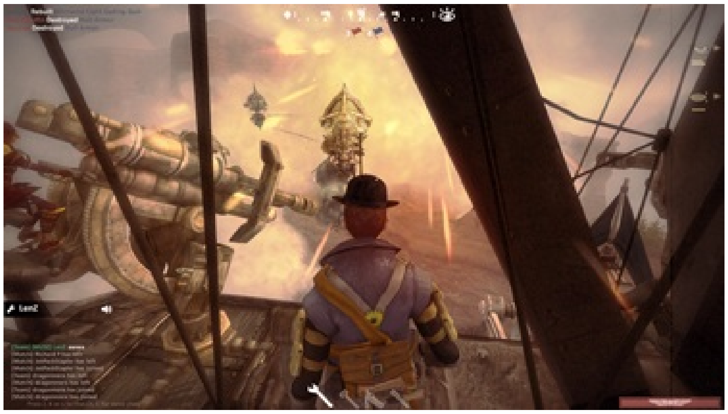
You've heard about the pros and cons of Unity, but you might still need convincing that the development tools in Unity can give first-rate results. Visit the Unity gallery at http://unity3d.com/showcase/gallery to see a constantly updated list of hundreds of games and simulations developed using Unity.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

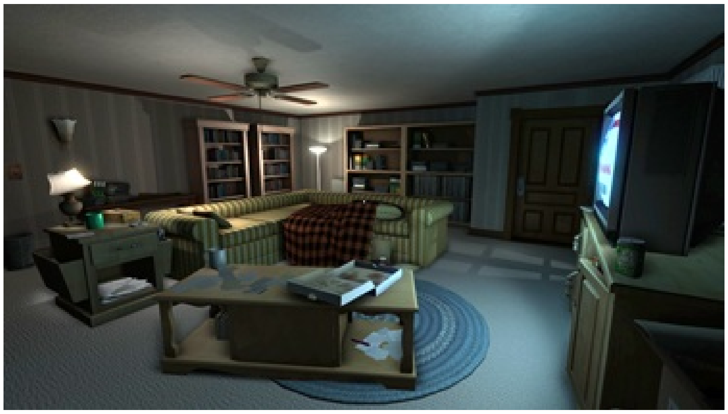
- Guns of Icarus Online (figure 1.2), a first-person shooter developed by Muse Games

  Figure 1.2. Guns of Icarus Online



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

  Figure 1.3. Gone Home



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

  Figure 1.4. Dead Trigger

- Bad Piggies (figure 1.5), a physics puzzle game developed by Rovio

  Figure 1.5. Bad Piggies

- Tyrant Unleashed (figure 1.6), a collectible card game developed by Synapse Games

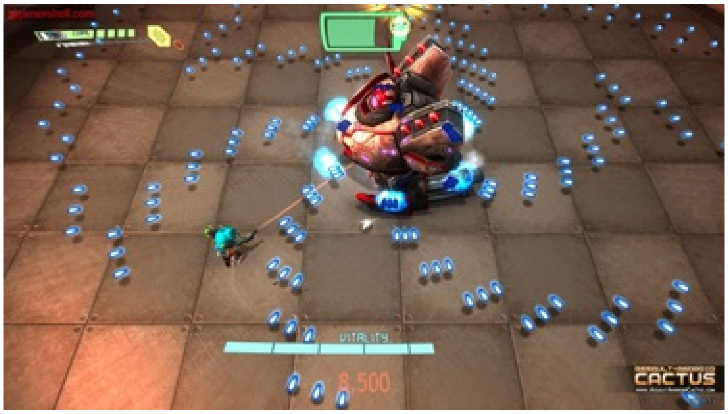
  Figure 1.6. Tyrant Unleashed

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:

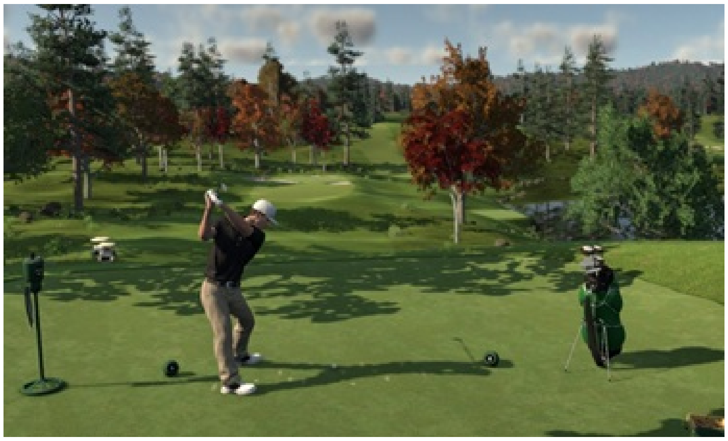
- Assault Android Cactus (figure 1.7), an arcade shooter developed by Witch Beam

  Figure 1.7. Assault Android Cactus



- The Golf Club (figure 1.8), a sports simulation developed by HB Studios

  Figure 1.8. The Golf Club



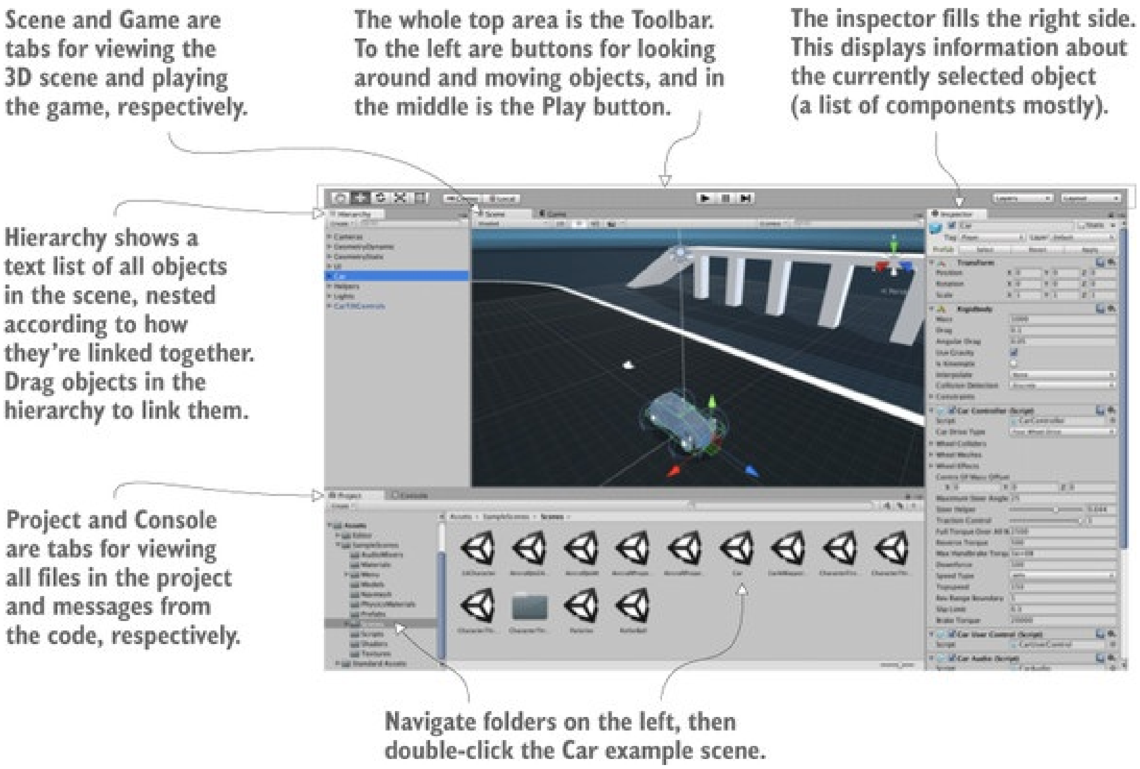
As you can see from these examples, Unity's strengths definitely can translate into commercial-quality games. But even with Unity's significant advantages over other game development tools, newcomers may have a misunderstanding about the involvement of programming in the development process. Unity is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

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.

Figure 1.9. Parts of the interface in Unity



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.

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.

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*.

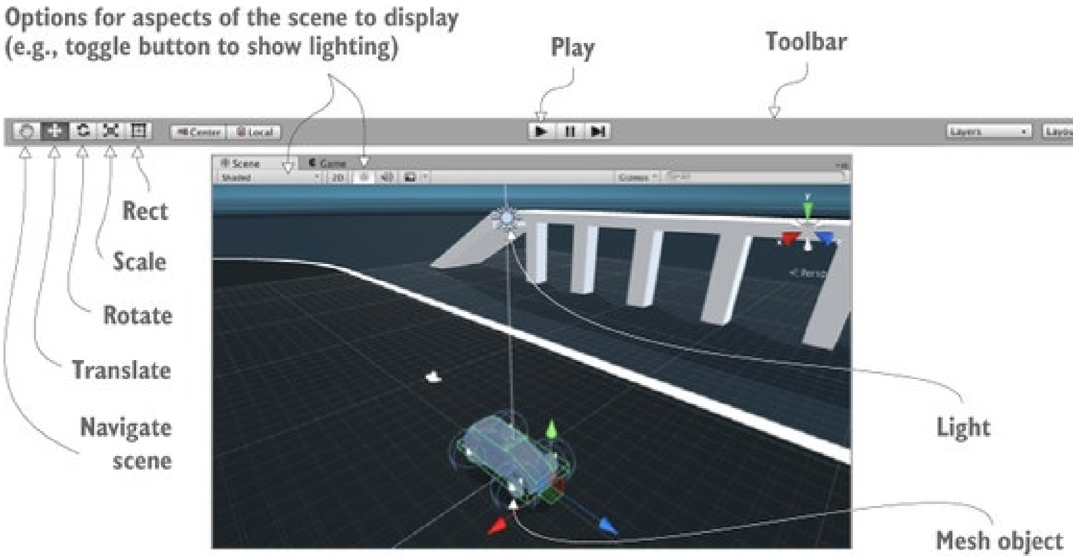
The Game view isn't a separate part of the screen but rather another tab located right next to Scene (look for tabs at the top left of views). A couple of places in the interface have multiple tabs like this; if you click a different tab, the view is replaced by the new active tab. When the game is running,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

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


Options for aspects of the scene to display (e.g., toggle button to show lighting)
Play
Toolbar
Rect
Scale
Rotate
Translate
Navigate scene
Light
Mesh object

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

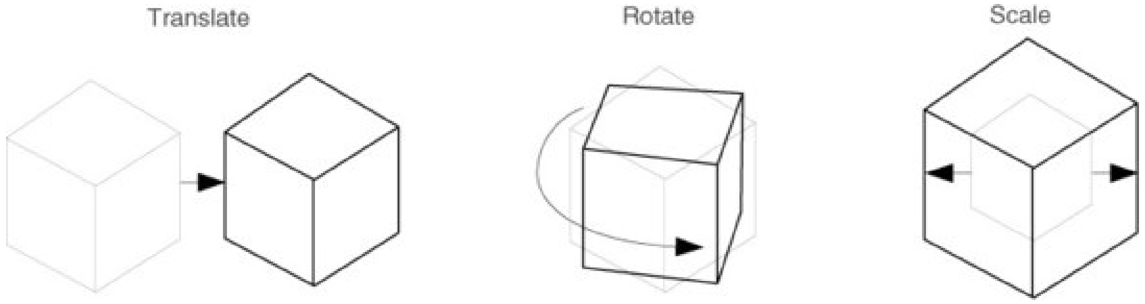
Scene navigation is primarily done using the mouse, along with a few modifier keys used to modify what the mouse is doing. The three main navigation maneuvers are Move, Orbit, and Zoom. The specific mouse movements for each are described in appendix A at the end of this book, because they vary depending 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).

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.)



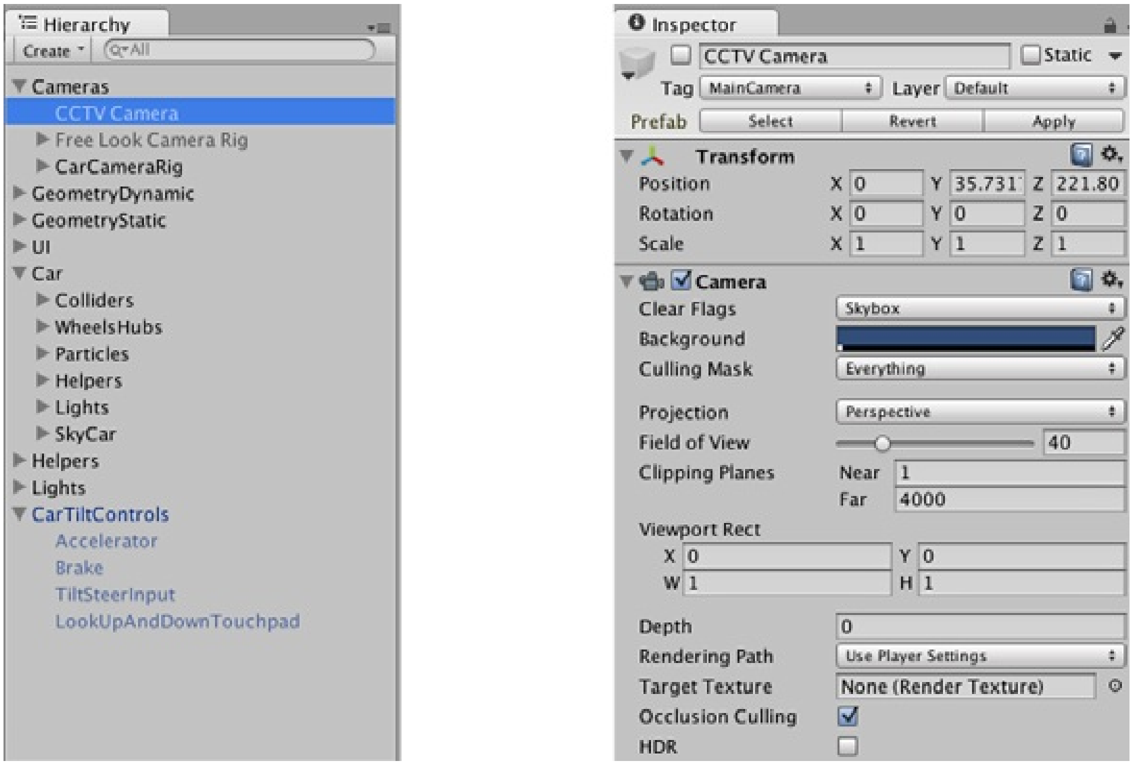
When you select an object in the scene, you can then move it around (the mathematically accurate technical term is *translate*), rotate the object, or scale how big it is. Relating back to scene navigation, Move is when you Translate the camera, Orbit is when you Rotate the camera, and Zoom is when you 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 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.

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



```
▼ Cameras
    CCTV Camera
  ▶ Free Look Camera Rig
  ▶ CarCameraRig
▶ GeometryDynamic
▶ GeometryStatic
▶ UI
▼ Car
  ▶ Colliders
  ▶ WheelsHubs
  ▶ Particles
  ▶ Helpers
  ▶ Lights
  ▶ SkyCar
▶ Helpers
▶ Lights
▼ CarTiltControls
    Accelerator
    Brake
    TiltSteerInput
    LookUpAndDownTouchpad
```

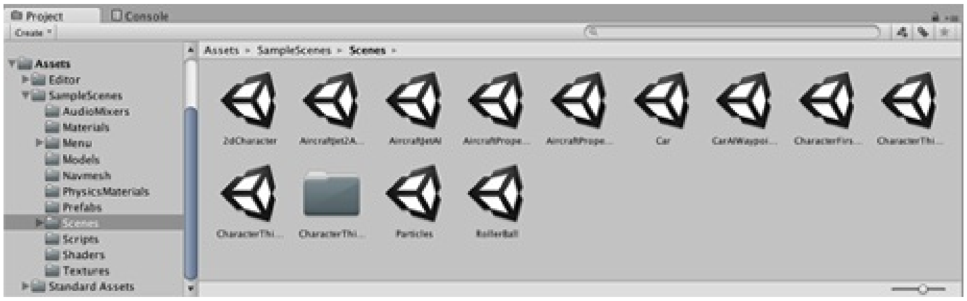
```
□ CCTV Camera                        □ Static
Tag  MainCamera        Layer  Default

▼ 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               □
```

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 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!).

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



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

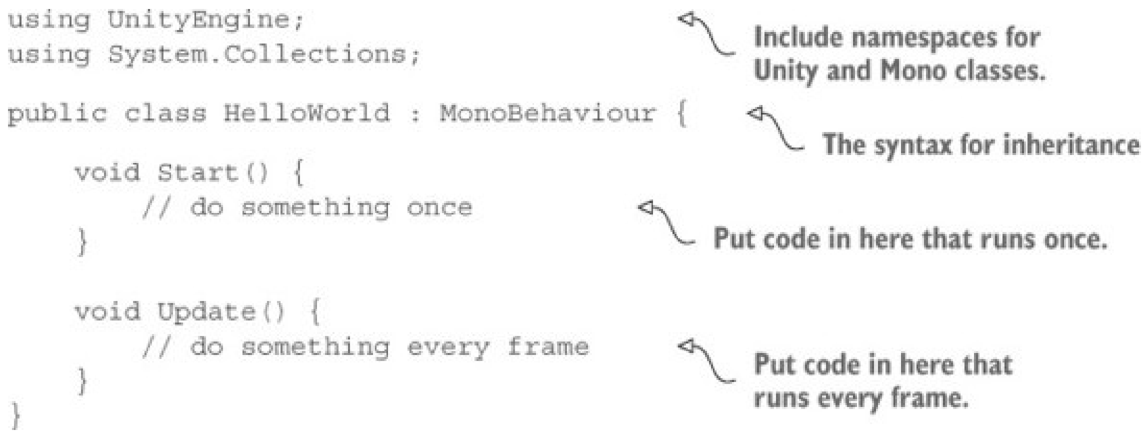
Unity refers to the code files as scripts, using a definition of “script” that’s most commonly encountered with JavaScript running in a browser: the code is executed within the Unity game engine, versus compiled code that runs as its own executable. But don’t get confused because many people define the 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;
using System.Collections;

public class HelloWorld : MonoBehaviour {

    void Start() {
        // do something once
    }

    void Update() {
        // do something every frame
    }
}
```

Include namespaces for Unity and Mono classes.

The syntax for inheritance

Put code in here that runs once.

Put code in here that runs every frame.

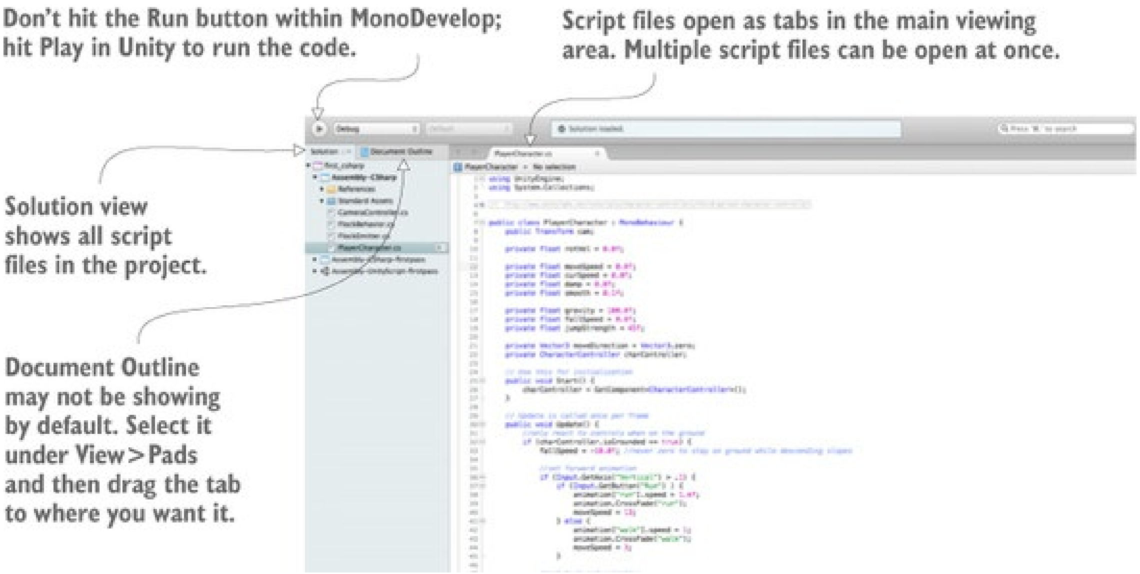
This is what the file contains when you create a new C# script: the minimal boilerplate code that defines a valid Unity component. Unity has a script template tucked away in the bowels of the application, and when you create a new script it copies that template and renames the class to match the name 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.

Figure 1.14. Parts of the interface in MonoDevelop



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.

```
using UnityEngine;
using System.Collections;

// [illegible]

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 = 5;
                }
```

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 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.

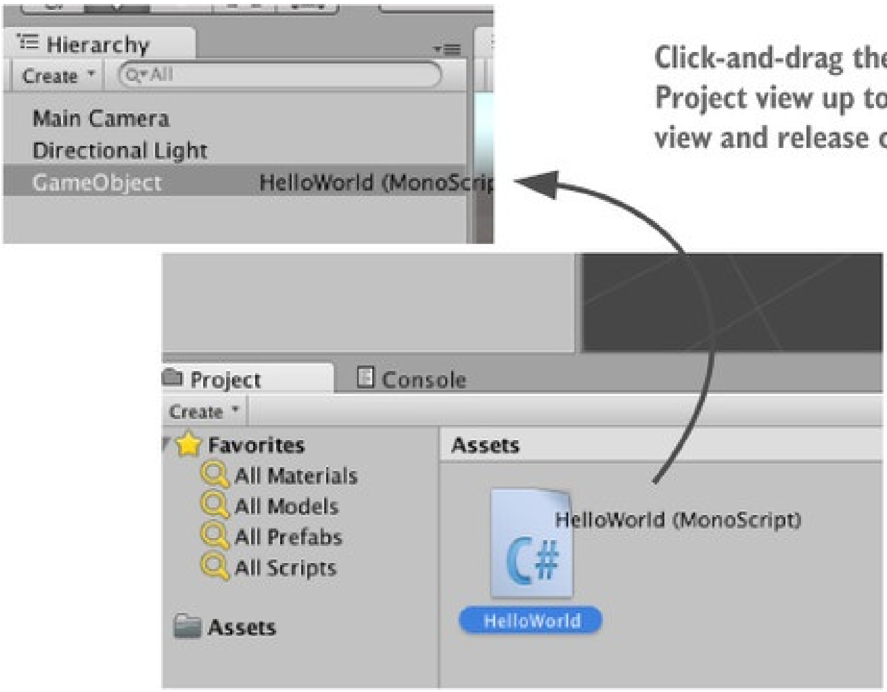
Always keep in mind that although the code is written in MonoDevelop, the code isn’t actually run there. The IDE is pretty much a fancy text editor, and the code is run when you hit Play within Unity.

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 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.

Figure 1.15. How to link a script to a GameObject



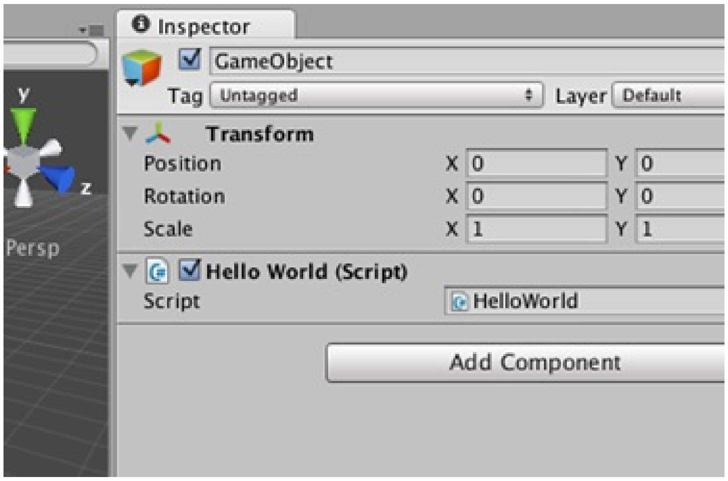
Click-and-drag the script from the Project view up to the Hierarchy view and release on the GameObject.

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



☑ GameObject

Tag Untagged Layer Default

Transform

Position X 0 Y 0

Rotation X 0 Y 0

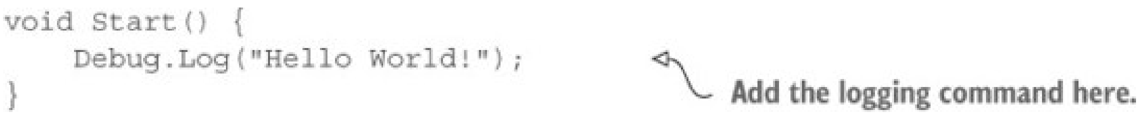
Scale X 1 Y 1

☑ Hello World (Script)

Script HelloWorld

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:

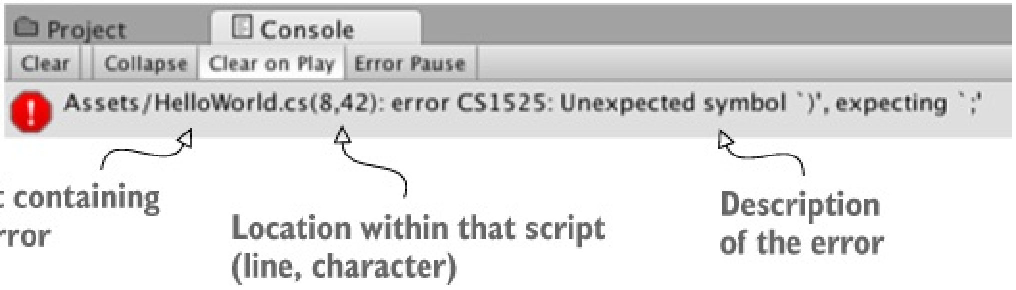
1. Create a new project.

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.

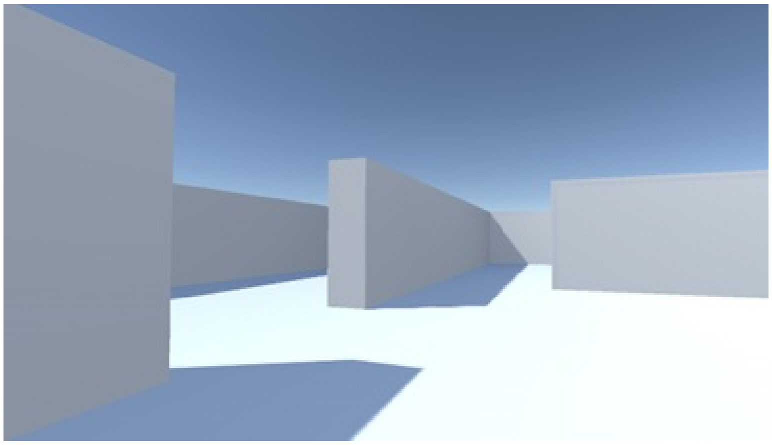
Chapter 2. Building a demo that puts you in 3D space

This chapter covers

- Understanding 3D coordinate space
- Putting a player in a scene
- Writing a script that moves objects
- Implementing FPS controls

Chapter 1 concluded with the traditional “Hello World!” introduction to a new programming tool; now it’s time to dive into a nontrivial Unity project, a project with interactivity and graphics. You’ll put some objects into a scene and write code to enable a player to walk around that scene. Basically, it’ll be Doom without the monsters (something like what figure 2.1 depicts). The visual editor in Unity enables new users to start assembling a 3D prototype right away, without needing to write a lot of boilerplate code first (for things like initializing a 3D view or establishing a rendering loop).

Figure 2.1. Screenshot of the 3D demo (basically, Doom without the monsters)



It’s tempting to immediately start building the scene in Unity, especially with such a simple (in concept!) project. But it’s always a good idea to pause at the beginning and plan out what you’re going to do, and this is especially important right now because you’re new to the process.

2.1. Before you start...

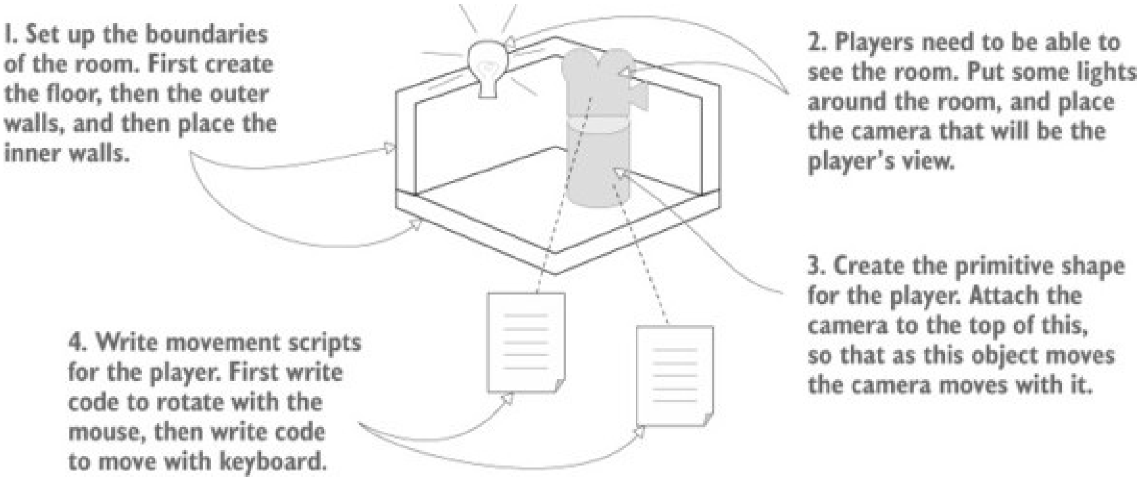
Unity makes it easy for a newcomer to get started, but let’s go over a couple of points before you build the complete scene. Even when working with a tool as flexible as Unity, you do need to have some sense of the goal you’re working toward. You also need a grasp of how 3D coordinates operate or you could get lost as soon as you try to position an object in the scene.

2.1.1. Planning the project

Before you start programming anything, you always want to pause and ask yourself, “So what am I building here?” Game design is a huge topic unto itself, with many impressively large books focused on how to design a game. Fortunately for our purposes, you only need a brief outline of this simple demo in mind in order to develop a basic learning project. These initial projects won’t be terribly complex designs anyway, in order to avoid distracting you from learning programming concepts; you can (and should!) worry about higher-level design issues after you’ve mastered the fundamentals of game development.

For this first project you’ll build a basic FPS (first-person shooter) scene. There will be a room to navigate around, players will see the world from their character’s point of view, and the player can control the character using the mouse and keyboard. All the interesting complexity of a complete game can be stripped away for now in order to concentrate on the core mechanic: moving around in a 3D space. Figure 2.2 depicts the roadmap for this project, basically laying out the mental checklist I built in my head:

Figure 2.2. Roadmap for the 3D demo


1. Set up the boundaries of the room. First create the floor, then the outer walls, and then place the inner walls.
2. Players need to be able to see the room. Put some lights around the room, and place the camera that will be the player's view.
3. Create the primitive shape for the player. Attach the camera to the top of this, so that as this object moves the camera moves with it.
4. Write movement scripts for the player. First write code to rotate with the mouse, then write code to move with keyboard.

1. Set up the room: create the floor, outer walls, and inner walls.

2. Place the lights and camera.

3. Create the player object (including attaching the camera on top).

4. Write movement scripts: rotate with the mouse and move with the keyboard.

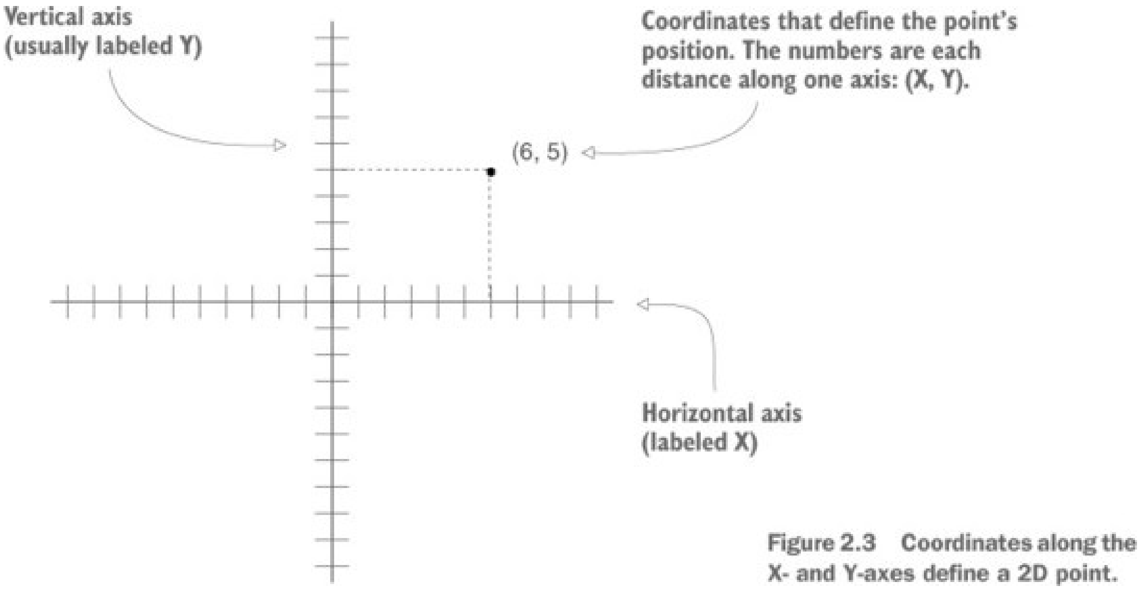
Don’t be scared off by everything in this roadmap! It sounds like there’s a lot in this chapter, but Unity makes it easy. The upcoming sections about movement scripts are so extensive only because we’ll be going through every line to understand all the concepts in detail. This project is a first-person demo in order to keep the art requirements simple; because you can’t see yourself, it’s fine for “you” to be a cylindrical shape with a camera on top! Now you just need to understand how 3D coordinates work, and it will be easy to place everything in the visual editor.

2.1.2. Understanding 3D coordinate space

If you think about the simple plan we’re starting with, there are three aspects to it: a room, a view, and controls. All of those items rely on you understanding how positions and movements are represented in 3D computer simulations, and if you’re new to working with 3D graphics you might not already know that stuff.

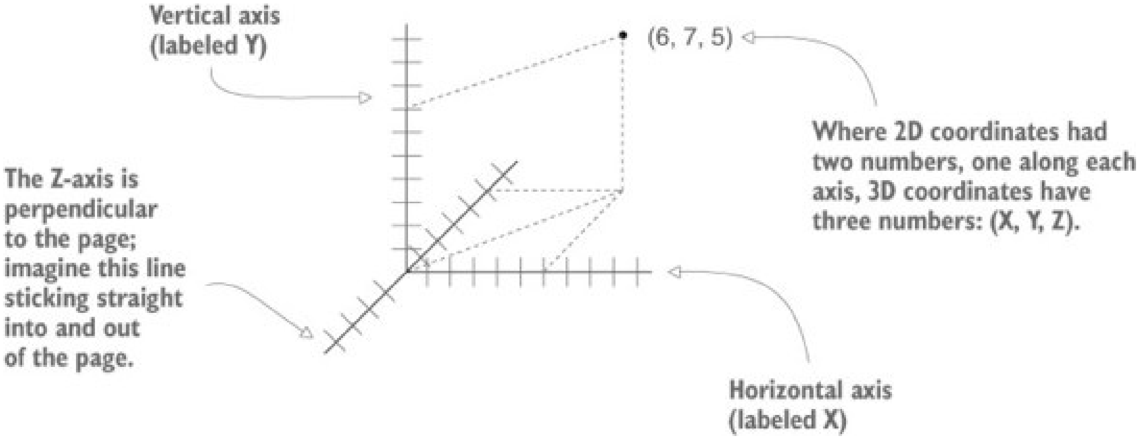
It all boils down to numbers that indicate points in space, and the way those numbers correlate to the space is through coordinate axes. If you think back to math class, you’ve probably seen and used X-and Y-axes (see figure 2.3) for assigning coordinates to points on the page, which is referred to as a Cartesian coordinate system.

Figure 2.3. Coordinates along the X-and Y-axes define a 2D point.


Vertical axis
(usually labeled Y)
Coordinates that define the point’s position. The numbers are each distance along one axis: (X, Y).
(6, 5)
Horizontal axis
(labeled X)
Figure 2.3 Coordinates along the X- and Y-axes define a 2D point.

Two axes give you 2D coordinates, with all points in the same plane. Three axes are used to define 3D space. Because the X-axis goes along the page horizontally and the Y-axis goes along the page vertically, we now imagine a third axis that sticks straight into and out of the page, perpendicular to both the X and Y axes. Figure 2.4 depicts the X-, Y-, and Z-axes for 3D coordinate space. Everything that has a specific position in the scene will have XYZ coordinates: position of the player, placement of a wall, and so forth.

Figure 2.4. Coordinates along the X-, Y-, and Z-axes define a 3D point.

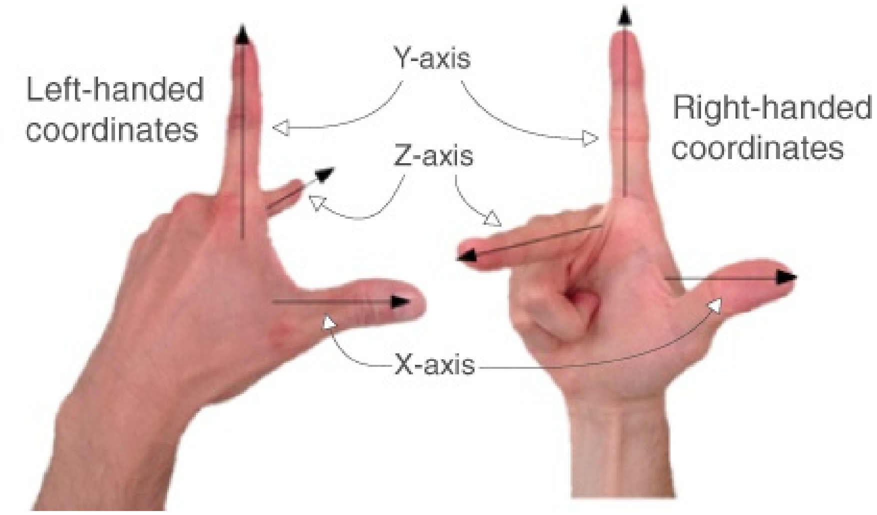

Vertical axis
(labeled Y)
(6, 7, 5)
Where 2D coordinates had two numbers, one along each axis, 3D coordinates have three numbers: (X, Y, Z).
The Z-axis is perpendicular to the page; imagine this line sticking straight into and out of the page.
Horizontal axis
(labeled X)

In Unity’s Scene view you can see these three axes displayed, and in the Inspector you can type in the three numbers to position an object. Not only will you write code to position objects using these three-number coordinates, but you can also define movements as a distance to move along each axis.

Left-handed vs. right-handed coordinates

The positive and negative direction of each axis is arbitrary, and the coordinates still work no matter which direction the axes point. You simply need to stay consistent within a given 3D graphics tool (animation tool, game development tool, and so forth).

But in almost all cases X goes to the right and Y goes up; what differs between different tools is whether Z goes into or comes out of the page. These two directions are referred to as “left-handed” or “right-handed”; as this figure shows, if you point your thumb along the X-axis and your index finger along the Y-axis, then your middle finger points along the Z-axis.



The Z-axis points in a different direction on the left hand versus the right hand.

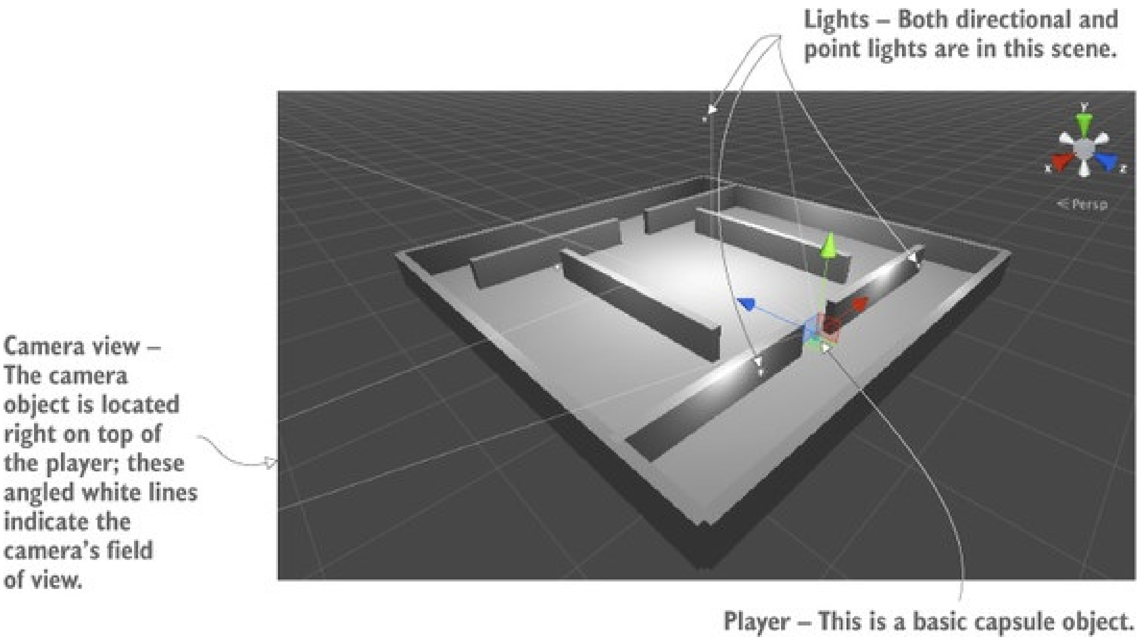
Unity uses a left-handed coordinate system, as do many 3D art applications. Many other tools use right-handed coordinate systems (OpenGL, for example), so don’t get confused if you ever see different coordinate directions.

Now that you have a plan in mind for this project and you know how coordinates are used to position objects in 3D space, it’s time to start building the scene.

2.2. Begin the project: place objects in the scene

All right, let’s create and place objects in the scene. First you’ll set up all the static scenery—the floor and walls. Then you’ll place lights around the scene and position the camera. Last you’ll create the object that will be the player, the object to which you’ll attach scripts to walk around the scene. Figure 2.5 shows what the editor will look like with everything in place.

Figure 2.5. Scene in the Editor with floor, walls, lights, a camera, and the player



Lights – Both directional and point lights are in this scene.

Camera view – The camera object is located right on top of the player; these angled white lines indicate the camera's field of view.

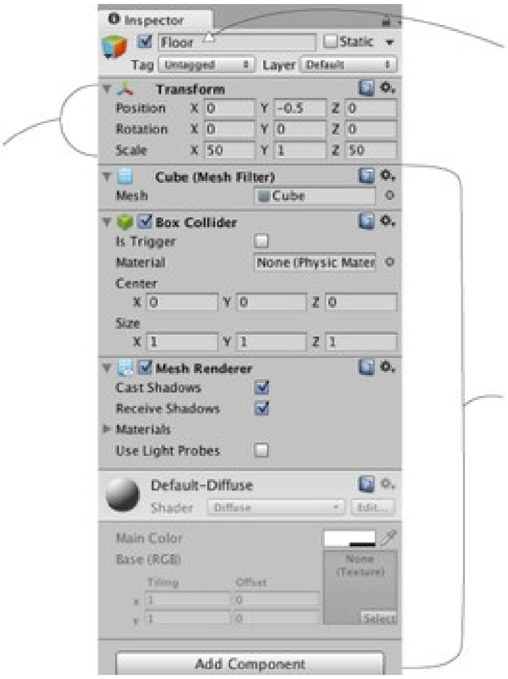
Player – This is a basic capsule object.

Chapter 1 showed how to create a new project in Unity, so you'll do that now. Remember: Choose File > New Project and then name your new project in the window that pops up. After creating the new project, immediately save the current empty default scene, because the project doesn't have any Scene file initially. The scene starts out empty, and the first objects to create are the most obvious ones.

2.2.1. The scenery: floor, outer walls, inner walls

Select the GameObject menu at the top of the screen, and then hover over 3D Object to see that drop-down menu. Select Cube to create a new cube object in the scene (later we'll use other shapes like Sphere and Capsule). Adjust the position and scale of this object, as well as its name, in order to make the floor; figure 2.6 shows what values the floor should be set to in the Inspector (it's only a cube initially, before you stretch it out).

Figure 2.6. Inspector view for the floor



☑ Floor □ Static
Tag Untagged Layer Default

Transform
Position X 0 Y -0.5 Z 0
Rotation X 0 Y 0 Z 0
Scale X 50 Y 1 Z 50

Cube (Mesh Filter)
Mesh Cube

☑ Box Collider
Is Trigger □
Material None (Physic Mater
Center X 0 Y 0 Z 0
Size X 1 Y 1 Z 1

☑ Mesh Renderer
Cast Shadows ☑
Receive Shadows ☑
Materials
Use Light Probes □

Default-Diffuse
Shader Diffuse Edit...
Main Color
Base (RGB) None (Texture) Select
Tiling Offset
x 1 0
y 1 0

1. At the top you can type in a name for the object. For example, call the floor object "Floor."

2. Position and scale the cube in order to create a floor for the room. Or rather "cube," since it won't look like a cube anymore after being stretched out with differing scale values on different axes.

Meanwhile the position is lowered very slightly to compensate for the height; we set the Y scale to 1, and the object is positioned around its center.

The remaining components filling the view come with a new Cube object but don't need to be adjusted right now. These components include a Mesh Filter (to define the geometry of the object), a Mesh Renderer (to define the material on the object), and a Box Collider (so that the object can be collided with during movement).

Note

The numbers for position can be any units you want, as long as you're consistent throughout the scene. The most common choice for units is meters and that's what I generally choose, but I also use feet sometimes and I've even seen other people decide that the numbers are inches!

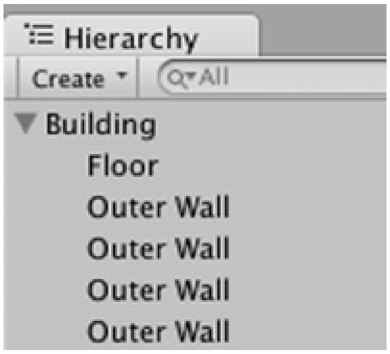
Repeat the same steps in order to create outer walls for the room. You can create new cubes each time, or you can copy and paste existing objects using the standard shortcuts. Move, rotate, and scale the walls to form a perimeter around the floor, as shown in figure 2.5. Experiment with different numbers (for example, 1, 4, 50 for scale) or use the transform tools first seen in section 1.2.2 (remember that the mathematical term for moving and rotating in 3D space is "transform").

Tip

Also recall the navigation controls so that you can view the scene from different angles or zoom out for a bird's-eye view. If you ever get lost in the scene, press F to reset the view on the currently selected object.

The exact transform values the walls end up with will vary depending on how you rotate and scale the cubes in order to fit, and on how the objects are linked together in the Hierarchy view. For example, in figure 2.7 the walls are all children of an empty root object, so that the Hierarchy list will look organized. If you need an example to copy working values from, download the sample project and refer to the walls there.

Figure 2.7. The Hierarchy view showing the walls and floor organized under an empty object



Tip

Drag objects on top of each other in the Hierarchy view to establish linkages. Objects that have other objects attached are referred to as *parent*; objects attached to other objects are referred to as *children*. When the parent object is moved (or rotated or scaled), the child objects are transformed along with it.

Tip

Empty game objects can be used to organize the scene in this way. By linking visible objects to a root object, their Hierarchy list can be collapsed. Be warned: before linking any child objects to it, you want to position the empty root object at 0, 0, 0 to avoid any positioning oddities later.

What is `GameObject`?

All scene objects are instances of the class `GameObject`, similar to how all script components inherit from the class `MonoBehaviour`. This fact was more explicit with the empty object actually named `GameObject` but is still true regardless of whether the object is named `Floor`, `Camera`, or `Player`.

`GameObject` is really just a container for a bunch of components. The main purpose of `GameObject` is so that `MonoBehaviour` has something to attach to. What exactly the object is in the scene depends on what components have been added to that `GameObject`. `Cube` objects have a Cube component, `Sphere` objects have a Sphere component, and so on.

Once the outer walls are in place, create some inner walls to navigate around. Position the inner walls however you like; the idea is to create some hallways and obstacles to walk around once you write code for movement.

Now the scene has a room in it, but without any lights the player won't be able to see any of it. Let's take care of that next.

2.2.2. Lights and cameras

Typically you light a 3D scene with a directional light and then a series of point lights. First start with a directional light; the scene probably already has one by default, but if not then create one by choosing GameObject > Light and selecting Directional Light.

Types of lights

You can create several types of light sources, defined by how and where they project light rays. The three main types are point, spot, and directional.

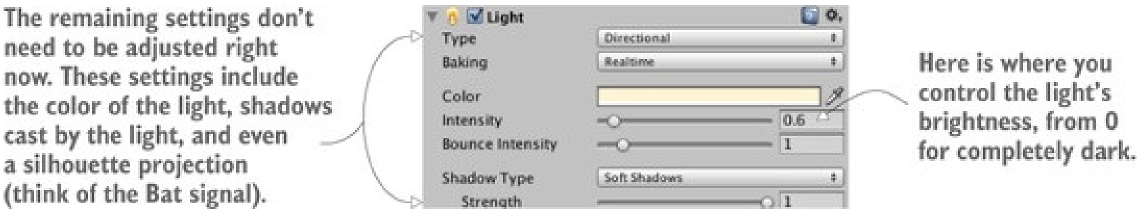
Point lights are a kind of light source where all the light rays originate from a single point and project out in all directions, like a lightbulb in the real world. The light is brighter up close because the light rays are bunched up.

Spot lights are a kind of light source where all the light rays originate from a single point but only project out in a limited cone. No spot lights are used in the current project, but these lights are commonly used to highlight parts of a level.

Directional lights are a kind of light source where all the light rays are parallel and project evenly, lighting everything in the scene the same way. This is like the sun in the real world.

The position of a directional light doesn’t affect the light cast from it, only the rotation the light source is facing, so technically you could place that light anywhere in the scene. I recommend placing it high above the room so that it intuitively feels like the sun and so that it’s out of the way when you’re manipulating the rest of the scene. Rotate this light and watch the effect on the room; I recommend rotating it slightly on both the X-and Y-axes to get a good effect. You can see an Intensity setting when you look in the Inspector (see figure 2.8). As the name implies, that setting controls the brightness of the light. If this were the only light, it’d have to be more intense, but because you’ll add a bunch of point lights as well, this directional light can be pretty dim, like 0.6 Intensity.

Figure 2.8. Directional light settings in the Inspector



☑ Light

Type: Directional

Baking: Realtime

Color

Intensity: 0.6

Bounce Intensity: 1

Shadow Type: Soft Shadows

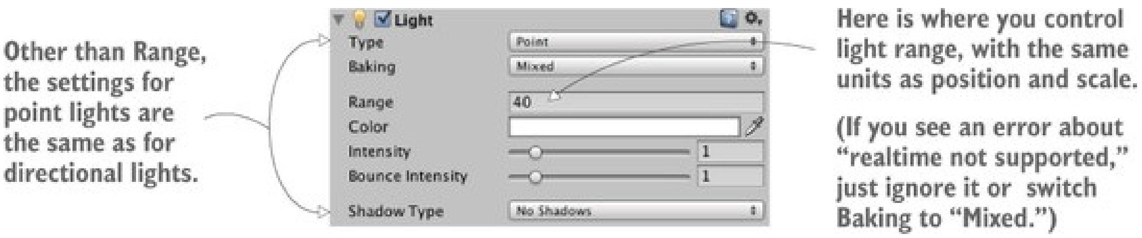
Strength: 1

The remaining settings don’t need to be adjusted right now. These settings include the color of the light, shadows cast by the light, and even a silhouette projection (think of the Bat signal).

Here is where you control the light’s brightness, from 0 for completely dark.

As for point lights, create several using the same menu and place them around the room in dark spots in order to make sure all the walls are lit. You don’t want too many (performance will degrade if the game has lots of lights), but one near each corner should be fine (I suggest raising them to the tops of the walls), plus one placed high above the scene (like a Y of 18) to give some variety to the light in the room. Note that point lights have a setting for Range added to the Inspector (see figure 2.9). This controls how far away the light reaches; whereas directional lights cast light evenly throughout the entire scene, point lights are brighter when an object is closer. The point lights lower to the floor should have a range around 18, but the light placed high up should have a range of around 40 in order to reach the entire room.

Figure 2.9. Point light settings in the Inspector



Light
Type: Point
Baking: Mixed
Range: 40
Color
Intensity: 1
Bounce Intensity: 1
Shadow Type: No Shadows

Other than Range, the settings for point lights are the same as for directional lights.

Here is where you control light range, with the same units as position and scale.

(If you see an error about "realtime not supported," just ignore it or switch Baking to "Mixed.")

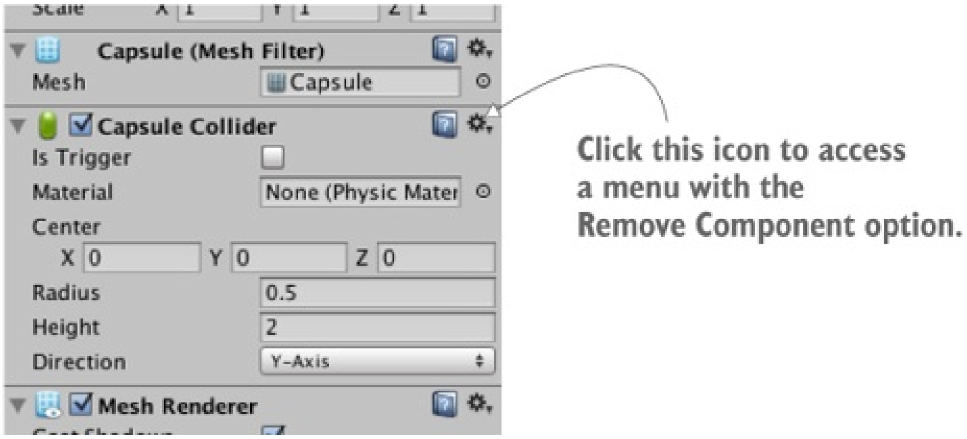
The other kind of object needed in order for the player to see the scene is a camera, but the "empty" scene already came with a main camera, so you'll use that. If you ever need to create new cameras (such as for split-screen views in multiplayer games), Camera is another choice in the same GameObject menu as Cube and Lights. The camera will be positioned around the top of the player so that the view appears to be the player's eyes.

2.2.3. The player's collider and viewpoint

For this project, a simple primitive shape will do to represent the player. In the GameObject menu (remember, hover over 3D Object to expand the menu) click Capsule. Unity creates a cylindrical shape with rounded ends; this primitive shape will represent the player. Position this object at 1.1 on the Y-axis (half the height of the object, plus a bit to avoid overlapping the floor). You can move the object on X and Z wherever you like, as long as it's inside the room and not touching any walls. Name the object `Player`.

In the Inspector you'll notice that this object has a capsule collider assigned to it. That's a logical default choice for a capsule object, just like cube objects had a box collider by default. But this particular object will be the player and thus needs a slightly different sort of component than most objects. Remove the capsule collider by clicking the gear icon toward the top-right of that component, shown in figure 2.10; that will display a menu that includes the option Remove Component. The collider is a green mesh surrounding the object, so you'll see the green mesh disappear after deleting the capsule collider.

Figure 2.10. Removing a component in the Inspector



Scale X 1 Y 1 Z 1

Capsule (Mesh Filter)

Mesh: Capsule

☑ Capsule Collider

Is Trigger □

Material: None (Physic Mater

Center: X 0 Y 0 Z 0

Radius: 0.5

Height: 2

Direction: Y-Axis

☑ Mesh Renderer

Click this icon to access a menu with the Remove Component option.

Instead of a capsule collider we're going to assign a character controller to this object. At the bottom of the Inspector there's a button labeled Add Component; click that button to open a menu of components that you can add. In the Physics section of this menu you'll find Character Controller; select that option. As the name implies, this component will allow the object to behave like a character.

You need to complete one last step to set up the player object: attaching the camera. As mentioned in the earlier section on floors and walls, objects can be dragged onto each other in the Hierarchy view. Drag the camera object onto the player capsule to attach the camera to the player. Now position the camera so that it'll look like the player's eyes (I suggest a position of 0, 0.5, 0). If necessary, reset the camera's rotation to 0, 0, 0 (this will be off if you rotated the capsule).

You've created all the objects needed for this scene. What remains is writing code to move the player object.

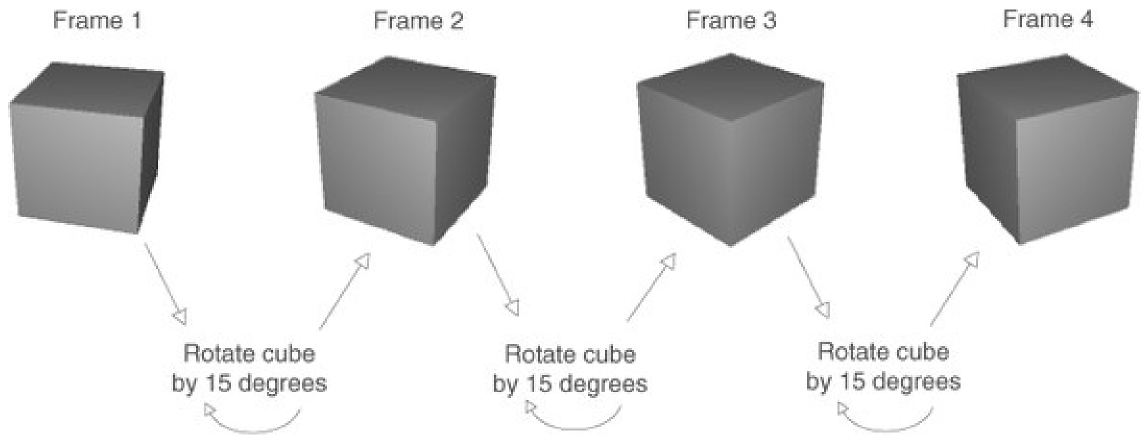
2.3. Making things move: a script that applies transforms

To have the player walk around the scene, you'll write movement scripts attached to the player. Remember, components are modular bits of functionality that you add to objects, and scripts are a kind of component. Eventually those scripts will respond to keyboard and mouse input, but first just make the player spin in place. This beginning will teach you how to apply transforms in code. Remember that the three transforms are Translate, Rotate, and Scale; spinning an object means changing the rotation. But there's more to know about this task than just "this involves rotation."

2.3.1. Diagramming how movement is programmed

Animating an object (such as making it spin) boils down to moving it a small amount every frame, with the frames playing over and over. By themselves transforms apply instantly, as opposed to visibly moving over time. But applying the transforms over and over causes the object to visibly move, just like a series of still drawings in a flipbook. Figure 2.11 diagrams how this works.

Figure 2.11. The appearance of movement: cyclical process of transforming between still pictures

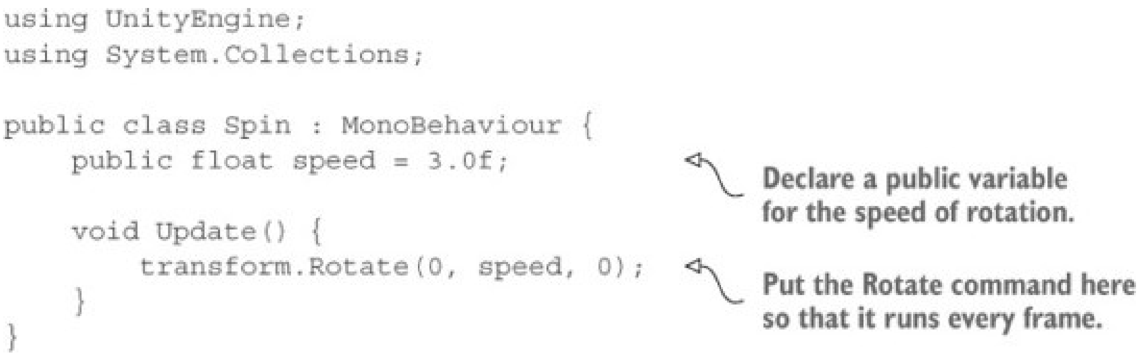

Frame 1
Frame 2
Frame 3
Frame 4
Rotate cube by 15 degrees
Rotate cube by 15 degrees
Rotate cube by 15 degrees

Recall that script components have an `Update()` method that runs every frame. To spin the cube, add code inside `Update()` that rotates the cube a small amount. This code will run over and over every frame. Sounds pretty simple, right?

2.3.2. Writing code to implement the diagram

Now let's put in action the concepts just discussed. Create a new C# script (remember it's in the Create submenu of the Assets menu), name it Spin, and write in the code from the following listing (don't forget to save the file after typing in it!).

Listing 2.1. Making the object spin



```
using UnityEngine;
using System.Collections;

public class Spin : MonoBehaviour {
    public float speed = 3.0f;

    void Update() {
        transform.Rotate(0, speed, 0);
    }
}
```

Declare a public variable for the speed of rotation.

Put the Rotate command here so that it runs every frame.

To add the script component to the player object, drag the script up from the Project view and drop it onto Player in the Hierarchy view. Now hit Play and you'll see the view spin around; you've written code to make an object move! This code is pretty much the default template for a new script plus two new added lines, so let's examine what those two lines do.

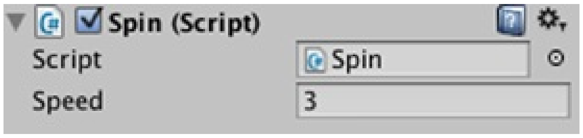
First there's the variable for speed added toward the top of the class definition. There are two reasons for defining the rotation speed as a variable: one is the standard "no magic numbers" programming rule, and the second reason is specific to how Unity displays public variables. Unity does something handy with public variables in script components, as described in the following tip.

Tip

Public variables are exposed in the Inspector so that you can adjust the component's values after adding a component to a game object. This is referred to as "serializing" the value, because Unity saves the modified state of the variable.

Figure 2.12 shows what the script component looks like in the Inspector. You can type in a new number, and then the script will use that value instead of the default value defined in the code. This is a handy way to adjust settings for the component on different objects, working within the visual editor instead of hardcoding every value.

Figure 2.12. The Inspector displaying a public variable declared in the script



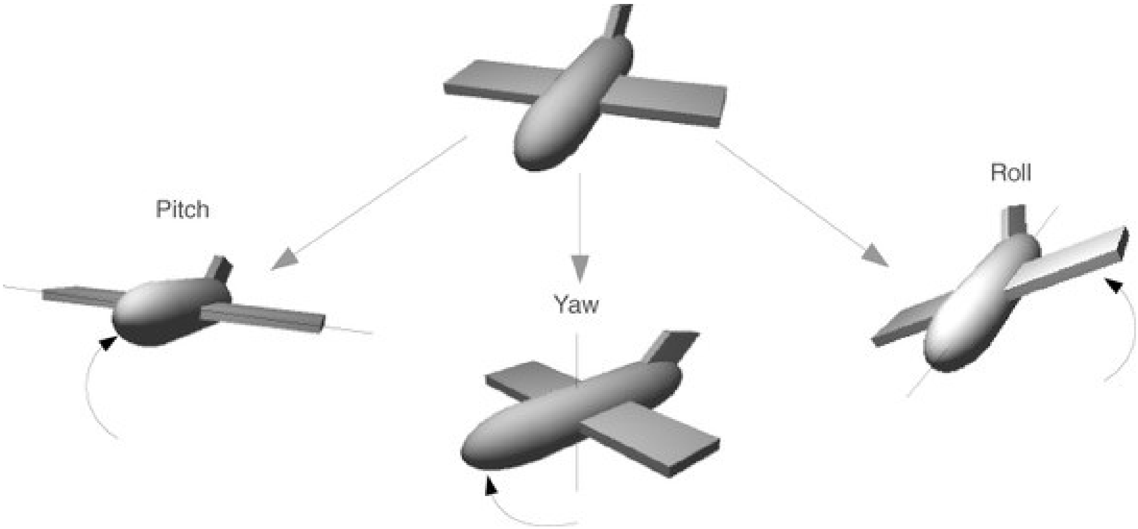
Script Spin

Speed 3

The second line to examine from listing 2.1 is the `Rotate()` method. That's inside `Update()` so that the command runs every frame. `Rotate()` is a method of the `Transform` class, so it's called with dot notation through the transform component of this object (as in most object-oriented languages, `this.transform` is implied if you type `transform`). The transform is rotated by `speed` degrees every frame, resulting in a smooth spinning movement. But why are the parameters to `Rotate()` listed as (0, speed, 0) as opposed to, say, (speed, 0, 0)?

Recall that there are three axes in 3D space, labeled X, Y, and Z. It's fairly intuitive to understand how these axes relate to positions and movements, but these axes can also be used to describe rotations. Aeronautics describes rotations in a similar way, so programmers working with 3D graphics often use a set of terms borrowed from aeronautics: pitch, yaw, and roll. Figure 2.13 illustrates what these terms mean; pitch is rotation around the X-axis, yaw is rotation around the Y-axis, and roll is rotation around the Z-axis.

Figure 2.13. Illustration of pitch, yaw, and roll rotation of an aircraft



Given that we can describe rotations around the X-, Y-, and Z-axes, that means the three parameters for `Rotate()` are X, Y, and Z rotation. Because we only want the player to spin around sideways, as opposed to tilting up and down, there should only be a number given for the Y rotation, and just 0 for X and Z rotation. Hopefully you can guess what will happen if you change the parameters to (speed, 0, 0) and then play it; try that now!

There's one other subtle point to understand about rotations and 3D coordinate axes, embodied in an optional fourth parameter to the `Rotate()` method.

2.3.3. Local vs. global coordinate space

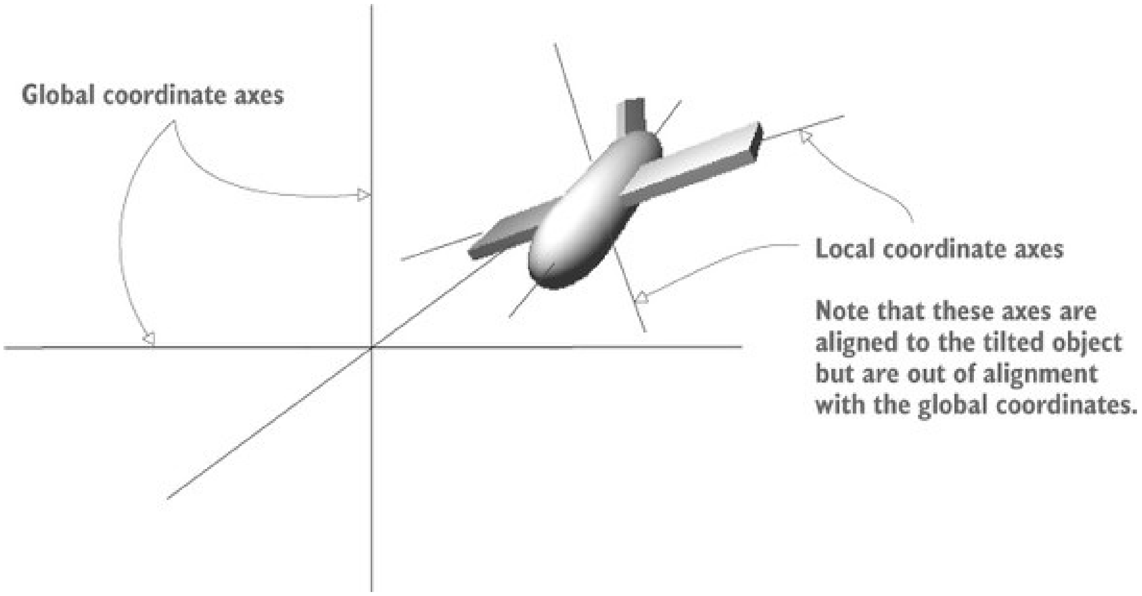
By default, the `Rotate()` method operates on what are called local coordinates. The other kind of coordinates you could use are global. You tell the method whether to use local or global coordinates using an optional fourth parameter by writing either `Space.Self` or `Space.World` like so:

```
Rotate(0, speed, 0, Space.World)
```

Refer back to the explanation about 3D coordinate space, and ponder these questions: Where is (0, 0, 0) located? What direction is the X-axis pointed in? Can the coordinate system itself move around?

It turns out that every single object has its own origin point, as well as its own direction for the three axes, and this coordinate system moves around with the object. This is referred to as local coordinates. The overall 3D scene also has its own origin point and its own direction for the three axes, and this coordinate system never moves. This is referred to as global coordinates. Thus, when you specify local or global for the `Rotate()` method, you're telling it whose X-, Y-, and Z-axes to rotate around (see figure 2.14).

Figure 2.14. Local vs. global coordinate axes


Global coordinate axes
Local coordinate axes
Note that these axes are aligned to the tilted object but are out of alignment with the global coordinates.

If you’re new to 3D graphics, this is somewhat of a mind-bending concept. The different axes are depicted in figure 2.14 (notice how “left” to the plane is a different direction than “left” to the world) but the easiest way to understand local and global is through an example.

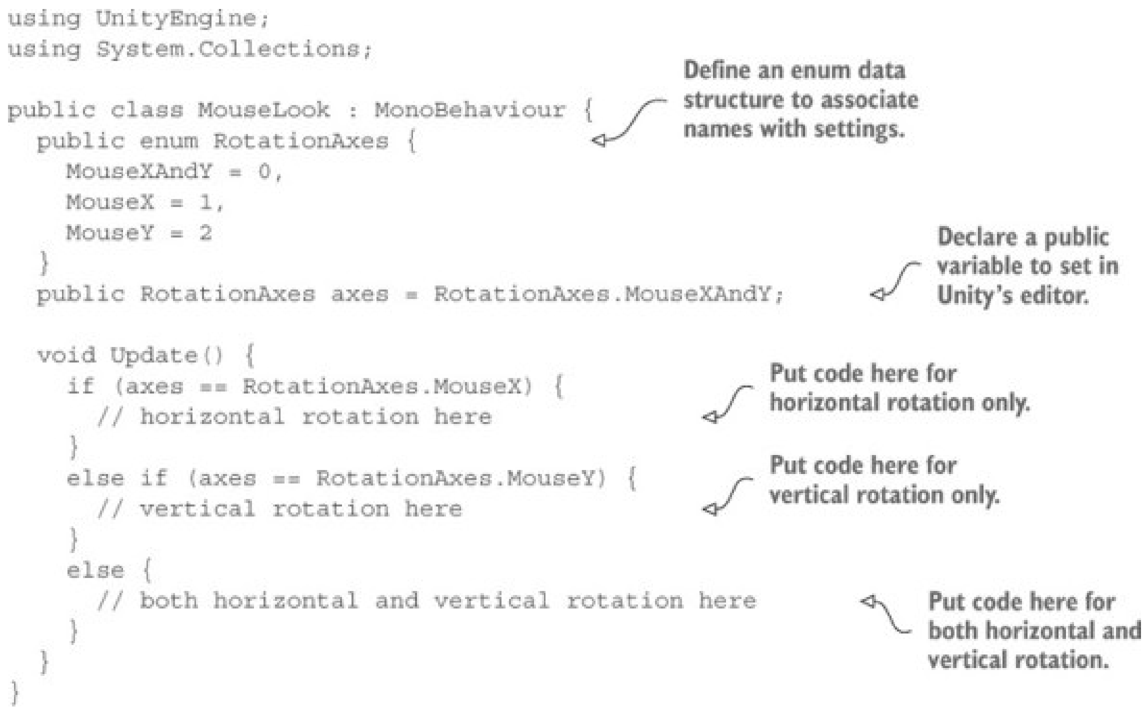
First, select the player object and then tilt it a bit (something like 30 for X rotation). This will throw off the local coordinates, so that local and global rotations will look different. Now try running the Spin script both with and without `Space.World` added to the parameters; if it’s too hard for you to visualize what’s happening, try removing the spin component from the player object and instead spin a tilted cube placed in front of the player. You’ll see the object rotating around different axes when you set the command to local or global coordinates.

2.4. Script component for looking around: MouseLook

Now you’ll make rotation respond to input from the mouse (that is, rotation of the object this script is attached to, which in this case will be the player). You’ll do this in several steps, progressively adding new movement abilities to the character. First the player will only rotate side to side, and then the player will only rotate up and down. Eventually the player will be able to look around in all directions (rotating horizontally and vertically at the same time), a behavior referred to as *mouse-look*.

Given that there will be three different types of rotation behavior (horizontal, vertical, and both), you’ll start by writing the framework for supporting all three. Create a new C# script, name it MouseLook, and write in the code from the next listing.

Listing 2.2. MouseLook framework with enum for the Rotation setting



```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {
  public enum RotationAxes {
    MouseXAndY = 0,
    MouseX = 1,
    MouseY = 2
  }
  public RotationAxes axes = RotationAxes.MouseXAndY;

  void Update() {
    if (axes == RotationAxes.MouseX) {
      // horizontal rotation here
    }
    else if (axes == RotationAxes.MouseY) {
      // vertical rotation here
    }
    else {
      // both horizontal and vertical rotation here
    }
  }
}
```

Define an enum data structure to associate names with settings.

Declare a public variable to set in Unity’s editor.

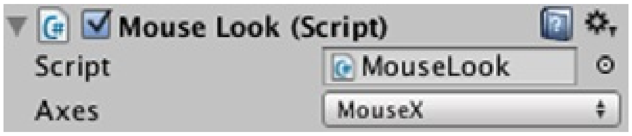
Put code here for horizontal rotation only.

Put code here for vertical rotation only.

Put code here for both horizontal and vertical rotation.

Notice that an enum is used to choose horizontal or vertical rotation for the MouseLook script. Defining an enum data structure allows you to set values by name, rather than typing in numbers and trying to remember what each number means (is 0 horizontal rotation? Is it 1?). If you then declare a public variable typed to that enum, that will display in the Inspector as a drop-down menu (see figure 2.15), which is useful for selecting settings.

Figure 2.15. The Inspector displays public enum variables as a drop-down menu.



☑ Mouse Look (Script)

Script MouseLook

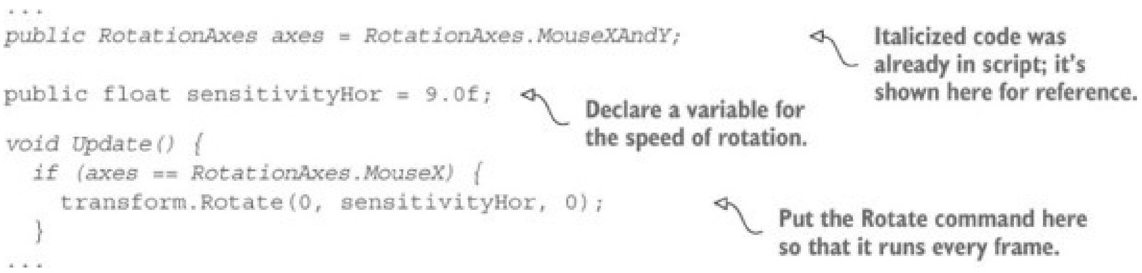
Axes MouseX

Remove the Spin component (the same way you removed the capsule collider earlier) and attach this new script to the player object instead. Use the Axes menu to switch between code branches while working through the code. With the horizontal/vertical rotation setting in place, you can fill in code for each branch of the conditional.

2.4.1. Horizontal rotation that tracks mouse movement

The first and simplest branch is horizontal rotation. Start by writing the same rotation command you used in listing 2.1 to make the object spin. Don't forget to declare a public variable for the rotation speed; declare the new variable after `axes` but before `Update()`, and call the variable `sensitivityHor` because speed is too generic a name once you have multiple rotations involved. Increase the value of the variable to 9 this time because that value needs to be bigger once the code starts scaling it (which will be soon). The adjusted code should look like the following listing.

Listing 2.3. Horizontal rotation, not yet responding to the mouse



```
...
public RotationAxes axes = RotationAxes.MouseXAndY;

public float sensitivityHor = 9.0f;

void Update() {
  if (axes == RotationAxes.MouseX) {
    transform.Rotate(0, sensitivityHor, 0);
  }
...
```

Italicized code was already in script; it's shown here for reference.

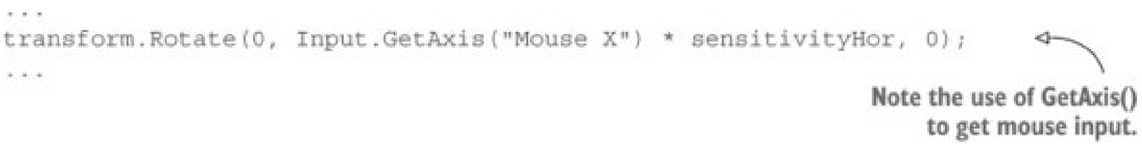
Declare a variable for the speed of rotation.

Put the Rotate command here so that it runs every frame.

If you play the script now, the view will spin around just like before (only a lot faster, because the Y rotation is 9 instead of 3). The next step is to make the rotation react to mouse movement, so let's introduce a new method: `Input.GetAxis()`. The `Input` class has a bunch of methods for handling input devices (such as the mouse) and the method `GetAxis()` returns numbers correlated to the movement of the mouse (positive or negative, depending on the direction of movement). `GetAxis()` takes the name of the axis desired as a parameter, and the horizontal axis is called Mouse X.

If you multiply the rotation speed by the axis value, the rotation will respond to mouse movement. The speed will scale according to mouse movement, scaling down to zero or even reversing direction. The `Rotate` command now looks like the next listing.

Listing 2.4. `Rotate` command adjusted to respond to the mouse



```
...
transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
...
```

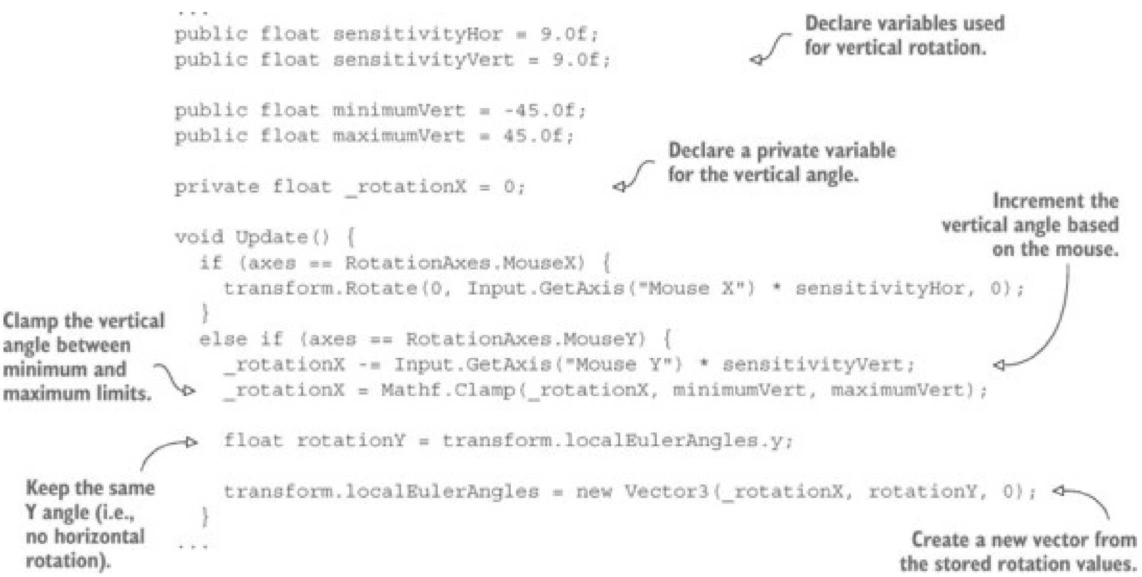
Note the use of GetAxis() to get mouse input.

Hit Play and then move the mouse around. As you move the mouse from side to side, the view will rotate from side to side. That’s pretty cool! The next step is to rotate vertically instead of horizontally.

2.4.2. Vertical rotation with limits

For horizontal rotation we’ve been using the `Rotate()` method, but we’ll take a different approach with vertical rotation. Although that method is convenient for applying transforms, it’s also kind of inflexible. It’s only useful for incrementing the rotation without limit, which was fine for horizontal rotation, but vertical rotation needs limits on how much the view can tilt up or down. The following listing shows the vertical rotation code for MouseLook; a detailed explanation of the code will come right after.

Listing 2.5. Vertical rotation for MouseLook



```
...
public float sensitivityHor = 9.0f;
public float sensitivityVert = 9.0f;

public float minimumVert = -45.0f;
public float maximumVert = 45.0f;

private float _rotationX = 0;

void Update() {
  if (axes == RotationAxes.MouseX) {
    transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivityHor, 0);
  }
  else if (axes == RotationAxes.MouseY) {
    _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
    _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

    float rotationY = transform.localEulerAngles.y;

    transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
  }
...
```

Declare variables used for vertical rotation.

Declare a private variable for the vertical angle.

Increment the vertical angle based on the mouse.

Clamp the vertical angle between minimum and maximum limits.

Keep the same Y angle (i.e., no horizontal rotation).

Create a new vector from the stored rotation values.

Set the Axes menu of the MouseLook component to vertical rotation and play the new script. Now the view won’t rotate sideways, but it’ll tilt up and down when you move the mouse up and down. The tilt stops at upper and lower limits.

There are several new concepts in this code that need to be explained. First off, we’re not using `Rotate()` this time, so we need a variable (called `_rotationX` here, because vertical rotation goes around the X-axis) in which to store the rotation angle. The `Rotate()` method increments the current rotation, whereas this code sets the rotation angle directly. In other words, it’s the difference between saying “add 5 to the angle” and “set the angle to 30.” We do still need to increment the rotation angle, but that’s why the code has the -= operator: to subtract a value from the rotation angle, rather than set the angle to that value. By not using `Rotate()` we can manipulate the rotation angle in various ways aside from only incrementing it. The rotation value is multiplied by `Input.GetAxis()` just like in the code for horizontal rotation, except now we ask for Mouse Y because that’s the vertical axis of the mouse.

The rotation angle is manipulated further on the very next line. We use `Mathf.Clamp()` to keep the rotation angle between minimum and maximum limits. Those limits are public variables declared earlier in the code, and they ensure that the view can only tilt 45 degrees up or down. The `Clamp()` method isn’t specific to rotation, but is generally useful for keeping a number variable between limits. Just to see what happens, try commenting out the `Clamp()` line; now the tilt doesn’t stop at upper and lower limits, allowing you to even rotate completely upside down! Clearly, viewing the world upside down is undesirable; hence the limits.

Because the angles property of `transform` is a Vector3, we need to create a new Vector3 with the rotation angle passed in to the constructor. The `Rotate()` method was automating this process for us, incrementing the rotation angle and then creating a new vector.

Definition

A *vector* is multiple numbers stored together as a unit. For example, a Vector3 is 3 numbers (labeled x, y, z).

Warning

The reason why we need to create a new Vector3 instead of changing values in the existing vector in the transform is because those values are read-only for transforms. This is a common mistake that can trip you up.

Euler angles vs. quaternion

You’re probably wondering why the property is called `localEulerAngles` and not `localRotation`. First you need to know about a concept called *quaternions*.

Quaternions are a different mathematical construct for representing rotations. They’re distinct from Euler angles, which is the name for the X-, Y-, Z-axes approach we’ve been taking. Remember the whole discussion of pitch, yaw, and roll? Well, that method of representing rotations is Euler angles. Quaternions are...different. It’s hard to explain what quaternions are, because they’re an obscure aspect of higher math, involving movement through four dimensions. If you want a detailed explanation, try reading the document found here:

www.flipcode.com/documents/matrfaq.html#Q47

It’s a bit easier to explain why quaternions are used to represent rotations: interpolating between rotation values (that is, going through a bunch of in-between values to gradually change from one value to another) looks smoother and more natural when using quaternions.

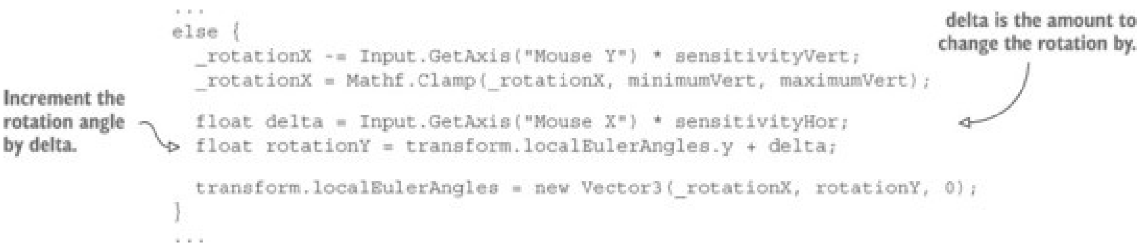
To return to the initial question, it’s because `localRotation` is a quaternion, not Euler angles. Unity also provides the Euler angles property to make manipulating rotations easier to understand; the Euler angles property is converted to and from quaternion values automatically. Unity handles the harder math for you behind the scenes, so you don’t have to worry about handling it yourself.

There’s one more rotation setting for MouseLook that needs code: horizontal and vertical rotation at the same time.

2.4.3. Horizontal and vertical rotation at the same time

This last chunk of code won’t use `Rotate()` either, for the same reason: the vertical rotation angle is clamped between limits after being incremented. That means the horizontal rotation needs to be calculated directly now. Remember, Rotate() was automating the process of incrementing the rotation angle (see the next listing).

Listing 2.6. Horizontal and vertical MouseLook



```
...
else {
  _rotationX -= Input.GetAxis("Mouse Y") * sensitivityVert;
  _rotationX = Mathf.Clamp(_rotationX, minimumVert, maximumVert);

  float delta = Input.GetAxis("Mouse X") * sensitivityHor;
  float rotationY = transform.localEulerAngles.y + delta;

  transform.localEulerAngles = new Vector3(_rotationX, rotationY, 0);
}
...
```

delta is the amount to change the rotation by.

Increment the rotation angle by delta.

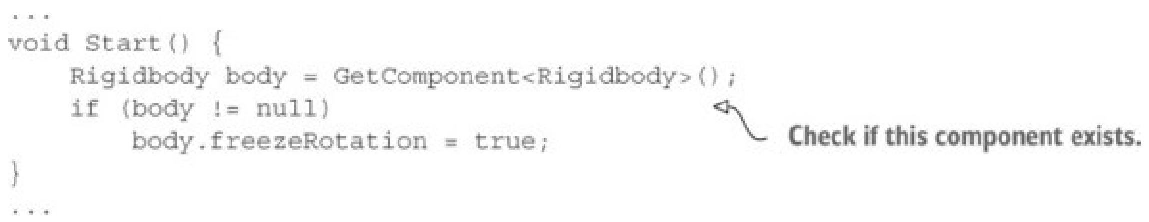
The first couple of lines, dealing with _rotationX, are exactly the same as in the last section. Just remember that rotating around the object's X-axis is vertical rotation. Because horizontal rotation is no longer being handled using the Rotate() method, that's what the delta and rotationY lines are doing. *Delta* is a common mathematical term for "the amount of change," so our calculation of delta is the amount that rotation should change. That amount of change is then added to the current rotation angle to get the desired new rotation angle.

Finally, both angles, vertical and horizontal, are used to create a new vector that's assigned to the transform component's angle property.

Disallow physics rotation on the player

Although this doesn't matter quite yet for this project, in most modern FPS games there's a complex physics simulation affecting everything in the scene. This will cause objects to bounce and tumble around; this behavior looks and works great for most objects, but the player's rotation needs to be solely controlled by the mouse and not affected by the physics simulation.

For that reason, mouse input scripts usually set the freezeRotation property on the player's Rigidbody. Add this Start() method to the MouseLook script:



```
...
void Start() {
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null)
        body.freezeRotation = true;
}
...
```

Check if this component exists.

(A Rigidbody is an additional component an object can have. The physics simulation acts on Rigidbodies and manipulates objects they’re attached to.)

In case you’ve gotten lost on where to make the various changes and additions we’ve gone over, the next listing has the full finished script. Alternatively, download the example project.

Listing 2.7. The finished MouseLook script

```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {
  public enum RotationAxes {
    MouseXAndY = 0,
    MouseX = 1,
    MouseY = 2
  }
  public RotationAxes axes = RotationAxes.MouseXAndY;

  public float sensitivityHor = 9.0f;
  public float sensitivityVert = 9.0f;

  public float minimumVert = -45.0f;
  public float maximumVert = 45.0f;

  private float _rotationX = 0;

  void Start() {
    Rigidbody body = GetComponent<Rigidbody>();
    if (body != null)
        body.freezeRotation = true;
  }

  void Update() {
    if (axes == RotationAxes.MouseX) {
      transform.Rotate(0, Input.GetAxis("Mouse X")  sensitivityHor, 0);
    }
    else if (axes == RotationAxes.MouseY) {
      _rotationX -= Input.GetAxis("Mouse Y")  sensitivityVert;
      rotationX = Mathf.Clamp(rotationX, minimumVert, maximumVert);
```

```
   float rotationY = transform.localEulerAngles.y;

   transform.localEulerAngles = new Vector3(_rotationX, rotationY,
0);
  }
  else {
   _rotationX -= Input.GetAxis("Mouse Y")  sensitivityVert;
   rotationX = Mathf.Clamp(rotationX, minimumVert, maximumVert);

   float delta = Input.GetAxis("Mouse X")  sensitivityHor;
   float rotationY = transform.localEulerAngles.y + delta;

   transform.localEulerAngles = new Vector3(_rotationX, rotationY,
0);
  }
 }
}
```

When you run the new script, you’re able to look around in all directions while moving the mouse. Great! But you’re still stuck in one place, looking around as if mounted on a turret. The next step is moving around the scene.

2.5. Keyboard input component: first-person controls

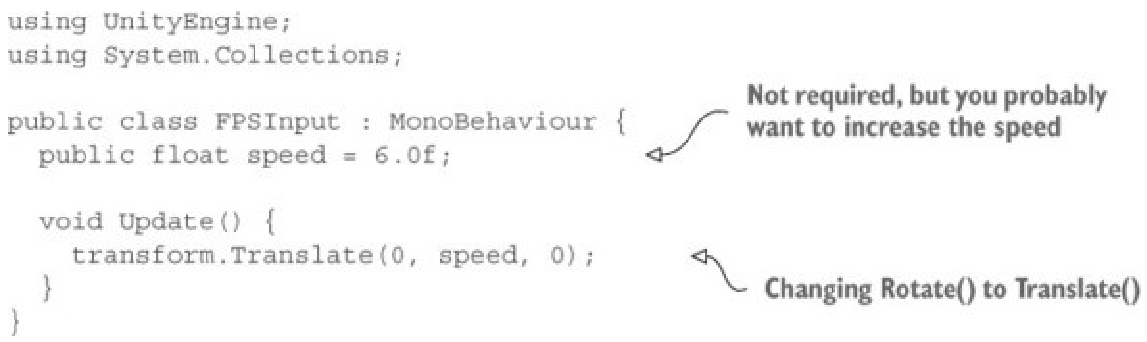
Looking around in response to mouse input is an important part of first-person controls, but you’re only halfway there. The player also needs to move in response to keyboard input. Let’s write a keyboard controls component to complement the mouse controls component; create a new C# script called FPSInput and attach that to the player (alongside the MouseLook script). For the moment set the MouseLook component to horizontal rotation only.

Tip

The keyboard and mouse controls explained here are split up into separate scripts. You don’t have to structure the code this way, and you could have everything bundled into a single “player controls” script, but a component system (such as the one in Unity) tends to be most flexible and thus most useful when you have functionality split into several smaller components.

The code you wrote in the previous section affected rotation only, but now we’ll change the object's position instead. As shown in listing 2.8, refer back to the rotation code from before we added mouse input; type that into FPSInput, but change `Rotate()` to `Translate()`. When you hit Play, the view slides up instead of spinning around. Try changing the parameter values to see how the movement changes (in particular, try swapping the first and second numbers); after experimenting with that for a bit, you can move on to adding keyboard input.

Listing 2.8. Spin code from the first listing, with a couple of minor changes



```
using UnityEngine;
using System.Collections;

public class FPSInput : MonoBehaviour {
  public float speed = 6.0f;

  void Update() {
    transform.Translate(0, speed, 0);
  }
}
```

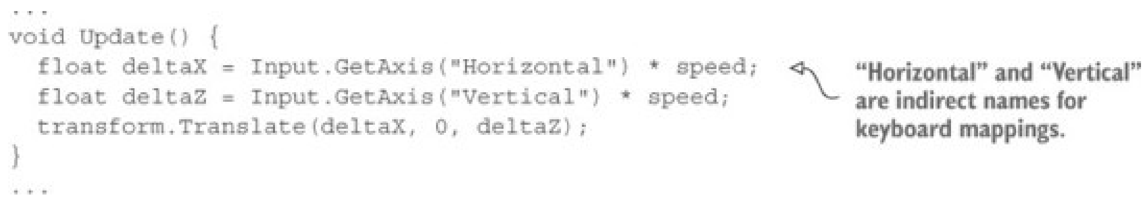
Not required, but you probably want to increase the speed

Changing Rotate() to Translate()

2.5.1. Responding to key presses

The code for moving according to key presses (shown in the following listing) is similar to the code for rotating according to the mouse. The `GetAxis()` method is used here as well, and in a very similar way. The following listing demonstrates how to use that command.

Listing 2.9. Positional movement responding to key presses



```
...
void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed;
  float deltaZ = Input.GetAxis("Vertical") * speed;
  transform.Translate(deltaX, 0, deltaZ);
}
...
```

"Horizontal" and "Vertical" are indirect names for keyboard mappings.

As before, the `GetAxis()` values are multiplied by speed in order to determine the amount of movement. Whereas before the requested axis was always "Mouse something," now we pass in either Horizontal or Vertical. These names are abstractions for input settings in Unity; if you look in the Edit menu under Project Settings and then look under Input, you'll find a list of abstract input names and the exact controls mapped to those names. Both the left/right arrow keys and the letters A/D are mapped to Horizontal, whereas both the up/down arrow keys and the letters W/S are mapped to Vertical.

Note that the movement values are applied to the X and Z coordinates. As you probably noticed while experimenting with the `Translate()` method, the X coordinate moves from side to side and the Z coordinate moves forward and backward.

Put in this new movement code and you should be able to move around by pressing either the arrow keys or WASD letter keys, the standard in most FPS games. The movement script is nearly complete, but we have a few more adjustments to go over.

2.5.2. Setting a rate of movement independent of the computer’s speed

It’s not obvious right now because you’ve only been running the code on one computer (yours), but if you ran it on different machines it’d run at different speeds. That’s because some computers can process code and graphics faster than others. Right now the player would move at different speeds on different computers because the movement code is tied to the computer’s speed. That is referred to as *frame rate dependent*, because the movement code is dependent on the frame rate of the game.

For example, imagine you run this demo on two different computers, one that gets 30 fps (frames per second) and one that gets 60 fps. That means `Update()` would be called twice as often on the second computer, and the same speed value of 6 would be applied every time. At 30 fps the rate of movement would be 180 units/second, and the movement at 60 fps would be 360 units/second. For most games, movement speed that varies like this would be bad news.

The solution is to adjust the movement code to make it *frame rate independent*. That means the speed of movement is not dependent on the frame rate of the game. The way to achieve this is by not applying the same speed value at every frame rate. Instead, scale the speed value higher or lower depending on how quickly the computer runs. This is achieved by multiplying the speed value by another value called `deltaTime`, as shown in the next listing.

Listing 2.10. Frame rate independent movement using `deltaTime`

```
...
void Update() {
  float deltaX = Input.GetAxis("Horizontal")  speed;
  float deltaZ = Input.GetAxis("Vertical")  speed;
  transform.Translate(deltaX  Time.deltaTime, 0, deltaZ
Time.deltaTime);
}
...
```

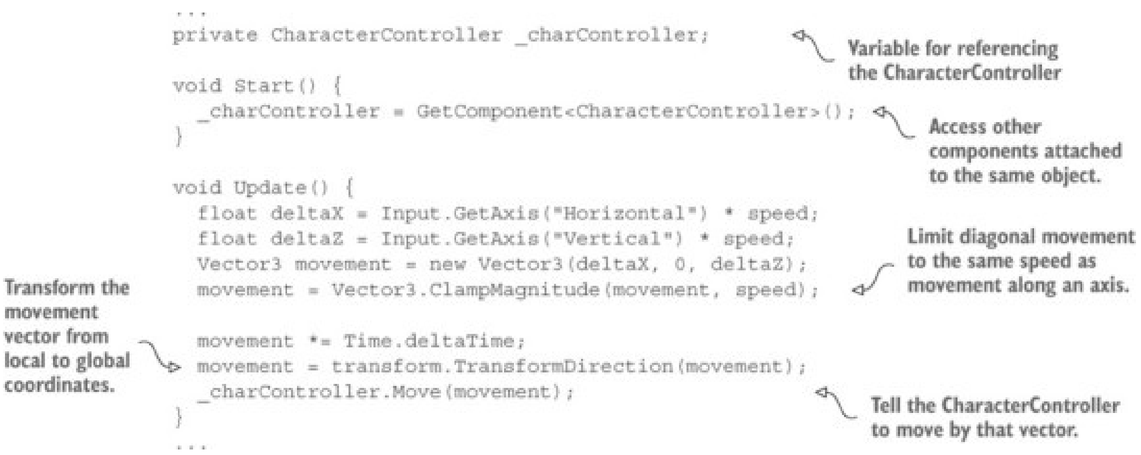
That was a simple change. The `Time` class has a number of properties and methods useful for timing, and one of those properties is `deltaTime`. Because we know that delta means the amount of change, that means `deltaTime` is the amount of change in time. Specifically, `deltaTime` is the amount of time between frames. The time between frames varies at different frame rates (for example, 30 fps is a `deltaTime` of 1/30th of a second), so multiplying the speed value by `deltaTime` will scale the speed value on different computers.

Now the movement speed will be the same on all computers. But the movement script is still not quite done; when you move around the room you can pass through walls, so we need to adjust the code further to prevent that.

2.5.3. Moving the CharacterController for collision detection

Directly changing the object's transform doesn't apply collision detection, so the character will pass through walls. To apply collision detection, what we want to do instead is use CharacterController. CharacterController is a component that makes the object move more like a character in a game, including colliding with walls. Recall that back when we set up the player, we attached a CharacterController, so now we'll use that component with the movement code in FPSInput (see the following listing).

Listing 2.11. Moving CharacterController instead of Transform



```
...
private CharacterController _charController;

void Start() {
  _charController = GetComponent<CharacterController>();
}

void Update() {
  float deltaX = Input.GetAxis("Horizontal") * speed;
  float deltaZ = Input.GetAxis("Vertical") * speed;
  Vector3 movement = new Vector3(deltaX, 0, deltaZ);
  movement = Vector3.ClampMagnitude(movement, speed);

  movement *= Time.deltaTime;
  movement = transform.TransformDirection(movement);
  _charController.Move(movement);
}
...
```

Variable for referencing the CharacterController

Access other components attached to the same object.

Limit diagonal movement to the same speed as movement along an axis.

Transform the movement vector from local to global coordinates.

Tell the CharacterController to move by that vector.

This code excerpt introduces several new concepts. The first concept to point out is the variable for referencing the CharacterController. This variable simply creates a local reference to the object (code object, that is—not to be confused with scene objects); multiple scripts can have references to this one CharacterController instance.

That variable starts out empty, so before you can use the reference you need to assign an object to it for it to refer to. This is where `GetComponent()` comes into play; that method returns other components attached to the same `GameObject.` Rather than pass a parameter inside the parentheses, you use the C# syntax of defining the type inside angle brackets, <>.

Once you have a reference to the CharacterController, you can call `Move()` on the controller. Pass in a vector to that method, similar to how the mouse rotation code used a vector for rotation values. Also similar to how rotation values were limited, use `Vector3.ClampMagnitude()` to limit the vector's magnitude to the movement speed; the clamp is used because otherwise diagonal movement would have a greater magnitude than movement directly along an axis (picture the sides and hypotenuse of a right triangle).

But there's one tricky aspect to the movement vector here, and it has to do with local versus global, as we discussed earlier for rotations. We'll create the vector with a value to move, say, to the left. That's the *player's* left, though, which may be a completely different direction from the *world's* left. That is, we're talking about left in local space, not global space. We need to pass a movement vector defined in global space to the `Move()` method, so we're going to need to convert the local space vector into global space. Doing that conversion is extremely complex math, but fortunately for us Unity takes care of that math for us, and we simply need to call the method `TransformDirection()` in order to, well, transform the direction.

Definition

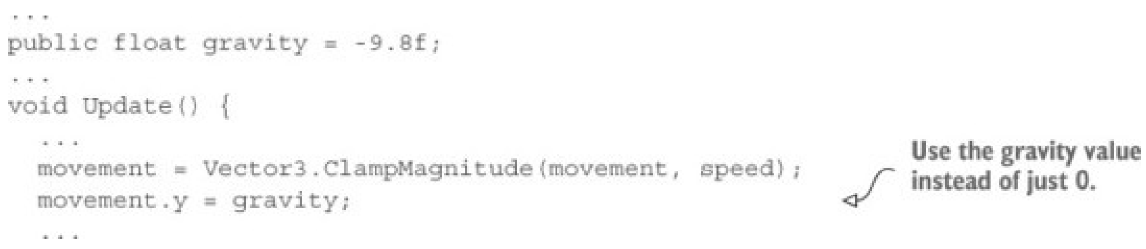
Transform used as a verb means to convert from one coordinate space to another (refer back to section 2.3.3 if you don't remember what a coordinate space is). Don't get confused with the other definitions of transform, including both the Transform component and the action of moving the object around the scene. It's sort of an overloaded term, because all these meanings refer to the same underlying concept.

Test playing the movement code now. If you haven't done so already, set the MouseLook component to both horizontal and vertical rotation. You can look around the scene fully and fly around the scene using keyboard controls. This is pretty great if you want the player to fly around the scene, but what if you want the player walking around on the ground?

2.5.4. Adjusting components for walking instead of flying

Now that collision detection is working, the script can have gravity and the player will stay down against the floor. Declare a gravity variable and then use that gravity value for the Y-axis, as shown in the next listing.

Listing 2.12. Adding gravity to the movement code



```
...
public float gravity = -9.8f;
...
void Update() {
  ...
  movement = Vector3.ClampMagnitude(movement, speed);
  movement.y = gravity;
  ...
```

Use the gravity value instead of just 0.

Now there's a constant downward force on the player, but it's not always pointed straight down, because the player object can tilt up and down with the mouse. Fortunately everything we need to fix that is already in place, so we just need to make some minor adjustments to how components are set up on the player. First set the MouseLook component on the player object to horizontal rotation only. Next add the MouseLook component to the camera object, and set that one to vertical rotation only. That’s right; you’re going to have two different objects responding to the mouse!

Because the player object now only rotates horizontally, there’s no longer any problem with the downward force of gravity being tilted. The camera object is parented to the player object (remember when we did that in the Hierarchy view?), so even though it rotates vertically independently from the player, the camera rotates horizontally along with the player.

Polishing the finished script

Use the `RequireComponent()` method to ensure that other components needed by the script are also attached. Sometimes other components are optional (that is, code that says “If this other component is also attached, then...”), but sometimes you want to make the other components mandatory. Add the method to the top of the script in order to enforce that dependency and give the required component as a parameter.

Similarly, if you add the method `AddComponentMenu()` to the top of your scripts, that script will be added to the component menu in Unity’s editor. Tell the command the name of the menu item you want to add, and then the script can be selected when you click Add Component at the bottom of the Inspector. Handy!

A script with both methods added to the top would look something like this:

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
...
```

Listing 2.13 shows the full finished script. Along with the small adjustments to how components are set up on the player, the player can walk around the room. Even with the gravity variable being applied, you can still use this script for flying movement by setting Gravity to 0 in the Inspector.

Listing 2.13. The finished FPSInput script

```
using UnityEngine;
using System.Collections;

[RequireComponent(typeof(CharacterController))]
[AddComponentMenu("Control Script/FPS Input")]
public class FPSInput : MonoBehaviour {
  public float speed = 6.0f;
  public float gravity = -9.8f;

  private CharacterController charController;

  void Start() {
    charController = GetComponent<CharacterController>();
  }

  void Update() {
    float deltaX = Input.GetAxis("Horizontal")  speed;
    float deltaZ = Input.GetAxis("Vertical")  speed;
    Vector3 movement = new Vector3(deltaX, 0, deltaZ);
    movement = Vector3.ClampMagnitude(movement, speed);

    movement.y = gravity;

    movement *= Time.deltaTime;
    movement = transform.TransformDirection(movement);
    _charController.Move(movement);
  }
}
```

Congratulations on building this 3D project! We covered a lot of ground in this chapter, and now you’re well-versed in how to code movement in Unity. As exciting as this first demo is, it’s still a long way from being a complete game. After all, the project plan described this as a basic FPS scene, and what’s a shooter if you can’t shoot? So give yourself a well-deserved pat on the back for this chapter’s project, and then get ready for the next step.

2.6. Summary

In this chapter you learned that

- 3D coordinate space is defined by X-, Y-, and Z-axes.
- Objects and lights in a room set the scene.
- The player in a first-person scene is essentially a camera.
- Movement code applies small transforms repeatedly in every frame.
- FPS controls consist of mouse rotation and keyboard movement.

Chapter 3. Adding enemies and projectiles to the 3D game

This chapter covers

- Taking aim and firing, both for the player and for enemies
- Detecting and responding to hits
- Making enemies that wander around
- Spawning new objects in the scene

The movement demo from the previous chapter was pretty cool but still not really a game. Let’s turn that movement demo into a first-person shooter. If you think about what else we need now, it boils down to the ability to shoot, and things to shoot at. First we’re going to write scripts that enable the player to shoot objects in the scene. Then we’re going to build enemies to populate the scene, including code to both wander around aimlessly and react to being hit. Finally we’re going to enable the enemies to fight back, emitting fireballs at the player. None of the scripts from chapter 2 need to change; instead, we’ll add scripts to the project—scripts that handle the additional features.

I’ve chosen a first-person shooter for this project for a couple of reasons. One is simply that FPS games are popular; people like shooting games, so let’s make a shooting game. A subtler reason has to do with the techniques you’ll learn; this project is a great way to learn about several fundamental concepts in 3D simulations. For example, shooting games are a great way to teach raycasting. In a bit we’ll get into the specifics of what *raycasting* is, but for now you just need to know that it’s a tool that’s useful for many different tasks in 3D simulations. Although raycasting is useful in a wide variety of situations, it happens that using raycasting makes the most intuitive sense for shooting.

Creating wandering targets to shoot at gives us a great excuse to explore code for computer-controlled characters, as well as use techniques for sending messages and spawning objects. In fact, this wandering behavior is another place that raycasting is valuable, so we’re already going to be looking at a different application of the technique after having first learned it with shooting. Similarly, the approach to sending messages that’s demonstrated in this project is also useful elsewhere. In future chapters you’ll see other applications for these techniques, and even within this one project we’ll go over alternative situations.

Ultimately we’ll approach this project one new feature at a time, with the game always playable at every step but also always feeling like there’s a missing part to work on next. This roadmap breaks down the steps into small, understandable changes, with only one new feature added in each step:

1. Write code enabling the player to shoot into the scene.
2. Create static targets that react to being hit.
3. Make the targets wander around.
4. Spawn the wandering targets automatically.
5. Enable the targets/enemies to shoot fireballs at the player.

Note

This chapter’s project assumes you already have a first-person movement demo to build on. We created a movement demo in chapter 2, but if you skipped to this chapter then you will need to download the sample files for chapter 2.

3.1. Shooting via raycasts

The first new feature to introduce into the 3D demo is shooting. Looking around and moving are certainly crucial features for a first-person shooter, but it’s not a game until players can affect the simulation and apply their skills. Shooting in 3D games can be implemented with a few different approaches, and one of the most important approaches is raycasting.

3.1.1. What is raycasting?

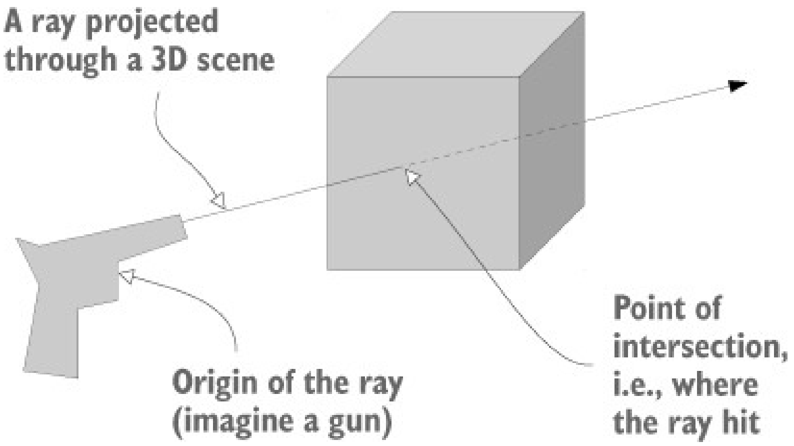
As the name indicates, raycasting is when you cast a ray into the scene. Clear, right? Well, okay, so what exactly is a *ray*?

Definition

A *ray* is an imaginary or invisible line in the scene that starts at some origin point and extends out in a specific direction.

Raycasting is when you create a ray and then determine what intersects that ray; figure 3.1 illustrates the concept. Consider what happens when you fire a bullet from a gun: the bullet starts at the position of the gun and then flies forward in a straight line until it hits something. A ray is analogous to the path of the bullet, and raycasting is analogous to firing the bullet and seeing where it hits.

Figure 3.1. A ray is an imaginary line, and raycasting is finding where that line intersects.


A ray projected through a 3D scene
Origin of the ray (imagine a gun)
Point of intersection, i.e., where the ray hit

As you can imagine, the math behind raycasting often gets complicated. Not only is it tricky to calculate the intersection of a line with a 3D plane, but you need to do that for all polygons of all mesh objects in the scene (remember, a mesh object is a 3D visual constructed from lots of connected lines and shapes). Fortunately, Unity handles the difficult math behind raycasting, but you still have to worry about higher-level concerns like where the ray is being cast from and why.

In this project the answer to the latter question (why) is to simulate a bullet being fired into the scene. For a first-person shooter, the ray generally starts at the camera position and then extends out through the center of the camera view. In other words, you're checking for objects straight in front of the camera; Unity provides commands to make that task simple. Let's take a look at these commands.

3.1.2. Using the command ScreenPointToRay for shooting

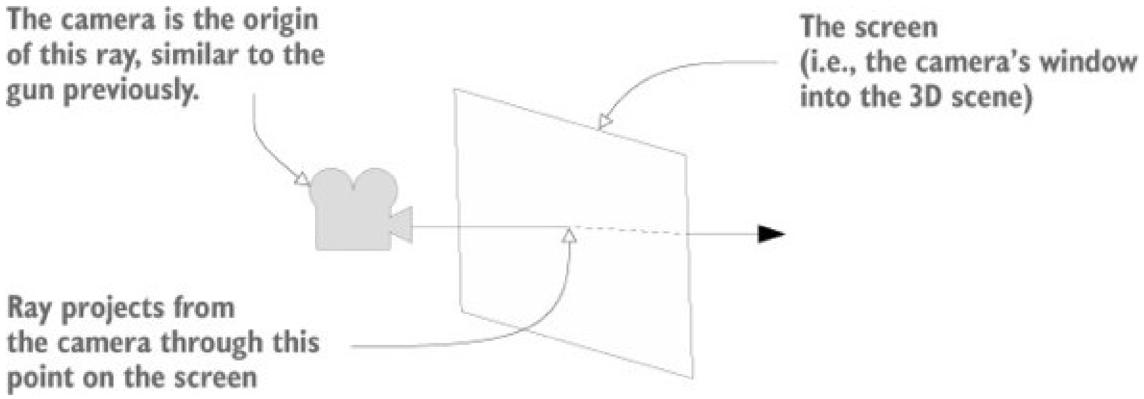
You’ll implement shooting by projecting a ray that starts at the camera and extends forward through the center of the view. Projecting a ray through the center of the camera view is a special case of an action referred to as *mouse picking*.

Definition

Mouse picking is the action of picking out the spot in the 3D scene directly under the mouse cursor.

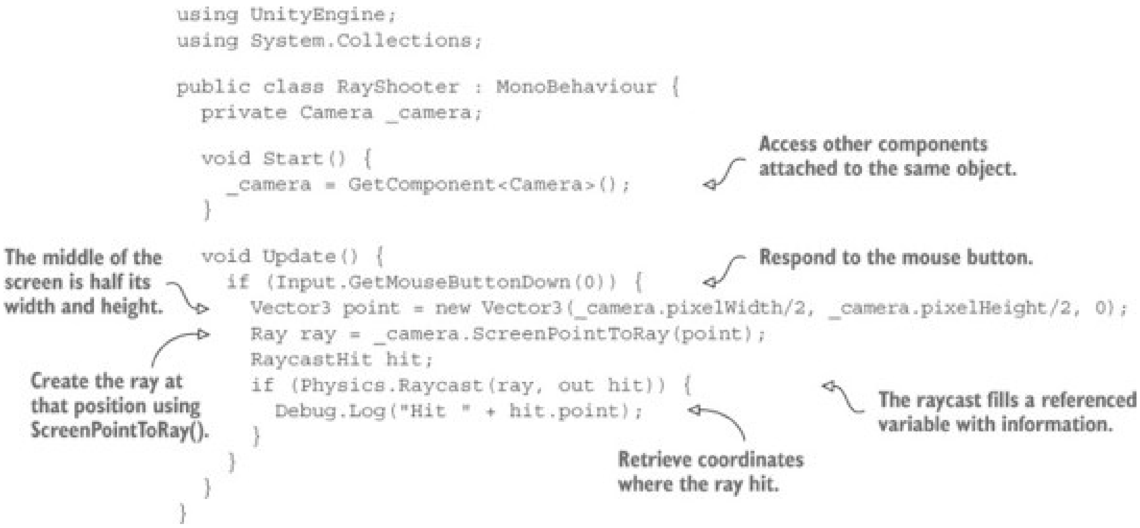
Unity provides the method `ScreenPointToRay()` to perform this action. Figure 3.2 illustrates what happens. The method creates a ray that starts at the camera and projects at an angle passing through the given screen coordinates. Usually the coordinates of the mouse position are used for mouse picking, but for first-person shooting the center of the screen is used. Once you have a ray, it can be passed to the method `Physics.Raycast()` to perform raycasting using that ray.

Figure 3.2. `ScreenPointToRay()` projects a ray from the camera through the given screen coordinates.


The camera is the origin of this ray, similar to the gun previously.
The screen (i.e., the camera’s window into the 3D scene)
Ray projects from the camera through this point on the screen

Let’s write some code that uses the methods we just discussed. In Unity create a new C# script, attach that script to the camera (not the player object), and then write the code from the next listing in it.

Listing 3.1. RayShooter script to attach to the camera



```
using UnityEngine;
using System.Collections;

public class RayShooter : MonoBehaviour {
  private Camera _camera;

  void Start() {
    _camera = GetComponent<Camera>();
  }

  void Update() {
    if (Input.GetMouseButtonDown(0)) {
      Vector3 point = new Vector3(_camera.pixelWidth/2, _camera.pixelHeight/2, 0);
      Ray ray = _camera.ScreenPointToRay(point);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit)) {
        Debug.Log("Hit " + hit.point);
      }
    }
  }
}
```

Access other components attached to the same object.

The middle of the screen is half its width and height.

Respond to the mouse button.

Create the ray at that position using ScreenPointToRay().

The raycast fills a referenced variable with information.

Retrieve coordinates where the ray hit.

You should note a number of things in this code listing. First, the camera component is retrieved in `Start()`, just like the CharacterController in the previous chapter. Then the rest of the code is put in `Update()` because it needs to check the mouse over and over repeatedly, as opposed to just one time. The method `Input.GetMouseButtonDown()` returns `true` or `false` depending on whether the mouse has been clicked, so putting that command in a conditional means the enclosed code runs only when the mouse has been clicked. You want to shoot when the player clicks the mouse; hence the conditional check of the mouse button.

A vector is created to define the screen coordinates for the ray (remember that a vector is several related numbers stored together). The camera’s `pixelWidth` and `pixelHeight` values give you the size of the screen, so dividing those values in half gives you the center of the screen. Although screen coordinates are 2D, with only horizontal and vertical components and no depth, a Vector3 was created because `ScreenPointToRay()` requires that data type (presumably because calculating the ray involves arithmetic on 3D vectors). `ScreenPointToRay()` was called with this set of coordinates, resulting in a `Ray` object (code object, that is, not a game object; the two can be confusing sometimes).

The ray is then passed to the `Raycast()` method, but it’s not the only object passed in. There’s also a `RaycastHit` data structure; `RaycastHit` is a bundle of information about the intersection of the ray, including where the intersection happened and what object was intersected. The C# syntax `out` ensures that the data structure manipulated within the command is the same object that exists outside the command, as opposed to the objects being separate copies in the different function scopes.

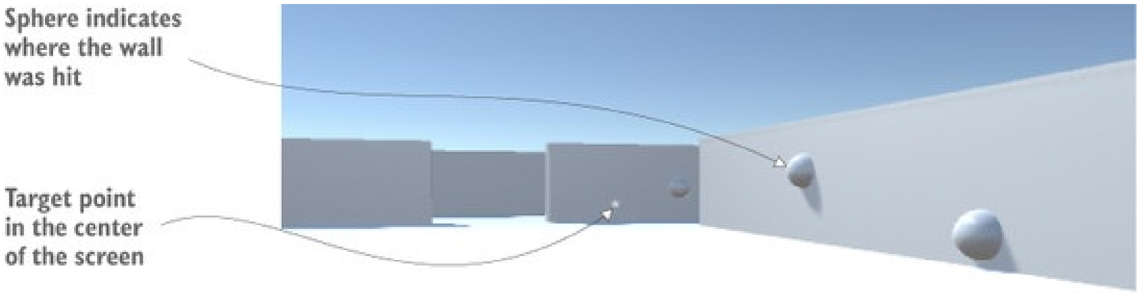
Finally the code calls the `Physics.Raycast()` method. This method checks for intersections with the given ray, fills in data about the intersection, and returns `true` if the ray hit anything. Because a Boolean value is returned, this method can be put in a conditional check, just as you used `Input.GetMouseButtonDown()` earlier.

For now the code emits a console message to indicate when an intersection occurred. This console message displays the 3D coordinates of the point where the ray hit (the XYZ values we discussed in chapter 2). But it can be hard to visualize where exactly the ray hit; similarly, it can be hard to tell where the center of the screen is (that is, where the ray shoots through). Let's add visual indicators to address both problems.

3.1.3. Adding visual indicators for aiming and hits

Our next step is to add two kinds of visual indicators: an aiming spot on the center of the screen, and a mark in the scene where the ray hit. For a first-person shooter the latter is usually bullet holes, but for now you're going to put a blank sphere on the spot (and use a coroutine to remove the sphere after one second). Figure 3.3 shows what you'll see.

Figure 3.3. Shooting repeatedly after adding visual indicators for aiming and hits

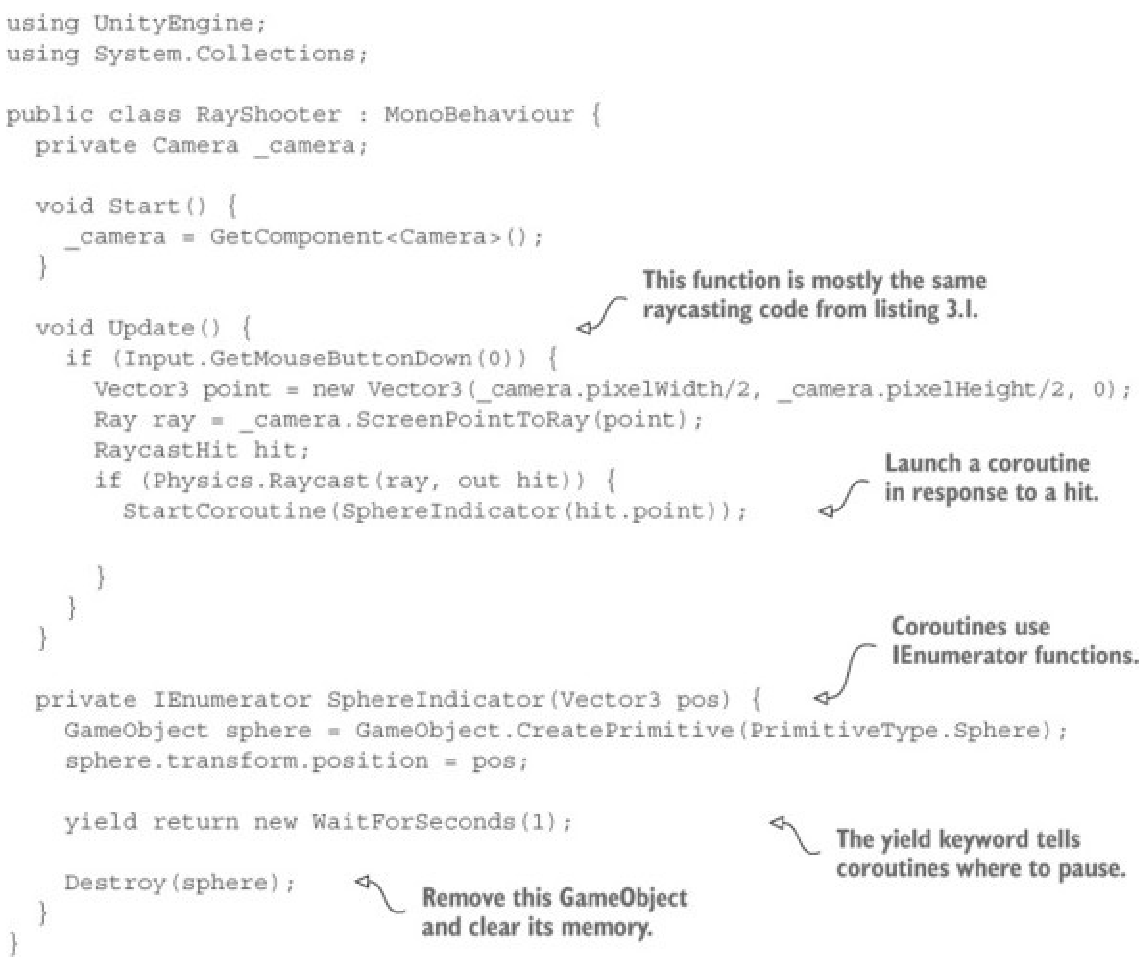

Sphere indicates where the wall was hit
Target point in the center of the screen

Definition

Coroutines are a Unity-specific way of handling tasks that execute incrementally over time, as opposed to how most functions make the program wait until they finish.

First let’s add indicators to mark where the ray hits. Listing 3.2 shows the script after making this addition. Run around the scene shooting; it’s pretty fun seeing the sphere indicators!

Listing 3.2. RayShooter script with sphere indicators added



```
using UnityEngine;
using System.Collections;

public class RayShooter : MonoBehaviour {
  private Camera _camera;

  void Start() {
    _camera = GetComponent<Camera>();
  }

  void Update() {
    if (Input.GetMouseButtonDown(0)) {
      Vector3 point = new Vector3(_camera.pixelWidth/2, _camera.pixelHeight/2, 0);
      Ray ray = _camera.ScreenPointToRay(point);
      RaycastHit hit;
      if (Physics.Raycast(ray, out hit)) {
        StartCoroutine(SphereIndicator(hit.point));

      }
    }
  }

  private IEnumerator SphereIndicator(Vector3 pos) {
    GameObject sphere = GameObject.CreatePrimitive(PrimitiveType.Sphere);
    sphere.transform.position = pos;

    yield return new WaitForSeconds(1);

    Destroy(sphere);
  }
}
```

This function is mostly the same raycasting code from listing 3.1.

Launch a coroutine in response to a hit.

Coroutines use IEnumerator functions.

The yield keyword tells coroutines where to pause.

Remove this GameObject and clear its memory.

The new method is `SphereIndicator()`, plus a one-line modification in the existing `Update()` method. This method creates a sphere at a point in the scene and then removes that sphere a second later. Calling `SphereIndicator()` from the raycasting code ensures that there will be visual indicators showing exactly where the ray hit. This function is defined with `IEnumerator`, and that type is tied in with the concept of coroutines.

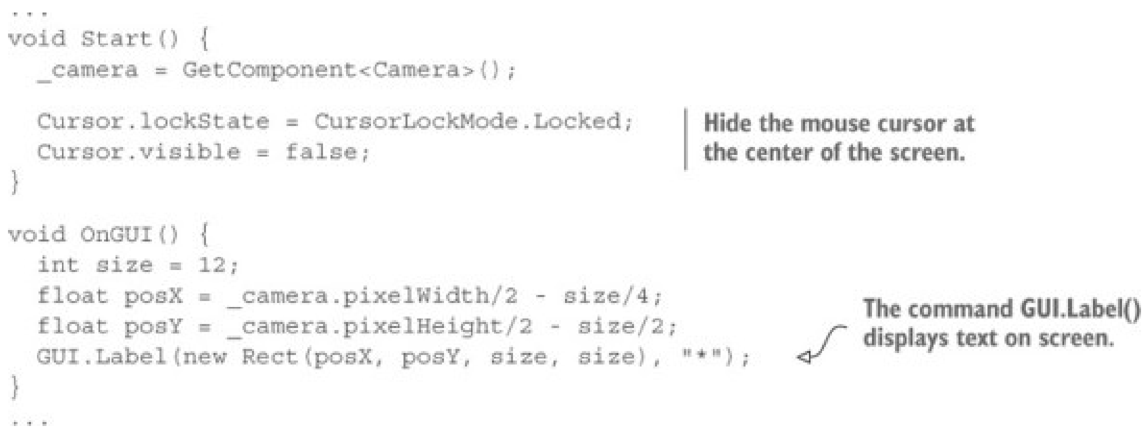
Technically, coroutines aren't asynchronous (asynchronous operations don't stop the rest of the code from running; think of downloading an image in the script of a website), but through clever use of enumerators, Unity makes coroutines behave similarly to asynchronous functions. The secret sauce in coroutines is the `yield` keyword; that keyword causes the coroutine to temporarily pause, handing back the program flow and picking up again from that point in the next frame. In this way, coroutines seemingly run in the background of a program, through a repeated cycle of running partway and then returning to the rest of the program.

As the name implies, `StartCoroutine()` sets a coroutine in motion. Once a coroutine is started, it keeps running until the function is finished; it just pauses along the way. Note the subtle but significant point that the method passed to `StartCoroutine()` has a set of parentheses following the name: this syntax means you're calling that function, as opposed to passing its name. The called function runs until it hits a `yield` command, at which point the function pauses.

`SphereIndicator()` creates a sphere at a specific point, pauses for the `yield` statement, and then destroys the sphere after the coroutine resumes. The length of the pause is controlled by the value returned at `yield`. A few different types of return values work in coroutines, but the most straightforward is to return a specific length of time to wait. Returning `WaitForSeconds(1)` causes the coroutine to pause for one second. Create a sphere, pause for one second, and then destroy the sphere: that sequence sets up a temporary visual indicator.

Listing 3.2 gave you indicators to mark where the ray hits. But you also want an aiming spot in the center of the screen, so that's done in the next listing.

Listing 3.3. Visual indicator for aiming



```
...
void Start() {
  _camera = GetComponent<Camera>();

  Cursor.lockState = CursorLockMode.Locked;
  Cursor.visible = false;
}

void OnGUI() {
  int size = 12;
  float posX = _camera.pixelWidth/2 - size/4;
  float posY = _camera.pixelHeight/2 - size/2;
  GUI.Label(new Rect(posX, posY, size, size), "*");
}
...
```

Hide the mouse cursor at the center of the screen.

The command GUI.Label() displays text on screen.

Another new method has been added to the `RayShooter` class, called `OnGUI()`. Unity comes with both a basic and more advanced user interface (UI) system; because the basic system has a lot of limitations, we'll build a more flexible advanced UI in future chapters, but for now it's much easier to display a point in the center of the screen using the basic UI. Much like `Start()` and `Update()`, every `MonoBehaviour` automatically responds to an `OnGUI()` method. That function runs every frame right after the 3D scene is rendered, resulting in everything drawn during `OnGUI()` appearing on top of the 3D scene (imagine stickers applied to a painting of a landscape).

Definition

Render is the action of the computer drawing the pixels of the 3D scene. Although the scene is defined using XYZ coordinates, the actual display on your monitor is a 2D grid of colored pixels. Thus in order to display the 3D scene, the computer needs to calculate the color of all the pixels in the 2D grid; running that algorithm is referred to as *rendering*.

Inside `OnGUI()` the code defines 2D coordinates for the display (shifted slightly to account for the size of the label) and then calls `GUI.Label()`. That method displays a text label; because the string passed to the label is an asterisk (*), you end up with that character displayed in the center of the screen. Now it's much easier to aim in our nascent FPS game!

Listing 3.3 also added some cursor settings to the `Start()` method. All that's happening is that the values are being set for cursor visibility and locking. The script will work perfectly fine if you omit the cursor values, but these settings make first-person controls work a bit more smoothly. The mouse cursor will stay in the center of the screen, and to avoid cluttering the view it will turn invisible and will only reappear when you hit Esc.

Warning

Always remember that you can hit Esc to unlock the mouse cursor. While the mouse cursor is locked, it’s impossible to click the Play button and stop the game.

That wraps up the first-person shooting code...well, that wraps up the player’s end of the interaction, anyway, but we still need to take care of targets.

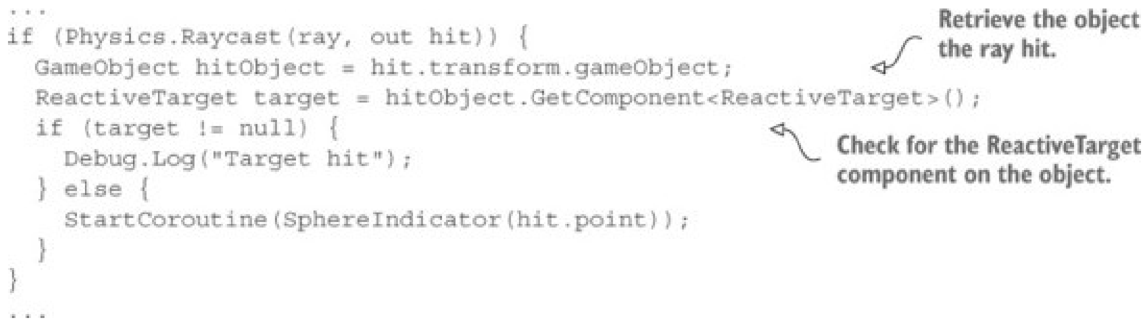
3.2. Scripting reactive targets

Being able to shoot is all well and good, but at the moment players don’t have anything to shoot at. We’re going to create a target object and give it a script that will respond to being hit. Or rather, we’ll slightly modify the shooting code to notify the target when hit, and then the script on the target will react when notified.

3.2.1. Determining what was hit

First you need to create a new object to shoot at. Create a new cube object (GameObject > 3D Object > Cube) and then scale it up vertically by setting the Y scale to 2 and leaving X and Z at 1. Position the new object at 0, 1, 0 to put it on the floor in the middle of the room, and name the object Enemy. Create a new script called ReactiveTarget and attach that to the newly created box. Soon you’ll write code for this script, but leave it at the default for now; you’re only creating the script file because the next code listing requires it to exist in order to compile. Go back to RayShooter.cs and modify the raycasting code according to the following listing. Run the new code and shoot the new target; debug messages appear in the console instead of sphere indicators in the scene.

Listing 3.4. Detecting whether the target object was hit



```
...
if (Physics.Raycast(ray, out hit)) {
  GameObject hitObject = hit.transform.gameObject;
  ReactiveTarget target = hitObject.GetComponent<ReactiveTarget>();
  if (target != null) {
    Debug.Log("Target hit");
  } else {
    StartCoroutine(SphereIndicator(hit.point));
  }
}
...
```

Retrieve the object the ray hit.

Check for the ReactiveTarget component on the object.

Notice that you retrieve the object from `RaycastHit`, just like the coordinates were retrieved for the sphere indicators. Technically, the hit information doesn't return the game object hit; it indicates the Transform component hit. You can then access `gameObject` as a property of `transform`.

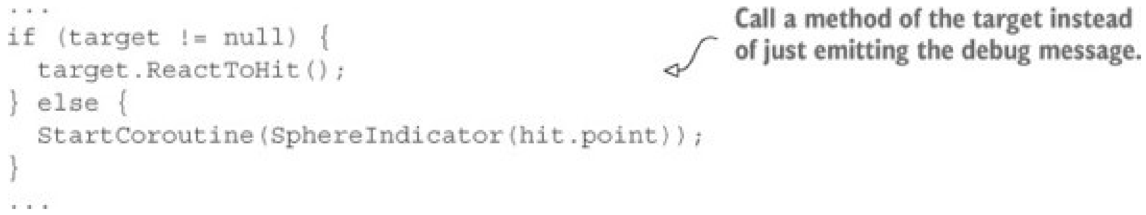
Then, you use the method `GetComponent()` on the object to check whether it's a reactive target (that is, if it has the ReactiveTarget script attached). As you saw previously, that method returns components of a specific type that are attached to the `GameObject`. If no component of that type is attached to the object, then `GetComponent()` won't return anything. You check whether `null` was returned and run different code in each case.

If the hit object is a reactive target, the code emits a debug message instead of starting the coroutine for sphere indicators. Now let's inform the target object about the hit so that it can react.

3.2.2. Alert the target that it was hit

All that's needed in the code is a one-line change, as shown in the following listing.

Listing 3.5. Sending a message to the target object

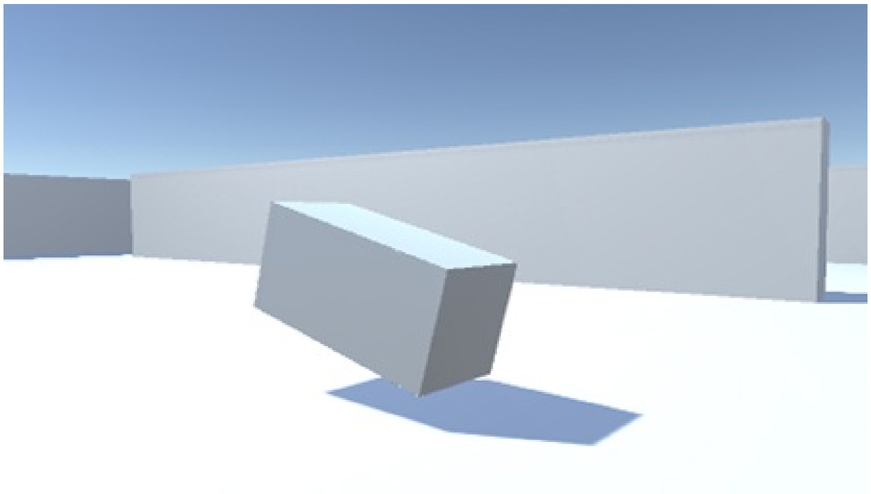


```
...
if (target != null) {
  target.ReactToHit();
} else {
  StartCoroutine(SphereIndicator(hit.point));
}
...
```

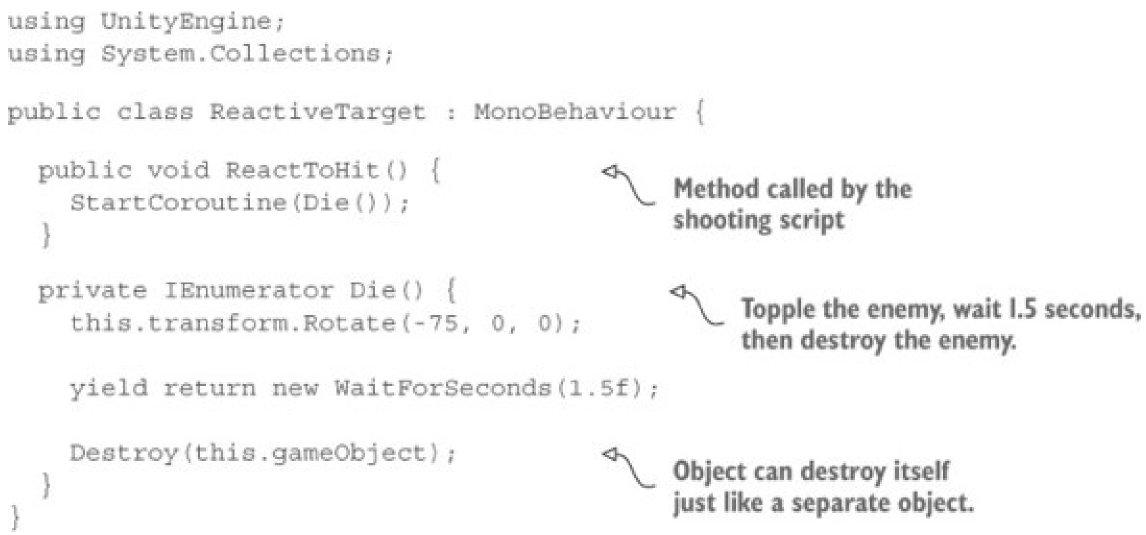
Call a method of the target instead of just emitting the debug message.

Now the shooting code calls a method of the target, so let's write that target method. In the ReactiveTarget script, write in the code from the next listing. The target object will fall over and disappear when you shoot it; refer to figure 3.4.

Figure 3.4. The target object falling over when hit



Listing 3.6. ReactiveTarget script that dies when hit



```
using UnityEngine;
using System.Collections;

public class ReactiveTarget : MonoBehaviour {

  public void ReactToHit() {
    StartCoroutine(Die());
  }

  private IEnumerator Die() {
    this.transform.Rotate(-75, 0, 0);

    yield return new WaitForSeconds(1.5f);

    Destroy(this.gameObject);
  }
}
```

Method called by the shooting script

Topple the enemy, wait 1.5 seconds, then destroy the enemy.

Object can destroy itself just like a separate object.

Most of this code should already be familiar to you from previous scripts, so we'll only go over it briefly. First, you define the method `ReactToHit()`, because that's the method name called in the shooting script. This method starts a coroutine that's similar to the sphere indicator code from earlier; the main difference is that it operates on the object of this script rather than creating a separate object. Expressions like `this.gameObject` refer to the `GameObject` that this script is attached to (and the `this` keyword is optional, so code could refer to `gameObject` without anything in front of it).