Jack Donovan

Mastering Oculus Rift Development

Explore the new frontier of virtual reality with the Oculus Rift and bring the VR revolution to your own projects

Packt>

Mastering Oculus Rift Development

Explore the new frontier of virtual reality with the Oculus Rift and bring the VR revolution to your own projects

Jack Donovan

Packt

BIRMINGHAM - MUMBAI

Mastering Oculus Rift Development

First published: February 2017

Production reference: 1310117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78646-115-5

www.packtpub.com

Credits

Author

Jack Donovan

Reviewer

Ezra Smith

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Smeet Thakkar

Content Development Editor

Divij Kotian

Technical Editor

Prajakta Mhatre

Copy Editors

Safis Editing

Sameen Siddiqui

Project Coordinator

Sheejal Shah

Proofreader

Safis Editing

Indexer

Pratik Shirodkar

Graphics

Tania Datta

Jason Monteiro

Production Coordinator

Aparna Bhagat

About the Author

Jack Donovan is a 24-year-old software engineer living in Brooklyn, New York. He graduated from Champlain College in Burlington, Vermont with a BS in game programming and soon after joined IrisVR, a startup founded in Burlington to create one-click software for architects to visualize models in virtual reality. IrisVR moved to New York City after being accepted into the startup incubator program Techstars and has since moved to an independent office in NoHo, where Jack continues to work.

Jack also wrote *OUYA Game Development By Example*, a book focused on the basics of developing games for the Android-powered OUYA console. Both books use the Unity3D engine for development, which Jack has been using to develop games and software in since 2010.

Thanks to Matt Gooding for being an invaluable friend and mentor, and Ezra Smith for his dedication and guidance in helping me put the finishing touches on this book. I'd also like to thank my parents, Paul Donovan and Jody Petersen, who have always supported me and inspired me to work hard on what I love. Lastly, I'd like to thank all of the brilliant people that I've been fortunate to work and grow with at IrisVR, especially the founders Shane Scranton and Nate Beatty for working tirelessly to make it all possible.

About the Reviewer

Ezra Smith is a software developer at IrisVR and one of the co-founders of FridgeCat Software. He's been writing mobile apps with Unity since 2011, and started experimenting with virtual reality in 2013. Ezra has been dreaming about playing video games in VR since he was a child, and is very excited to see those dreams becoming a reality.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thank you for purchasing this Packt book. We take our commitment to improving our content and products to meet your needs seriously—that's why your feedback is so valuable. Whatever your feelings about your purchase, please consider leaving a review on this book's Amazon page. Not only will this help us, more importantly it will also help others in the community to make an informed decision about the resources that they invest in to learn.

You can also review for us on a regular basis by joining our reviewers' club. **If you're interested in joining, or would like to learn more about the benefits we offer, please contact us**: `customerreviews@packtpub.com`.

Table of Contents

Preface

Welcome to *Mastering Oculus Rift Development*! With this book, you'll learn the fundamentals of creating complete virtual reality experiences for the Oculus Rift VR headset using the Unity3D game engine. Each chapter will guide new and experienced developers alike in building and expanding a project with basic gameplay, spatialized audio, and networked matchmaking. It also covers in-depth optimization and the process of publishing your game, ensuring you have everything you need to release a polished experience.

What this book covers

`Chapter 1`, *Exploring a New Reality with the Oculus Rift*, briefly outlines the basics of virtual reality and guides the user through getting set up with their Rift.

`Chapter 2`, *Stepping into Virtual Reality*, introduces the reader to the Unity3D game engine and implementing simple navigation in a new VR project.

`Chapter 3`, *Improving Performance and Avoiding Discomfort*, explains the importance of avoiding VR sickness by ensuring the game is optimized and always running smoothly.

`Chapter 4`, *Interacting with Virtual Worlds*, covers how to handle interaction and different input devices in Unity, including the controller and remote that ship with the Oculus Rift.

`Chapter 5`, *Establishing Presence*, teaches the reader about several factors that make VR experiences immersive and interesting, including 3D environments, particle effects, and interactive objects.

`Chapter 6`, *Adding Depth and Intuition to a User Interface*, guides the reader through setting up a user interface (UI) in VR and explains what makes it different from conventional game UIs.

`Chapter 7`, *Hearing and Believing with 3D Audio*, introduces the reader to the concept of spatialized audio and its importance in establishing presence and scale in any VR world.

`Chapter 8`, *Adding Tone and Realism with Graphics*, provides a detailed overview of rendering the pipelines and shaders that help establish both realism and artistic flair.

`Chapter 9`, *Bringing Players Together in VR*, teaches the reader how to create networked games and interact with other players in VR over the Web.

`Chapter 10`, *Publishing on the Oculus Store*, outlines all of the steps involved in packaging a VR game and publishing it on the Oculus Store for the world to download and play.

What you need for this book

The following software is recommended for use with this book:

- Unity 5.4.0
- CrazyBump 1.22

We will also need the following hardware:

- Oculus Rift CV1
- Xbox controller and Oculus remote (included with Rift)

Make sure you also have the following:

- Internet connection
- Windows 7+

Who this book is for

This book is for aspiring indie developers and VR enthusiasts who want to bring their ideas into virtual reality with a new platform that provides an unprecedented level of realism and immersion.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Create a folder called `Scripts` to put your game's code in."

A block of code is set as follows:

```
void Update()
{
  if(Input.GetKeyDown(KeyCode.Space))
  {
    Teleport();
  }
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Expand the **Horizontal** tab to view its preset configuration."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Oculus-Rift-Development`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/MasteringOculusRiftDevelopment_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Exploring a New Reality with the Oculus Rift

What made you feel like you were truly immersed in a game world for the first time? Was it graphics that looked impressively realistic, ambient noise that perfectly captured the environment and mood, or the way the game's mechanics just started to feel like a natural reflex? Game developers constantly strive to replicate scenarios that are as real and as emotionally impactful as possible, and they've never been as close as they are now with the advent of **virtual reality** (**VR**).

VR has been a niche market since the early 1950s, often failing to evoke a meaningful sense of presence that the concept hinges on—that is, until the first Oculus Rift prototype was designed in 2010 by Oculus founder Palmer Luckey. The Oculus Rift proved that modern rendering and display technology was reaching a point that immersive VR could be achieved, and that's when the new era of VR development began.

Today, VR development is as accessible as ever, comprehensively supported in the most popular off-the-shelf game development engines such as Unreal Engine and Unity3D. In this book, you'll learn all of the essentials that go into a complete VR experience, and master the techniques that will enable you to bring any idea you have into VR.

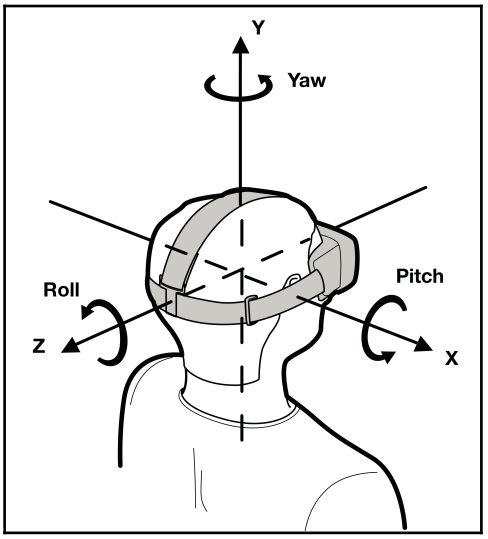
This chapter will cover everything you need to know to get started with VR, including the following points:

- The concept of VR
- The importance of intent in VR design
- Common limitations of VR games
- Vergence-accommodation conflict
- The anatomy of a VR headset

- Constellation tracking
- The importance of input
- Meeting required frame rates
- Asynchronous Timewarp
- Installing the Oculus runtime
- Configuring your Oculus Rift
- Stepping into Oculus home

The concept of VR

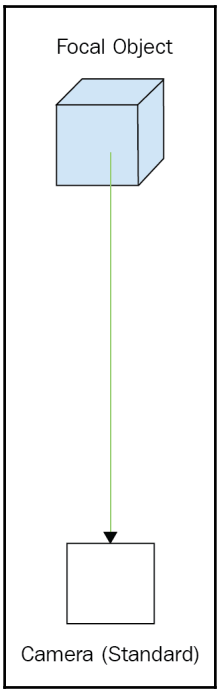
VR has taken many forms and formats since its inception, but this book will be focused on modern VR experienced with a **Head-Mounted Display** (**HMD**). HMDs such as the Oculus Rift are typically treated like an extra screen attached to your computer (more on that later), but with some extra components that enable it to capture its own orientation (and position, in some cases). This essentially amounts to a screen that sits on your head and knows how it moves, so it can mirror your head movements in the VR experience and enable you to look around the virtual world, making you feel like you're actually there:



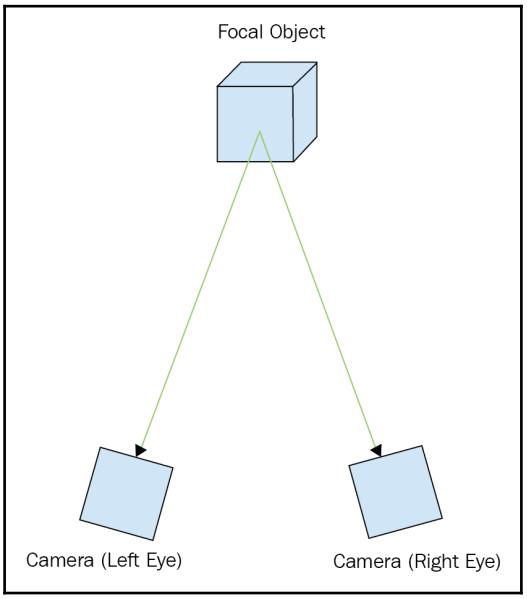
Depth perception

Depth perception is another big principle of VR. Because the display of the HMD is always positioned right in front of the user's eyes, the rendered image is typically split into two images, one per eye, with each individual image rendered from the position of that eye.

You can observe the difference between normal rendering and VR rendering in the following two images. This first image is how normal 3D video games are rendered to a computer screen, created based on the position and direction of a virtual camera in the game world:



This next image shows how VR scenes are rendered, using a different virtual camera for each eye to create a stereoscopic depth effect:



Common limitations of VR games

While VR provides the ability to immerse a player's senses like never before, it also creates some new, unique problems that must be addressed by responsible VR developers.

Locomotion sickness

VR headsets are meant to make you feel like you're somewhere else, and it only makes sense that you'd want to be able to explore that somewhere. Unfortunately, common game mechanics such as traditional joystick locomotion are problematic for VR. Our inner ears are accustomed to sensing inertia while we move from place to place, so if you were to push a joystick forward to walk in VR, your body would get confused when it sensed that you're still stationary.

Typically when there's a mismatch between what we're seeing and what we're feeling, our bodies assume that a nefarious poison or illness is at work, and they prepare to rid the body of the culprit; that's the motion sickness you feel when reading in a car, standing on a boat, and yes, moving in VR. This doesn't mean that we have to prevent users from moving in VR, we just might want to be more clever about it—more on that later.

The primary cause of nausea with a traditional joystick movement in VR is acceleration and smooth movement; your brain gets confused easily when picking up speed or slowing down, and even constant smooth motion can cause nausea (car sickness, for instance)—rotation is even more complicated, because rotating smoothly creates discomfort almost immediately. Some developers get around this using hard increments instead of gradual motion, such as rotating in 30 degree “snaps” once per second instead of rotating smoothly.

Lack of real-world vision

One of the potentially clumsiest aspects of VR is getting your hands where they need to be without being able to see them. Whether you're using a gamepad, keyboard, or motion controller, you'll probably need to use your hands to interact with VR, which you can't see with an HMD sitting over your eyes. It's good practice to centralize input around resting positions (that is, the buttons naturally closest to your thumbs on a gamepad or the home row of a computer keyboard), but shy away from anything that requires complex precise input, such as writing sentences on a keyboard or hitting button combos on a controller.

Some VR headsets, such as the HTC Vive, have a forward-facing camera (sometimes called a passthrough camera) that users can choose to view in VR, enabling basic interaction with the real world without taking the headset off. The Oculus Rift doesn't feature a built-in camera, but you could still display the feed from an external camera on any surface in (we'll play with that idea later in the book).

Even before modern VR, developers were creating applications that overlay smart information over what a camera is seeing; that's called **augmented reality** (**AR**). Experiences that ride the line between camera output and virtual environments are called **mixed reality** (**MR**).

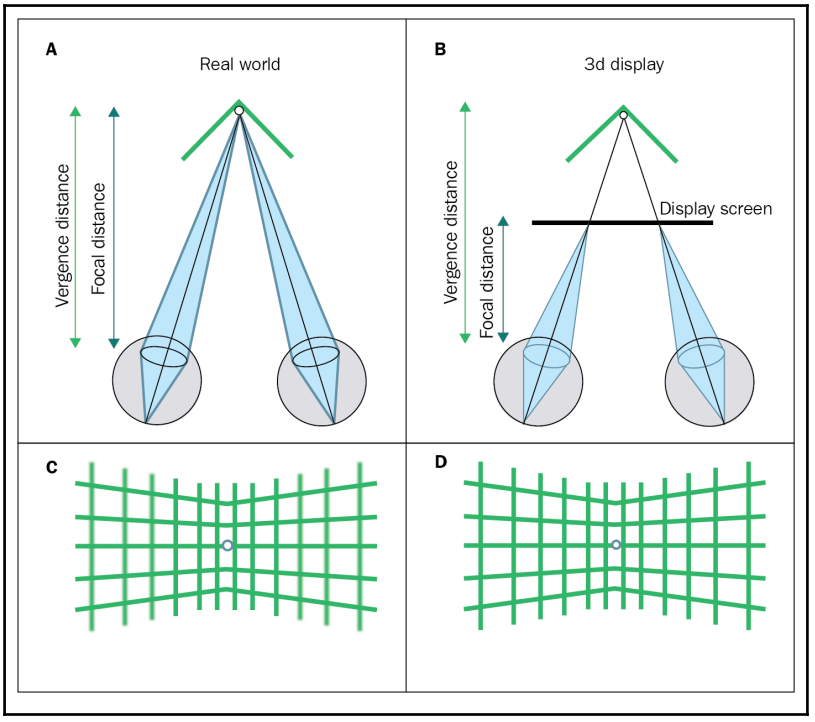
Unnatural head movements

You may not have thought about it before, but looking around in a traditional **first-person shooter** (**FPS**) is quite different than looking around using your head. The right analog stick is typically used to direct the camera and make adjustments as necessary, but in VR, players actually move their head instead of using their thumb to move their virtual head. Don't expect players in VR to be able to make the same snappy pivots and 180-degree turns on a dime that are simple in a regular console game.

Vergence-accommodation conflict

Another limitation to consider when designing your VR game is what's called a **vergence-accommodation conflict**. This is basically what happens when the distance to the point of your focus, or vergence (that is, an object in VR), is notably different than your focal distance, or where your eyes are actually focusing on the screen in front of you.

This image from a research article in the *Journal of Vision* demonstrates the conflicting difference:



Forcing the user to focus on objects that are too close or too far away for extended periods of time can cause symptoms of eye fatigue, including sore eyes or headaches. Therefore, it's important to consider placement of the pieces of your game that will draw a lot of attention.

The full article, titled *Vergence-accommodation conflicts hinder visual performance and cause visual fatigue* by David M. Hoffman, Ahna R. Girshick, Kurt Akeley, and Martin S. Banks, is available at `http://jov.arvojournals.org/article.aspx?articleid=2122611`. It is a valuable resource in avoiding depth cues that may cause eye fatigue.

Constellation tracking

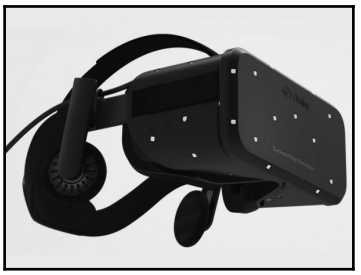
Earlier in this chapter, we briefly mentioned that HMDs sometimes monitor positional movement as well as rotational. There are a few different methods of tracking an HMD positionally, but so far, every solution includes an external component not connected to the headset. In the future as the hardware gets better, we can expect a solution to positional tracking as part of the headset itself (referred to as **inside-out tracking**).

The Oculus Rift's solution to positional tracking is called **constellation tracking**. It uses an infrared camera that faces the user to detect small infrared LED markers, invisible to the naked eye, and extrapolate positional movement values based on the number of pixels those markers move in a frame.

Here's what the tracking camera of the Oculus Rift looks like:

As long as the Rift is in view of this camera, the user can laterally move their head and the HMD's display will update to reflect it; this can be used for mechanics such as leaning into or understanding something or sticking your head out from behind a corner. The constellation tracker is capable of tracking the Rift in a seated or standing experience, which means you could even engage the player in limited full body movement. The Oculus Touch controllers, shipping in late 2016, will include an additional camera to improve the quality of tracking further.

This is an image of an early Oculus Rift prototype that shows the exposed IR trackers covering the outside of the device:



The consumer version of the Oculus Rift has these markers embedded in the strap on the back of the headset as well, so there are markers that can help the constellation system track you no matter which direction you're facing.

Generally, the more realistic a VR experience is, the more the user forgets about the world outside of it. Positional tracking adds a lot of realism to the feeling of looking around in VR, so it's a good idea to design your game in a way that will take full advantage of it.

Input with controllers, remotes, and more

What you choose as the primary input device for your VR experience will have a lot of bearing on how it feels. Out of the box, you have three possible solutions for controlling your game: the Xbox controller, the Oculus remote, or your keyboard and mouse. Once the Oculus Touch controllers mentioned in the last section are released at the end of 2016, you'll have a much more immersive way of interacting with VR experiences, and the implementation will be similar to the Xbox controller's implementation but with the added ability of tracking the user's hands just like the Rift headset is tracked.

The Xbox controller, included with the Oculus Rift and pictured here, is ergonomic but still has numerous unique inputs, meaning it's a decent choice for a game with complex controls:

For simpler games that emphasize observation and exploration over mechanical input, the Oculus remote is the perfect simple solution. It only has a few buttons, making it easy to use in tandem with the headset, and lending itself to head-based interfaces (which we'll cover in a later chapter).

This is an image of the Oculus remote, also included with the Oculus Rift:

If all else fails, you can use your keyboard and mouse, but because the keyboard is complicated and somewhat restrictive, you should always look to simpler input methods unless you have a strong particular reason for using the keyboard.

The importance of frame rate

Drops in frame rate are much more acceptable outside of VR than in VR; if a computer is good enough to run a game at 30 frames a second but not at 60 frames a second, some players are able to push through it and get accustomed to the lower frame rate. After all, we watch movies at a frame rate, sometimes even lower than 30 frames per second, so it's not unreasonable to think that a game at 30 frames per second would still be enjoyable.

Suboptimal frame rates are much more important when it comes to VR. Since the HMD takes over the entirety of what our eyes see, it needs to update the world virtually as quickly as our eyes could. If the world we're perceiving doesn't update as fast as we look around it, our brain starts to get confused again and cue the nausea. This is sometimes referred to as VR sickness, and not only does it decrease the feeling of immersion the player gets, but it can also leave them feeling ill even after removing the headset.

With the hardware in the Oculus Rift, we can update the display with a new frame up to 90 times in a second. While 120 times a second would be even more ideal, a steady frame rate of 90 will be adequate in mitigating the vast majority of nausea.

Complex, asset-dense games that require a large number of calculations every frame can start to have a monopolistic impact on a computer's hardware, meaning your VR experience may start to drop frames. This should be avoided at all costs, because sporadic frame choppiness, known as judder, is one of the fastest ways to induce VR sickness.

Asynchronous Timewarp

The good news regarding the high performance cost of the Oculus Rift is that you're not without help. **Asynchronous Timewarp** (**ATW**) is a rendering technique that helps fill delays in rendering with a calculated intermediary frame.

In essence, ATW warps an image based on the user's head movement, giving the appearance of multiple rendered frames but actually only modifying one while the rest are generated. All rendered images need to be warped at a baseline level so they don't appear skewed when viewed through the lenses, so adding an extra step to the warping process is relatively inexpensive for your hardware.

ATW can go a long way in making up for lost time in your VR experience, but it's important not to rely too heavily on it, because like every other rendering trick, it has its limits. For instance, with purely rotational ATW, the user can experience positional judder, which causes objects close to the user to appear blurry or doubled noticeable.

This screenshot of a submarine interior, provided by Oculus, demonstrates the perceived effect of positional judder:

ATW calculations are relatively simple because they only take rotation into account. Lateral movement is an entirely different problem; in 2016 Oculus announced work on a complementary feature called Asynchronous Spacewarp, which will perform predictions for head position as well. We'll see Asynchronous Timewarp at work in a later chapter, when we focus on performance and optimization of VR experiences.

Installing the Oculus runtime

Now that we've covered the basics of VR in theory, it's time for some practice. The first thing you'll have to do is download the Oculus runtime, which is the background process responsible for handling all of the activity in your Rift.

Open a web browser and navigate to `http://www.oculus.com/setup`. This page will provide a download link to the Rift setup package, which includes the runtime you need and Oculus home, the central hub for browsing and starting all of your Oculus software.

Setting up the Rift requires an Internet connection and about 30-60 minutes of setup time. The setup utility will guide you through the process step by step on screen.

After all of the software required to run the Rift is installed, you'll run through a quick hardware calibration step that will set up your constellation tracker, remote, and the headset itself. It will also help you set values such as height and **interpupillary distance** (**IPD**).

IPD is the distance between the pupils of your eyes, and even though it may seem like everyone's IPDs are similar to each other, significant inaccuracies can affect the way we perceive scale. Scale is everything in VR, especially when you're trying to use size to convey a mood or purpose, so ensuring your IPD is accurate is important in experiencing content exactly as it was meant to be experienced.

Summary

In this chapter, we approached the topic of virtual reality from a fundamental level. The HMD is the crux of modern VR simulation, and it uses motion tracking components as well as peripherals such as the constellation system to create immersive experiences that transport the player into a virtual world.

Now that we've covered the hardware, development techniques, and use cases of virtual reality—particularly the Oculus Rift—you're probably beginning to think about what you'd like to create in virtual reality yourself. In the next chapter, we'll be diving into implementations using the Unity3D engine, and you'll take your first step into VR development.

2
Stepping into Virtual Reality

Now that you're familiar with the concepts that make up the Oculus Rift and virtual reality in general, you're ready to get your feet wet. In this chapter, we'll be diving into Unity3D, which is one of the most popular engines to develop VR on for beginners and experts alike.

This chapter will cover the following topics:

- Getting started with Unity
- Incorporating the Oculus Utilities package
- Taking your first steps in VR
- Enabling interaction with gaze-based mechanics
- Unlocking the power of Unity's VR API
- Building and running your first VR app

Getting started with Unity

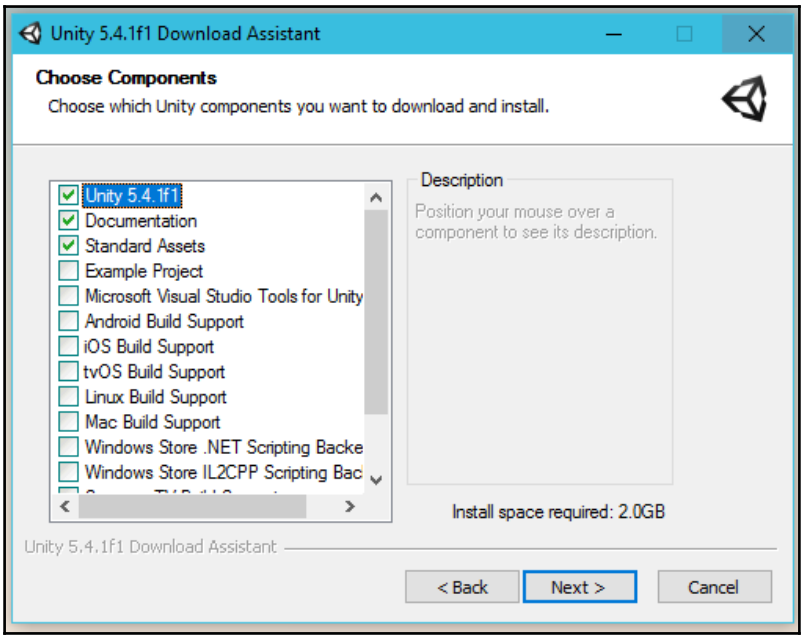
Unity3D is a complex engine, but it's very easy to get started with. In this section, we'll cover every aspect of the engine you need in order to build and deploy your first VR experience, but we'll be exploring different capabilities of the engine every chapter. Unity3D offers support for a few different languages, but all code we write in this book will be C#.

Installing the Unity3D engine

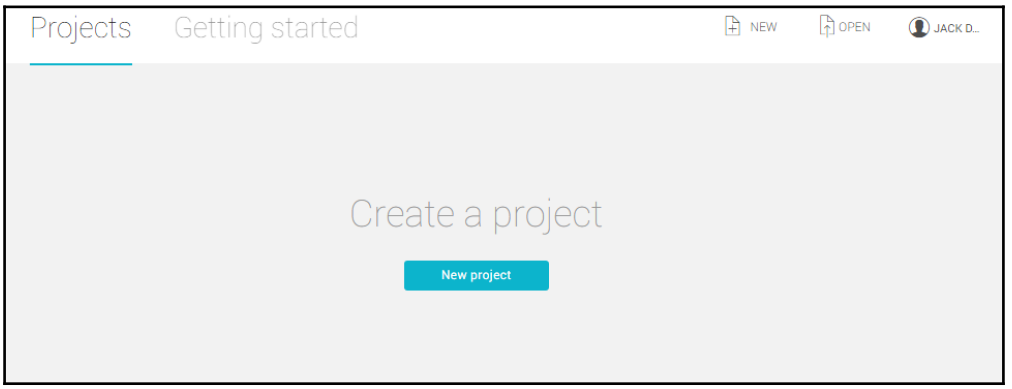
To install Unity, navigate to http://www.unity3d.com/ and download the installation assistant from the front page. Unity offers a Personal and Professional edition; the Personal edition is free and contains everything you need to create a VR application. Also, create a Unity account on the page to manage your license and other account-based services.

Unity's Professional edition isn't required for virtual reality development, but it does provide some helpful tools for finalizing your game before you sell it, such as the ability to remove the Unity splash screen and the option to make builds on Unity's cloud as opposed to making them on your local computer. The good news is that you can upgrade a Personal project to Professional relatively easily at any time. You will, however, need to upgrade to Professional if your game made in Unity Personal grosses over USD$100,000.

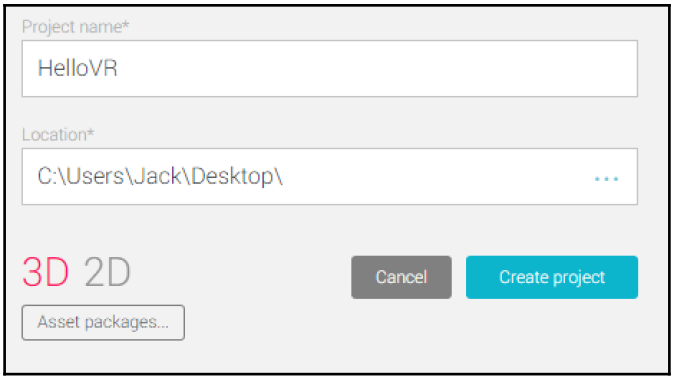
Download and run the Unity installer. When it prompts you for what components you want to install, it wouldn't hurt to keep **Documentation** and **Example Project** checked if you want to have a few in-depth Unity references on hand.



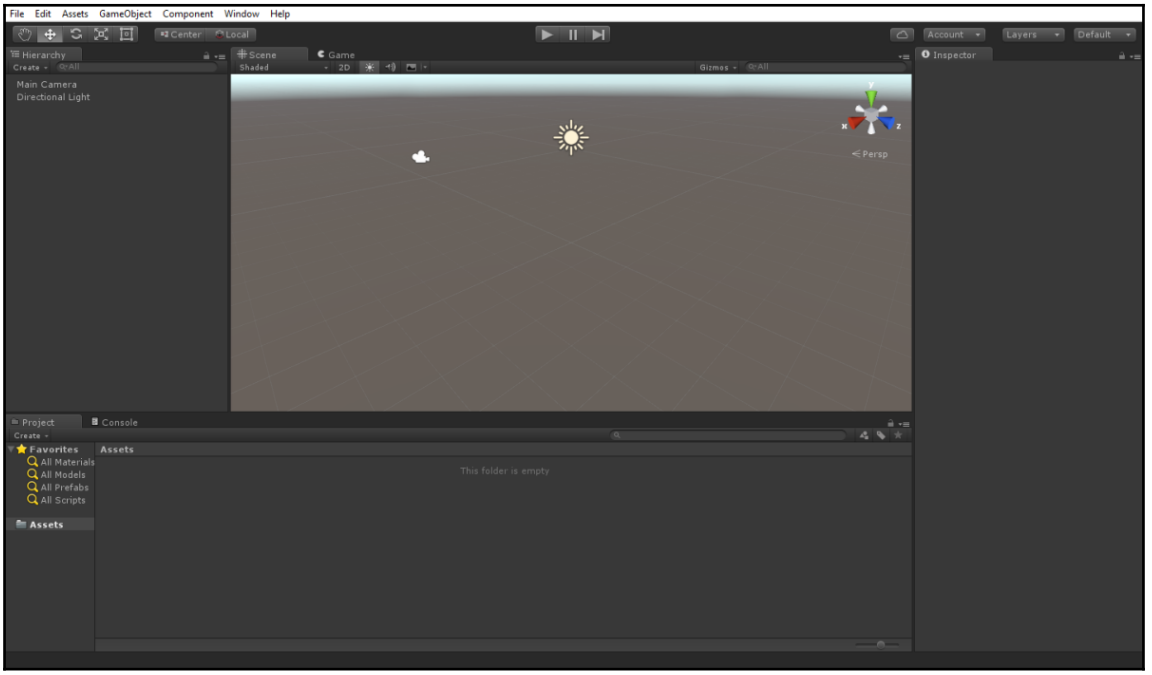
Once Unity has finished installing, launch it and log in with your Unity account. Select the **New project** button in the center of the launcher window or the **NEW** button in the upper-right corner. You can see both buttons in the following screenshot:



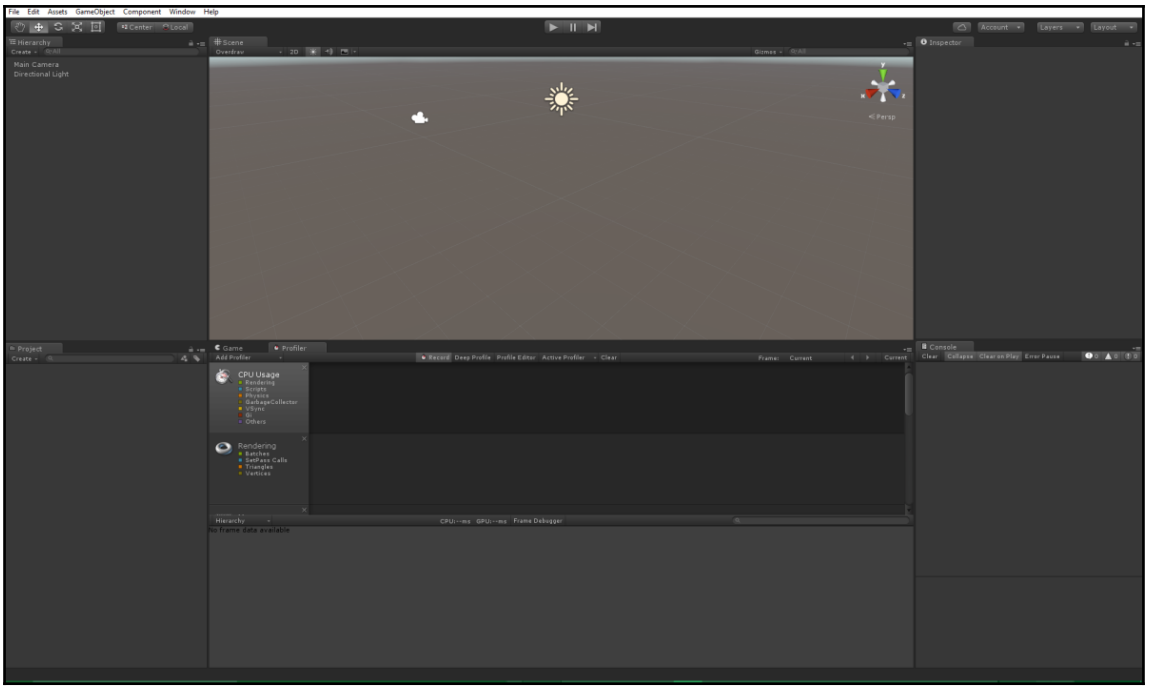
Choose a name for your project and specify a folder for the project to exist in. Also, be sure the **3D** option is selected, not the **2D** option, as shown in the following screenshot:



Once you click on **Create project**, the Unity Editor will open with a layout that looks something like the next screenshot:



Every window inside the Unity Editor can be dragged around by the window's tab and positioned wherever you like. It's largely a matter of preference, but for the sake of consistency, this book will be using a simple symmetrical layout, as shown in the following screenshot:



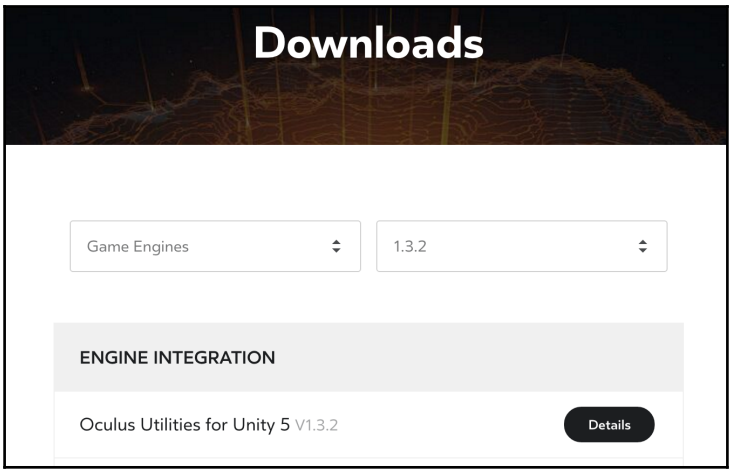
The majority of this layout consists of the same windows as the default layout, simply rearranged. However, you'll notice a new window in a tab right next to the **Game** window: the **Profiler** window. The **Profiler** window can be added by clicking on **Profiler** in the **Windows** menu on the top toolbar. We'll use the Unity profiler to record and analyze the performance of our game to make sure we're always running at optimal frame rates. We'll dig more into its features in Chapter 3, *Improving Performance and Avoiding Discomfort*.

If you find a Unity layout that you totally fall in love with, you can save it by clicking on the **Layout** button in the upper-right section of the Editor and selecting the **Save Layout...** option.

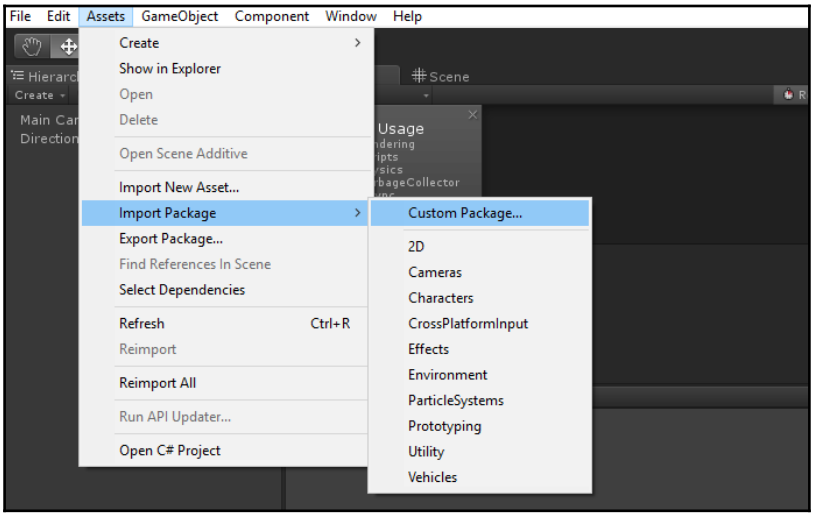
Incorporating the Oculus Utilities package

Unity supports native VR development out of the box, so you don't need anything besides the engine itself to build VR experiences. However, Oculus provides a set of optional tools to developers that take care of some of the boilerplate functionality. In this section, we'll download the Oculus Utilities package for Unity and import it to our project.

Open a web browser and navigate to https://developer.oculus.com/downloads/. Select **Game Engines** in the category dropdown and then select the most recent release version (1.3.2 at the time of writing). Look for a package called Oculus Utilities for Unity 5, as shown in the following screenshot:

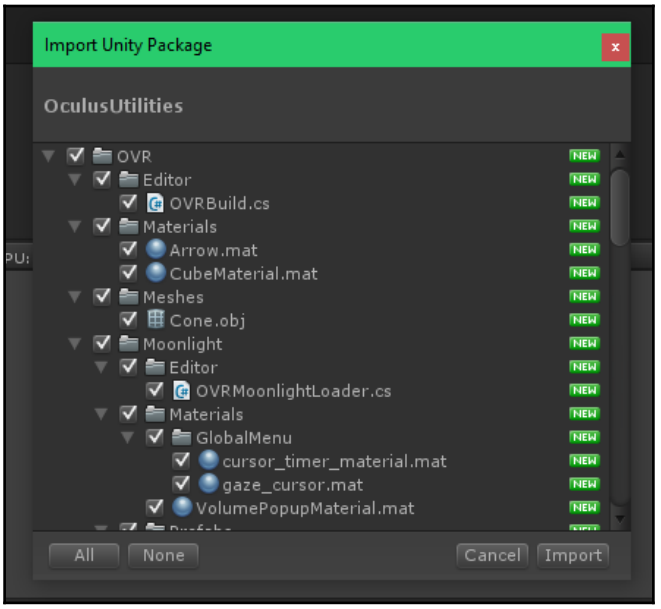


Click on **Details**, accept the agreement, and download the package and save it to a location where you can easily access it later, like your desktop. Once it has finished downloading, you can import it by mousing over **Import Package** in the **Assets** dropdown and selecting the **Custom Package…** option, as shown in the following screenshot:



Once you've navigated to the location of the Oculus Utilities package and selected it, you'll see a window that lets you pick what components of the package are imported into the project, just in case you only want a certain part of a package.

In this case, you want all of the components of the package selected for import, as shown in the following screenshot:



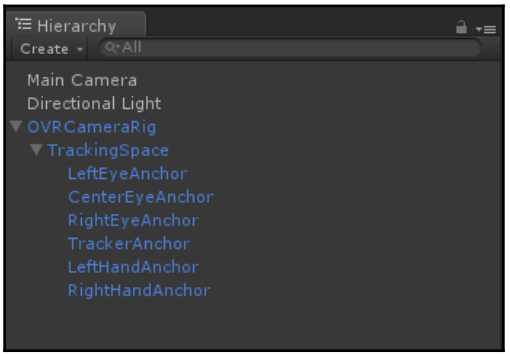
Import all of the components within the package and then you'll see them show up in your **Project** window, organized within the `OVR` and `Plugins` folders.

There are a few folder names that Unity treats as special cases; `Plugins` is one of them. Files in the `Plugins` folder can be used as libraries or packages that your game depends on during compilation, such as the DLL required for communication between the Oculus Rift and the Unity Engine.

We'll focus on each of the components in this package as we need them, but for now, turn your attention to the `Prefabs` folder within the `OVR` folder in your **Project** window. The **OVRCameraRig** prefab is an object that contains a minimal configuration for viewing a scene with the Rift, and it will serve as a good jumping off point for us in this chapter.

Click and drag the **OVRCameraRig** from your **Project** window and release it over the **Hierarchy** window to add an instance of that prefab to your scene. You'll notice an arrow to the left of the object in your hierarchy; click on this arrow once to expand the object and view its children.

If you fully expand the **OVRCameraRig**, you'll notice that it contains several anchors, as shown in the following screenshot:



Later in development, these anchors can be treated as attachment points for objects attached to the player. For instance, if you wanted a user interface to stay oriented to wherever the player is looking, you could position it in front of them and then parent it to the **CenterEyeAnchor**.

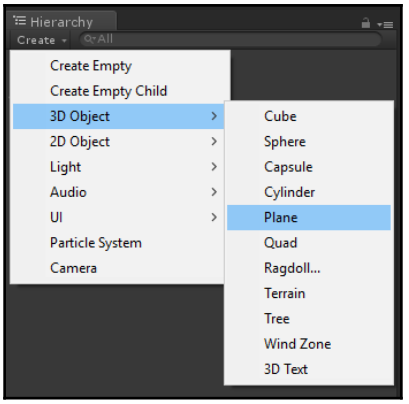
Now that we've got the ability to view a scene in VR, let's set it up with some prop objects to look at and then take our first step inside.

Taking your first steps in VR

The first step we'll take in setting up our first VR scene is to delete **Main Camera** from the hierarchy. This is a standard object that Unity adds to every scene in order to render it, but since the Rift is responsible for rendering VR scenes, we don't need a standard camera anymore.

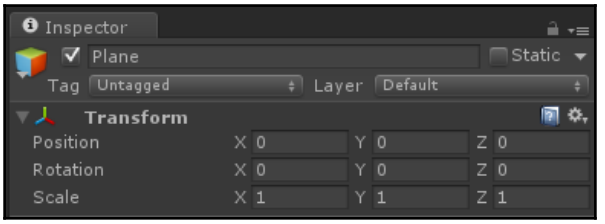
After you delete the default camera, it's time to add a place for the player to stand. We'll get into creating natural terrains later in this book, but for now we'll just make a simple flat floor.

Click on the **Create** button in the upper-left corner of the **Hierarchy** window and select **Plane** from the **3D Object** menu, as shown in the following screenshot:



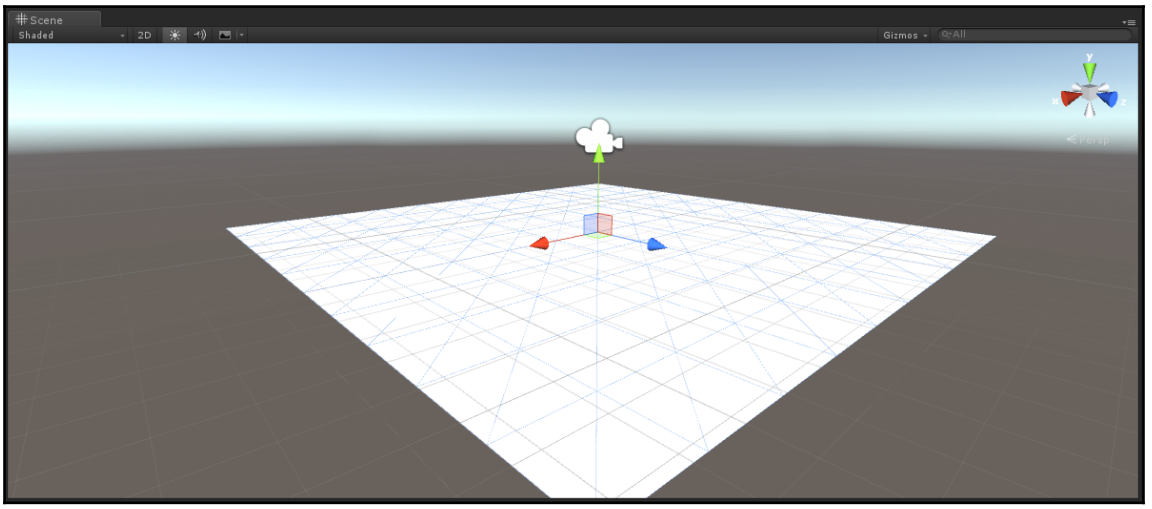
Once your plane has been created, click on it once in the hierarchy to display its properties in the **Inspector** window, which include its position and rotation. In this book's Unity layout, the **Inspector** window is at the same height as the **Hierarchy** window on the far right of the editor window.

At the top of the **Inspector** window with the new plane selected, you'll see a **Transform** component, which is required for any object that occupies physical space in our scene, as shown in the following screenshot:



Right now, the plane is at the origin point of the scene, or (0, 0, 0). Our **OVRCameraRig** is also at the origin point, but the position of the **OVRCameraRig** represents the eye height of the player, so we should move the plane down to be at a natural floor height.

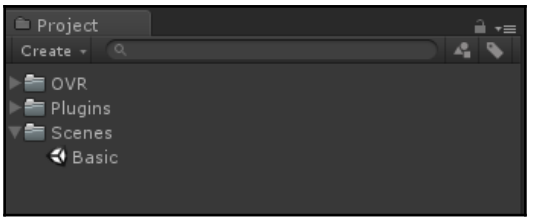
Set the **Y** component of the plane's position to `-1.7`. You can then look in the **Scene** window to see the player's position relative to the height of the plane, as shown in the following screenshot:



There are a few different ways to navigate within the **Scene** window in order to get a better look. Holding the middle mouse button and dragging will let you pan the view of the scene, whereas holding the right mouse button and dragging will let you rotate the scene from its current position. You can also hold the right mouse button and use the *W*, *A*, *S*, and *D* keys to move forward, backward, left, and right, respectively.

Now's a good time to save your scene. To keep things organized, we'll create a folder named `Scenes` in your **Project** window that will hold this scene and any others we may create.

Click on **Create** in the upper-left corner of your **Project** window and select **Folder**. Enter `Scenes` as the folder name and press *Enter*. Then click on the **Save** option in the **File** menu at the top of the editor (or use the shortcut *Ctrl* + *S*) and save the scene as `Basic.unity` in your newly created folder. Once the save is complete, you'll see it in the editor underneath the `Scenes` folder, as shown in the following screenshot:



Scripting basic player movement

Now that we've got a floor, it's time to walk around on it. This will be the first code we'll write to add functionality to the **OVRCameraRig**, and the first step is to create a script and attach it.

Create a new folder and name it `Scripts`. Right-click on the `Scripts` folder and select **C# Script** under the **Create** menu. Name the script `PlayerController` and double-click on it to open it in the code editor that came with your Unity installation. You'll notice that the script isn't empty; it already has a class declaration named after the script name as well as `Start` and `Update` functions.

The `Start` and `Update` functions are two examples of Unity's built-in functions. `Start` is called once when the script is initialized, and `Update` is called once per frame for systems that require constant action.

Since we want to be constantly checking for user input, the code that we write right now will be in our `Update` function. Add an `if` statement on the first line of `Update` that checks if the player is pressing the *W* key:

```
void Update()
{
  if(Input.GetKey(KeyCode.W))
  {
  }
}
```

The code inside our `if` statement will be run every frame as long as the *W* key is held down, so we want to move the player forward only slightly with Unity's `Translate` function. Add the following logic to the `Update` function:

```
void Update()
{
  if(Input.GetKey(KeyCode.W))
  {
    gameObject.transform.Translate
    (transform.forward * Time.deltaTime);
  }
}
```

A lot happens on this single line. The `gameObject` keyword gets a reference to whatever object this script is attached to, the `transform` keyword gets a reference to that object's **Transform** component (if it has one), and the `Translate` function moves the object according to the parameter, which in this case is the object's forward direction.

You'll also notice that we multiply the translation value by `Time.deltaTime`. This value is the amount of time in seconds since the last frame, and it guarantees that the amount of space traveled remains the same even if something causes a delay in frame rendering while the *W* key is held.

Expand the logic in the `Update` function to handle backward, left, and right movement in addition to the forward movement you've already coded:

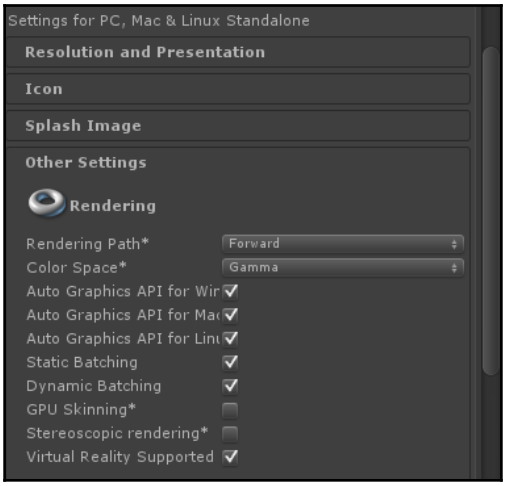
```
void Update()
{
  if(Input.GetKey(KeyCode.W))
  {
    gameObject.transform.Translate
    (transform.forward * Time.deltaTime);
  }
  if(Input.GetKey(KeyCode.S))
  {
    gameObject.transform.Translate(-transform.forward *
    Time.deltaTime);
  }
  if(Input.GetKey(KeyCode.A))
  {
    gameObject.transform.Translate
    (-transform.right * Time.deltaTime);
  }
  if(Input.GetKey(KeyCode.D))
  {
    gameObject.transform.Translate
```

```
        (transform.right * Time.deltaTime);
    }
  }
```

To add this script's functionality to the player, simply drag the **PlayerController** script from the **Project** window and release it over the **OVRCameraRig**. You will see the script as another component in the **Inspector** window.

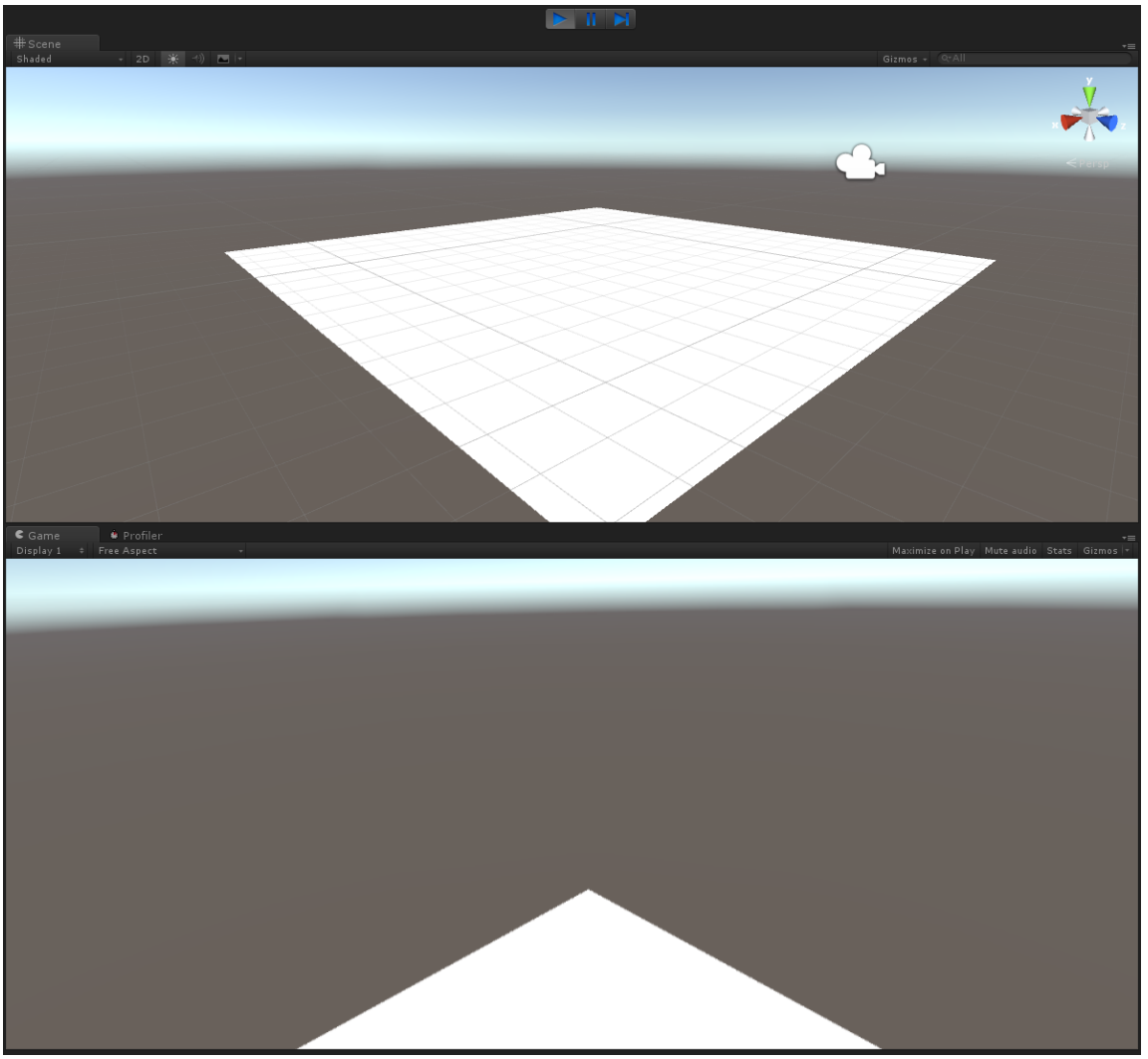
Without further ado, it's time to experience your VR world for the first time. As minimal as it is, it's always a good idea to test new code, and you'll use this code to take a few steps around your ground plane.

To test your build, you'll need to enable VR support in Unity. Open the **Edit** menu on the top toolbar and select **Player** within **Project Settings**. Scroll to find the **Virtual Reality Supported** property and check it:



Next, put on your Rift and press the Play button at the top of the Unity Editor. Because Unity supports native development for the Rift, it will render your scene to the headset before you even build an executable (we'll get to that process later). Use the *W*, *A*, *S*, and *D* keys to take a few steps around your scene.

In the editor window on your primary monitor, you'll see that the **Game** window displays a mirrored image of what's being sent to the Rift, and the **Scene** window shows the position of the **OVRCameraRig** and updates it as you move it:



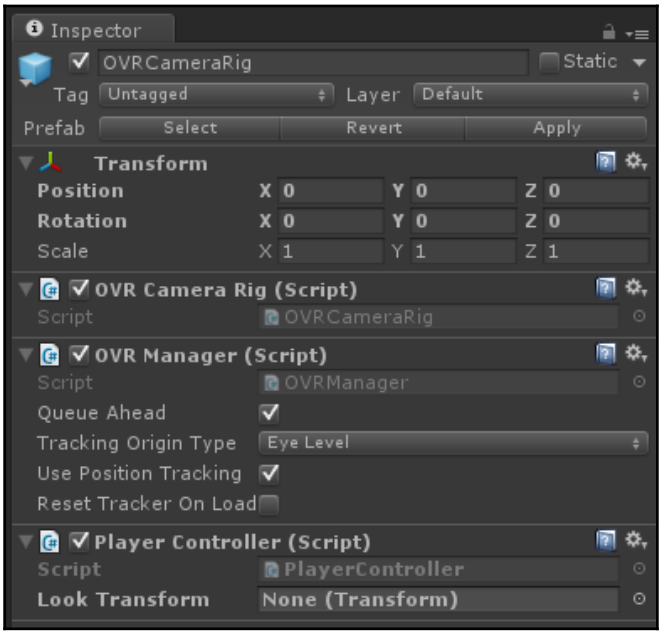
If you looked around, you may have noticed that the directionality doesn't change; the *W* key will move you in the same direction no matter where you're looking. It would make more sense for the forward direction to always be based on the way the player is looking, so let's modify our `PlayerController` to take that into account.

If we're going to use a transform that doesn't belong to the object this script is attached to, we'll need to create a reference to it. Add the following variable to the top of your class, right before the `Start` function:

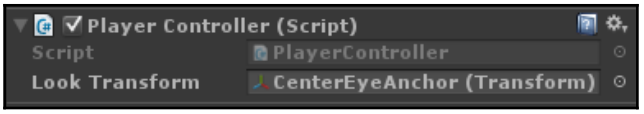
```
[SerializeField]
private Transform lookTransform;

// Use this for initialization
Start()
{

}
```

The `[SerializeField]` attribute that you specified tells Unity that it's a value we want to see exposed in the **Inspector** window. Not only will this let us see the value of this variable in **Inspector**, but it will also let us set the variable from within the Unity editor as well.

Sure enough, save the script and highlight your **OVRCameraRig** in the **Inspector**, and you'll see the **Look Transform** field on the **PlayerController** script component at the bottom of the component list, as shown in the following screenshot:



Here's where the anchors on the **OVRCameraRig** we mentioned earlier come in. We need a reference to a transform that reflects where the user's head is pointing, so a reference to the eyes is just the ticket. Click and drag the **CenterEyeAnchor** object from the **OVRCameraRig** into the **Look Transform** field to assign it. You'll then see the name of the anchor reflected in the property field, as shown in the following screenshot:



Now that the field is linked, you can access this transform directly in your script by referencing the `lookTransform` variable that corresponds to the field.

All that's left is to swap out the transform references in the `Update` function, like so:

```
void Update()
{
  if(Input.GetKey(KeyCode.W))
  {
    gameObject.transform.Translate( lookTransform
    .forward * Time.deltaTime);
  }
  if(Input.GetKey(KeyCode.S))
  {
    gameObject.transform.Translate(-lookTransform
    .forward * Time.deltaTime);
  }
  if(Input.GetKey(KeyCode.A))
  {
    gameObject.transform.Translate(-lookTransform
    .right * Time.deltaTime);
  }
  if(Input.GetKey(KeyCode.D))
  {
    gameObject.transform.Translate(lookTransform
    .right * Time.deltaTime);
  }
}
```

Press Play again and walk around your scene. Your forward direction is now based on where you're looking, so you can change direction while holding *W* simply by moving your head.

Now that you've created a simple player controller, you can sample a more complex one that Oculus has provided in their Utilities package. To do this, replace the **OVRCameraRig** in your scene with an instance of the **OVRPlayerController** prefab from the same folder. This prefab already has a complex script for handling all kinds of navigation, including walking, turning, and jumping.

Enabling interaction with gaze-based mechanics

If you've played a video game before (chances are high if you're reading this book), navigating with *W*, *A*, *S*, and *D* might seem familiar. It's a proven input method that has been a solid common denominator for many keyboard-operated games. However, VR enables us to implement input methods that haven't been possible before now, for instance, using your eyes.

One of the simplest ways that gaze is used in virtual reality experiences is to teleport or jump from point to point. This is sometimes seen as more ideal than the movement we implemented in the last section since any kind of smooth locomotion without a response from our inner ear can make us feel nauseated.

Gaze-based teleporting circumvents this issue by instantaneously moving the player to whatever point they were looking at when a key or button was pressed. Because there's no perceived inertia, there is no eye/ear disconnect; the player is stationary throughout the whole process.

Implementing gaze-based teleporting

In this section, we'll build upon the `PlayerController` class that we've already created and add the ability to instantly jump to the player's focal point. We'll do this using a method called **raycasting**, which essentially involves drawing a line from a point in a specific direction and recording information about what it hit.

Begin by adding a new function to your `PlayerController` class called `Teleport` just after the `Start` and `Update` functions. Add two lines to the beginning of it, one to initialize a `Ray` class that we can cast based on our look transform and another to initialize a `RaycastHit` object that will collect our results:

```
private void Teleport()
{
  Ray ray = new Ray(lookTransform.position, lookTransform.forward);
  RaycastHit hit;
}
```

The `ray` object is initialized at the position of our eye anchor and cast in its forward direction. The `RaycastHit` object doesn't need to be initialized; it will be initialized and set automatically by the raycasting function.

Create an `if` statement that contains the actual raycast logic:

```
private void Teleport()
{
  Ray ray = new Ray(lookTransform.position,
  lookTransform.forward);
  RaycastHit hit;
  if(Physics.Raycast(ray, out hit, Mathf.Infinity))
  {
  }
}
```

Anything within that raycast block will be run whenever the ray hits a collider in the scene. The `out` keyword before our `hit` variable tells it to route all outgoing information from the raycast to hit, and the third parameter tells it how far to cast (we'll use an infinite distance for now).

Next, we'll add some logic to move the player by assigning the location of the raycast hit to their position:

```
if(Physics.Raycast(ray, out hit, Mathf.infinity))
{
  gameObject.transform.position = hit.point;
}
```

We want the player to maintain a constant height from the ground, so we'll add `1.7` to the `hit` coordinate, just as we lowered the ground plane by `1.7` down the player when we created it:

```
if(Physics.Raycast(ray, out hit, Mathf.infinity))
{
  gameObject.transform.position = hit.point;
  gameObject.transform.Translate(0f, 1.7f, 0f);
}
```

All that's left is to call the `Teleport` function whenever a button is pressed; we'll use the space bar. Add logic to your `Update` function to call `Teleport` whenever the space bar is pressed:

```
void Update()
{
  ...
  if(Input.GetKeyDown(KeyCode.Space))
  {
    Teleport();
  }
}
```

Notice that the `Input` call is to `GetKeyDown` instead of `GetKey`; this is because we don't want to perform an action for every frame that the key is held down, but instead perform an action once right when the key has just been pressed.

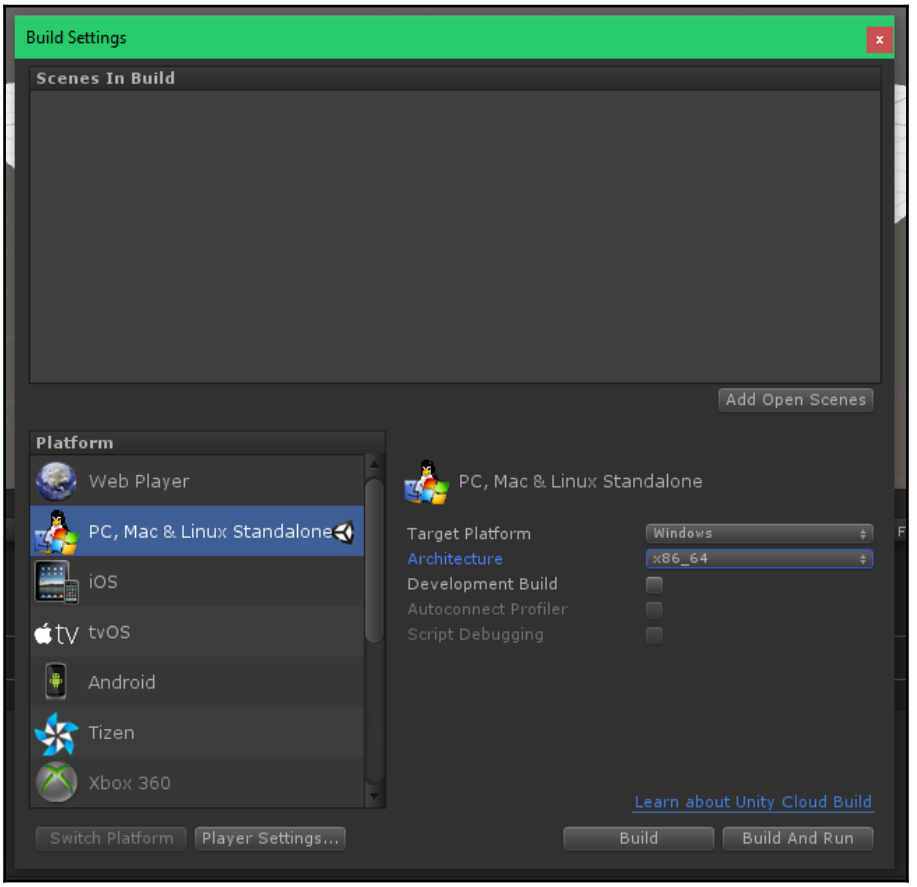
Building and running your first VR app

Up until now, testing in the Unity Editor has sufficed for getting a taste of the features you're working on. That being said, it's always good to make a proper build and run it outside of Unity to test the game exactly as your users will be experiencing it.

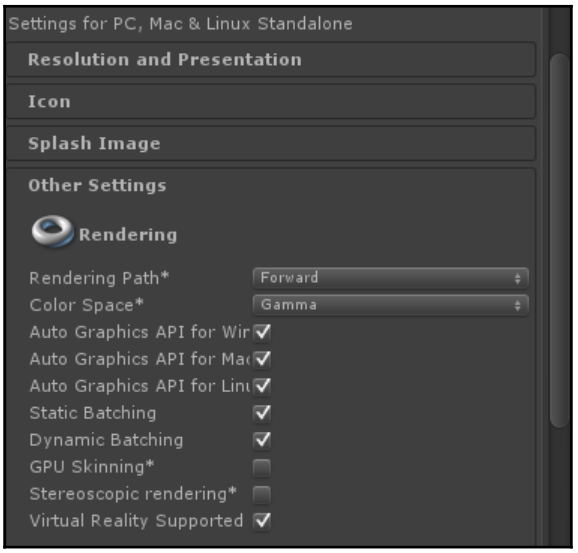
In this section, we'll go over the configuration of different settings in Unity to prepare and build your first VR executable.

Configuring Unity Build Settings for VR

Open the **File** menu at the top of the Unity Editor and select the **Build Settings…** option. In the window that appears, set **Target Platform** to `Windows` and **Architecture** to `x86_64`, as shown in the following screenshot:



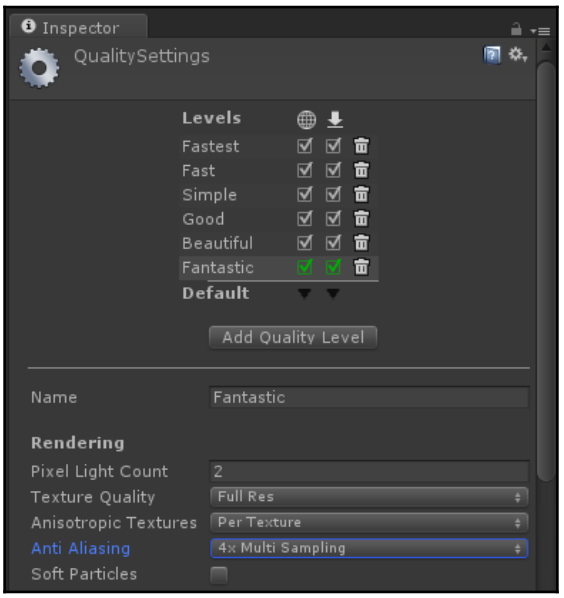
Next, click on **Quality** within the **Project Settings** section of the **Edit** menu to move onto the next configuration section, as shown in the following screenshot:



Configuring Unity Quality settings for VR

The quality settings of your project are not crucial for the functionality of your build, but they can go a long way in ensuring your game is performing at the highest fidelity possible while still maintaining a consistent and comfortable frame rate.

Open the quality settings by opening the **Edit** dropdown at the top of the editor and selecting **Quality** in the **Project Settings** menu. The quality settings will then appear in **Inspector**. Make sure your settings mirror the ones shown in the following screenshot:



After the quality settings have been squared away, reopen the **Build Settings** menu by clicking on it under **File** or using the *Ctrl* + *Shift* + *B* shortcut. Click on the **Build** button and specify a location for your build. After a moment, an executable and an accompanying data folder will appear at the specified location. From here, you can double-click on the executable and step right into your game from outside of Unity.

Summary

In this chapter, you took your first steps in VR and built your first VR executable of many. You became familiar with Unity's basic workflow, added different methods of locomotion to the player object, and, finally, configured Unity to build a standalone executable. In the next chapter, we'll tackle some performance optimizations that will keep your game running smoothly as we continue to add complex functionality to it.

3
Improving Performance and Avoiding Discomfort

Virtual reality is a powerful medium, and with great power comes great responsibility. Because virtual reality is tied directly to your senses, it's important to consider anything that might cause interruptions or appear strange to the user.

Back in `Chapter 1`, *Exploring a New Reality with the Oculus Rift*, we briefly covered the adverse effects of inconsistent or low frame rates, and the concept of Asynchronous Timewarp that the Oculus Rift uses to mitigate these effects. However, Asynchronous Timewarp does take away from the immersion of the experience and should be treated as a last resort for the best results.

In this chapter, we'll go over several techniques you can use as you develop more complex games to keep them running optimally, and ensure your users are able to experience a large amount of detail comfortably.

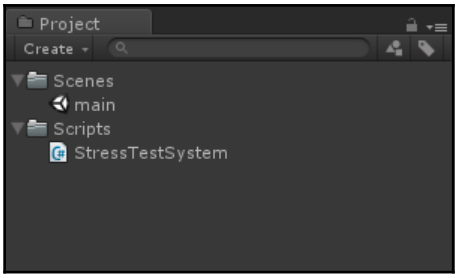
This chapter will cover the following topics:

- Using the Unity profiler
- Using coroutines to split up complex work
- Mesh optimization
- Displaying dynamic detail with LODs
- Keeping a handle on memory allocation
- Avoiding memory waste with object pools

Using the Unity profiler

The Unity3D engine comes with a full-featured profiler that's a great tool for dissecting everything that it's doing behind the scenes every frame. In this section, we'll write some functions that tax the processor and track them in the Unity profiler.

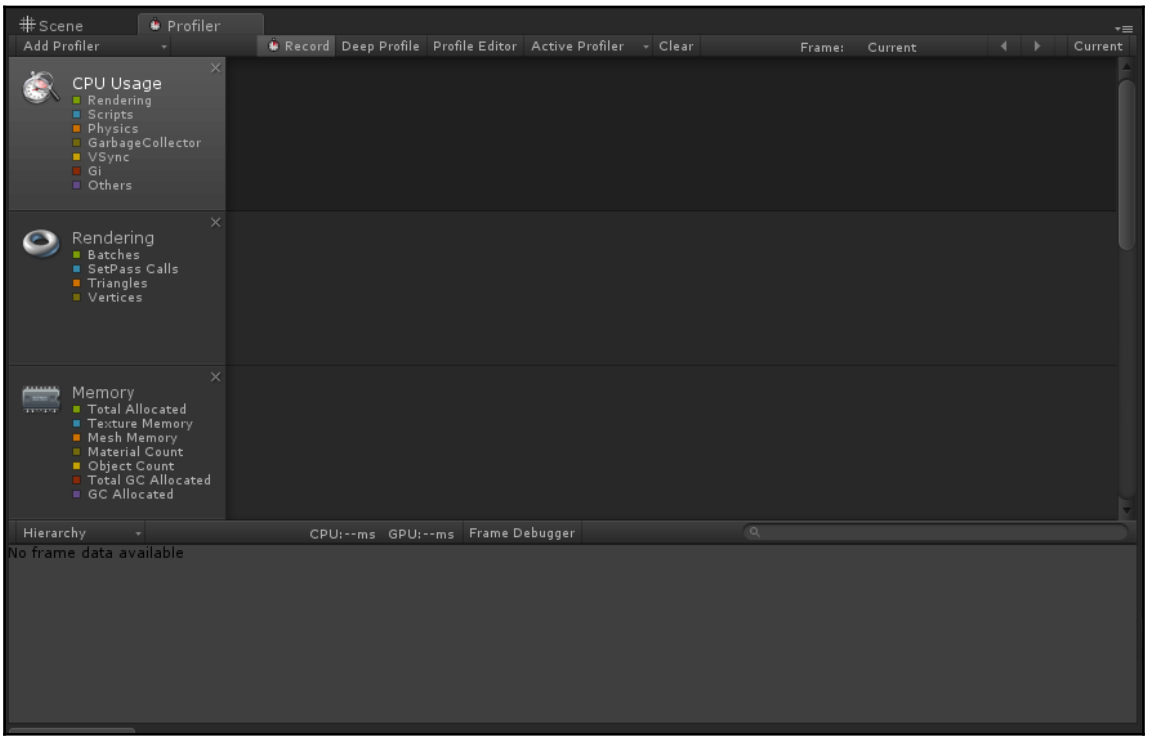
Create a new Unity project called `PerformanceSandbox`. Save a scene called `main.unity` in a new folder called `Scenes`. Next, create a folder called `Scripts` and create a new C# script in it named `StressTestSystem`. When you're done, your **Project** window should look something like the following screenshot:



Now is a good time to add the **Profiler** window to your Editor layout. Select **Profiler** from the **Window** menu on the toolbar at the top of the editor. Add the window as a new tab next to the **Scene** window. It's a good idea to keep the **Profiler** window in a place where you can see it at the same time as your **Game** view, so you can see what Unity is rendering while you analyze it.

You can quickly open the profiler at any time by pressing *Ctrl + 7* on Windows or *Command + 7* on OS X.

You'll see several different rows within the profiler for analyzing performance from different angles, but the ones we'll focus on in this chapter are the top three: **CPU Usage**, **Rendering**, and **Memory**:



CPU Usage refers to how much the processor needs to manage to keep your game running; this includes game logic that you write in scripts as well as the upkeep of the game itself, such as sending camera information to be rendered.

This brings us to rendering. The **Rendering** section of the profiler covers everything that your GPU (video card) does. The CPU tells the GPU what's being rendered, and the GPU is responsible for the complex math involved in rendering it. Last but not least, the **Memory** section is where you'll find all of the allocations Unity makes to store data for every object in your game.

Analyzing CPU usage

To sample some data from the profiler, we'll need to add some functionality to our `StressTestSystem.cs`. Open up the script and create a new function called `GenerateRandomNumbers`:

```
private float[] GenerateRandomNumbers(int numberCount)
{

}
```

The preceding function will return an array of floating-point numbers based on a provided count. Initialize an array and populate each entry with a random number between 0 and 100:

```
private float[] GenerateRandomNumbers(int numberCount)
{
  float[] randomNumbers = new float[numberCount];
  for(int i = 0; i < numberCount; ++i)
  {
    randomNumbers[i] = Random.Range(0f, 100f);
  }

  return randomNumbers;
}
```

Now, let's add a call to generate one hundred random numbers in the `Start` function of `StressTestSystem`:

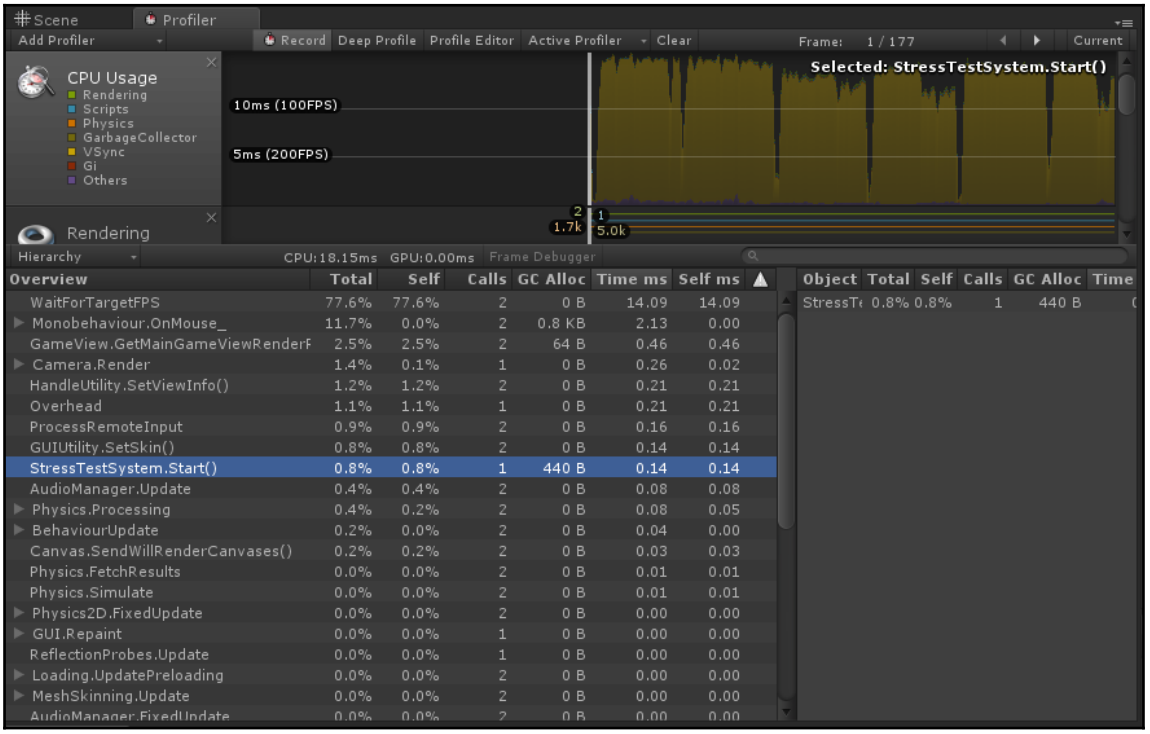
```
// Use this for initialization
void Start()
{
  GenerateRandomNumbers(100);
}
```

As with all Unity scripts, we'll have to attach it as a component to an object before it can run in our scene. Click on **Create Empty** in the **Create** menu at the top of the **Hierarchy** window to create a new object and name it `StressTestSystem`. Then, drag your **StressTestSystem** script from the **Project** window and drop it over your new object to add it.

Lastly, make sure the **Record** toggle in the **Profiler** window is switched on and then press Play. Let the data collect in the profiler for a few seconds, but not long enough for the timeline to run off the left side, and then press Play again to stop it; the most recent data in the profiler will stay even after playback has ended.

Click your mouse on top of the **CPU Usage** row to select a frame and then drag your mouse to move the selection back to the first frame. You'll see a list of all of the functions that were running during that frame, and somewhere in your list, you'll see something familiar: `StressTestSystem.Start()`.

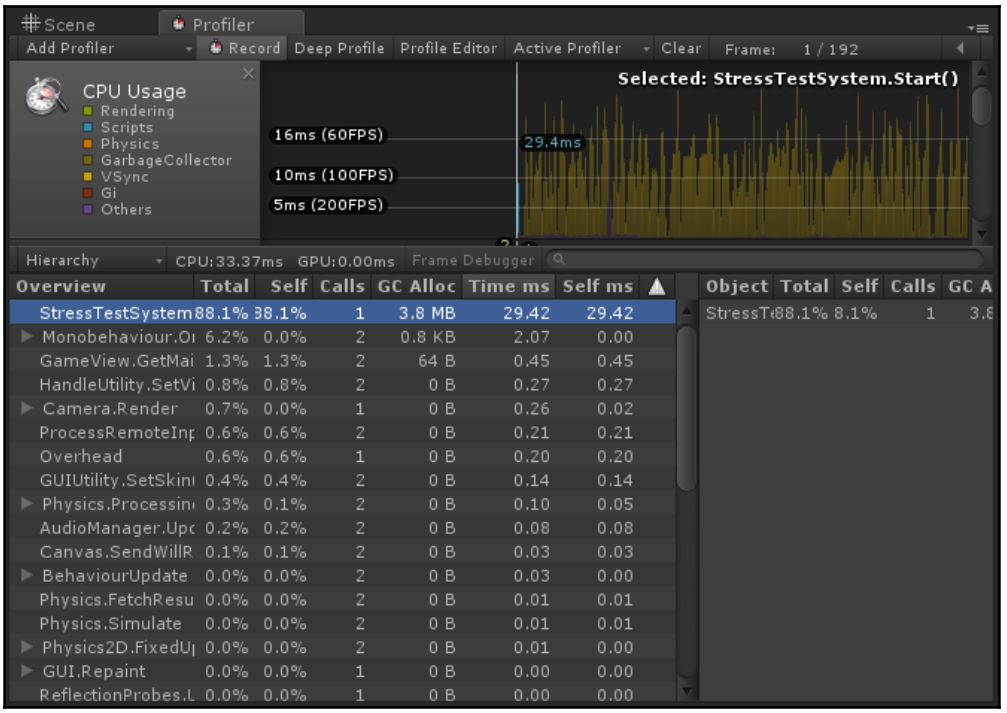
In the following screenshot, we've clicked our function once to highlight it, which isolates it in the graph of activity and displays more precise information to the right of the **Hierarchy** window:



You'll notice that the total CPU usage of the function is very small, only **0.8%** in the pictured profiler. We are, after all, only generating 100 random numbers, which is trivial for a modern CPU. Let's edit our call in the `Start` function to generate one million random numbers instead:

```
void Start()
{
  GenerateRandomNumbers(1000000);
}
```

Now press **Clear** to reset the profiler's timeline and then let Unity run for a few more frames. Stop it again to focus on the first frame. You'll notice a clear representation of the function in the graph now (colored blue), and a substantially higher percentage of CPU usage:



If your game ever begins to lag, this is the first place you should come to figure out what's taking the most energy. It's very common for the CPU activity to be the bottleneck for performance of Unity games; always be on the lookout for functions that monopolize the CPU in the way that our random number function did.

Using coroutines to split up complex work

So you've got a function that taxes the CPU too much. What do you do now? Sometimes you can pare down your experience to create a little more breathing room for your CPU, but obviously it's preferable to figure out a way to split all of the workup so that it can be done in a more conservative manner, even if it takes slightly longer.

This is where Unity's coroutine system comes in. Coroutines are functions that can run for a while, yield to other functions, and then pick up right where they left off. In this section, we'll modify the `GenerateRandomNumbers` function to be a coroutine so that it can generate one million random numbers without ever blocking the other functions the CPU is responsible for.

There are a few caveats to coroutine functions as well, which we'll cover in this section. One that will immediately impact our function is the required return type; all coroutines must return an `IEnumerator` object, so we'll have to change `GenerateRandomNumbers` to return that instead of `float[]`. Also, remove the `return` line after the `for` loop:

```
private IEnumerator GenerateRandomNumbers(int numberCount)
{
  float[] randomNumbers = new float[numberCount];
  for(int i = 0; i < numberCount; ++i)
  {
    randomNumbers[i] = Random.Range(0f, 100f);
  }
}
```

Speaking of returns, they work a little differently in coroutines. Typically, the `yield` keyword is used before `return` or any other yield parameters. The simplest valid returns are yield break, which ends a coroutine immediately and does not come back to it, and yield returns null, which ends a coroutine's execution for the current frame and picks it up in the same place in the next frame.

The skill of making coroutines is all about when to yield. You want to find a good balance of yields in your coroutines: too few and your function will put too much work on the CPU, too many and the function will take too long to complete while leaving CPU to waste.

We already know that it didn't take long to compute 100 random numbers in a single frame, so let's use that information to place our yield by yielding every thousand iterations. We can use the `%` operator to check if the `i` variable in the `for` loop is divisible by `1,000`, effectively returning 0 every thousandth number. Add the following logic in your `for` loop:

```
private IEnumerator GenerateRandomNumbers(int numberCount)
{
  float[] randomNumbers = new float[numberCount];
  for(int i = 0; i < numberCount; ++i)
  {
    randomNumbers[i] = Random.Range(0f, 100f);
    // Yield every thousand iterations
    if(i % 1000 == 0) yield return null;
  }
}
```

The last thing we'll have to change to run this function as a coroutine is how it's invoked. You'll use a function called `StartCoroutine` to initiate it, which takes the desired function as a parameter with that function's parameters included.

Change your call in the `Start` function so that it looks like the following:

```
void Start()
{
  StartCoroutine(GenerateRandomNumbers(1000000));
}
```

Your function can still be called without the `StartCoroutine` function, but it won't be able to continue running after its first yield. If it ever seems that a coroutine stops in the middle of a task, make sure it's being invoked properly with `StartCoroutine`.

Press Play, let it run for a few seconds and stop it again. You'll notice that your function is being run over the course of several frames instead of all at once and it's taking up a relatively small percentage of the CPU's power.

There's no single right way to yield in a coroutine. There are plenty of variables to consider, for instance, how many other coroutines are running at the same time? How long can we afford to wait for the information that the coroutine is providing? The best way to be sure that your coroutine is running as optimally as possible is to yield precisely and intelligently based on the operation at hand.

Mesh optimization

At this point, we've focused quite a bit on CPU usage, but we haven't covered optimizations for rendering, which is crucial for high-detail scenes that contain a lot of complex objects. In this section, we'll look at how to analyze the cost of rendering 3D models, and the tools that can help us render as efficiently as possible.

Optimizing a mesh using Simplygon

Meshes use vertices and faces to define the shape of an object. The more complex and detailed the object is, the more vertices and faces are needed to describe it, and as you probably guessed, more vertices and faces lead to high render costs.

Fortunately, there are tools available that allow for simple algorithmic optimization with the click of a button. These algorithms generate new meshes that sacrifice a small level of aesthetic accuracy to drastically reduce the number of vertices and faces in the model. One such tool is called *Simplygon*, which we'll use in this section to optimize a dense high-poly model.

The model that we'll use is the famous Stanford bunny. The Stanford bunny was created in 1994 by Greg Turk and Marc Levoy and has been used since to test all kinds of 3D mesh algorithms.

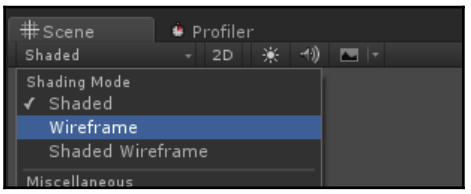
The following is a sample rendering of the model we'll be working with from the Stanford University Computer Graphics Laboratory, with basic texturing and color applied:

Create a new folder in your Unity project called `Models`. Find the `StanfordBunny.obj` file included with this book and drag it over the Unity Editor to the `Models` folder to import it into your project.

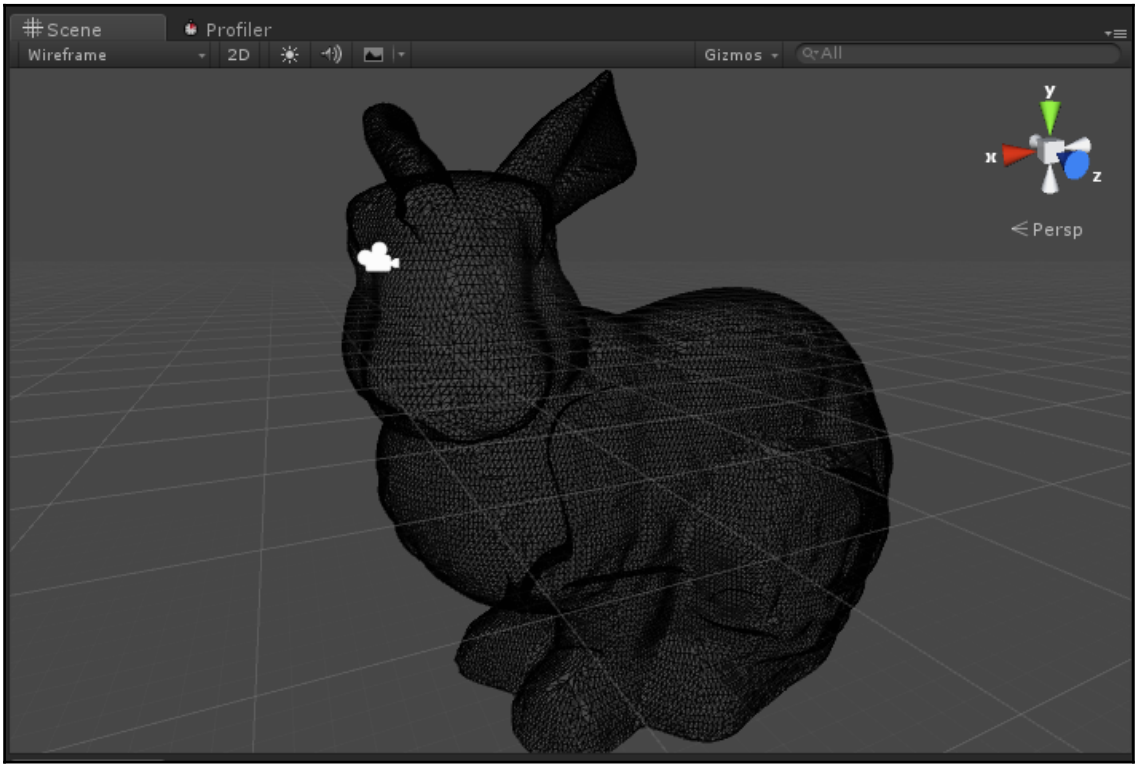
From the `Models` folder, drag **StanfordBunny** over the **Hierarchy** window and release to add an instance of that model to our scene. Locate the model in the **Scene** view and position it so that you can see it up close.

To quickly move the **Scene** window's camera to an object, left-click on it in the **Hierarchy** window, move the cursor over the **Scene** window, and press the *F* key.

Click on the drop-down menu in the upper-left corner of the **Scene** window that reads **Shaded**. This setting dictates how your models are displayed in the **Scene** view. Select the **Wireframe** option, as shown in the following screenshot:

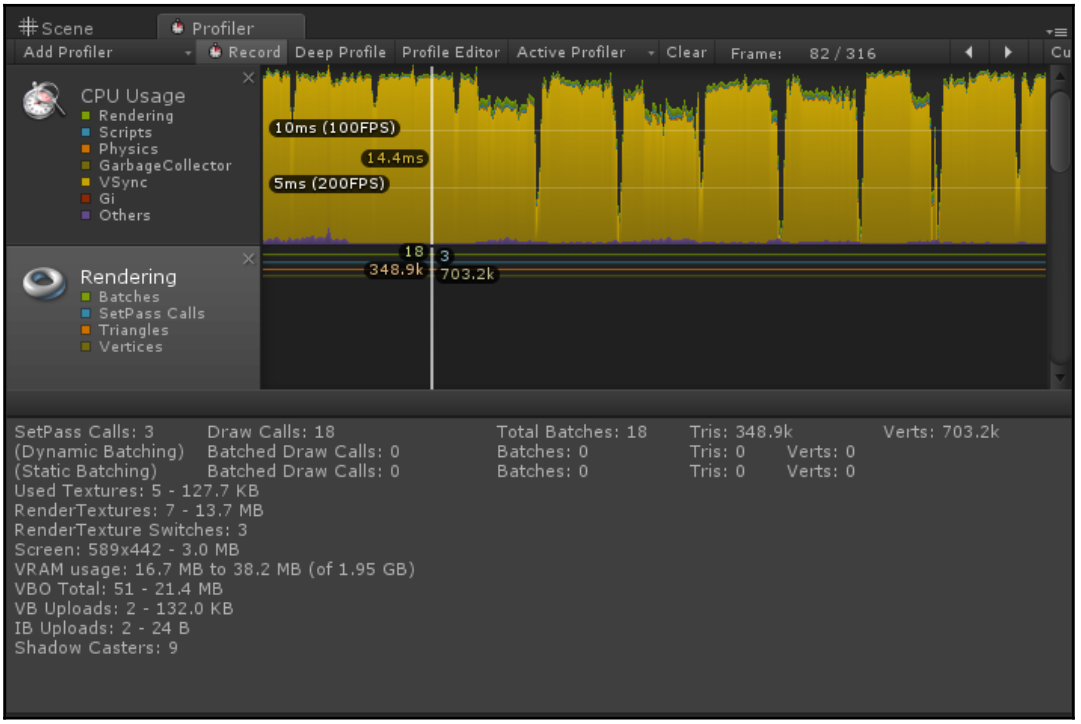


You can now clearly see the vertices and faces that make up the mesh:



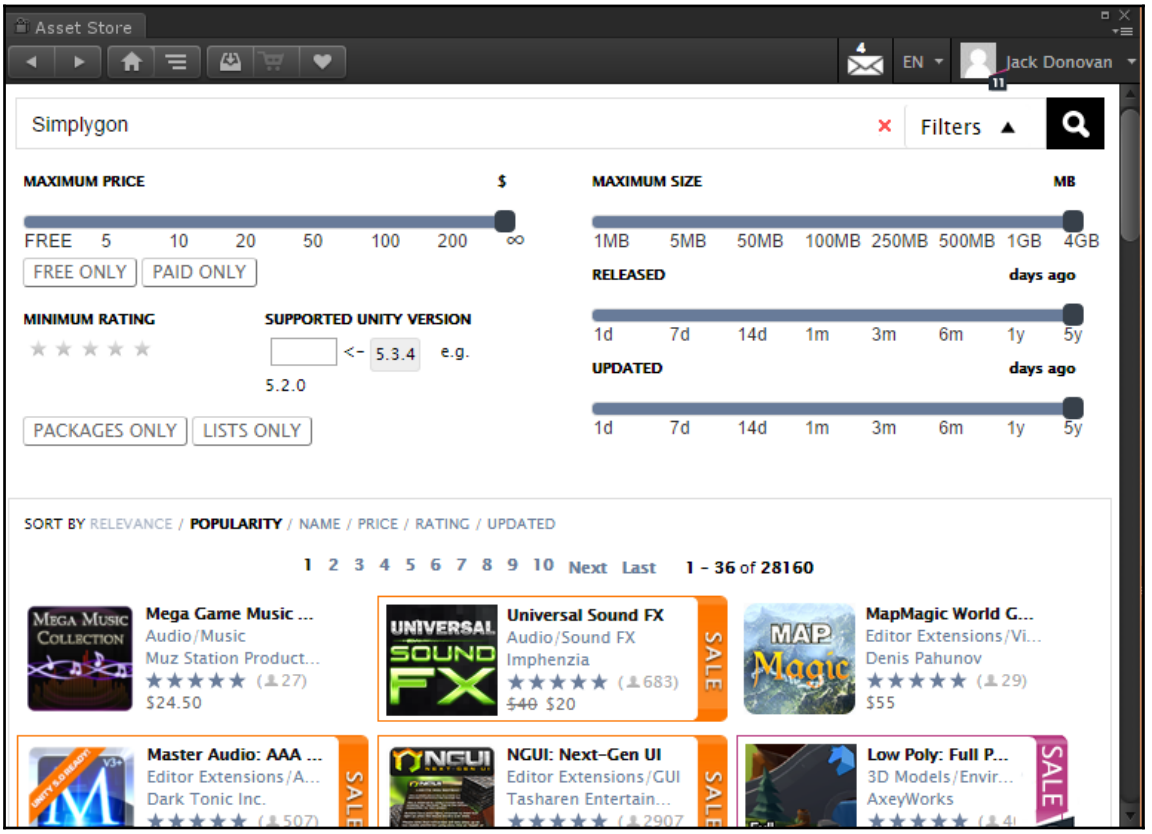
As you can see in the preceding screenshot, there's an immense amount of detail in the model—actually too much for Unity to even fit into a single mesh internally. Objects like these can hinder performance very quickly, and based on the object, the player might not even be in a position to notice that level of detail at any time.

Let's see how a model like this impacts the **Rendering** section of the Unity profiler. With the model fully in view of the game camera, let Unity run for a few seconds and then stop it. Click anywhere on the **Rendering** row to view per-frame data. You should see something similar to the following:



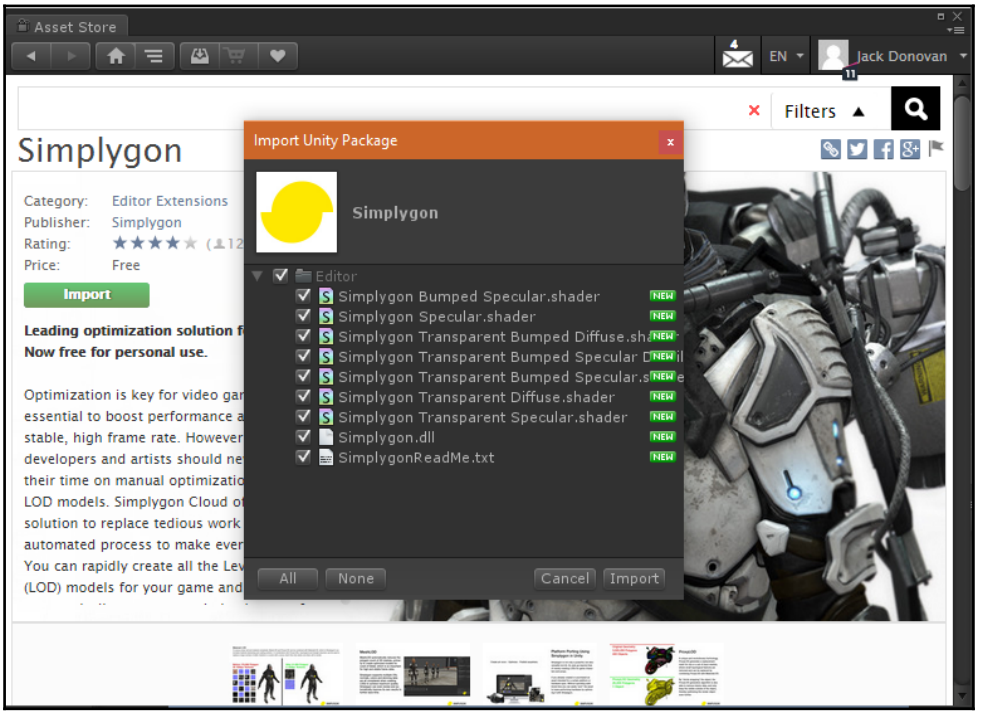
Hundreds of thousands of vertices is simply too many for a model like this (or any single model for that matter). Most game assets are made with as few vertices and faces as possible, leaning on the textures to add an appearance of realism. In a professional game, a bunny like this might have around 2,500 vertices, maybe even fewer. Let's see what we can do about making this bunny more render-friendly by installing the Simplygon plugin.

Open the **Window** menu on the top toolbar of Unity and select **Asset Store**. The Unity Asset Store is a great place to look for anything you might need for your game that you can't create yourself. You can find models, textures, scripts, and even developer tools like Simplygon. Enter `Simplygon` in the search bar at the top, as we've done in the following screenshot, and press *Enter*:

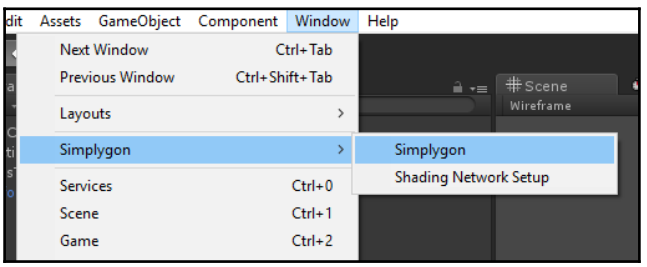


You can quickly open the asset store at any time by pressing *Ctrl* + *9* on Windows or *Command* + *9* on OS X.

Select the package named **Simplygon** with a yellow icon, download the package, and import it just as you imported the Oculus Utilities package in `Chapter 2`, *Stepping into Virtual Reality*. You should see the following files as part of the package:

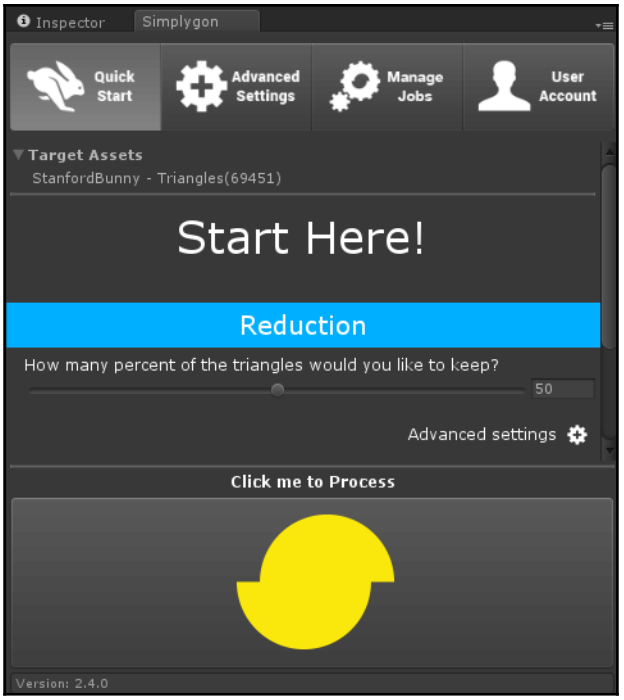


You'll notice that all of the files are being imported into the `Editor` folder. Like some of the files in the Oculus Utilities package, the files that are in the `Editor` folder interact directly with the Unity Editor. Sure enough, as soon as the import is completed, you'll find a new menu option in your **Window** menu to open the Simplygon interface:



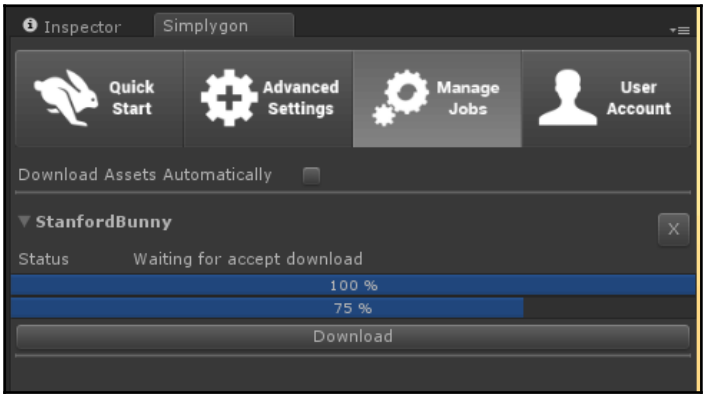
Click on the **Window** menu to open the **Simplygon** window, which you can position in your editor wherever you'd like. You'll need to make an account to upload meshes to be processed on the cloud, so follow the **Create Account** button to make a username and password and then use them to sign in.

Once you've signed in, Simplygon will open the **Quick Start** tab with the **Reduction** mode enabled. Select your **StanfordBunny** object in **Hierarchy** by left-clicking on it and then click on the button in the **Simplygon** window that says **Click me to Process**, as shown in the following screenshot:



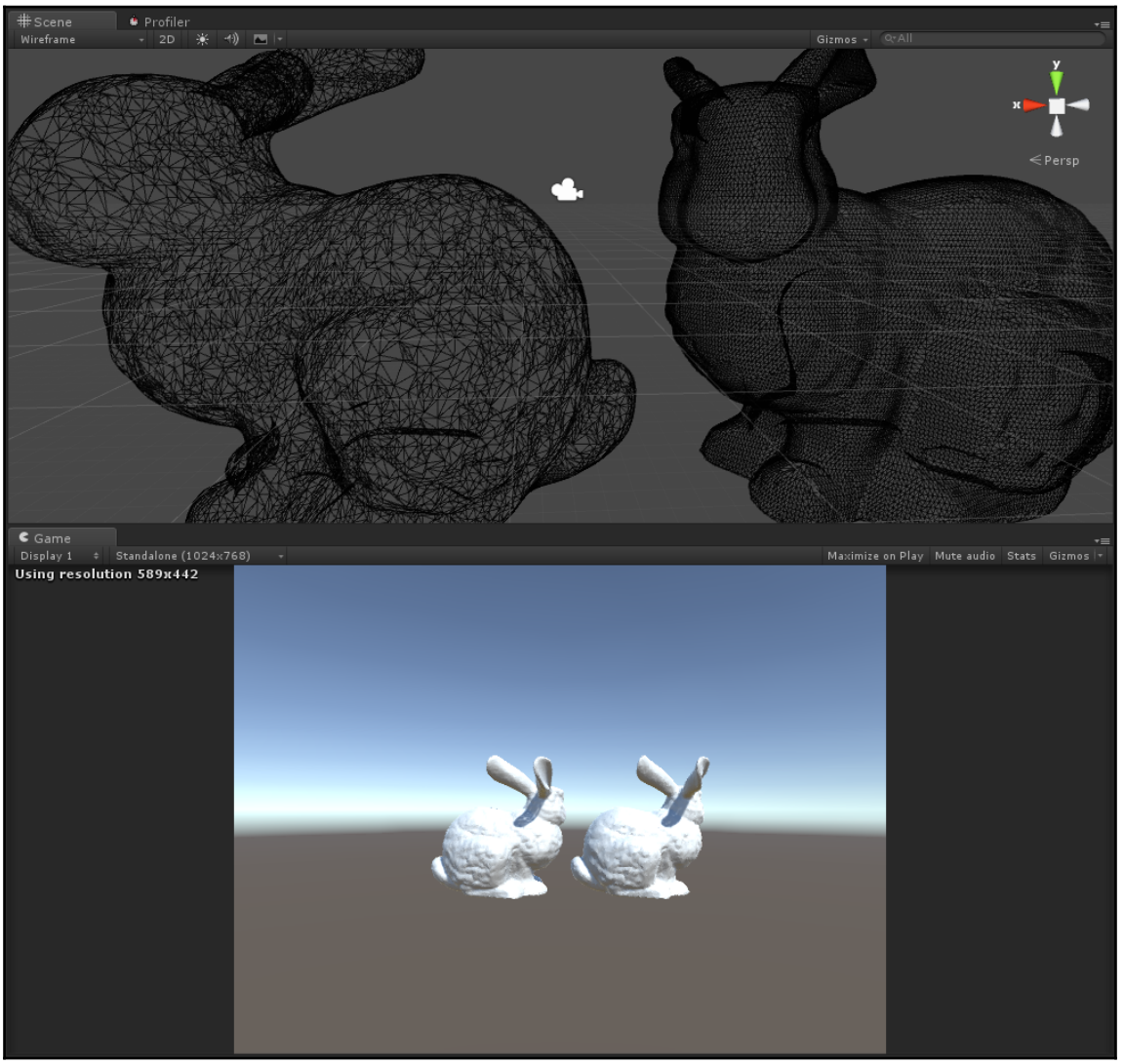
Notice that under the **TargetAssets** list at the top, we see our **StanfordBunny** and its triangle count of 69,451 (that's quite a lot). By default, Simplygon's reduction is going to cut that number in half, which will still leave quite a few triangles, but it's a good start.

You'll notice that the icon next to the **Manage Jobs** tab at the top begins to animate, which means that a mesh optimization job is currently being processed. Click on that tab to view Simplygon's progress compressing our bunny—don't worry, it doesn't hurt it! Once it completes, click on the button that says **Download** to save the new compressed mesh:



The new mesh will save to a folder that Simplygon auto creates in your **Project** window, called `LODs`. Expand this folder and find the subfolder that begins with `StanfordBunny_` and ends in a timestamp of the end time of the job. You'll see two created meshes inside it, `Stanfordbunny_LOD1` and `StanfordBunny_LODGroup`.

Drag an instance of `Stanfordbunny_LOD1` into your scene, position it next to the original high-poly bunny, and compare the two in **Wireframe** mode and in the game view:



As you can see in the preceding screenshot, the bunny on the left side of the **Scene** view is much less dense than the original bunny on the left side, but they are virtually indistinguishable in the game view, which means they'll be indistinguishable to anyone playing our game and viewing the compressed mesh from that distance.

But what if they walked closer? Surely, the player could reach a distance where they would want to see every detail of the rabbit. On the other hand, if they got further away, we could probably get away with a greater reduction than 50% of the mesh's triangles while still maintaining the object's general appearance. This is where **level of detail** (**LOD**) chains come in.

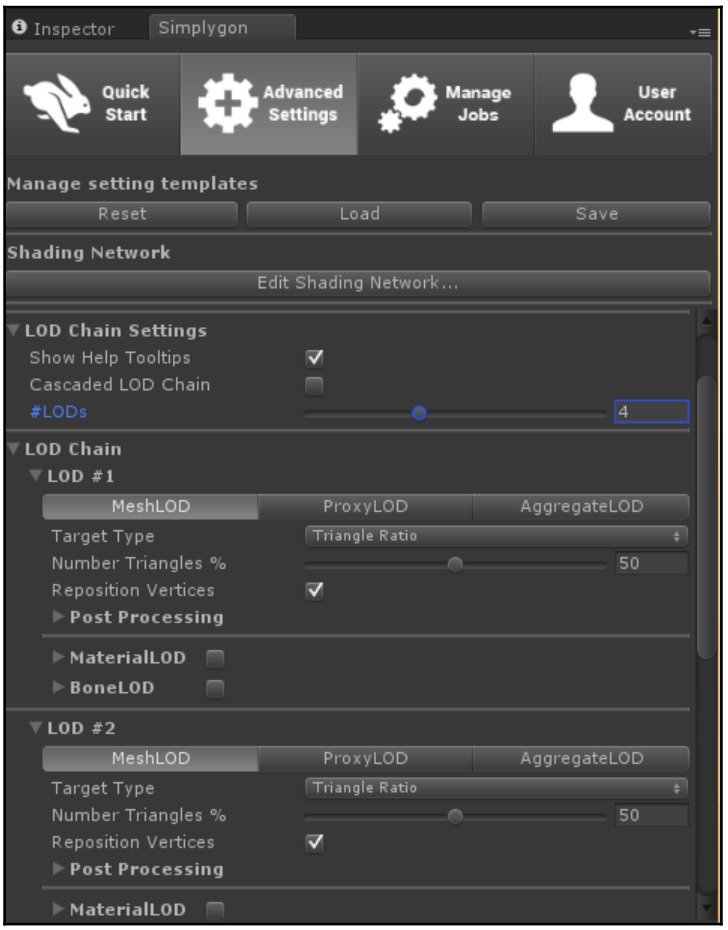
Displaying dynamic detail with LODs

An **LOD chain** is simply a collection of versions of a single mesh, all optimized to be viewed at different distances. If the player is far away from the mesh, it can be rendered using the mesh with the smallest level of detail because the player can't see details from far away anyway. As the player gets closer, the LOD chain replaces the mesh with others in the chain that get progressively more detailed.

To see this technique in action, we'll tweak some Simplygon settings that will enable us to write out several different LODs in a chain at one time.

Delete your old compressed **StanfordBunny** mesh, leaving just the original behind. Reselect the original in your **Hierarchy** and go back to the **Simplygon** menu, but select the **Advanced Settings** tab instead of the **Quick Start** tab this time.

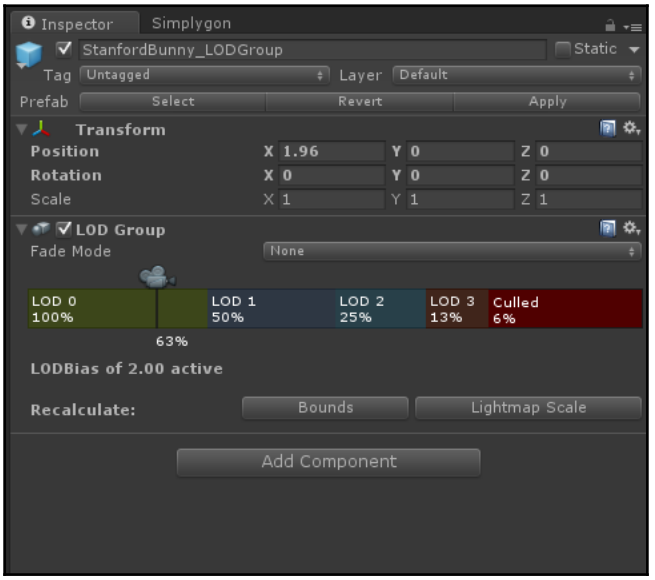
Under the **LOD Chain Settings** menu, you'll see a property called **#LODs**, which tells Simplygon how many different levels of detail to render. Set this value to **4** and press *Enter*, and you'll notice four total LOD definitions under **LOD Chain** in a list:



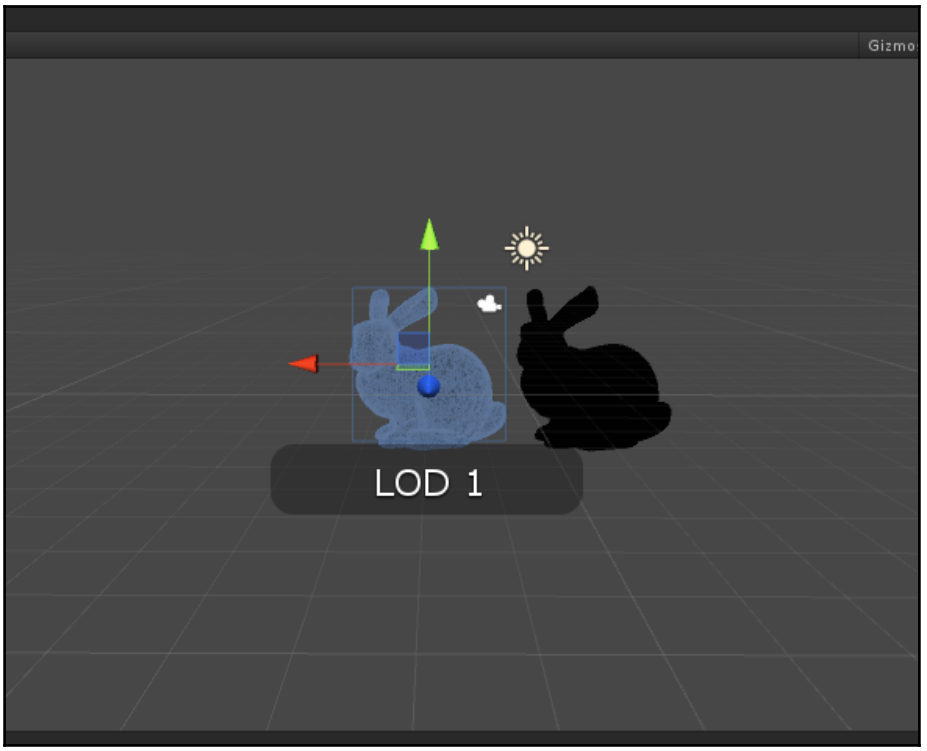
Set the **Number Triangles %** field of **LOD # 1** to `100`, set **LOD # 2** to `75`, **LOD # 3** to `50`, and **LOD # 4** to `25`. Press the **Click me to Process** button again and sit back; as we're generating four different LODs, this might take slightly longer than the single bunny.

Once the process has completed, find your new set of models again in the `LODs` folder and this time drag an instance of `StanfordBunny_LODGroup` into the scene.

If you select the new instance of `StanfordBunny_LODGroup` to view it in the **Inspector** window, you'll notice a new component type that we haven't worked with yet: the **LOD Group** component. You'll notice that it shows a gradient of all of the LODs we created, with a line depicting what LOD the camera is rendering based on the distance from the object:

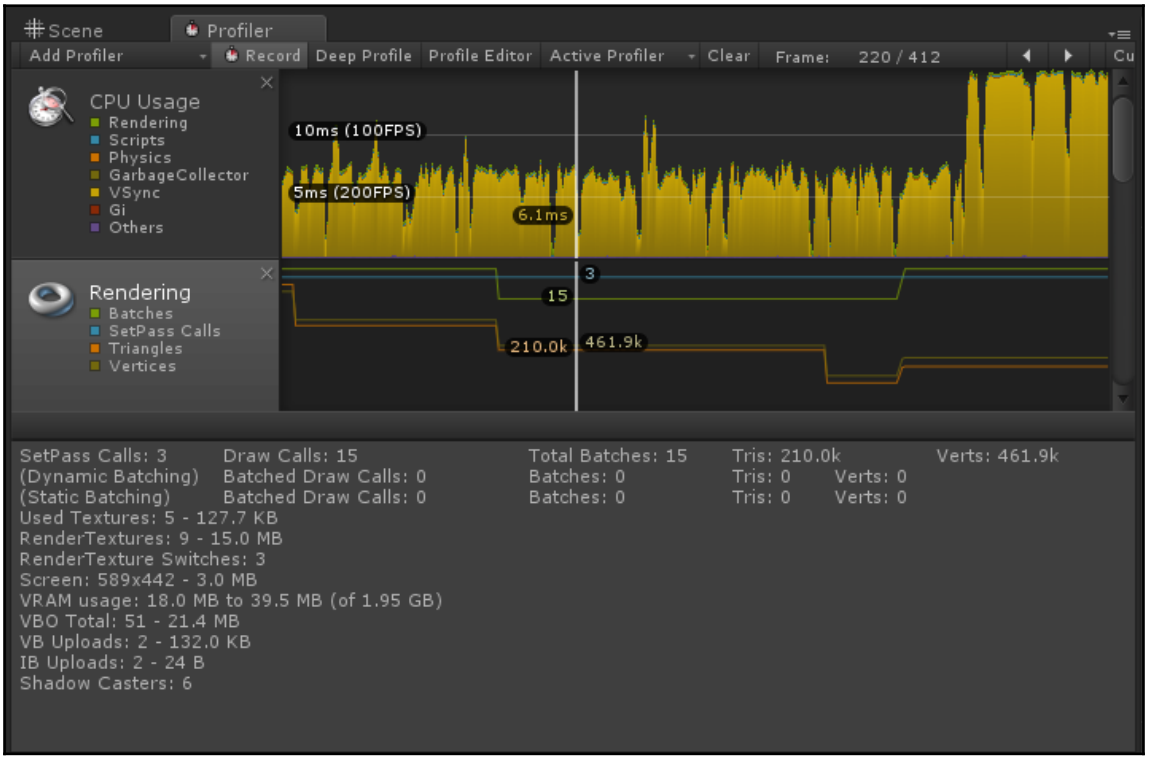


If you zoom the scene camera in and out from the bunny model, you'll notice that the camera line will move left and right, switching over to the lower LODs as it gets further away from the model. In the **Scene** view, you'll see a box around the bunny that displays what LOD you're currently viewing:



Move the camera toward and away from the bunny and see if you can detect where the meshes are being swapped out; you may be able to see some subtle shifts, but certainly not anything a player would see while navigating your game world.

Let's see how our LODs help our rendering load. Delete the original `StanfordBunny` object, leaving only `StanfordBunny_LODGroup`. With the bunny fully within the game camera's view, press Play. Slowly drag the camera away from the bunny in the **Scene** view while Unity is running so that you can see the LOD switches in the profiler. You'll end up with something like the following:



Setting up LOD chains can be time consuming, but the benefit you get in ensuring that you're not putting too much strain on the GPU at any given time is well worth the cost and will enable you to fill your virtual world with as much vibrant, immersive content as possible.

Keeping a handle on memory allocation

The last major topic we'll cover in this chapter is memory allocation and deallocation. The material we cover in this section gets relatively complex and you won't need to know it right away, so feel free to revisit this section later if you'd rather get a stronger handle on the basics first.

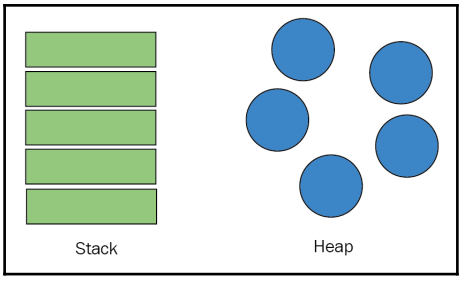
C# is a garbage-collected language, which means the actual process of allocation and deallocation is done by a garbage collector automatically according to an algorithm, but there are ways to manage your data to make sure you avoid common memory issues (even in a garbage-collected language) and keep everything optimized.

Stack and heap memory

To understand optimization of memory, first you have to understand the different types: **stack memory** and **heap memory**.

Imagine stack memory as a neatly organized stack of books you're going to read. You set the order, but you need to read through the books on top before you can access the books in the middle.

Heap memory, by contrast, can be imagined as a scattered pile of books. You can access any of them at any time, but you need to remember where each one is. When it comes to memory, you need to remember which books you've already read and get rid of them; if you were using a stack, you could throw the book on top away as soon as you were done with it and move onto the next one. Picture it like the following diagram:



Whether a variable is allocated on the stack or the heap depends largely on the type. Every basic type in C# falls into one of two categories: value types or reference types.

Value types and reference types

Value types are all simple objects that are allocated directly on the stack. These aren't garbage-collected, which means they won't stick around waiting to be destroyed; they're gone the second our code execution goes beyond the scope the variable was declared in.

The following are some examples of types treated as value types by C#:

- `bool`
- `byte`
- `char`
- `int`
- `float`
- `struct`

`struct` is perhaps the most notable of the value types since it is commonly used in tandem with classes to describe complex game-specific data structures in your code.

While the `struct` objects can be allocated on the stack, the `class` objects are allocated on the heap because they are reference type. Reference types are generally more expensive to allocate and deallocate.

Here are some examples of types treated as value types by C#:

- `class`
- `delegate`
- `string`
- `object`

Because reference types are allocated on the heap, they'll occupy that space until the garbage collector comes around to pick them up. However, the garbage collector only disposes of heap memory when no references to it remain, so if even one reference is left intact, that memory will remain monopolized and unusable.

Boxing and unboxing

It's possible to store value types in reference types, and reference types in compatible value types; these processes are called **boxing** and **unboxing**, respectively.

Boxing a value type essentially allocates space on the heap for a container that holds the variable's value. The boxed value will then be accessible from a memory address, and it won't be flagged for deletion until all references to it are removed.

The following is an example of boxing an integer:

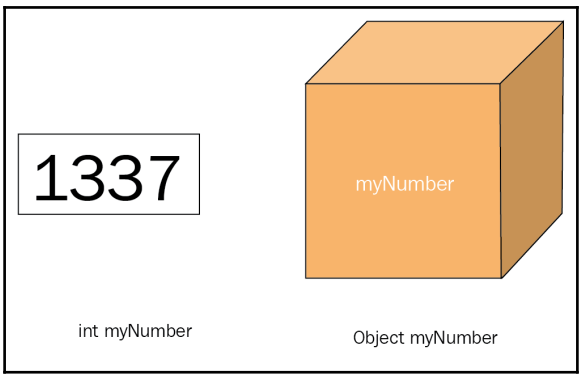
```
private void BoxInteger()
{
  int numberToBox = 1337;
  object boxedNumber = numberToBox;
}
```

The process for unboxing a reference value is very similar:

```
private void UnboxInteger()
{
  object boxedNumber = 1337;
  int unboxedInteger = (int)boxedNumber;
}
```

Note that in order to unbox our integer, we had to explicitly specify what value type we were expecting from the object. Boxing objects doesn't require specification because you're wrapping all of something in something else; to unbox, you need to specify what you're taking out.

Boxing and unboxing is best thought of like its literal real-world counterpart; you can either organize your belongings in the open or store them in a box that you label so that you can take things out later, as shown in the following figure:



Now that you understand the basic principles of boxing and unboxing, remember one golden rule: avoid both as much as possible. Boxing and unboxing is a very expensive operation, and only using it when absolutely necessary is the key to keeping your game running smoothly.

Unity specifics

We've covered memory pitfalls and limitations that span across the C# language, what about the limitations of Unity itself? Unity's C# framework is based on a dated version of Mono that can create a couple of pitfalls that seem counterintuitive; we'll go over those in this section.

foreach loops versus for loops

When it comes to iterative loops in C#, generally the options are foreach loops and for loops. In Unity's version of Mono, foreach loops are deceptively inefficient.

The major difference between foreach and for is how they iterate through a given list. The standard syntax for for loops generally initialize an integer and increase it by a certain value every time a loop completes, enabling the iterator to be used as an access index for the list or array.

A foreach loop creates a new instance of an enumerator object that is solely responsible for looping through a collection. foreach loops are generally better to use when the data in the collection doesn't need to be modified, but unfortunately a bug in Unity's version of Mono causes extra wasteful memory to be allocated every time foreach is used. Therefore, we should always use for loops in place of foreach loops in our Unity projects.

There's one exception to this rule: foreach loops that iterate through an array (not a List or other collection). This is because foreach loops for arrays are automatically converted to for loops behind the scenes in C#.

Material references

Remember how we used Unity's `GetComponent` function in `Chapter 2`, *Stepping into Virtual Reality*, to get a reference to the material of an object? As it turns out, every time the material property is accessed on a `Renderer` component, it's duplicated in memory—even if you don't change anything about it.

For instance, the following code would inadvertently create a new clone of a material just by accessing it:

```
private void GetMaterialReference()
{
  Material myMaterial = GetComponent<Renderer>().material;
}
```

To access the properties of a material without cloning it, it's important to use the `sharedMaterial` property instead:

```
private void GetMaterialReference()
{
  Material myMaterial = GetComponent<Renderer>().sharedMaterial;
}
```

This may seem like a trivial modification to make, but depending on the size of your in-game materials and how often you access them, you can dig yourself a memory trap relatively quickly.

Comparing tags efficiently

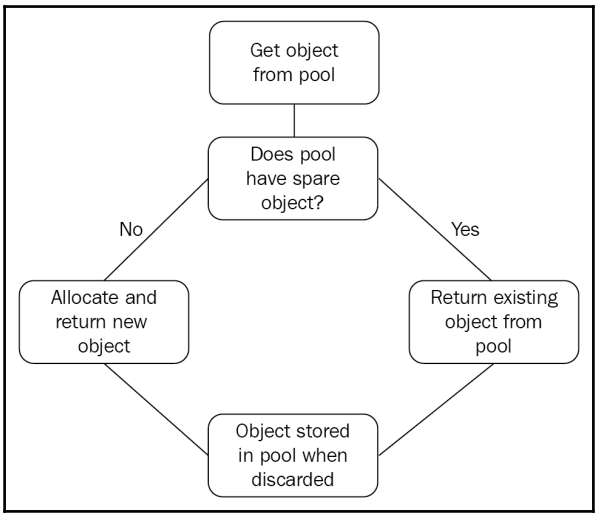
Another property that has a notable impact on allocations is the `tag` property on any object in your Unity scene. Instead of accessing the object's tag through this property directly, use Unity's `CompareTag` function, which takes a `string` tag and returns a `bool` value that reflects whether the tag matches the object's or not.

Object pooling

There's one thing better than optimal deallocation: avoiding deallocation completely. This is where the concept of object pooling comes in. The idea behind object pooling is that we have a finite collection of reference types in memory, and instead of deallocating an object when we're done with it, we add it to the collection to be reused later.

With this method, new allocations only need to be made when there aren't any objects in the collection not being used. Even when new memory needs to be allocated, it goes into the pool to be reused when we're done with it, effectively increasing the capacity of the pool permanently and reducing the odds that we'll need to allocate again in the future.

The following diagram displays how this system works at a high level:



Let's implement a very basic object pool that we can use in all of our future projects to see this principle in action.

Creating an object pool for strings

Since typically only reference types are allocated on the heap, our own custom class would be a good candidate for an object pool, especially when used for something like recycling enemies. In this section, we'll define a very basic `Enemy` class and create an object pool for it.

Create a new script in your `Scripts` folder and name it `EnemyPool`. Unlike our previous scripts that we built on top of Unity's MonoBehaviour template, we're going to write our own independent class, so erase everything in the script and replace it with a class declaration for our `Enemy` type:

```
public class Enemy
{
}
```

Our enemy doesn't need to be very complicated, so let's just give it an int for health, a bool to reflect whether it's been defeated or not, and a dummy `Update` function:

```
public class Enemy
{
  public int health;
  public bool isDefeated;
  public void UpdateEnemy()
  {
    return;
  }
}
```

Now that we've got our `Enemy` class, we can create a class for the object pool that will hold all of our `Enemy` objects. Add the following class declaration underneath your `Enemy` class declaration:

```
public class EnemyPool
{
}
```

The first thing to declare in our `EnemyPool` class is the pool collection itself. Add the following line at the top of your script to include the `Stack` class:

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

Now declare `Stack` of the `Enemy` type in `EnemyPool`:

```
public class EnemyPool
{
  Stack<Enemy> pool;
}
```

Now create two functions, one to get return a spare `Enemy` object and another to discard a used `Enemy` object once it has been defeated:

```
public class EnemyPool
{
  Stack<Enemy> pool;
  public Enemy GetEnemy()
  {
  }
  public void DiscardEnemy()
  {
  }
}
```

The `GetEnemy` function will need to check the pool to see if it has a spare enemy allocation to return and it will allocate a new enemy if it doesn't. Add this logic to the `GetEnemy` function:

```
public Enemy GetEnemy()
{
  if(pool.Count == 0)
  {
    return new Enemy();
  }
  else
  {
    return pool.Pop();
  }
}
```

Finally, define your `DiscardEnemy` function to return a used enemy to the pool:

```
public void DiscardEnemy(Enemy enemyToDiscard)
{
  pool.Push(enemyToDiscard);
}
```

And there you have it: a very basic implementation of an object pool. We'll expand on this and make it much more complex for our game systems later in the book, but the implementation we just created serves as a good simple example of a way to avoid unnecessary memory operations.

Pooling Unity GameObjects

The object pool system you just created pools your custom `Enemy` class, but that's not all you can pool; in fact, the Unity `GameObject` type, which is responsible for representing every discrete object in your scene, can be pooled. If you ever find yourself instantiating and uninstantiating several objects in your Unity scene—an army/horde defense game, perhaps consider using a pool to save on memory costs.

Summary

We've been through a lot of different optimizations in this chapter, and while they may not be as exciting as dazzling effects or complex mechanics to some, they pave the road for as many massive systems as the hardware can handle at its very best.

At the beginning of this chapter, we explored the Unity profiler and used it to practice debugging CPU-intensive script functions. We implemented a solution to an expensive function by implementing it as a coroutine.

In the second section, we looked at 3D geometry and the costs of rendering it. Objects with several faces in a scene can quickly add up and tax your GPU, but by sacrificing detail strategically using mesh optimization and LODs, we can avoid overloading the renderer.

Finally, we briefly covered several different techniques for memory optimizations, all leading up to a very basic object pool system that can help any game perform more efficiently by minimizing and optimizing allocations and deallocations.

Everything covered in this chapter is still just the tip of the iceberg in terms of optimization; entire volumes could be written on Unity optimization alone. For now, though, these techniques will serve as a good foundation for your projects moving forward.

4
Interacting with Virtual Worlds

In this chapter, we'll dive into the game project that we'll add features to throughout the rest of this book until it becomes the first full game you create in virtual reality. To begin, we'll cement a solid input system by building on the basics that you learned in `Chapter 2`, *Stepping into Virtual Reality*.

This chapter will cover the following topics:

- Designing basic player input
- Unity's input system
- Supporting joysticks and gamepads

Designing basic player input

Designing input that is intuitive, efficient, and simple is key to creating a game that players will come back to. Perhaps it's especially important in VR because your player won't be able to actually see any of the joysticks, buttons, and keys they press.

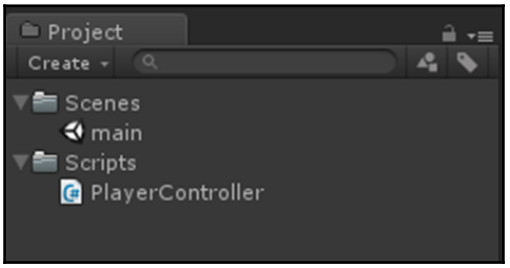
In this section, we'll put together the basics of input in a simple arena combat game that we'll continue to expand on throughout the rest of this book.

To get started, create a new Unity project called `ArenaCombat`. Once the editor opens, create a folder in your **Project** window called `Scenes` and save your blank scene in it with the name `main.unity`.

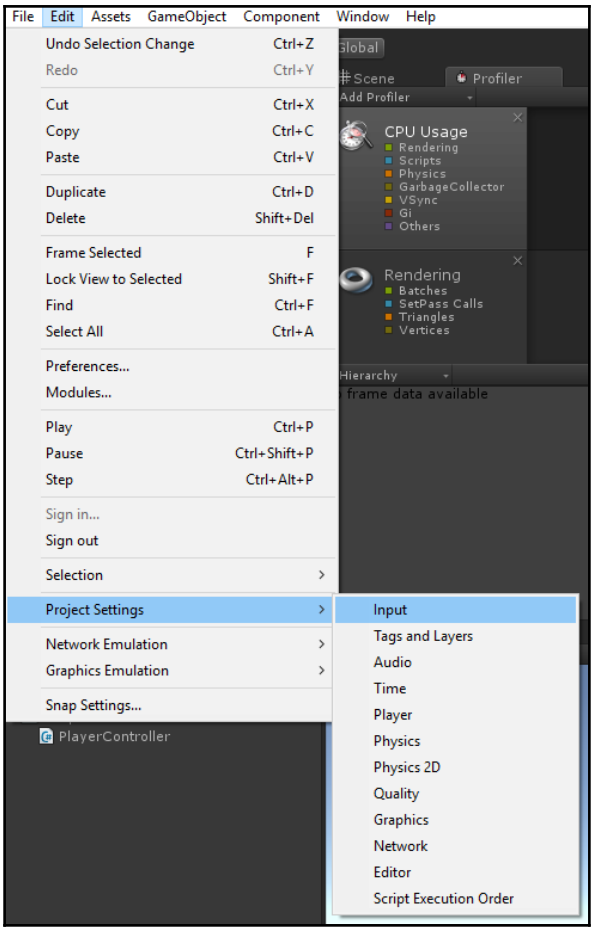
Using Unity input axes

If you recall, we structured the main input in our `PlayerController` script from `Chapter 2`, *Stepping into Virtual Reality*, by checking whether various keys had been pressed. This is a fine method for quickly testing functions, but in a full game, it's beneficial to have a more structured input system. Fortunately, Unity has a great input system that we can customize to capture input in our game from keyboards, mice, and joysticks. To start off this section, we'll modify our old script to use this system. Create a folder called `Scripts` to put your game's code in. Click and drag the `PlayerController.cs` file from your `HelloVR` project and release it over your `Scripts` folder to automatically copy it there and add it to your project.

At this point, your **Project** window should look like the following:

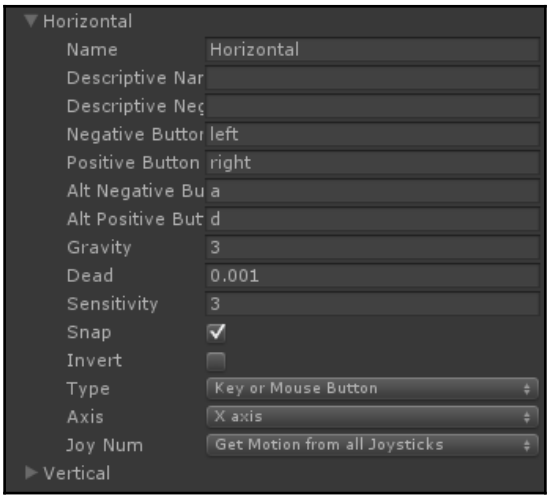


Mouse over the **Project Settings** option in the **Edit** menu on the top toolbar and select **Input**, as shown in the following screenshot:



In the **Inspector** window, you'll now see **InputManager** displayed, which is a collection of axes that each correspond to a different input action. There are several default axes already defined; we'll start by looking at the first two, **Horizontal** and **Vertical**, as shown in the following screenshot:

Expand the **Horizontal** tab to view its preset configuration, as shown in the following screenshot:



The **Positive Button** and **Negative Button** (and their **Alt** versions) represent what key or button Unity will be watching to trigger this input action. In this case, the left and right arrow keys represent each end of the **Horizontal** axis, and alternatively, the user can use the *A* and *D* keys (like they have already been using in the `PlayerController` script).

Let's add a reference to these axes in our `PlayerController` script. In your `Update` function, erase the four `if` statements that reference the *W*, *A*, *S*, and *D* keys. Your `Update` function should now only contain the `GetKeyDown` check for the space bar:

```
void Update()
{
  if(Input.GetKeyDown(KeyCode.Space))
  {
    Teleport();
  }
}
```

Next, store the value of our **Vertical** axis to represent forward and backward movement, and the value of our **Horizontal** axis to represent left and right movement:

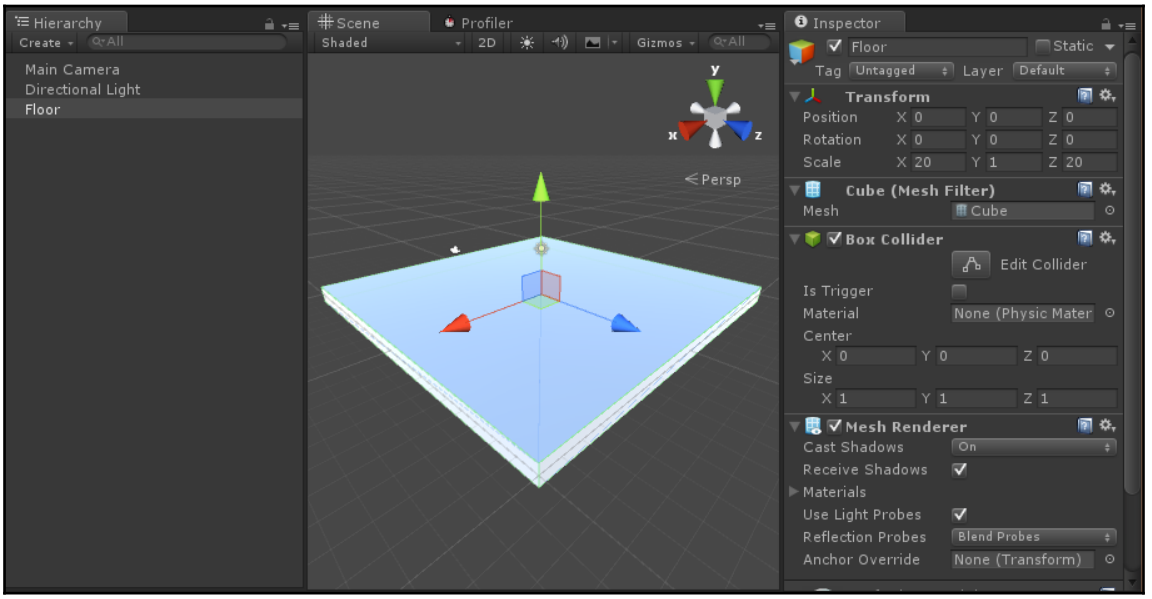
```
void Update()
{
  float forwardMovement = Input.GetAxis("Vertical");
  float strafeMovement = Input.GetAxis("Horizontal");

  if(Input.GetKeyDown(KeyCode.Space))
  {
    Teleport();
  }
}
```

In our previous input code, we had to have four separate checks for the four unique directions that our player could travel. By using Unity's input axes, we only need to have two checks because each axis has a positive and a negative pole. For instance, with our **Vertical** axis, a positive value would represent forward movement, and a negative value would represent backward movement.

Now re-add a call to our player's `Translate` function, using the following new values in place of the old ones:

```
void Update()
{
  float forwardMovement = Input.GetAxis("Vertical");
  float strafeMovement = Input.GetAxis("Horizontal");

  gameObject.transform.Translate(lookTransform.forward *
  forwardMovement * Time.deltaTime);
  gameObject.transform.Translate(lookTransform.right *
  strafeMovement * Time.deltaTime);
  ...
}
```

We're ready to test our new code, but we need to set up a minimal scene first. Create a platform for the player to walk on by creating a new cube, and name it `Floor`. Set the scale of its **Transform** component to `20`, `1`, `20`, and position it at `0`, `0`, `0`, as shown in the following screenshot:



Import the Oculus Utilities package into your `CombatArena` project. Click and drag a **CameraRig** prefab from the `Prefabs` folder inside **OVR** and drop it over your hierarchy to add it to the scene. Position it at `0`, `2.7`, `0` and click and drag your `PlayerController` script over it to add it as a component. At this point, feel free to delete the **Main Camera** object from your scene. Finally, expand the **CameraRig** object in your hierarchy and drag the nested object named **CenterEyeAnchor** into the **LookTransform** field of your **PlayerController** component.

Now, click on Play and use the *W*, *A*, *S*, and *D* keys to move around a bit. As you'll see, the script functions in the same way as it did earlier, but instead of hard-coding the key to check in the `Update` function, you're checking an entry in Unity's input system instead. That means you can change key and button mappings without touching your code, and enable the user to use up to two sets of buttons to trigger an action without the need for redundant code.

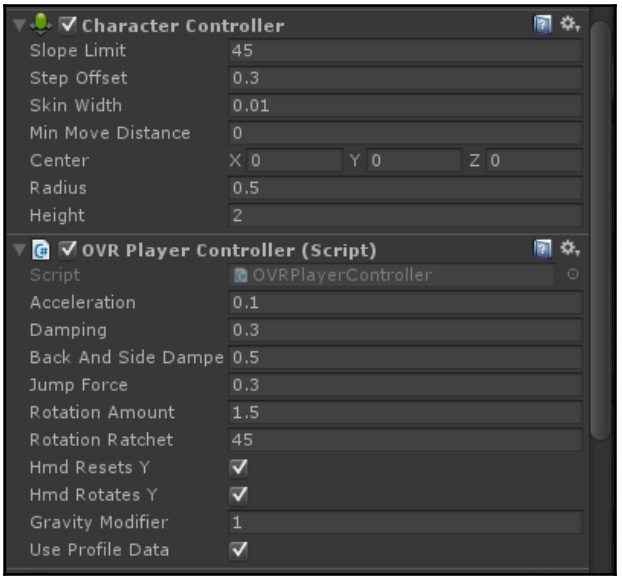
Using the OVRPlayerController script

There's still a lot missing from our `PlayerController` script; we don't have a way to rotate the user's body or have it reflect height, gravity, or physics. We could code all of this ourselves, but there's a basic script included in the Oculus Utilities package that will handle most of the boilerplate work for us. We'll still be able to add functionality in separate scripts by attaching them as components, or modify the existing functionality by editing the script provided in the package.

In this section, we'll replace our existing player object with the **OVRPlayerController** prefab and examine the key functions in the included `OVRPlayerController.cs` script.

Click and drag an instance of the **OVRPlayerController** prefab in the `OVR/Prefabs` folder into your hierarchy and position it at `0`, `1.5`, `0`. Delete your **OVRCameraRig** from the scene, as we'll be using **OVRPlayerController** from now on.

Click on the **OVRPlayerController** in the hierarchy to highlight its components in the **Inspector** window. There are two to note immediately, the **Character Controller** component and the **OVR Player Controller (Script)** component, as shown in the following screenshot:



The **Character Controller** component manages the physical properties of the character, including the collision volume and the slope limit, which is the steepest angle the character can climb.

The **OVR Player Controller (Script)** component is responsible for every way the player's input is converted into character movement. Here you can tweak several variables to get the right *feel* for movement in your game, including acceleration for how fast your character picks up speed and damping for how fast it falls off.

Notice that there isn't a variable for rotation speed, just **Rotation Amount** and **Rotation Ratchet**. To reduce nausea, it's a good idea to rotate your player in discrete steps rather than smooth animation.

Press Play and try navigating with the new controller with the *W*, *A*, *S*, and *D* keys. You'll notice that it's similar to before, but with a much more natural movement and some extra features, such as the ability to rotate with the *Q* and *E* keys.

You'll also notice that moving your mouse left and right will rotate the player. As mentioned earlier, this can cause nausea, so we'll turn it off for now and only re-enable it if we have a good reason.

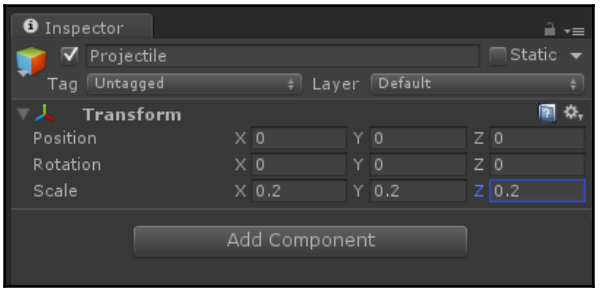
Double-click on the `OVRPlayerController` file in the **OVR Player Controller (Script)** component to open the script for editing. Scroll down to line 91, where you'll find the following line in a series of member declarations:

```
private bool SkipMouseRotation = false;
```

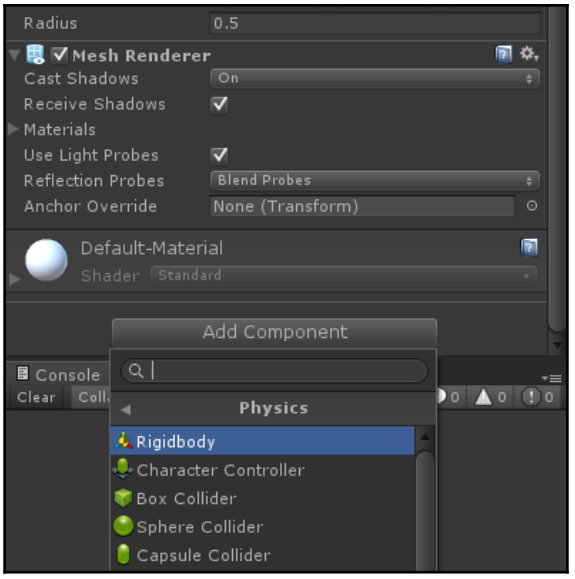
Replace the `false` value with `true`, save the script, and test again to make sure your mouse no longer rotates your controller.

Adding projectiles

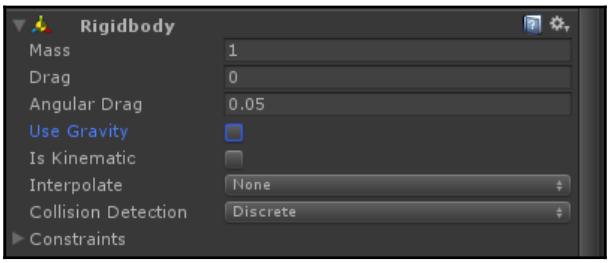
It's about time we added the ability to shoot something. We'll need a simple projectile object that we can turn into a prefab, so start by creating a new sphere from your hierarchy's **Create** menu. Select it in the hierarchy to display its properties in inspector and rename it `Projectile`. We don't want it to be too big, and Unity spheres are a meter in diameter by default, so edit the **Scale** property in the **Transform** component to be `0.2`, `0.2`, `0.2`, as shown in the following screenshot:



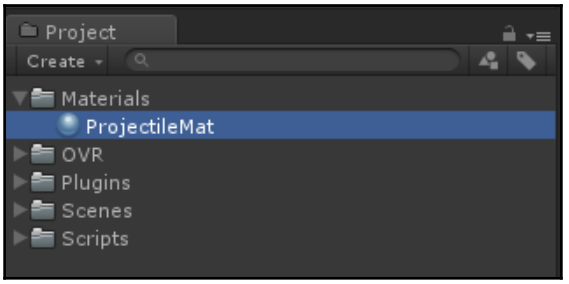
Next, we need to give our projectile a component to handle physics. Click on the **Add Component** button at the bottom of the projectile's inspector window and pick **Rigidbody** in the **Physics** category, as shown in the following screenshot:



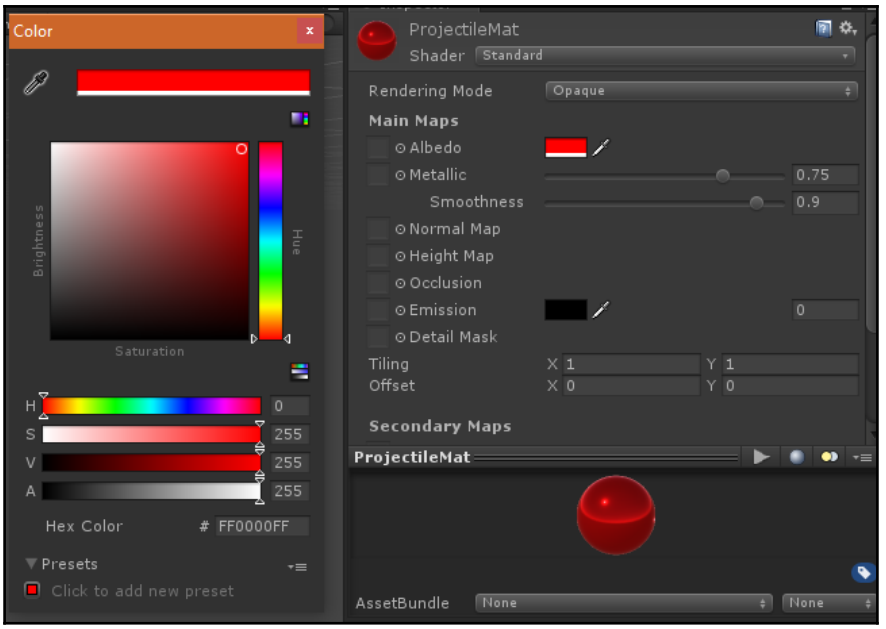
On your new **Rigidbody** component, uncheck the box that reads **Use Gravity**; for a simple projectile, we won't worry about gravity modifying its trajectory until we have a target to hit. Your **Rigidbody** component properties should now look like the following:



We should also give our projectile a unique material. Create a new folder from the **Create** menu in your **Project** window and name it `Materials`. Then, right-click on your `Materials` folder and add a new material by selecting **Material** in the **Create** menu; name it `ProjectileMat`, as shown in the following screenshot:



Click on the color box next to the **Albedo** property to open a color picker, where you can give our material a red hue. Also, increase the **Metallic** and **Smoothness** properties to make it stand out more. Your material should end up looking something like the following:



To add our new material to the projectile, drag it from the **Project** window to the **Hierarchy** window and release it over the **Projectile** object.

Our projectile is now ready to be saved as a prefab. Create a new folder named `Prefabs` and drag the **Projectile** object from the hierarchy to that folder in the **Project** window. Once you've done that, you can delete the original projectile from your scene; all of our projectiles in the game will be instantiated through code from now on.

Implementing a firing mechanic

Now that we've got our projectile prefab, we need to write a script to instantiate it at the player's location and add force in the direction that they're firing it. Create a new script in your `Scripts` folder and call it `FiringSystem`. Then double-click on the script to open it for editing.

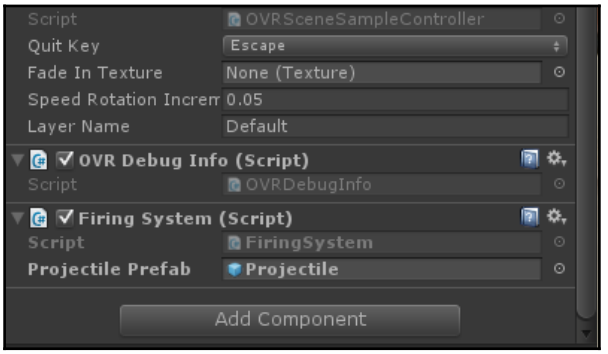
Add a new serialized field at the top of your class definition to serve as a reference to our projectile prefab:

```
Public class FiringSystem : MonoBehaviour
{
  [SerializeField]
  private GameObject projectilePrefab;
```

```
    // Use this for initialization
    void Start()
    {

    }
    ...
}
```

To link the prefab we just created to this reference, we'll first have to add an instance of the script to the scene. Click and drag the script from your **Project** window and release it over the **OVRPlayerController** prefab. Left-click on the prefab to display its components in inspector; you'll now see your **Firing System (Script)** component at the bottom of the list, along with an empty field for your serialized variable.

Click and drag the **Projectile** prefab into the slot to complete the link. Now, whenever you reference the `projectilePrefab` variable in your `FiringSystem` class, you'll be accessing this prefab. Using this method, the prefab can be swapped out for any other prefab without the need to change code; all you need to do to replace the referenced prefab is drag a different prefab to the serialized field:



While it's easy to swap serialized prefabs without changing code, it's important to keep in mind any unique elements of your prefab that might be accessed. For instance, if you swapped out your current **Projectile** prefab for a different object that didn't have a **Rigidbody** component, you'd run into an error as soon as your code tried to access it.

And now for the fun part—firing the projectile. Create a new function in your `FiringSystem` class called `FireProjectile`:

```
private void FireProjectile()
{

}
```

The first thing this function will do is instantiate a new projectile. Add the following lines to your script to create a new projectile according to the player's current orientation:

```
private void FireProjectile()
{
  GameObject newProjectile = Instantiate(projectilePrefab);
  newProjectile.transform.position = transform.position +
transform.forward;
  newProjectile.transform.rotation = transform.rotation;
}
```

In our `position` and `rotation` assignments, you may have noticed that we don't explicitly reference the object that the source `transform` belongs to. Every `MonoBehaviour` you create in Unity has an implicit reference to the `transform` that the script is attached to, so by using `transform.position`, we're using the position of whatever object our `FiringSystem` script is attached to (which is the **OVRPlayerController** right now).

Next, we need to apply force, and to add force, we need to get a reference to the `Rigidbody` component using Unity's `GetComponent` method. Add the following line to your `FireProjectile` method:

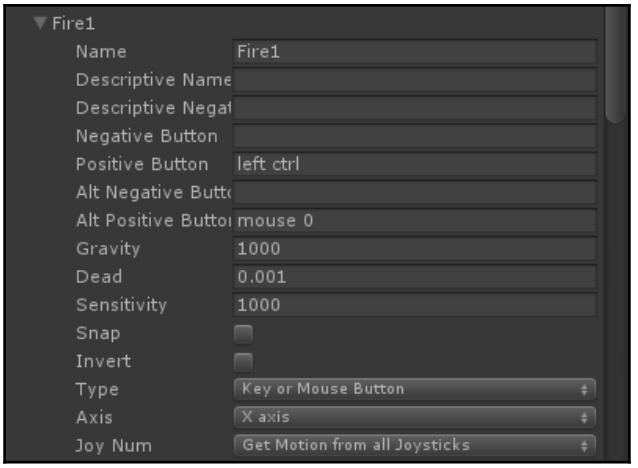
```
private void FireProjectile()
{
  GameObject newProjectile = Instantiate(projectilePrefab);
  newProjectile.transform.position = transform.position +
transform.forward;
  newProjectile.transform.rotation = transform.rotation;

  Rigidbody rigidbody = newProjectile.GetComponent<Rigidbody>();
}
```

Now that we have a reference to our `Rigidbody` component, we can call `AddForce` to shoot the projectile forward. Add the following line to the end of your `FireProjectile` method:

```
private void FireProjectile()
{
  ...
  newProjectile.GetComponent<Rigidbody>();
  rigidbody.AddForce(newProjectile.transform.forward,
    ForceMode.VelocityChange);
}
```

The `ForceMode` you choose specifies what kind of force is applied to your `Rigidbody` component. There are four modes: `Force`, which adds a continuous force relative to mass, `Impulse`, which adds an instant force relative to mass, `Acceleration`, which mirrors `Force` but ignores mass, and `VelocityChange`, which mirrors `Impulse` but ignores mass.

The last thing we need to do before we can test is create some input code for calling our `FireProjectile` method. Unity already has an input axis called `Fire1` that we'll use, which you may have noticed when you were looking at **InputManager** earlier:



Add the following line to your `Update` function in `FiringSystem` to call our firing function whenever the `Fire1` input is triggered:

```
void Update()
{
  if(Input.GetButtonDown("Fire1"))
  {
    FireProjectile();
```

```
    }
}
```

Press Play and click on the left mouse button or press the left *Ctrl* key to fire your projectile a few times. It's a good start, but you'll notice that it's kind of slow and you can't fire it higher or lower than straight ahead since it's using your body's transform, not your head's. Let's make a few modifications to make this a more satisfying mechanic.

First, add a new `float` variable to your class underneath your projectile prefab reference to represent firing speed, and initialize it with a value of `10`:

```
public class FiringSystem : MonoBehaviour
{
  [SerializeField]
  private GameObject projectilePrefab;
  private float projectileForce = 20f;
  ...
}
```

Now add the following variable as a coefficient to your `AddForce` call:

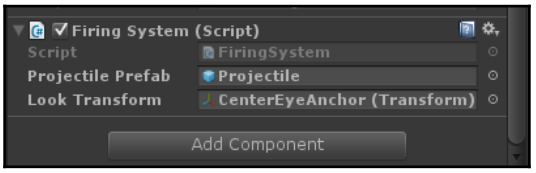
```
private void FireProjectile()
{
  ...
  newProjectile.GetComponent<Rigidbody>();
  rigidbody.AddForce(newProjectile.transform.forward *
  projectile Force, ForceMode.VelocityChange);
}
```

Next we'll have the projectile fire in the direction that the user is looking rather than the direction their controller *body* is facing. Create a new serialized `Transform` reference in your script:

```
public class FiringSystem : MonoBehaviour
{
  [SerializeField]
  private GameObject projectilePrefab;
  private float projectileForce = 20f;
  [SerializeField]
  private Transform lookTransform;
  ...
}
```

Return to the **Firing System (Script)** component in the inspector panel and you'll find your new serialized field right beneath the **Projectile Prefab** field.

Expand your **OVRPlayerController** in your **Hierarchy** window to reveal its **OVRCameraRig** child and then expand **OVRCameraRig** to reveal the **CenterEyeAnchor** child. Click and drag the **CenterEyeAnchor** object to the new field to add its **Transform** component as our reference:



Next, we'll tell our `FireProjectile` function to use this transform instead of the implicit transform it's been using from the **OVRPlayerController** object. Modify the references in the `FireProjectile` function as follows:

```
private void FireProjectile()
{
  GameObject newProjectile = Instantiate(projectilePrefab);
  newProjectile.transform.position = lookTransform.position +
  transform.forward;
  newProjectile.transform.rotation = lookTransform.rotation;
  ...
}
```

Finally, re-enable gravity on your projectile's **Rigidbody** component. Now that we have a target to aim for, we want to make it a little more difficult than simply firing straight ahead along the user's gaze.

Press Play again and try out your modifications. The projectiles should fly forward in whatever direction you're looking. The mechanic should feel a little more fun and natural, even if there's nothing to do with it yet (that's for the next section). There's one loose end we need to tie up before moving on, though, and that's the issue of cleaning up our projectiles when we're done with them. As it stands now, they're never destroyed after they're fired, so the user could theoretically create more projectiles until eventually there were enough to cause the game to crash.

Realistically, the player will probably never care about their projectiles after 10 seconds or so, so we'll do a timed cleanup. Enter the following line at the end of your `FireProjectile` function to schedule the projectile to be destroyed after `10` seconds:

```
private void FireProjectile()
{
  ...
  Destroy(newProjectile, 10);
}
```

That's it! You've coded a basic but complete projectile mechanic. Of course, it's pointless without something to fire at, so without further ado, we'll move onto making some targets.

Implementing gameplay around input

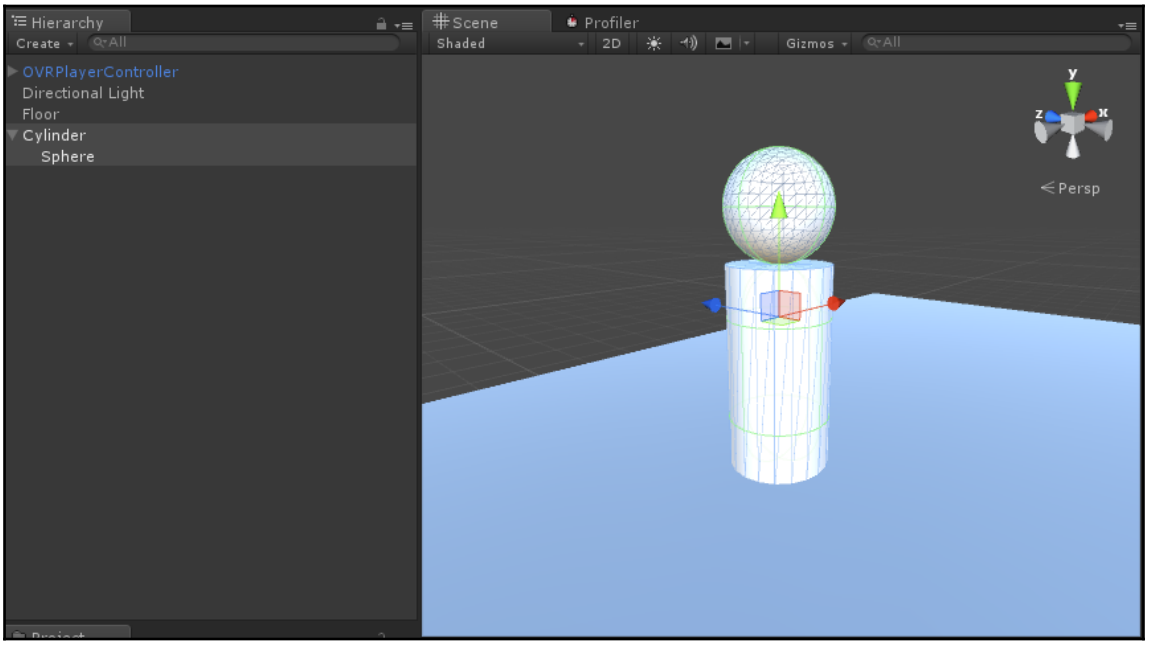
The controls to your game are the crux of your player's experience, so it's important to make each action feel like it impacts the game in a meaningful way. A control scheme that includes jumping wouldn't suit a level with no obstacles to jump over, and a control scheme that includes shooting won't do if there's nothing to shoot at.

In this section, we'll create some targets for the firing mechanic you created in the previous section, and the objectives of our game will start to take form (as will our level's environment).

Creating a target dummy

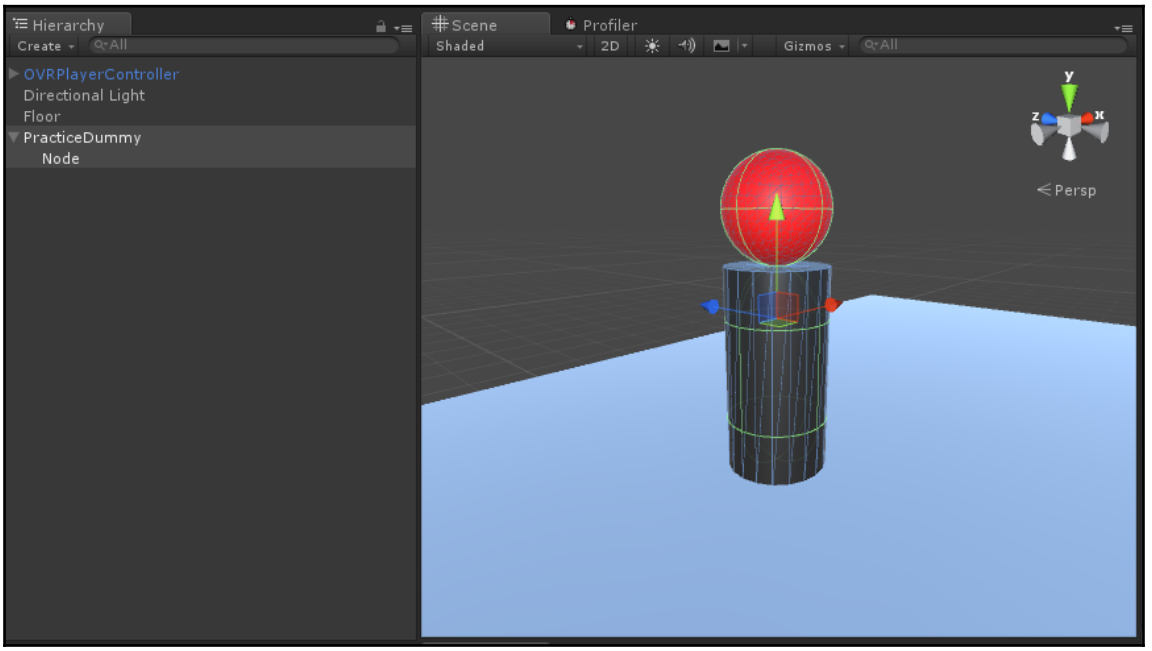
Eventually, the targets in our game will be other real players when we implement online multiplayer later in this book, but for now, we'll just implement a target dummy to test our mechanics.

Create a new **Cylinder** object from the **Create** menu in your **Hierarchy** window and position it on the floor at the origin of your scene. Center the scene view on it and then create a new **Sphere** object, positioning it directly on the **Cylinder** object; this will act as a node that will change color when our practice dummy is hit. Finally, drag the **Sphere** over the **Cylinder** to make it a child of the **Cylinder** so that you can move both as one object. When you're done, the new object should look like the following:



Rename the **Cylinder** object to `PracticeDummy` and rename the **Sphere** to `Node`. We should also give these objects some unique materials so they stand out; create two new materials, `PracticeDummyMat` and `NodeMat`. Give **PracticeDummyMat** a dark gray color and apply it to the cylinder, and then set the **NodeMat** material's color to red and apply it to the sphere.

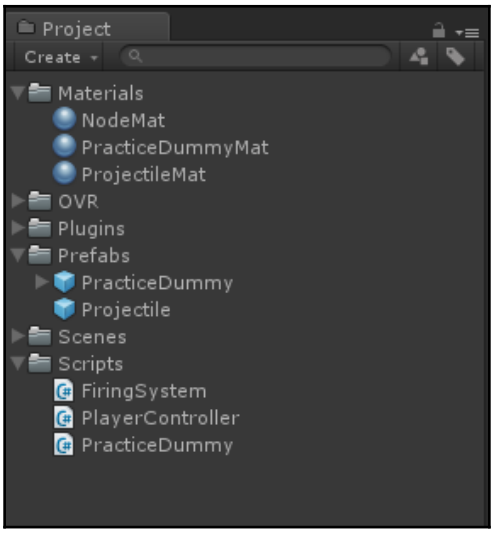
When this is done, you should have a more unique-looking object, something like the following:



To complete the base of what will become a new prefab, create a new script called `PracticeDummy` and drag it from your `Scripts` folder over the **PracticeDummy** object to add it as a component.

Now's a good time to save our **PracticeDummy** prefab before we get into coding. Click and drag the **PracticeDummy** object onto your `Prefabs` folder in the **Project** window and a prefab will automatically be created.

You just made a handful of new files, so let's take stock quickly. Your **Project** window should have the following items in it:



Now we'll add coded functionality to our new practice dummy. Double-click on your `PracticeDummy` script to open it for editing.

Declare a new function in your `PracticeDummy` class, called `OnCollisionEnter`, which takes a `Collision` object as a parameter:

```
private void OnCollisionEnter(Collision collisionInfo)
{

}
```

`OnCollisionEnter` is another one of Unity's built-in callback functions that you can use in any `MonoBehaviour` scripts simply by declaring it. In this case, it will be called when any object with a **Collider** component collides with another. The `Collision` parameter provides plenty of data about both objects involved, so we can invoke code based on the circumstances of the collision.

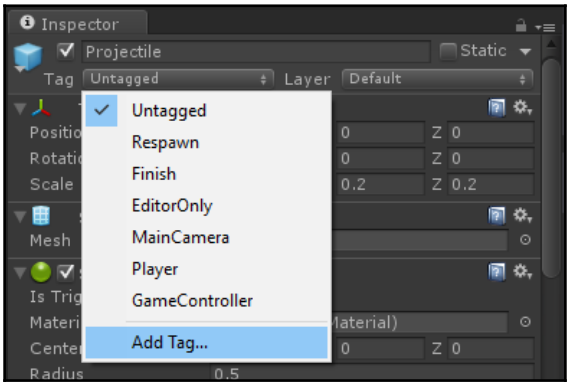
In this case, we want to check if the object colliding with the practice dummy is indeed a projectile fired by the user, and if so, we want to change the color of the node. We'll do this by associating a tag with our **Projectile** prefab; the `OnCollisionEnter` function can then check the tag of the colliding object and we can trigger a function if we get a positive result.

Add the following line to your `OnCollisionEnter` function to print a statement to Unity's console every time the dummy is hit by an object with the `Projectile` tag:

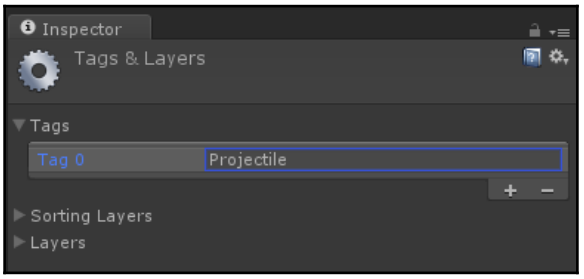
```
private void OnCollisionEnter(Collision collisionInfo)
{
  if(collisionInfo.collider.tag == "Projectile")
  {
    Debug.Log("Dummy hit by projectile!");
  }
}
```

This is almost ready to test, but first we need to create the `Projectile` tag and assign it to our prefab.

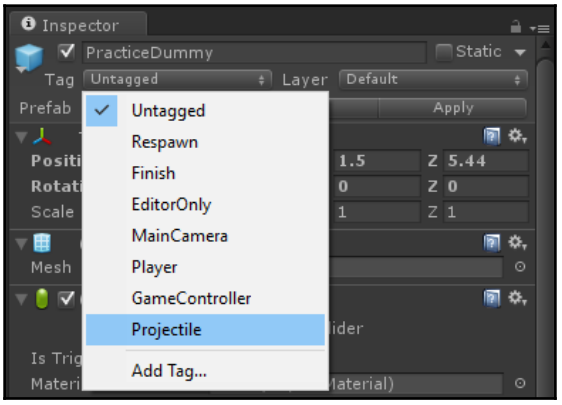
Left-click on the **Projectile** prefab in your **Project** window once, to display its properties in the inspector. Click on the drop-down menu next to the word **Tag** at the top and select the **Add Tag…** option, as shown in the following screenshot:



This will take you to the **Tags & Layers** menu within the inspector, and you can click on the **+** button underneath **Tag** to add a tag. Name the new tag `Projectile`, as shown in the following screenshot:

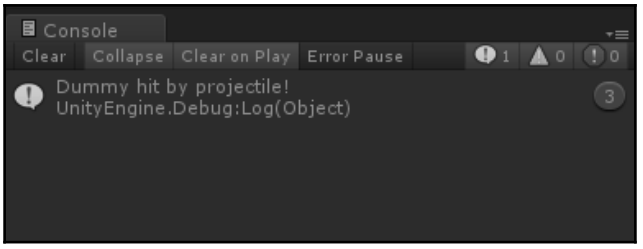


The tag has been added, so the final step is to assign the tag to our **Projectile** prefab. Left-click on the prefab once again in the **Project** window to get back to its main component display and select the **Projectile** tag from the drop-down **Tags** menu, as shown in the next screenshot:



Now, every instance of the **Projectile** prefab will be tagged as such, and the `OnCollisionEnter` function can do its job.

Drop an instance of your **PracticeDummy** prefab into the scene or use the one that you created the prefab from. Press Play in Unity and fire a few projectiles at it; every time you hit it, you'll see a new message appear in Unity's console:



Debug statements are a good way to test triggers and other functions that may interact with code that hasn't been written yet. For instance, we don't yet have a function to process a hit on the dummy, but we can verify that we've established our trigger correctly by putting a debug statement in place of where the function will eventually be called.

Now we'll write the function that gets called when we hit the practice dummy. Create a new function in your `PracticeDummy` class called `OnDummyHit`:

```
private void OnDummyHit()
{

}
```

In the preceding function, we want to change the color of the dummy's node from red to green. To do that, we'll need the renderer associated with the `Node` object, which is a child of the `PracticeDummy` object that our script is attached to.

We could get a reference to the renderer of the node dynamically as soon as it's hit, but we may want to call this function multiple times, so we'll create a member variable to hold onto the reference.

Add the following member declaration to your `PracticeDummy` class:

```
public class Practice Dummy : MonoBehaviour
{
  private Renderer nodeRenderer;
  ...
}
```

To initialize it, declare a new function in your script called `Awake` and call `GetComponent` within it:

```
private void Awake()
{
  nodeRenderer = GetComponentsInChildren<Renderer>()[1];
}
```

Notice the array accessor `[1]` at the end of our `GetComponentsInChildren` function call. The `GetComponentsInChildren` function also returns components on the calling object itself at the first index, `[0]`, so we can access the renderer of **Node** by reading the second returned component, which has index `[1]`.

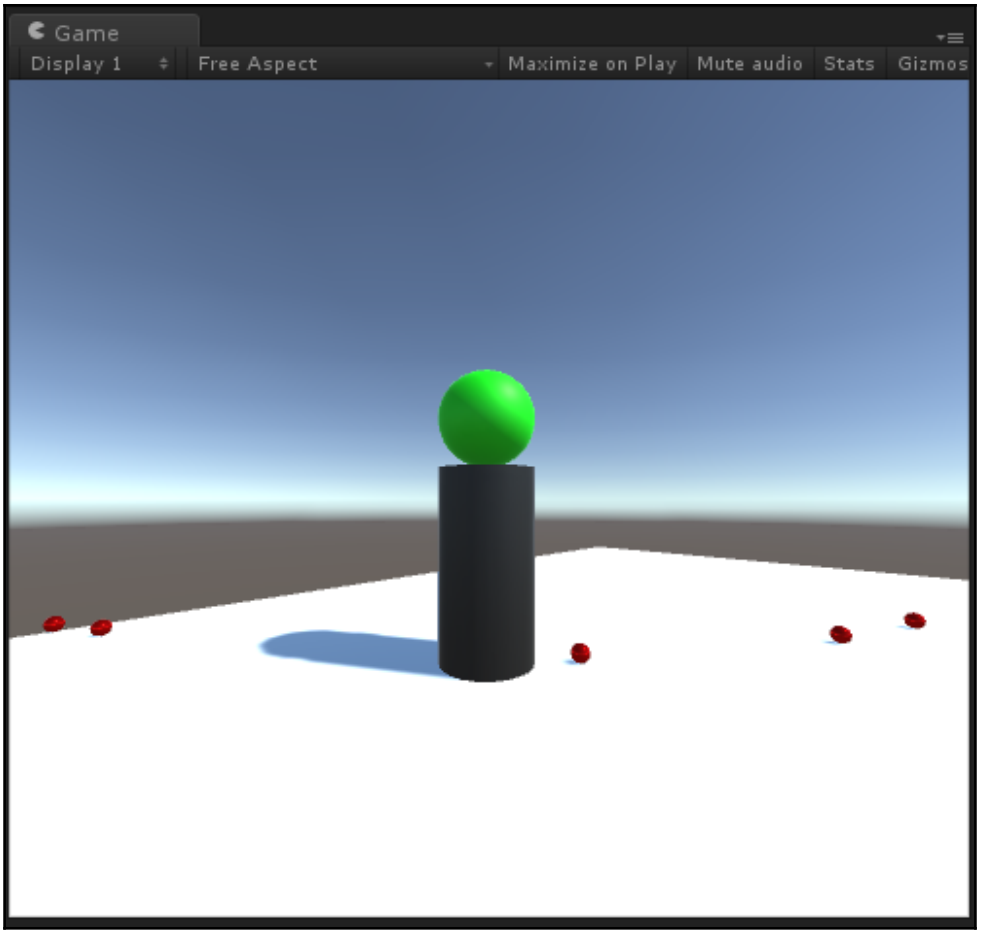
`Awake` is a special function name in Unity, like `Start` and `Update`. `Awake` is the first function called on all objects as soon as they are in an active scene. `Awake` is called on every object before `Start` is called, so it's a good idea to initialize references in `Awake` and use `Start` for initializations that depend on those references.

Now add the following line to your `OnDummyHit` function to access the renderer and change its color to green:

```
private void OnDummyHit()
{
  nodeRenderer.material.color = Color.green;
}
```

Finally, replace your `Debug.Log` call in the `OnCollisionEnter` function with a call to `OnDummyHit`.

Now run your scene again and hit your practice dummy with a projectile (or five). You'll see the node on top turn green after the first hit, as shown in the following screenshot:

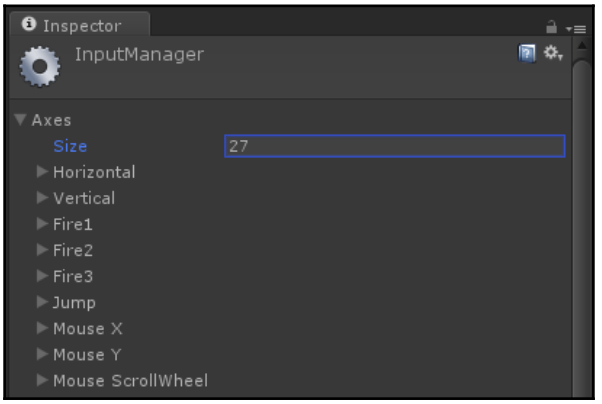


You've done it! You now have an input-based mechanic and an object in your environment that responds to it. Obviously, everything you've done in this chapter so far has been diluted down to technical implementations, but take some time to play around with what you've created and use your imagination to make a few modifications to the way the system works. Maybe the practice dummy moves, maybe the projectiles are slower, or maybe there are several platforms in different locations that each have dummies to hit on them.

Creating custom Unity input axes

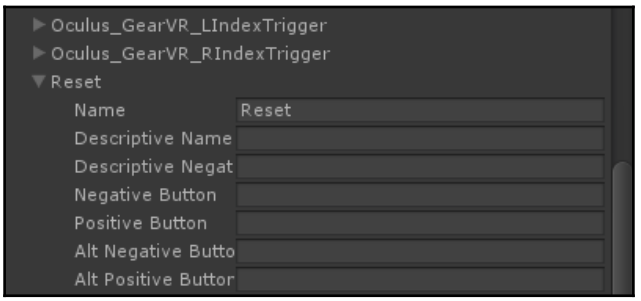
So far, you've coded your own input and used predefined axes in Unity's **InputManager**, but you haven't yet defined your own axis. In this section, we'll do this by adding a new axis called **Reset**, which we'll tie in with a dummy manager to find all practice dummies in a scene and set all of their nodes back to red.

Open the **InputManager** again by mousing over **Project Settings** in the **Edit** menu and selecting **Input**. Click on in the **Size** field directly beneath **Axes** and add one to the total count; in the following screenshot, we had 26 axes, so we updated it to `27`:



Now, scroll to the bottom of the list of axes and you'll see a duplicate of whatever the last axis was at the end of the list. When you create a new input axis, it will automatically be assigned all of the same values that the last one had, but we'll be replacing these values to make it unique.

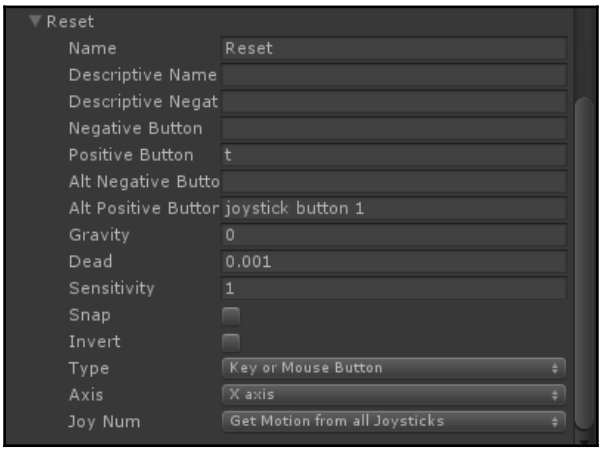
Expand the final input axis in the list and rename it `Reset`, as shown in the following screenshot:



For the **Positive Button** value, enter `t` for the *T* key on the keyboard. For the **Alt Positive Button** value, we'll use the *B* button on the Xbox controller, which can be represented with joystick button 1.

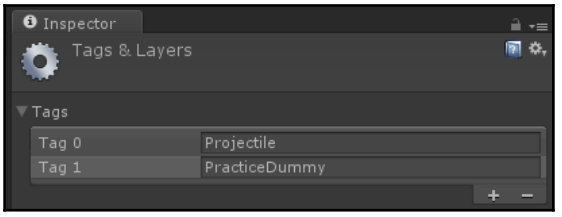
Every button on the Xbox controller has a unique identifier for Unity's input system. The mapping is as follows:

Change the **Type** to **Key or Mouse Button**; the **Axis** setting doesn't matter unless you're interpreting some kind of axial movement, such as one of the analog thumbsticks. When you're finished, your new axis should look like the following:

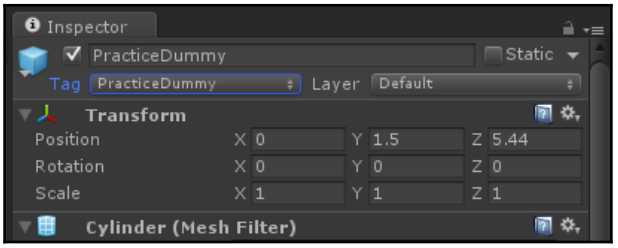


Now your axis is ready to use, so let's create our script to reset all of the practice dummies. We want this script to be able to automatically find any active practice dummies in a scene, so let's give our dummy prefab a unique tag to make instances easy to locate.

Add a new tag in the same way in which you added the **Projectile** tag, but name it `PracticeDummy`, as shown in the following screenshot:



Assign it to your **PracticeDummy** prefab definition in the **Inspector** window, as shown in the following screenshot:



The only other thing we need to add to our **PracticeDummy** prefab is a function to turn the renderer red again. Create a new public function in the `PracticeDummy` class, called `ResetDummy`, and add the following line to set the renderer's color to red:

```
public void ResetDummy()
{
  nodeRenderer.material.color = Color.red;
}
```

Next, create a new script called `PracticeDummyManager` and open it for editing. Declare a new function called `ResetAllDummies`:

```
private void ResetAllDummies()
{
}
```

Add the following line to create an array of all `PracticeDummy` instances in the scene:

```
private void ResetAllDummies()
{
  GameObject[] allDummies =
  GameObject.FindGameObjectsWithTag("PracticeDummy");
}
```

Now add a `for` loop that will loop through every dummy in the array:

```
private void ResetAllDummies()
{
  GameObject[] allDummies =
  GameObject.FindGameObjectsWithTag("PracticeDummy");
  for(int i = 0; i < allDummies.Length; ++i)
  {

  }
}
```

Inside the `for` loop, get a reference to each dummy's `PracticeDummy` component and call `ResetDummy` on it:

```
private void ResetAllDummies()
{
  GameObject[] allDummies =
  GameObject.FindGameObjectsWithTag("PracticeDummy");
  for(int i = 0; i < allDummies.Length; ++i)
  {
    allDummies[i].GetComponent<PracticeDummy>().ResetDummy();
  }
}
```

Next, add the input check for our custom axis. In the `Update` function for `PracticeDummyManager`, add the following lines:

```
void Update()
{
  if(Input.GetButtonDown("Reset"))
  {
    ResetAllDummies();
  }
}
```

Finally, we need an object to hold the `PracticeDummyManager` script. Create a new empty game object from the hierarchy's **Create** menu and name it `PracticeDummyManager`. Then, drag the `PracticeDummyManager` script on top of it to add it as a component.

You're ready to test! Add a few more instances of the **PracticeDummy** prefab to the scene and press Play so that you can hit a few with some projectiles. Press either of the two buttons you tied to the **Reset** input entry—the *T* key or the *B* button on the Xbox controller—and watch the nodes on all of your dummies reset back to red.

Summary

You've now worked with basic key-bound input and Unity's built-in input axes and have made your own. Key-bound input is fine for testing and debugging, but you'll thank yourself later if you follow Unity's input pattern (or your own) rigidly.

Unity's input system in particular allows you to check several different buttons or keys with a single input name. This is helpful for avoiding long blocks of code and also allows you to add more input methods later without changing the code.

In this chapter, we started what will eventually become a full game that we build upon in every chapter from here on. It's still small and simple, but we'll add to both our player and the world around them to make it an immersive and fun experience. In the next chapter, we'll add more presence to the game by establishing an actual level environment and tools that the player can pick up and use to augment their experience.

5
Establishing Presence

Up until this point, we've been designing several discrete systems in VR for every aspect of our game, including input mechanics, gameplay scripts, and simple objects. However, we're still missing a real environment to lend a sense of presence to the experience.

Presence can't be defined as easily as a number of singular technical components. It's a combination of several things, some blatant and some subtle, that all contribute to a player's feeling of immersion in a space. This includes the visible game world, lighting, interaction between the player and the world, and a feeling of context or purpose that gives the player the feeling like they belong there. In this chapter, we'll undertake the sizable task of making our audience feel at home in our game like they are part of a world, not just part of a simulation.

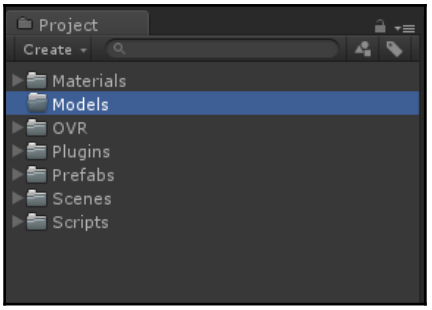
This chapter will cover the following topics:

- Setting up the game environment
- Accentuating depth with particle effects
- Customizing the skybox
- Enabling player interaction with the world
- Populating the game world with resources
- Using resources to interact with the environment

Setting up the game environment

We're creating an arena combat game, but up until this point, we've just been working on a simple square floor. In this section, we'll build out an environment for our game that will give our player somewhere to go.

Create a new folder in the **Project** window and name it `Models`:

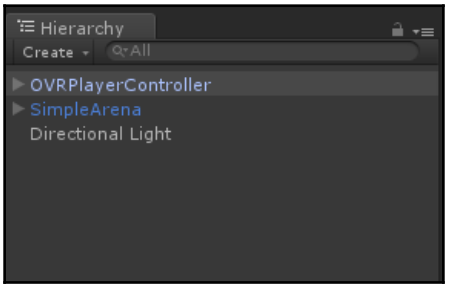


Find the file included with this project called `SimpleArena.skp` and drag it on top of your `Models` folder to add it to the project. This model is a structured but simple level that was created in Sketchup, a free modeling program that Unity is capable of importing and adding to projects.

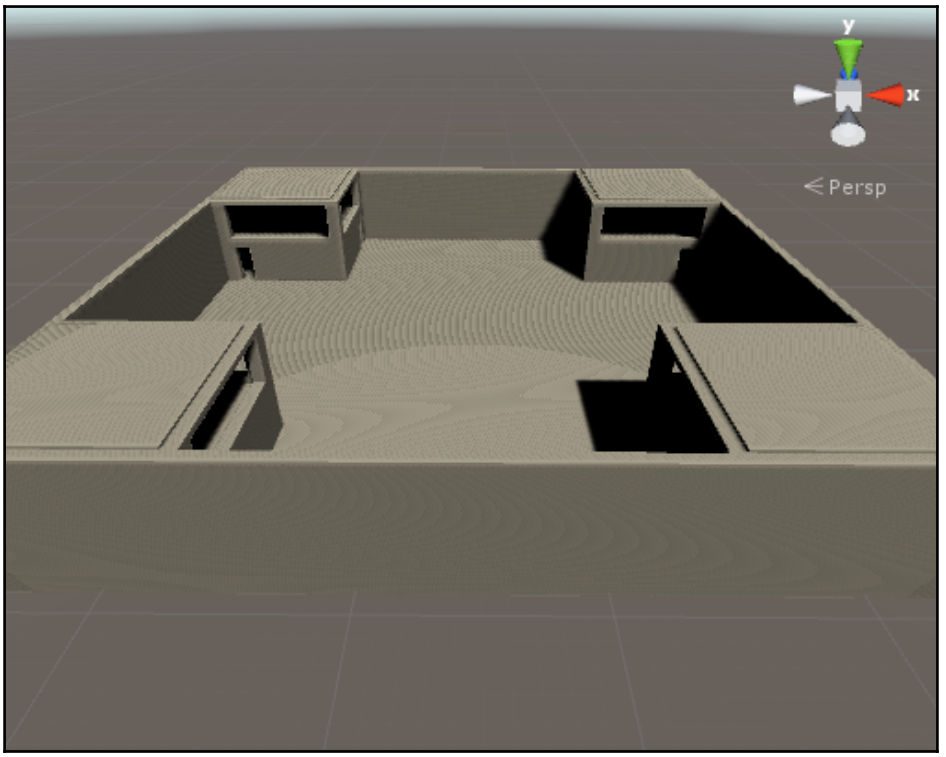
Sketchup is only one of many free modeling programs that you can use to create art assets for your game. Blender is a commonly-used modeling program that's less focused on architectural and environmental modeling; it can be downloaded from `http://www.blender.org/`. If you're looking for top-of-the-line tools and don't mind spending a bit of money, you can download Autodesk's Maya or 3DS Max, which are prevalently used by professionals in the game industry.

Drag and drop the model over your **Hierarchy** window to add an instance of it to the scene. Then delete the **Floor** object and position the arena model to where it used to be. You can also delete the **PracticeDummyManager** and any **PracticeDummy** instances since we'll be adding different targets in this chapter.

Your hierarchy should now look like the following:



And the scene with its automatically imported material will look like the following:

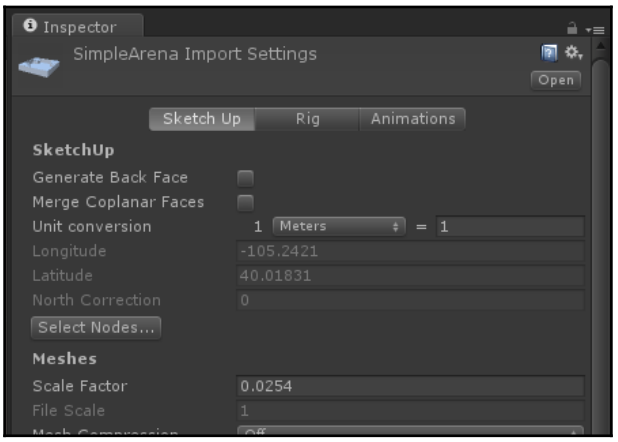


The level doesn't look very interesting yet, but next we'll take several steps to improve its appearance and give it some personality.

Configuring lighting properties

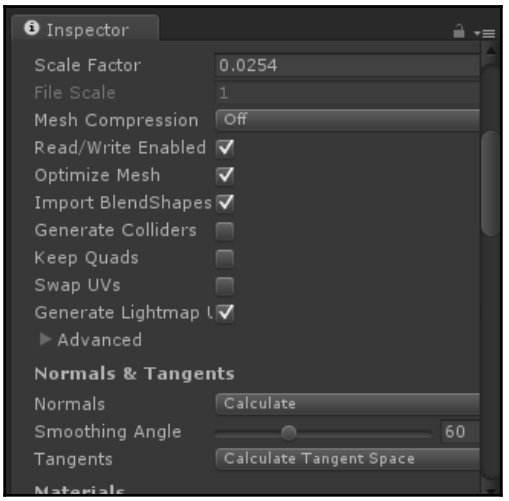
It's easy to functionally light a scene with a simple directional light like the one we've been using, but Unity comes with a built-in lighting engine called **Enlighten** that produces extremely realistic lighting effects including light bounces, ambient occlusion, and global illumination. All of this information is stored in what's called a **lightmap**, and creating these lightmaps is a process called **baking**.

Before we bake our model, though, we need to modify a few of its import settings. Left-click on the **SimpleArena** model in your **Project** window to display its import properties in the **Inspector** window. The properties menu will look like the following:

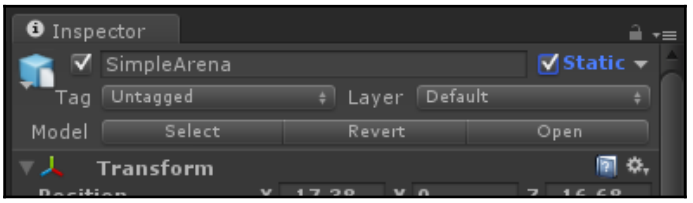


Scroll down to the **Meshes** section and check the box that says **Generate Lightmap UVs**. Then click on the drop-down menu next to **Normals** and select **Calculate**. Some models have custom normals that the modeler has specified for a certain look, but in our case, the normals were automatically generated by Sketchup, and it's fine to let Unity overwrite them for the purposes of lighting.

When you're done, the import settings will look like the following:



Our model is ready to be baked, but Unity won't include it in the baking process unless we mark its instance as **Static** in the scene. Left-click on the instance of **SimpleArena** in the hierarchy and check the box in the upper right of its **Inspector** window labeled **Static**. Accept the prompt to enable the static flags for child objects as well; this ensures that everything within the level will be given the same flag:



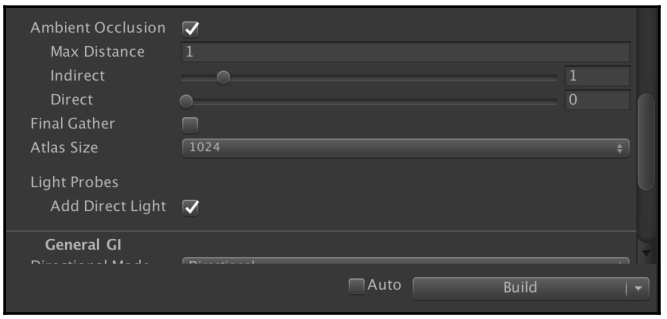
Any object tagged as a lightmap static will be included in the lightmap bake, so now that we've tagged our **SimpleArena** as static, we can finally bake lighting for it.

Unity is set to constantly bake lightmaps to make iteration convenient, but since this is our first bake, we'll disable this feature temporarily and configure all of our settings before we allow the process to start.

Open the **Window** menu on the top toolbar and select **Lighting**. This will open the lighting properties window, which you can use to change how every light affects your scene, including how they're baked. Start by unchecking the **Auto** box in the bottom right of the window.

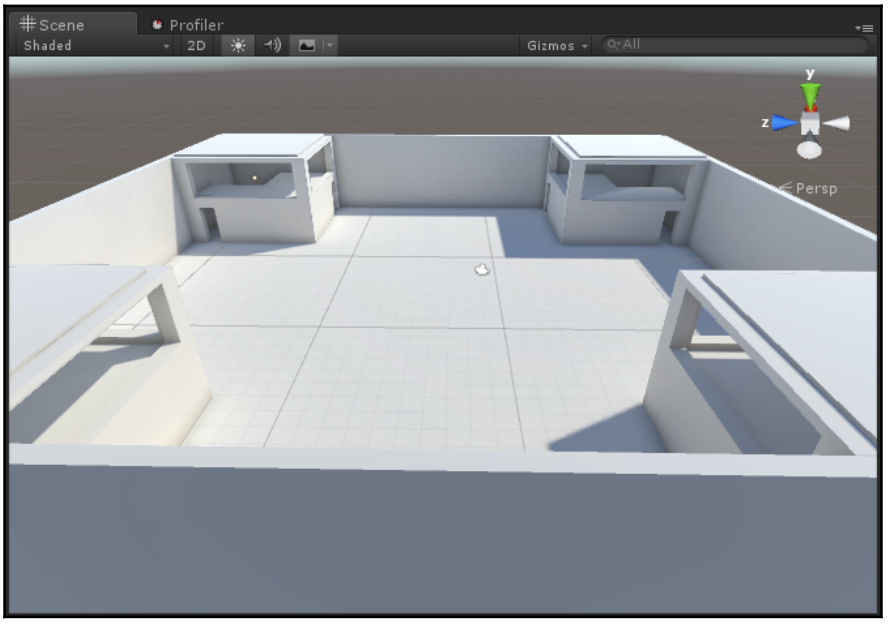
Drag the **Directional Light** in your scene from the hierarchy over the field marked **Sun** in the **Lighting** window. Then set the **Ambient GI** to **Baked**. The **Enviromental Lighting** section should now look like the following:

Next, scroll down to the section labeled **Baked GI** and check the box next to **Ambient Occlusion**. Feel free to leave the **Direct** and **Indirect** sliders at their default values. After that, click on the **Build** button in the bottom-right of the **Lighting** window (as shown in the following screenshot) to begin the baking process:



Baking is generally a time-consuming process, so plan your lightmap baking for a time when you don't need to do anything else for a while. Baking our simple scene will only take a few minutes right now, but complex scenes can take hours, depending on the density and the lighting settings.

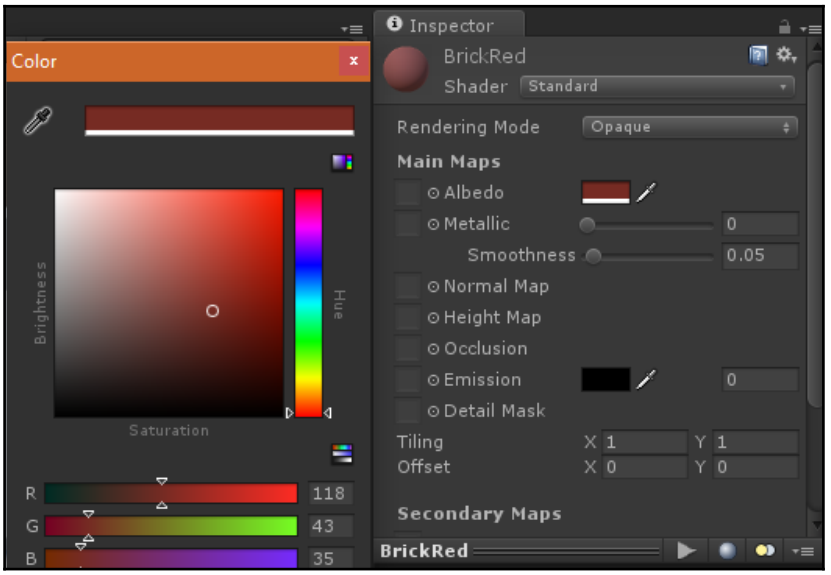
Once your bake has finished, the lighting in your scene will look much cleaner and realistic:



Our level's lighting looks quite nice now, but the environment still feels dull because it has a single default color applied to all objects in the scene. Fortunately, colors of individual meshes in a model can be changed without having to edit the source model and reimport it.

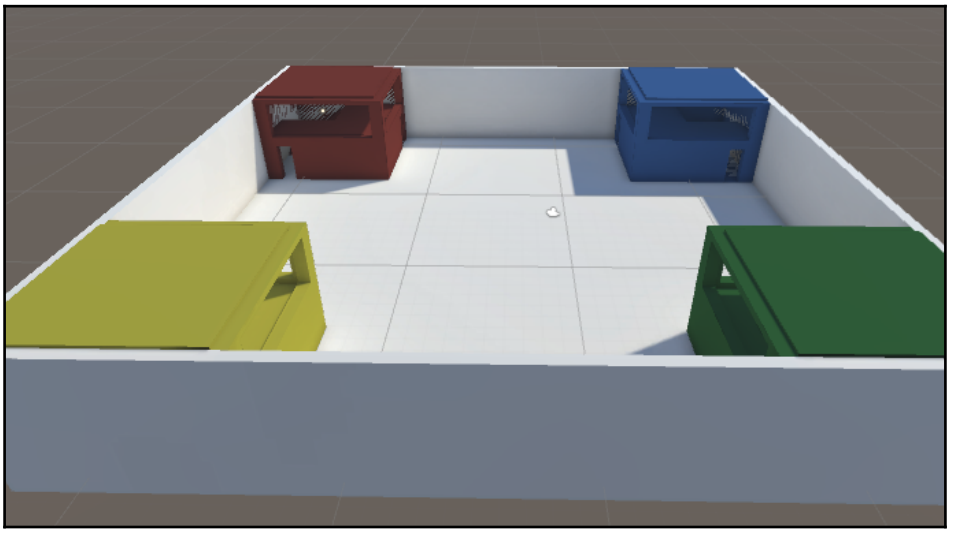
Adding some color to the scene

Right-click on the `Materials` folder, mouse over the **Create** submenu, and select **Material**; name the new material `BrickRed` and left-click on it once to display its properties in **Inspector**. Set the **Smoothness** to `0.05` and left-click on the **Albedo** property to select a dark red area on the color matrix, as shown in the following screenshot:



Now apply it to the scene by clicking and dragging the material from your **Project** window to your **Scene** window and releasing it on top of one of the buildings.

Create three more unique colors and apply one to each building; we used yellow, blue, and green, as shown in this screenshot:

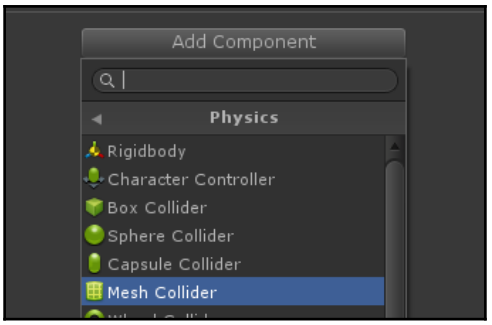


Now that our scene has some color, we'll give it some physical colliders so that our player can move around it.

Adding colliders to imported mesh objects

The primitive objects we've created so far from our hierarchy's **Create** menu are instantiated with their own collider component already added. However, custom meshes that we add to the scene from model resources don't create their own colliders by default, so we'll have to add them manually.

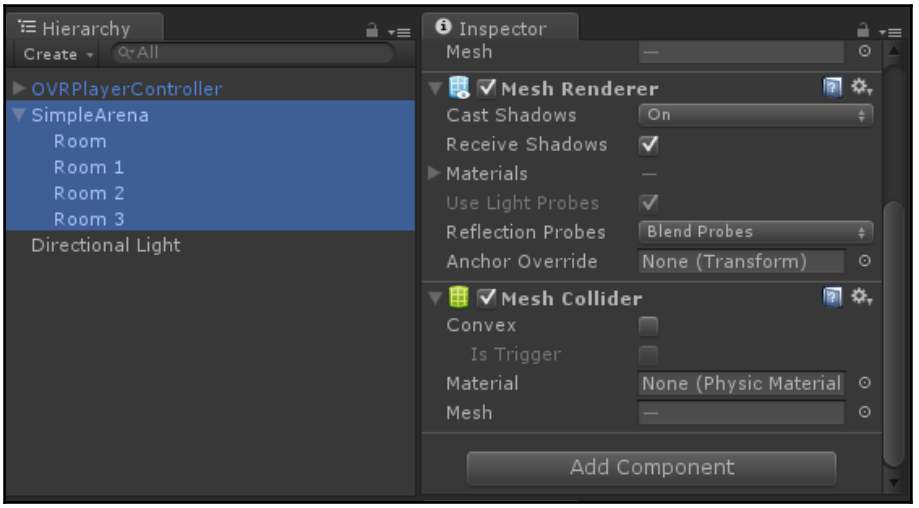
Expand the **SimpleArena** object in the hierarchy to reveal each mesh that makes up the scene. We'll start with the **SimpleArena** object itself; left-click on it to display it in **Inspector** and then click on the **Add Component** button. Select **Mesh Collider** in the **Physics** category, as shown in the following screenshot:



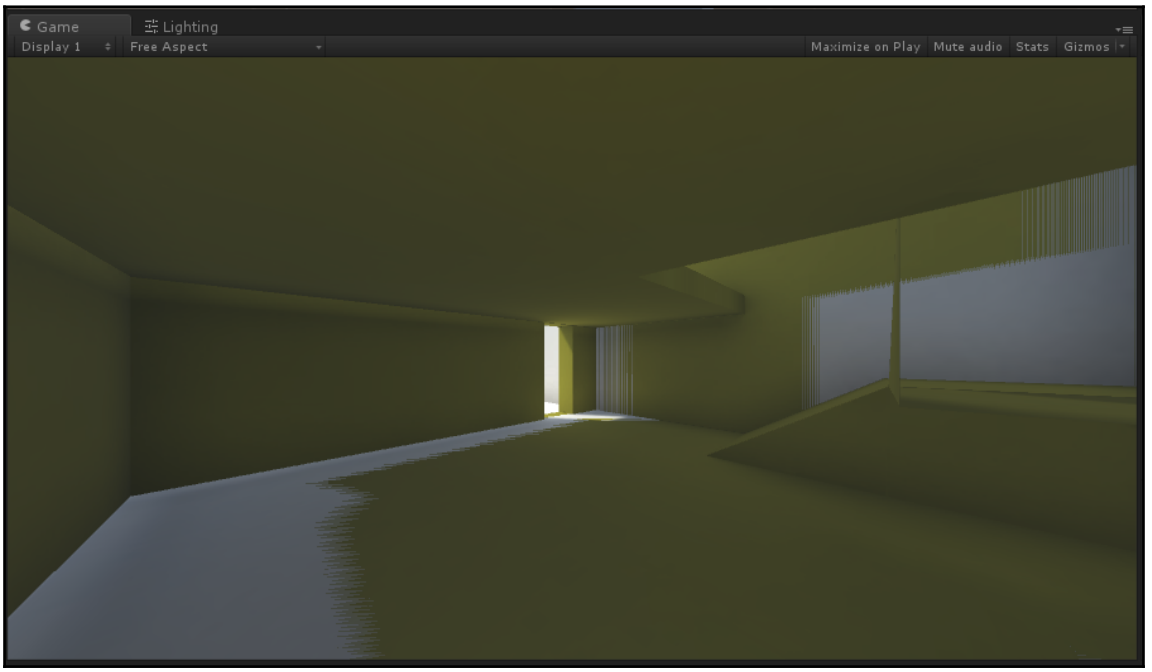
A mesh collider will form a collision shell around the mesh's geometry. This is the most accurate kind of collider, but because it has a complex shape instead of a primitive shape, it's more expensive to calculate collisions on.

As a rule, you should only use a mesh collider if no other collider will do, even on complex objects. For instance, consider a model of a refrigerator; a mesh collider would envelop the body, handles, doors, and feet of the model, but because the smaller elements don't influence the overall volume too much, we could probably just get away with a rectangular box collider that roughly surrounds the width, height, and depth of the model.

Now apply mesh colliders to the four rooms that are child objects of the **SimpleArena** object. When you're done, the **SimpleArena** object and all of its children should share a **Mesh Collider** component in common, as shown in the following screenshot:



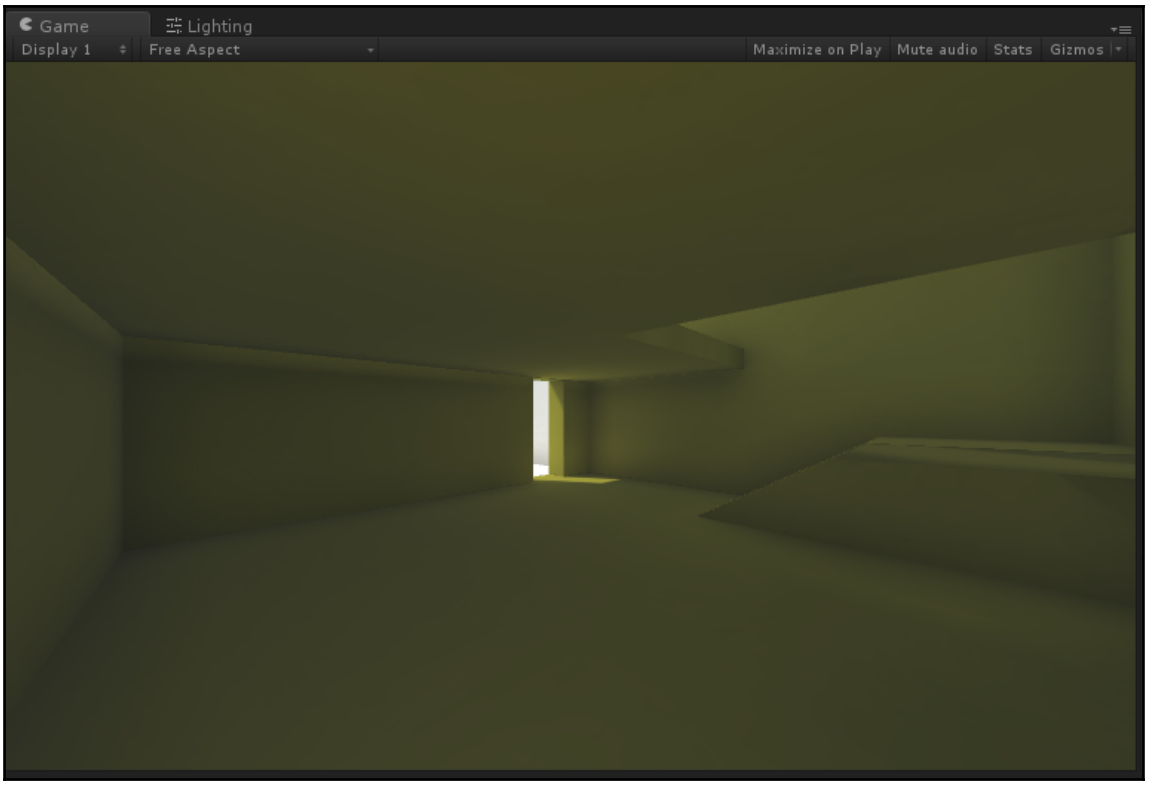
Your level is now navigable, so take this opportunity to walk around it in VR and get a sense of how it looks so far. You may notice some color flickering in the buildings, as shown in the following screenshot:



This graphical artifact is called **Z-fighting**, and it occurs when two planes are being rendered extremely close together. In this case, it's the walls and floors of the building fighting with the walls and floors of the white level base. Depending on the software that the model was rendered in, these artifacts may not be able to be seen until they're rendered in Unity, so you may have to modify a mesh after you import it. Fortunately, in this case, the fix is as simple as moving the rooms slightly away from the floor and walls.

Left-click on a room in the hierarchy and press the *W* key to activate the move tool. Move each room slightly up and away from the corner walls, but not so much that the mesh looks out of place.

After you've moved them away from the clashing surfaces a slight amount, press Play again and walk through each building to make sure no Z-fighting remains. The rooms should now display no flickering at all, as shown in the following screenshot:

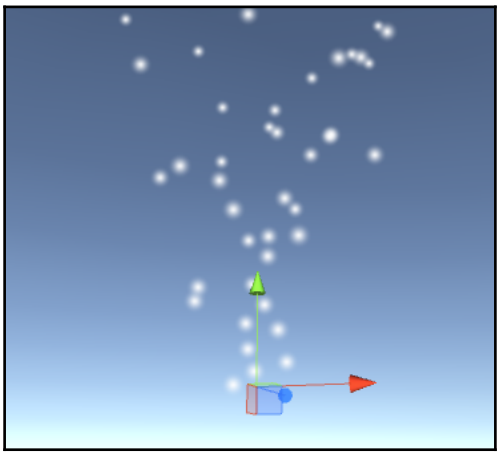


At this point, we've filled out the basic geometric and physical elements of our first level. Next, we'll look at a more subtle method of making our environment feel more complete—particle effects.

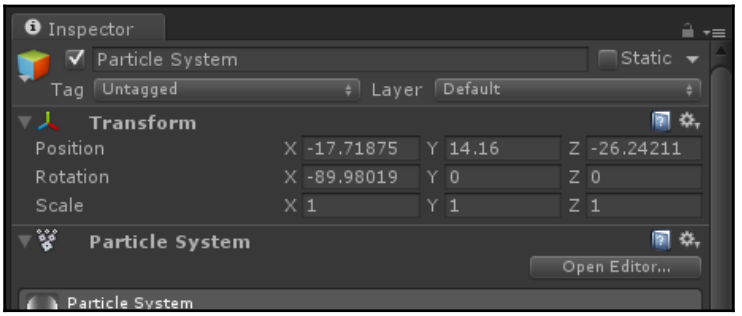
Accentuating depth with particle effects

It's easy to establish a sense of scale indoors, but sometimes when you're in a large open space, it's difficult to see how far away something is, even with the depth perception that the Oculus Rift enables. To mitigate this, many VR games and experiences use a subtle global particle effect to give players a depth reference independent of the level's geometry. In this section, we'll explore Unity's Shuriken particle system to explore how a technique like this is implemented.

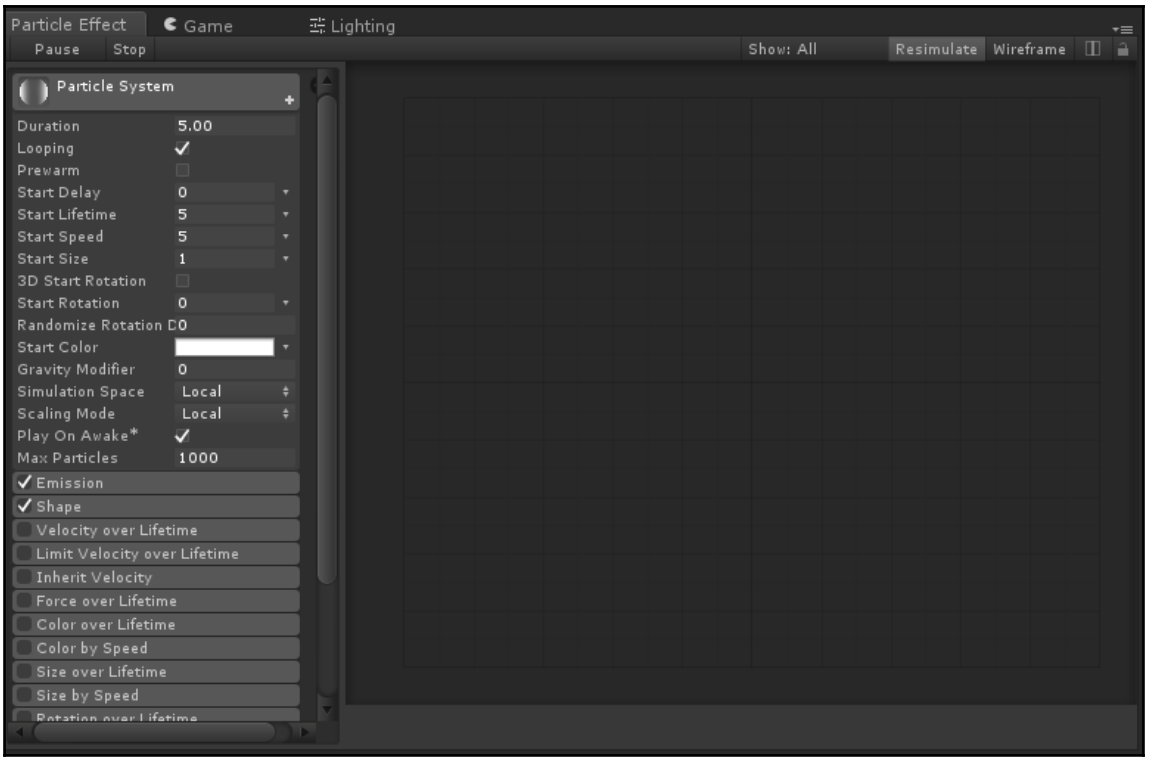
Open the **Create** menu in the **Hierarchy** window and select **Particle System** to create a new default particle system in your scene. It won't look like much to begin with, just simple white dots emitting from a point in a conical shape:



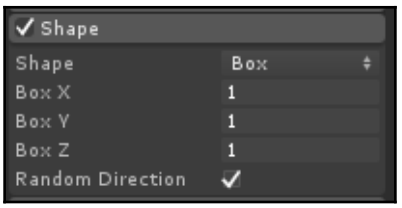
We can edit the particle properties directly in the **Inspector** panel, but the system is pretty complex, so we can also expand it out into its own menu to get a better view of what we're editing. Open the particle system's properties in **Inspector** and press the **Open Editor…** button in the top right of the **Particle System** component, as shown in the following screenshot:



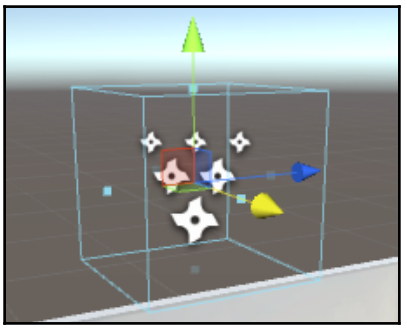
The particle effect's editor will pop open in a new window, which you can pin somewhere in your Unity window if you wish. The left column in the editor is a list of properties that you can edit on the system, and to the right is a graph area where you can customize particle animations, as shown in the following screenshot:



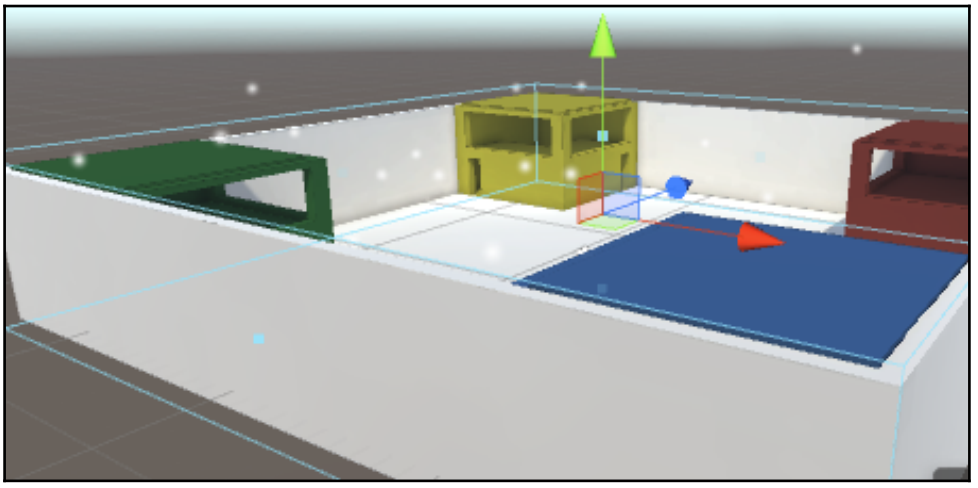
We'll start by specifying a new shape that will let our particles float around our entire level. Click on the **Shape** property in the left column, select the **Box** shape, and check the box labeled **Random Direction**:



We'll need to update the box's size to fit our level, but instead of setting each individual *X*, *Y*, and *Z* value manually, we'll adjust it by hand in the **Scene** view. Highlight the **Particle System** in the hierarchy and notice the blue box bounds that now appear around it with small dot handles in the middle of each face:



Click and drag each of these handles to resize the box so that it's large enough to envelop the level (you may have to reposition the box using the translation tool as well). When you're done, large white particles will be floating around your scene like a snowdrift:



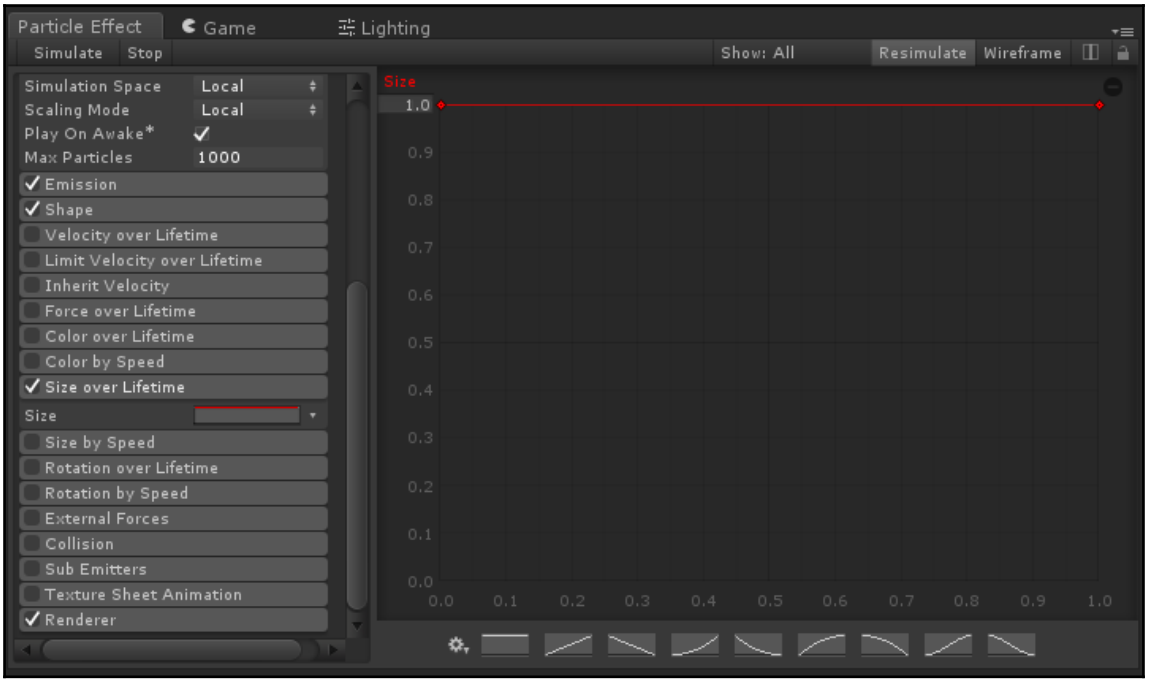
Since we're using the particle system for a subtle depth cue, we'll tone down some additional settings so that our effect looks more like small dust particles floating around the scene.

In the **Particle Effect** window under the **Particle System** label, set the **Start Lifetime** to `10`, the **Start Speed** to `1`, and the **Start Size** to `0.1`. Set the **Start Color** to a light sky blue. This will create very small particles that will slowly drift through the scene and blend in with the level and the sky.

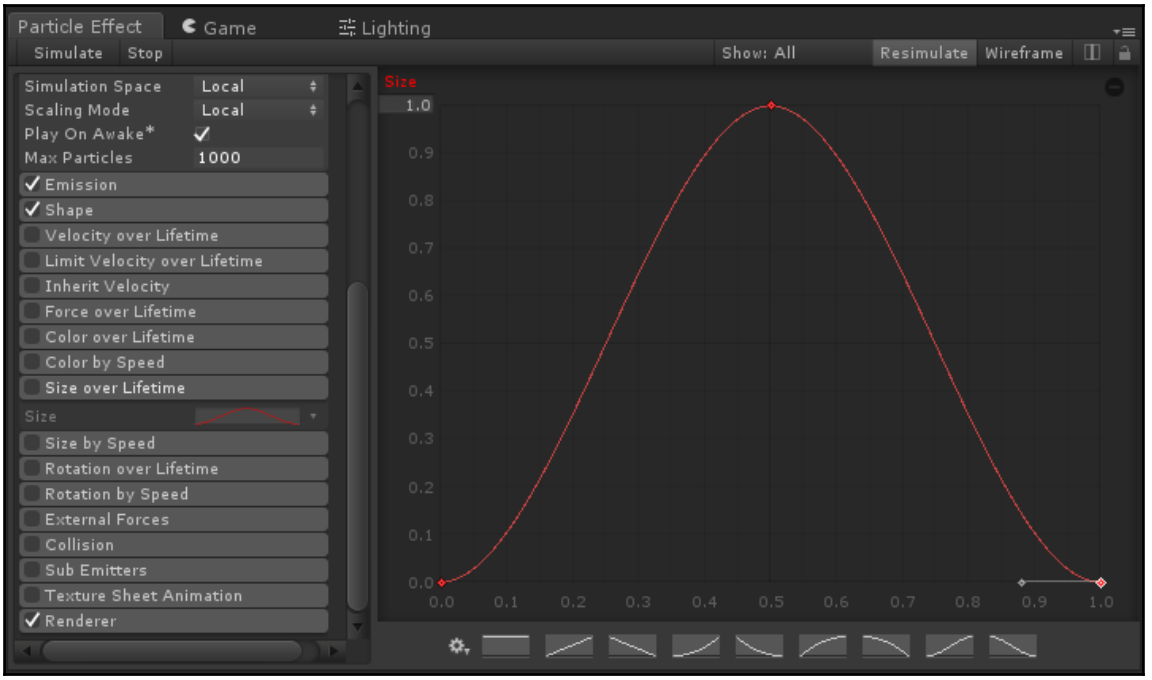
Next, under the **Emission** label, change the **Rate** value from **10** to `5`. This will spawn particles more infrequently, so there aren't so many that they distract the player from the rest of the environment.

Finally, we'll change the size over the particle's lifetime. We don't want any particles suddenly popping on or off in front of the user, so we'll start them off imperceptibly small, scale them up as they get older, and then scale them back down before they disappear.

Click on the checkbox on the **Size over Lifetime** property to enable it and again on the label to expand it. Then click on the field next to the **Size** label to display the animation curve in the graph view, as shown in the following screenshot:



The animation graph will form a spline between all points, or **keys**, on the line. By default, there are two points, one on the far left and one on the far right, both at **1.0** on the vertical **Size** axis. Right-click on the middle of the line (above the **0.5** horizontal mark) and then click on **Add Key**. Leaving the middle key at the **1.0** mark on the **Size** axis, drag the original two points both down to the **0.0** mark. Your new curve should look like the following:



We'll leave our basic effect as complete for now, so walk around the scene in VR for a while to see the effect. Subtle particles can add a lot to how a scene feels; it's all about finding the right balance between noticeable and unobtrusive. By customizing the properties further, you can add context to the scene—think sawdust particles in a wood mill or sand particles in a desert.

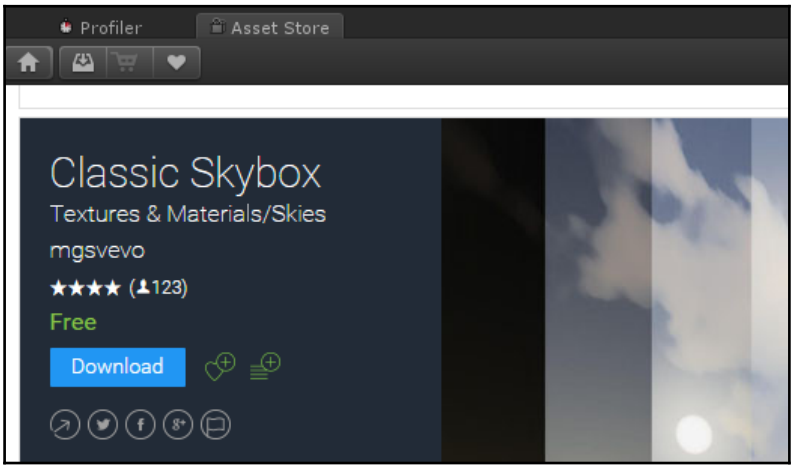
Play around with the different properties in the particle system and add some extra elements if you feel like it before moving onto the next section, where we'll add some detail to the plain blue sky we've been using.

Customizing the skybox

Unity's default skybox is nice and simple, but a little dull. The sun appears in different positions in the sky based on the angle of your scene's directional light, but the sky lacks clouds, and everything south of the horizon line is a plain gray. In this section, we'll look at using cubemap images to define your own skybox and add a little more flair to the sky.

Cubemaps are sets of six images that combine to form a cubic shell around the very furthest extents of your rendered environment. They can be hard to create without specialized renderers due to the warping that has to be applied at the corners, but fortunately there are several available for free in the Unity Asset Store that we can download and import to get a feel for how they work.

Open the **Asset Store** from the **Window** menu and search `Classic Skybox`. In the search results, find the following package:



Download and import the package into your project; when it's done, you'll see the `Classic Skybox` folder appear in your **Project** window. Left-click on the **sky09** material to display it in **Inspector**; you'll notice the **Skybox/6 Sided** shader at the top as well as the six texture slots that come with it (one per side of the cubemap).

Open your **Lighting** window and drag the **sky07** material from the `Classic Skybox` folder into the **Skybox** field, as shown in the following screenshot:

Now take a look at your scene; the standard Unity skybox has been replaced with the new cubemap, and now the sky is slightly more distinct.

The skybox is one of the ambient sources included in the light baking process, so to maintain 100% lighting accuracy, we'll need to rebake our light maps with our new skybox. The influence of the skybox on the baked lighting is most noticeable with bold sky colors; if you were to bake the scene with a sunset skybox, you'd notice a yellowish hue applied to all surfaces the light hit.

In a future chapter, we'll dig into methods of making our sky more full and animated with volumetric clouds and day/night cycles, but for now, we'll go back to other elements that our scene needs to make it feel like a real arena.

Enabling player interaction with the world

Right now, our environment completely lacks interaction. Our player can move around it, but there's nothing to do; there's no particular reason to be anywhere on the map. In this section, we'll create a dynamic wall system that players can interact with in order to add some strategic diversity to the map.

Creating the dynamic wall prefab

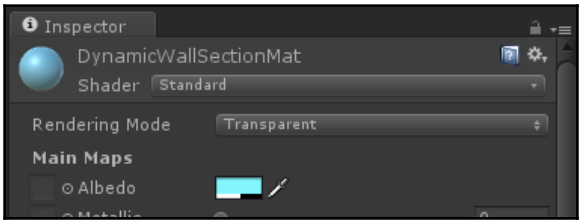
First, we'll create a prefab for a dynamic wall that players can activate to raise or lower a wall section in the level. For now, we'll make this prefab out of primitive shapes, but we'll structure it in a way that could easily be swapped out with complex models later.

Create a new cube in your scene and set its **X**, **Y**, and **Z** scale to be `8`, `0.5`, and `1`, respectively. Name it `DynamicWallBase`. This will serve as the base of the dynamic wall that the player interacts with to raise or lower the section.

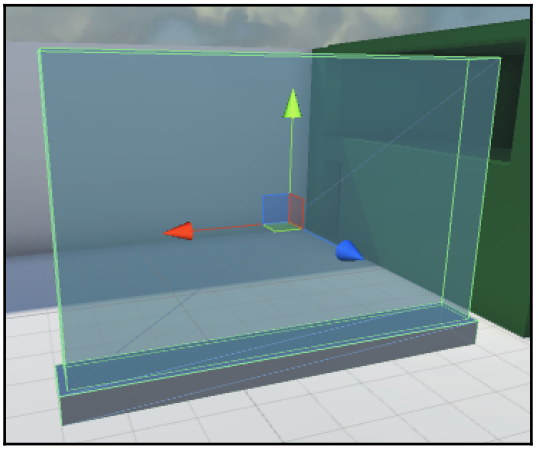
Next, we'll create the wall section, which is the part we'll eventually animate in code. Create another cube named `DynamicWallSection`, this time setting its scale to be `7.75`, `5`, `0.75`. Position it above the base section resting in the center of the top face.

Next, create two materials named `DynamicWallBaseMat` and `DynamicWallSectionMat` in your `Materials` folder. Give **DynamicWallBaseMat** a light gray color and assign it to your base object by dragging it over the mesh in the **Scene** view.

With the **DynamicWallSectionMat** selected in the **Inspector** panel, click on the drop-down menu next to the **Rendering Mode** field and change it from **Opaque** to **Transparent**, as shown in the following screenshot:



Now set the **Color** property to a light blue color and give it some transparency by halving the alpha (**A**) value in the color picker. Assign the material to your **DynamicWallSection** object. At this point, your dynamic wall should look like the following:



Click and drag the **DynamicWallSection** object onto your **DynamicWallBase** object in the hierarchy to make it a child of the base object. Finally, drag the **DynamicWallBase** object from your hierarchy to the `Prefabs` folder in your **Project** window to create a prefab of it.

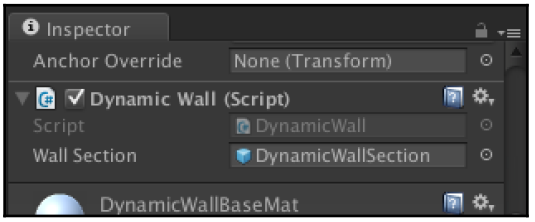
Scripting our prefab

Now that we've got an object for our dynamic wall, we can write a script with functions to expand and collapse it. Create a new script in your `Scripts` folder called `DynamicWall` and open it in your editor.

We'll need a reference to the section object in order to manipulate it, so add the following variable to your `DynamicWall` class definition:

```
public class DynamicWall : MonoBehaviour
{
  [SerializeField]
  private GameObject wallSection;
  ...
}
```

The `SerializeField` attribute above the `wallSection` variable will make it visible in **Inspector**, so we'll be able to assign an object to the script directly in the Unity Editor. Next, we'll need to associate the section object with this variable. Drag your **DynamicWall** script over the **DynamicWallBase** prefab in your **Project** window to add it as a component to the prefab definition. Then open the prefab in the inspector and drag the **DynamicWallSection** child into the **Wall Section** field of the script component (you may need to left-click on the arrow to the left of the prefab in the **Project** window to view its child) , as shown in the following screenshot:



By editing the prefab in the **Project** window, you ensure that all instances of that prefab will inherit the changes you make. However, if you make a change to an instance of a prefab in the **Hierarchy** window, the changes will stay unique to that instance until you manually apply the change to the prefab definition.

Move back to your code editor and create a new function called `MoveSectionDown`:

```
[SerializeField]
private GameObject wallSection;
public IEnumerator MoveSectionDown()
{
}
```

You'll notice that the return type of the function is `IEnumerator`; if you guessed that we'll be using this function as a coroutine, you guessed right. Previously, we've used coroutines to split up a large function over several frames to prevent performance drops, but in this case, we'll use it to gradually move the wall every frame, like an animation. Add the following code to move the wall down slightly every frame:

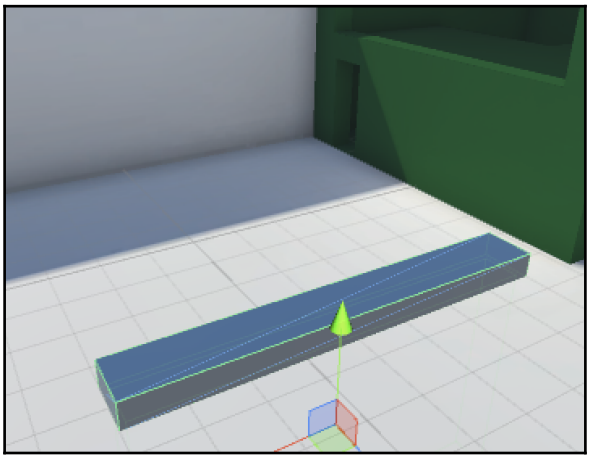
```
public IEnumerator MoveSectionDown()
{
  while(wallSection.transform.localPosition.y > -5f)
  {
    wallSection.transform.Translate(Vector3.down * 10f *
    Time.deltaTime);
    yield return null;
  }
}
```

You'll notice that we're using the `transform.localPosition` value instead of the `transform.position` value to keep track of where the wall is; the `localPosition` value refers to a child's offset from its parent instead of global position, so a `localPosition` value will be the same no matter where the parent object lies in the scene.

Now add a temporary check to the `Update` function that calls this function when the *F* key is pressed (we'll call it properly from the player later):

```
void Update()
{
  if(Input.GetKeyDown(KeyCode.F))
  {
    StartCoroutine(MoveSectionDown());
  }
}
```

Now press Play and hit the *F* key to test your function. You'll see the section collapse into the wall base, as shown in the following screenshot:



Next, we'll create a function to move the wall back up. Create a function called `MoveSectionUp` and define it as follows:

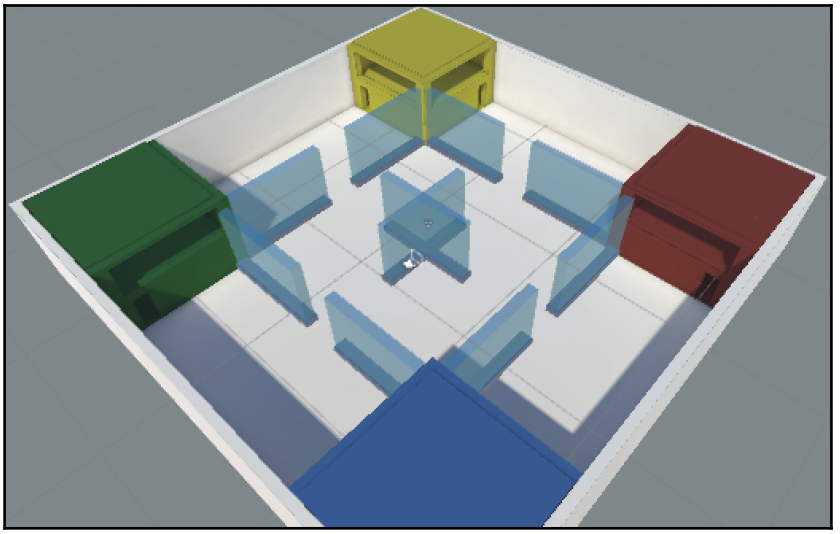
```
public IEnumerator MoveSectionUp()
{
  while(wallSection.transform.localPosition.y < 4.5f)
  {
    wallSection.transform.Translate(Vector3.up * 0.5f);
    yield return null;
  }
}
```

Now make another temporary check in your `Update` function for testing, just to be safe:

```
void Update()
{
  if(Input.GetKeyDown(KeyCode.F))
  {
    StartCoroutine(MoveSectionDown());
  }
  if(Input.GetKeyDown(KeyCode.R))
  {
    StartCoroutine(MoveSectionUp());
  }
}
```

Now press Play again and you should be able to move the wall up and down with the *F* and *G*, *R* keys, respectively.

Once you've verified that your functions work as intended, remove the temporary checks from the `Update` function, leaving it blank. Now that our prefab is more or less complete, let's place some around the map. Set up a configuration with two walls at each building's corner and two crossed in the middle as follows:



With this many walls, there's finally an opportunity for strategy in our map; the walls on each corner block the window views on the second floor of each building, and some can be activated to hinder or redirect the path from building to building.

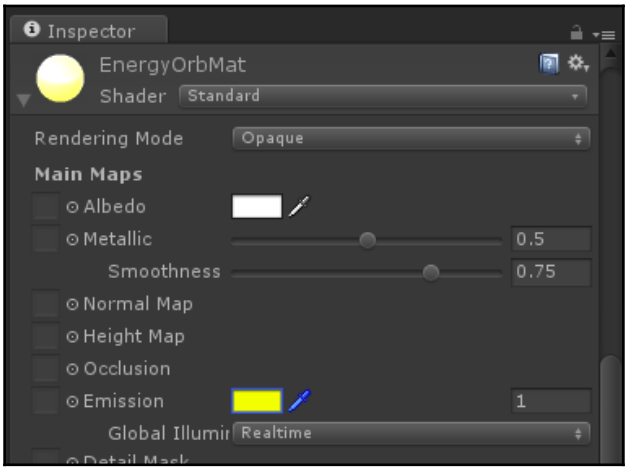
We'll give the player the ability to toggle the walls themselves soon, but since we have so many, we need something to limit that ability so that it doesn't become a chaotic game of constantly shifting walls. This is a good time to introduce in-game resources that the player can explore and collect in order to make a strategic move on the map.

Populating the game world with resources

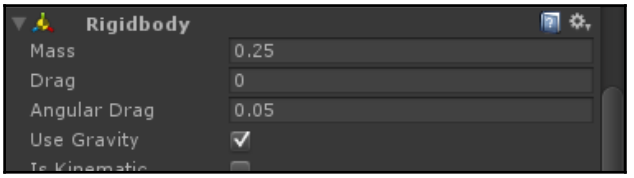
Adding resources to a game map adds an element of strategy and rewards players for exploring areas of the map where there might not be frequent combat. In this section, we'll build upon our old practice dummy prefab and modify it to drop resources that will be required to activate the walls we created in the last section.

To begin, we'll create a new prefab for the resource itself. Create a new **Sphere** object in your scene and name it `EnergyOrb`. Create a new material in your `Materials` folder, name it `EnergyOrbMat`, and apply it to the sphere.

Open the **EnergyOrbMat** material in the **Inspector**. Set the **Metallic** value to `0.5`, **Smoothness** value to `0.75`, and the **Emission** color to yellow, as shown in the following screenshot:



You should now have a shiny, glowing yellow orb—a perfect object to stand out as a collectible. Add a **Rigidbody** component to the **EnergyOrb** from the **Inspector**. Set the **Mass** field to `0.25`:



Create a new tag called **EnergyOrb** in the **Inspector** as well and apply it to the object. Finally, drag the **EnergyOrb** object from your hierarchy onto your `Prefabs` folder to save it as a prefab.

Next, we'll script the action of picking up the orbs. Create a new script called **OrbManager** and add it to the **OVRPlayerController** object.

Open the script and add a variable `totalOrbs` as well as a function `AddOrb`:

```
public class OrbManager : MonoBehaviour
{
  private int totalOrbs = 0;
  private void AddOrb()
  {
    totalOrbs++;
  }
  ...
}
```

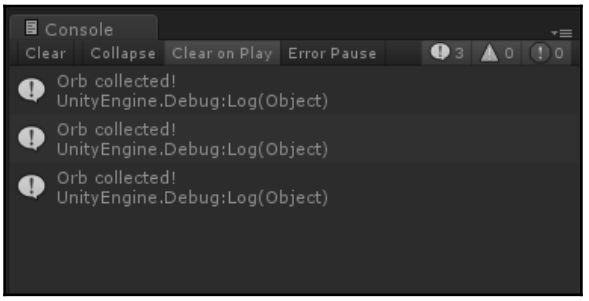
Next, create a definition of `OnControllerColliderHit`, which will be called when our character's **Character Controller** component collides with anything. Add a check for the tag of the object `hit`:

```
private void OnControllerColliderHit(ControllerColliderHit hit)
{
  if(hit.collider.tag == "EnergyOrb")
  {
    AddOrb();
    Destroy(hit.gameObject);
  }
}
```

Prepare your new collection mechanic for testing by adding a temporary debug line to the top of the `AddOrb` function:

```
private void AddOrb()
{
  Debug.Log("Orb collected!");
  totalOrbs++;
}
```

Now add a few orbs to your scene, press Play, and walk into the orbs. You should see the debug line printed in the console every time you successfully pick up an orb:



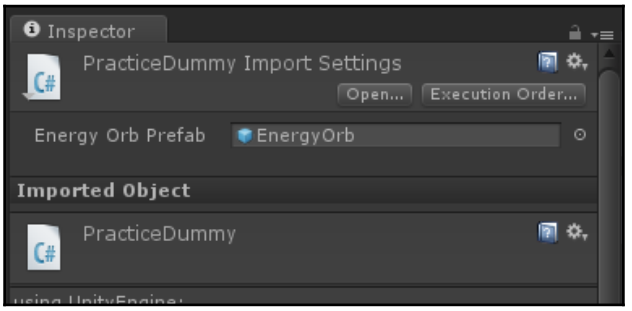
Once you've confirmed it works, remove the debug line from the `AddOrb` function and delete the test orbs you placed in your scene. Next, we'll make the practice dummies emit orbs when they're hit.

Making the practice dummies drop resources

We'll need to add some code to our `PracticeDummy` class, but what we've created from the previous chapters will serve as a good starting point. Open the **PracticeDummy** script and add a serialized field so that we can link the **EnergyOrb** prefab:

```
public class PracticeDummy : MonoBehaviour
{
  private Renderer nodeRenderer;
  [SerializeField]
  private GameObject energyOrbPrefab;
  ...
}
```

Now click on the **PracticeDummy** script in your **Project** window to display its import settings in the **Inspector** and drag the **EnergyOrb** prefab onto the **Energy Orb Prefab** field:



Next, create a new function in your `PracticeDummy` class called `SpawnOrb`. Begin the function by instantiating a new orb above the enemy:

```
private void SpawnOrb()
{
  GameObject newOrb = (GameObject)Instantiate(energyOrbPrefab,
  transform.position, transform.rotation);
  newOrb.transform.Translate(Vector3.up * 2.5f);
}
```

Add some initial force to the orb so that it flies off of the enemy in a random direction:

```
private void SpawnOrb()
{
  GameObject newOrb = (GameObject)Instantiate(energyOrbPrefab,
  transform.position, transform.rotation);
  newOrb.transform.Translate(Vector3.up * 2.5f);
  float xForce = Random.Range(-1f, 1f);
  float zForce = Random.Range(-1f, 1f);
  newOrb.GetComponent<Rigidbody>().AddForce(new Vector3(xForce,
  1f, zForce), ForceMode.Impulse);
}
```

Now we can add a call to `SpawnOrb` to `OnDummyHit`. Also, add a check that will exit the function early if the dummy has already turned green:

```
private void OnDummyHit()
{
  if(nodeRenderer.material.color.g > 0.95f)
  return;
  nodeRenderer.material.color = Color.green;
  SpawnOrb();
}
```

We'll give our dummy a reset time so that eventually it will return to being red and can be used to spawn resources once again. Add a variable `cooldownTimer` in `PracticeDummy`:

```
public class PracticeDummy : MonoBehaviour
{
  private Renderer nodeRenderer;
  [SerializedField]
  private GameObject energyOrbPrefab;
  private float cooldownTimer = 0;
  ...
}
```

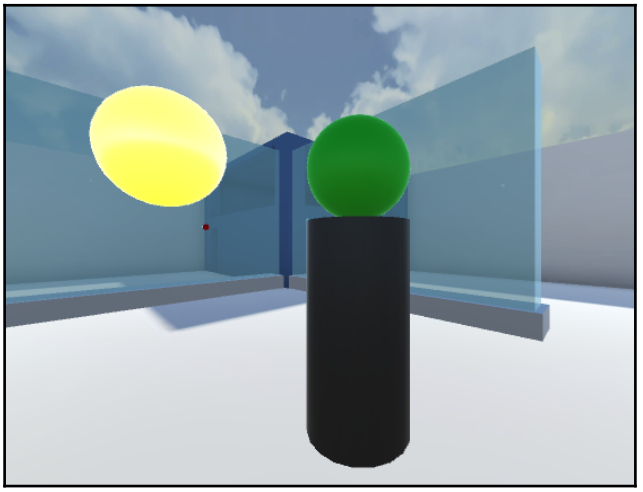
Add a line to the end of `OnDummyHit` to add 15 seconds to the `cooldown` timer:

```
private void OnDummyHit()
{
  if(nodeRenderer.material.color == Color.green)
  return;
  nodeRenderer.material.color = Color.green;
  SpawnOrb();
  cooldownTimer = 15f;
}
```

Add the following lines to the `Update` function to subtract the time between frames and reset the dummy when the timer reaches zero:

```
private void Update()
{
  if(cooldownTimer > 0)
  {
    cooldownTimer -= Time.deltaTime;
  }
  else if(nodeRenderer.material.color == Color.green)
  {
    ResetDummy();
  }
}
```

Feel free to drag a **PracticeDummy** into your scene and test it out; every 15 seconds you can hit it with a projectile to make it turn green and dispense one energy orb:



The last thing left to do is enable the player to spend orbs raising and lowering the walls in the level.

Using resources to interact with the environment

For the walls to be toggled on and off, their scripts will have to know their current state. Add the following variable to your `DynamicWall` class:

```
public class DynamicWall : MonoBehaviour
{
  [SerializeField]
  private GameObject wallSection;
  public bool isRaised = true;
}
```

Add a line to the end of `MoveSectionDown` to set `isRaised` to `false` after the wall has been lowered:

```
public IEnumerator MoveSectionDown()
{
  ...
  isRaised = false;
}
```

And an opposite line to set it to `true` at the end of `MoveSectionUp`:

```
public IEnumerator MoveSectionUp()
{
  ...
  isRaised = true;
}
```

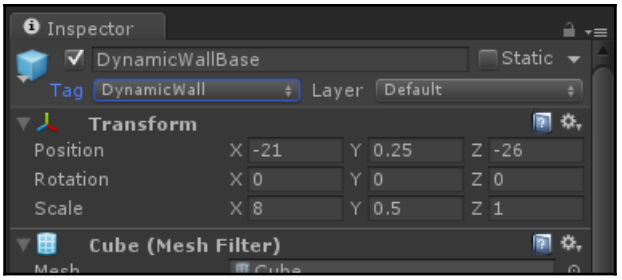
Now we can add a call to both of these functions from the player using the `OrbManager` script. Add a call to a new function called `ToggleWall` in the `Update` function when the *F* key is pressed. Also, define `ToggleWall` as an empty function after the `Update` function for now:

```
void Update()
{
  if(Input.GetKeyDown(KeyCode.F))
  {
    ToggleWall();
  }
}
private void ToggleWall()
{
}
```

We're going to require that the player be looking at the base of a wall in order to toggle it, so add a serialized reference `lookTransform` and drag the **ForwardDirection** child of **OVRPlayerController** onto its field in the **Inspector**:

```
public class OrbManager : MonoBehaviour
{
  private int totalOrbs;
  [SerializeField]
  private Transform lookTransform;
}
```

Now we'll define the `ToggleWall` function with a raycast to determine if the user is in fact looking at a wall. To do that, we'll need to create a tag for our walls, so add a new tag called **DynamicWall** in the **Inspector** and add it to the **DynamicWallBase** prefab definition:



Add the raycast to the `ToggleWall` function and call `MoveSectionUp` or `MoveSectionDown` on the `DynamicWall` component based on its state:

```
private void ToggleWall()
{
  Ray ray = new Ray(lookTransform.position, lookTransform.forward);
  RaycastHit hit;
  if(Physics.Raycast(ray, out hit, Mathf.Infinity))
  {
    if(hit.collider.tag == "DynamicWall")
    {
      DynamicWall wall = hit.collider.GetComponent<DynamicWall>();
      if(wall.isRaised)
        StartCoroutine(wall.MoveSectionDown());
      else
        StartCoroutine(wall.MoveSectionUp());
      totalOrbs--;
    }
  }
}
```

Now we'll add resource cost into the equation. At the beginning of your `ToggleWall` function, add an early return if the player doesn't have any energy orbs:

```
private void ToggleWall()
{
  if(totalOrbs == 0)
  return;
  ...
}
```

In a future chapter, we'll add a visual interface where the player will be able to see how many orbs they have, but for now, we'll still be able to test our new function. Add one practice dummy in each corner building of the map and then press Play. Shoot a practice dummy to collect one or two energy orbs from it, and then look at the base of a wall and press the *F* key. The player's orb count will go down by one and the wall will be toggled.

Summary

In this chapter, we broke out of the monotony of an empty scene and built up a basic but functional environment. We also solved Z-fighting in our imported model, a common rendering bug with faces that are close to each other. By using light baking and Unity's particle system, we were able to make the environment pleasing aesthetically, and we populated it with unique mechanics such as collectible resources and dynamic walls.

Now that we've spent a good deal of time building out our world, we'll be turning our focus to the user interfaces that will add interaction and depth to our scenes.

6
Adding Depth and Intuition to a User Interface

Every game needs a user interface. Whether it's a graphic to display ammunition count in a first-person shooter, an inventory screen in an RPG, or a speedometer in a racing game, dynamic and intuitive interfaces are what hold the complex mechanics of a game together.

In virtual reality, we have the opportunity to extend the user interface beyond the simplicity of a 2D screen and incorporate depth and real-world properties to our interface. In this chapter, we'll be building out a 3D UI for our combat arena game and expanding traditional 2D UI techniques into a better, more immersive format for VR.

This chapter will cover the following topics:

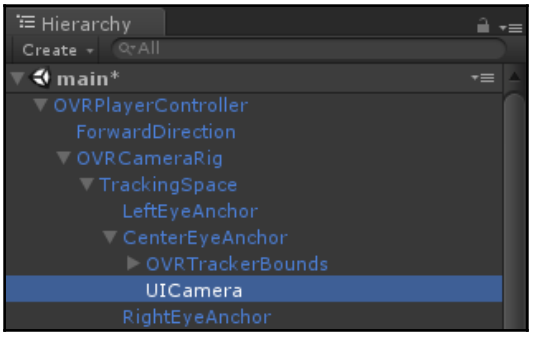
- Adding a VR input module
- Constructing a simple menu
- Creating a GameManager script
- Adding UI elements to the game world

Adding a VR input module

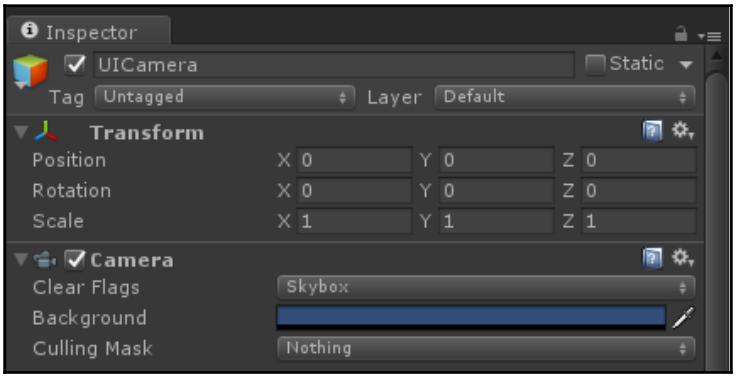
There are plenty of similarities between traditional game development and VR game development, but UI input isn't one of them. Most game input relies on screen bounds to position UI elements and capture mouse coordinates, but in VR, the “screen” is all around you, and the added depth gives you an entire new dimension to work with.

With that in mind, it takes a little extra work to define constraints that allow input to work in VR. In this section, we'll add a new input module to our player that will let them send messages to Unity's UI system.

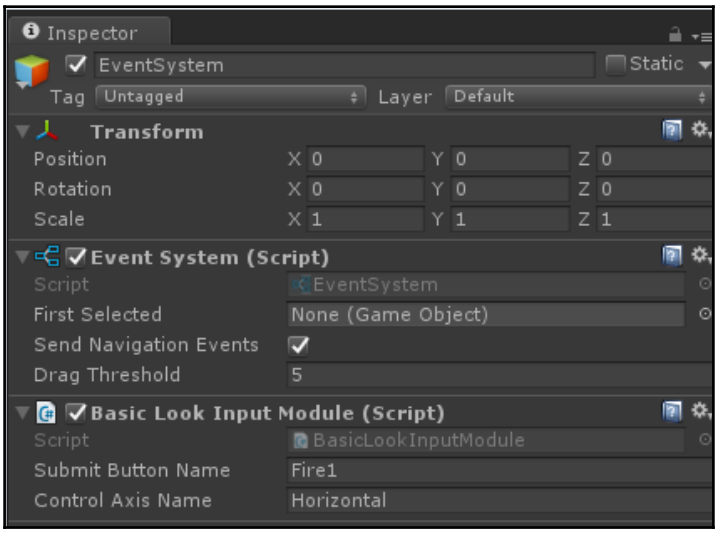
Expand your **OVRPlayerController** object in the hierarchy and find the **CenterEyeAnchor** object inside of the **OVRCameraRig** child. Create a new **Camera** object as a child of **CenterEyeAnchor** and name it `UICamera`:



Highlight the new camera in the **Inspector** panel and set the **Culling Mask** property to **Nothing**. Also, remove the **AudioListener** component from the camera by right-clicking on it and selecting **Remove Component**. Finally, set the **Target Eye** property to **None (Main Display)**; we don't need this camera to be able to render any actual objects, we just need its position and rotation to send messages to Unity's UI system.



Finally, create an **EventSystem** object from the **UI** list in the **GameObject** menu. Highlight it in the **Inspector** panel, right-click on the **Standalone Input Module** component, and click on **RemoveComponent**. In its place, add the **BasicLookInputModule** script (included with this chapter), as shown in the following screenshot:



That's all we need to do for our VR input module for now, but as we add UI elements, we'll need to assign our **UICamera** object to their canvases, which are responsible for containing all UI elements in a group.

Constructing a simple menu

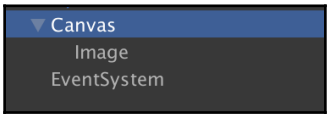
Now that we've got an input module to process input from our HMD, we can start constructing an interactive menu as our first UI element. We'll start simple and then move onto more complex functionality before establishing in-game UI elements in the next section.

Setting up a canvas

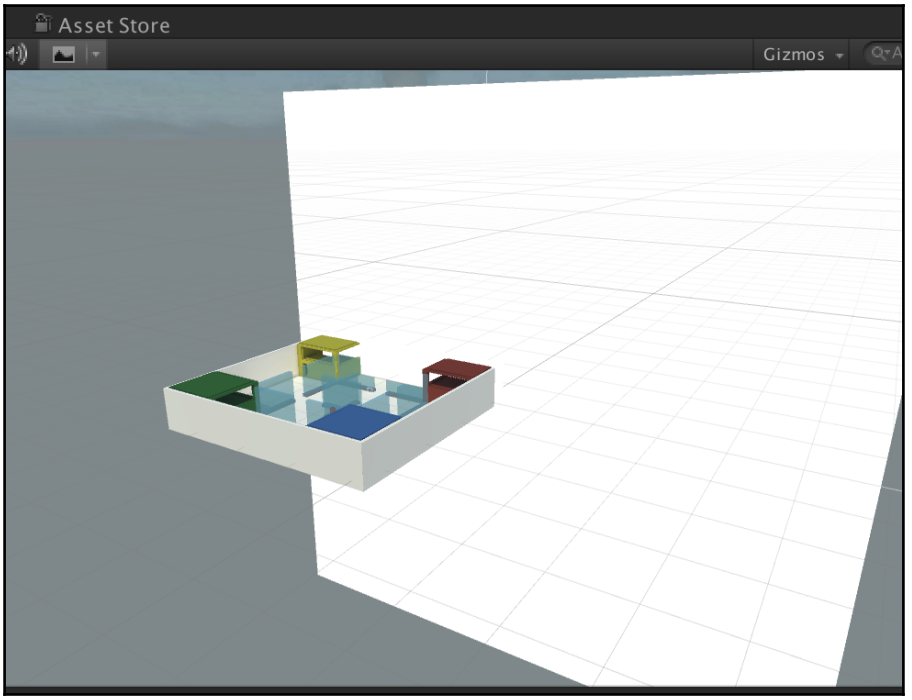
In Unity's UGUI system, every UI element must be a child of a **Canvas** object. A default canvas is created any time a UI element is added to a scene, but you can also add elements directly to an existing canvas and group individual elements together under one system.

In this section, we'll create a canvas to group our main menu elements together and some buttons to start and quit the game.

Open the **Create** menu in your hierarchy and select **Canvas** in the **UI** list. Right-click on your new **Canvas** object and select **Image** under **UI**; this will create an image-based UI element and make it a child of your new canvas. Your new objects in the hierarchy should now look as follows:



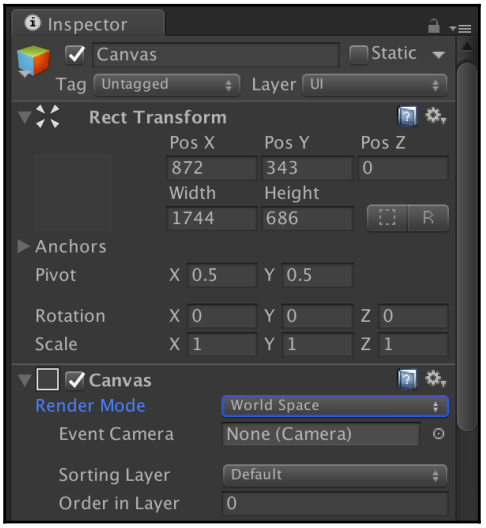
If you center the new canvas in your scene, you'll notice something strange; specifically, the currently blank image element is about twice the size of the level:



If you thought it was a little weird that an **Image** element was that large in the **Scene** view by default, you're not wrong; the current measurement units for the image are in pixels, but in the **Scene** view, everything is represented in meters, so an image with a mere 100 pixels in height or width could fit comfortably over a full-size football field.

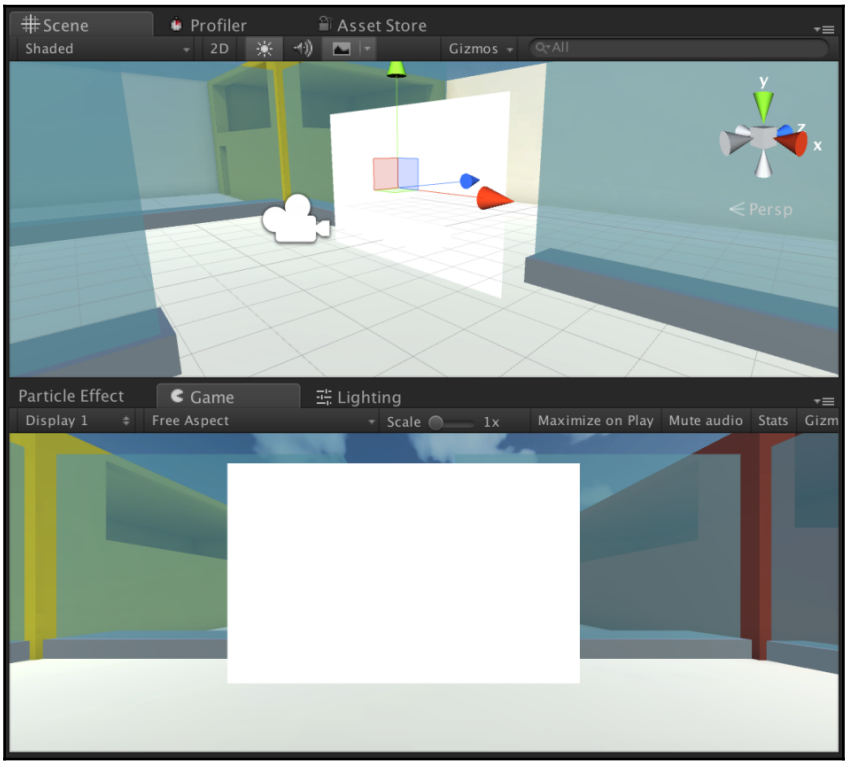
This discrepancy is a matter of **screen space** versus **world space**. Unity UI elements default to drawing on top of the screen after the 3D scene has already been rasterized to the 2D image displayed on your monitor. However, there's not really a conventional “screen” in VR because there are no boundaries or depth limits, so we need to make our UI exist in world space and give it a proper location and orientation in the scene instead.

Left-click on the **Canvas** object once to highlight it in **Inspector**. Scroll down to the **Canvas** component and change the **Render Mode** setting from **Screen Space – Overlay** to **World Space**, as shown in the following screenshot:

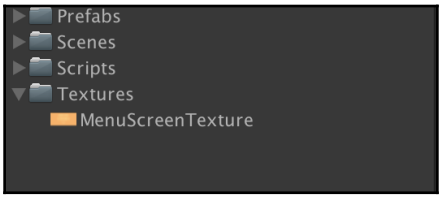


Now you'll see the image displayed the same way in the **Game** view as it is in the **Scene** view; altogether too large, but present in the world, not a screen overlay. Also notice the **Event Camera** field underneath the **Render Mode** property. Drag the **UICamera** we created earlier into this field to make sure it receives messages from our player in VR.

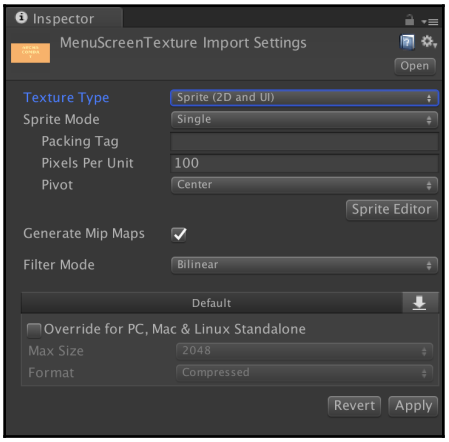
In the **Inspector** panel with the **Canvas** object highlighted, set the **Width** value to `5` and the **Height** value to `4`. Then highlight the child **Image** object and set its dimensions with the same values. Your UI should now appear at a reasonable size in the **Scene** and **Game** views:



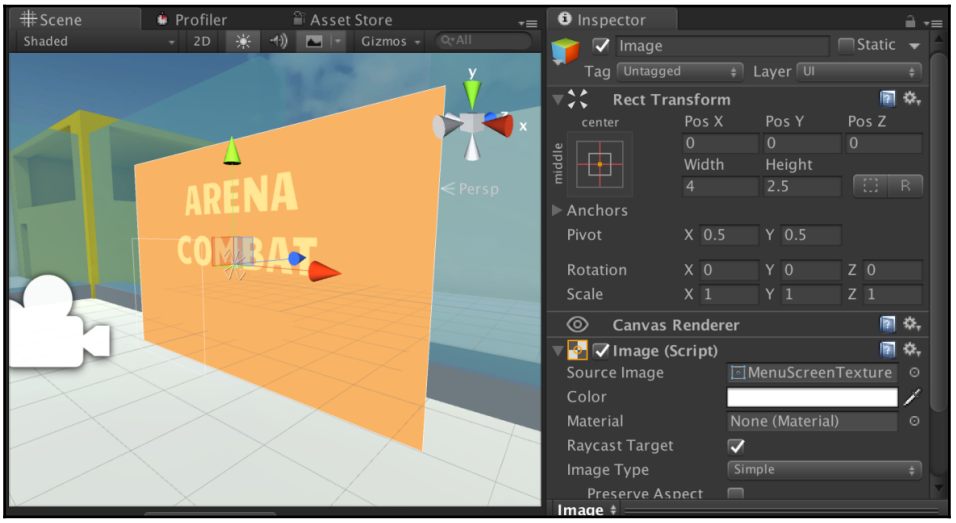
Next, we'll add a texture to our **Image** object to serve as the backdrop for the main menu. Create a new folder called `Textures` in your **Project** window and drag the file named `MenuScreenTexture.png` (included with this book) into the new folder, as shown in the following screenshot:



Before we apply it to our **Image** object, we need to specify it as a **Sprite**, which is the format used for all UI elements in Unity. Left-click on the image in your **Project** window to display its properties in **Inspector** and change the **Texture Type** to **Sprite (2D and UI)**, as shown in the following screenshot:



Click on **Apply** at the bottom-right corner to make sure your new **Import** settings are applied to the texture. To apply the texture to the **Image** object, highlight it in **Inspector** and drag the **MenuScreenTexture** asset from your **Project** window into the **Source Image** field, as shown in the following screenshot:

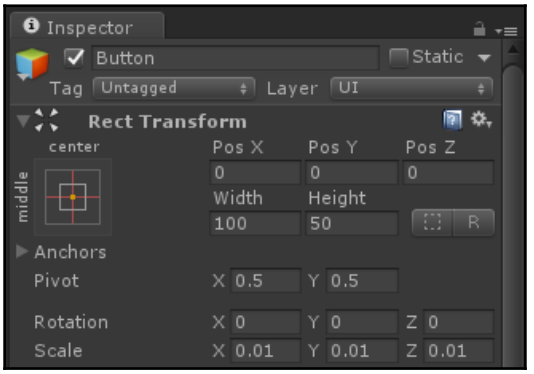


Now that we've got a blank menu screen, let's fill it with some buttons to add functionality.

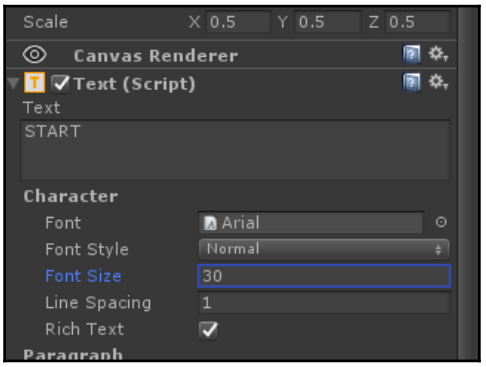
Adding buttons to a canvas

In this section, we'll add a **START** button and a **QUIT** button to our menu. Later in the book when we implement multiplayer, we'll tweak this menu to allow players to select a lobby, but for now just initiating or quitting the game will be enough to get started.

Right-click on your **Canvas** object and select **Button** in the **UI** list. Your button will appear oversized just like your canvas did initially, so set **Width** to `100`, **Height** to `50`, and the **X**, **Y**, and **Z** scale to `0.01`, as shown in the following screenshot:



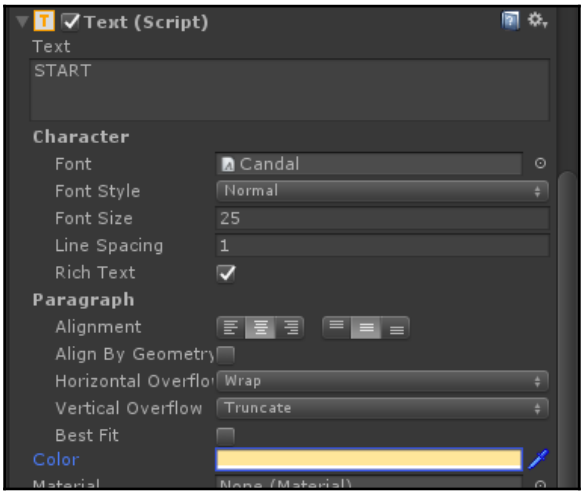
If you left-click on the arrow to the left of the Button object in the hierarchy, you'll notice that it has a **Text** child object; this is where we can label the button. Left-click on the **Text** object and type `START` in the **Text** field within the **Text (Script)** component. Also, set **Font Size** to `30` and the **Scale** values to `0.5` for greater pixel density and better appearance, as shown in the following screenshot:



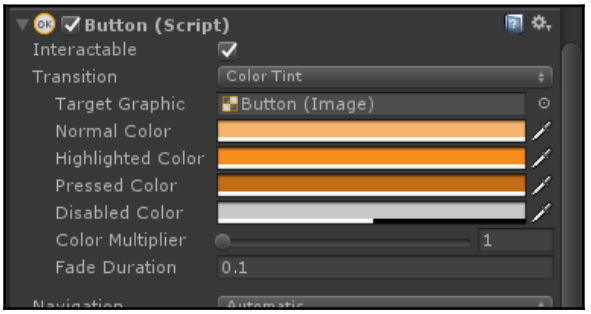
Finally, select the **Button** parent object again and set the **Pos X** value to `-1` and the **Pos Y** value to `-0.75` to position it in the bottom-left of the menu screen. If the text is hard to see, adjust the **Z** position of the **Text** child of the button so that it's in front of the button itself. When you're done, your new button should look like the following on your menu screen:

The button's default appearance doesn't match the menu screen, so we'll customize it to fit in. We'll start off by changing the font. Unity's GUI text uses **Arial** as a default font, but most standard fonts can be imported and used.

Create a new folder in your **Project** window called `Fonts`. Drag the `Candal.ttf` file (included with this book) into your new folder. Then left-click on the **Text** object nested in the **Button** object and drag the **Candal** font to the **Font** field in **Inspector**. Then click on the eye dropper tool to the right of the **Color** property and click anywhere on the menu screen's text to copy the color:



Next, use the **Button** object's color picker to make the **Normal Color** property match the menu screen's background. Make the **Highlighted Color** and **Pressed Color** properties darker versions of this color, as shown in the following screenshot:

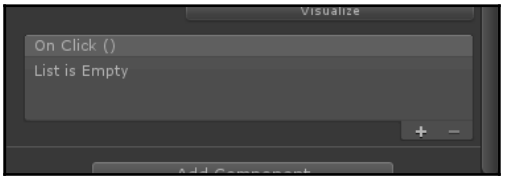


Finally, duplicate the button, set the duplicate's **Pos X** value to `1`, and change the text to read `QUIT`. Your menu screen should now look as follows:

Now that we've got a simple but complete menu, we'll add functionality to the buttons to initialize and exit the game.

Adding functionality to UI elements

All interaction facilitated by UI elements is done through **callback functions** that fire when certain actions are performed. In the case of our **Button** objects, the event is called `OnClick()`. On the **Button (Script)** component of either button in **Inspector**, you'll see a currently empty field where function calls can be attached to this event:



Before we add a callback, though, we'll need to write a script to define it.

Creating a GameManager script

Create a new script called `GameManager`. This script will be responsible for managing the state of the game by initializing the player and level when **START** is selected and shutting down the game when **QUIT** is selected.

We don't want our player to be able to move around or shoot before the game starts, so we'll need to disable these by default and write a function in our new script that enables them after **START** is pressed.

Add two variables to serve as references to the **OVRPlayerController** component and the **FiringSystem** component on the main player:

```
public class GameManager : MonoBehaviour
{
  private OVRPlayerController playerController;
  FiringSystem firingSystem;
  ...
}
```

Next, create a new function called `InitializePlayerReferences`. In this function, we'll get references to the defined components with the `GetComponent` function. We need a reference to the player before we search its components; we'll find the player using Unity's tag system, so highlight the **Player** prefab in the **Inspector**, add a new tag named **Player** from the **Tag** menu in the upper-left, and assign it as the tag for this prefab.

Now define your `InitializePlayerReferences` function as follows:

```
private void InitializePlayerReferences()
{
  GameObject player = GameObject.FindGameObjectWithTag("Player");
  playerController = player.GetComponent<OVRPlayerController>();
  firingSystem = player.GetComponent<FiringSystem>();
}
```

We want to initialize these references as soon as the game starts, so add a call to `InitializePlayerReferences` in the `Start` function as follows:

```
void Start()
{
  InitializePlayerReferences();
}
```

Next, create a function called `StartGame` to be called from the **START** button as follows:

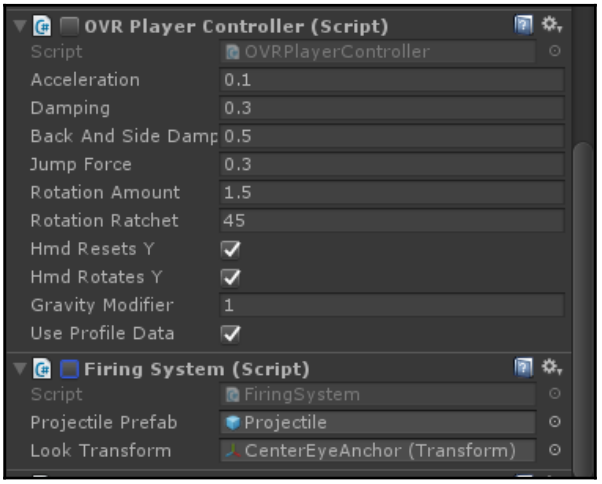
```
public void StartGame()
{
}
```

All functions that you want to call from a UI element need to be declared as `public` so that they can be accessed from the list of callback functions.

Add two lines to the `StartGame` function to enable the `OVRPlayerController` component and the `FiringSystem` component:

```
public void StartGame()
{
  playerController.enabled = true;
  firingSystem.enabled = true;
}
```

These components are currently enabled by default when we press Play, so disable them in **Inspector** so that they're not active until `StartGame` is called:



Next we'll add another line to your `StartGame` function that will disable the menu when the game has been started. First, we'll need a reference to the game object that the menu canvas is on, so define a new serialized `GameObject` variable as follows:

```
public class GameManager : MonoBehaviour
{
  private OVRPlayerController playerController;
  private FiringSystem firingSystem;
  [SerializeField] private GameObject menuObject;
}
```

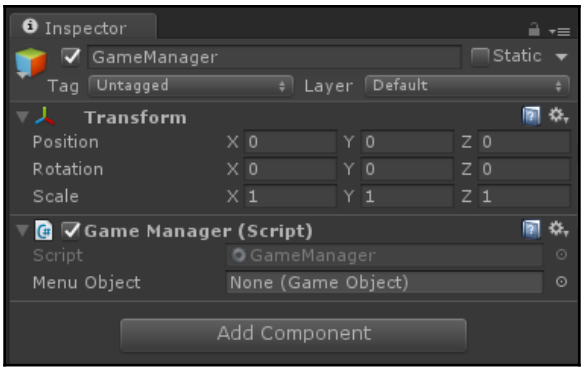
Now add the following line to `StartGame` to make the object inactive:

```
public void StartGame()
{
  playerController.enabled = true;
  firingSystem.enabled = true;
  menuObject.SetActive(false);
}
```

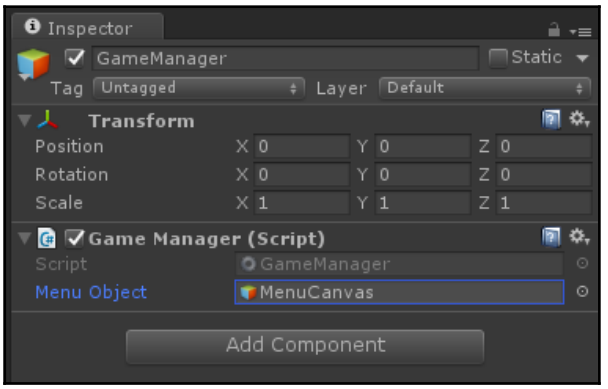
Let's create a function for the **QUIT** button, too. Declare a new function called `QuitGame` and add a line that will exit the application when it's called:

```
public void QuitGame()
{
  Application.Quit();
}
```

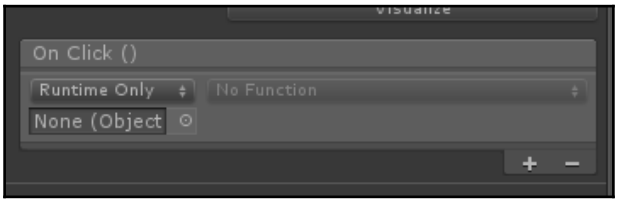
Our `GameManager` class is now ready to be added to the scene. Create a new empty object and name it `GameManager`, and then add a **GameManager (Script)** component to it, as shown in the following screenshot:



Rename the **Canvas** menu object to `MenuCanvas` to make it unique and then drag it into the **Menu Object** field on your **GameManager (Script)** component, as shown in the following screenshot:

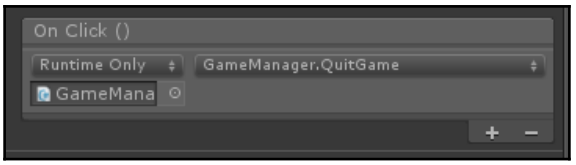


Finally, we'll link the `StartGame` and `QuitGame` functions to their callbacks on the **MenuCanvas** buttons. Highlight the **START** button in **Inspector** and click on the + sign to add a slot for a new function in the **On Click ()** list:



Drag the **GameManager** object into the field labeled **None (Object)** and then select the **StartGame** function from the function drop-down list:

Now do the same with the **On Click ()** list on the **QUIT** button, but select **GameManager.QuitGame** instead:



Now that both of your buttons have been linked to your `GameManager` functions, press Play and select **START** to test out the `StartGame` function. Pressing **QUIT** while running the game within the Unity Editor won't do anything, but if you want to test the `QuitGame` function as well, create a build and run it as a standalone executable.

We'll add much more to this simple menu system later, but now that we've created a basic functional UI menu, let's make some in-game UI elements as well.

Adding UI elements to the game world

A screen-shaped gaze-based menu is the very tip of the iceberg in terms of UI in VR; the lack of any actual screen creates a myriad of opportunities for interfaces directly integrated into the world, limited by just your creativity instead of the pixel count of the player's monitor.

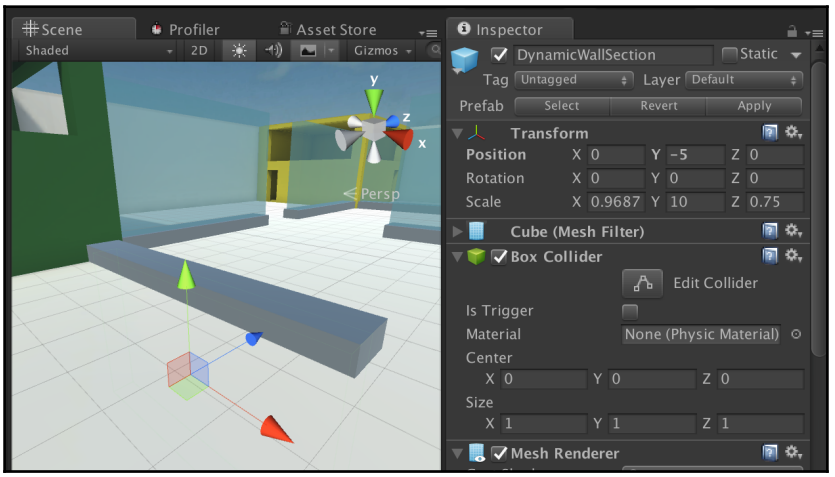
In this section, we'll add some complexity to our dynamic wall mechanic and add some interface elements to them in the process. Specifically, we're going to make the walls expand upward in a unique color based on the player's team and make players able to shoot through walls that match their team's color. We're also going to make the walls collapse when players on other teams lower their health enough, which will be displayed in a dynamic bar on top of them.

Adding complexity to the dynamic walls

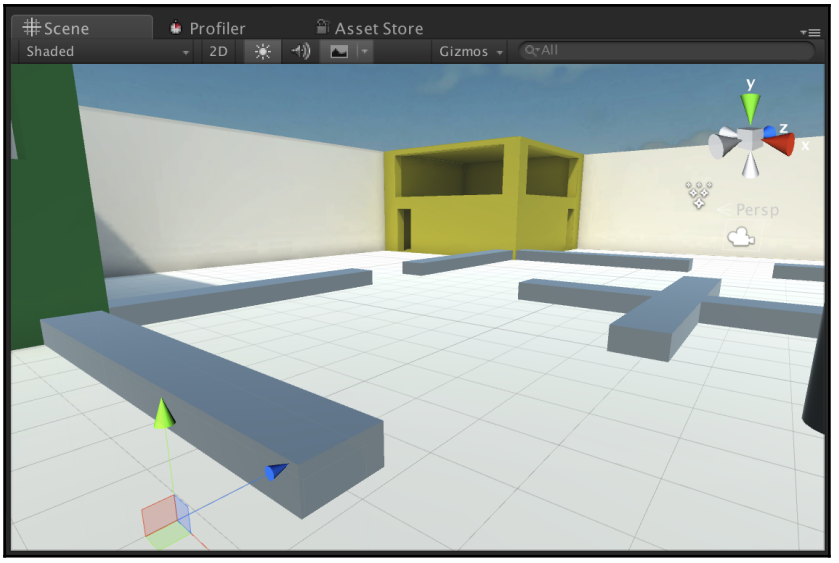
The first thing we'll do is make all walls start down by default. Open your **DynamicWall** script and edit the `isRaised` variable to be initialized as `false`:

```
public class DynamicWall : MonoBehaviour
{
  [SerializeField] private GameObject wallSection;
  public bool isRaised = false;
}
```

Next, click on one of the **DynamicWallSection** objects within an instance of **DynamicWallBase** and move its local **Y** position to `-5`, as shown in the following screenshot:

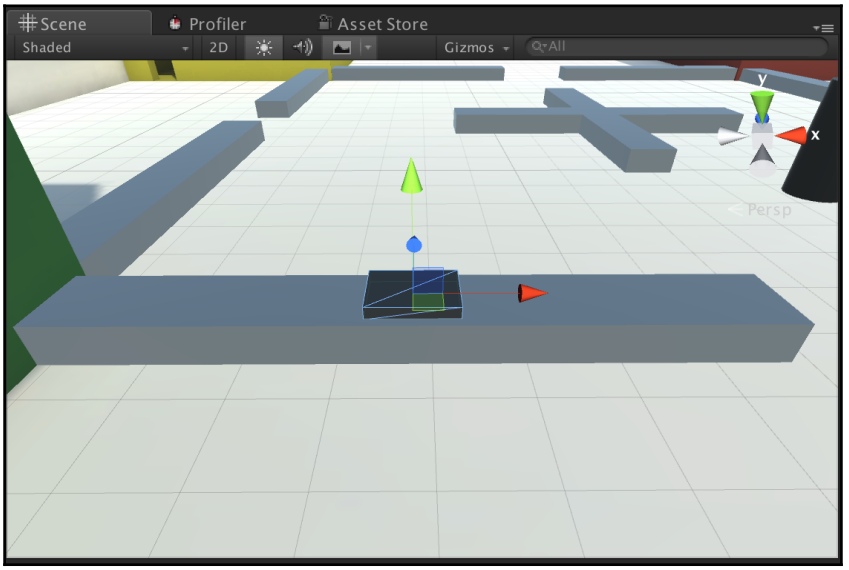


Next, click on the **Apply** button in the upper-right area of the **Inspector** window to apply the new **Y** position to all other instances of **DynamicWallBase**. All of the other walls will then drop to match the one you edited:

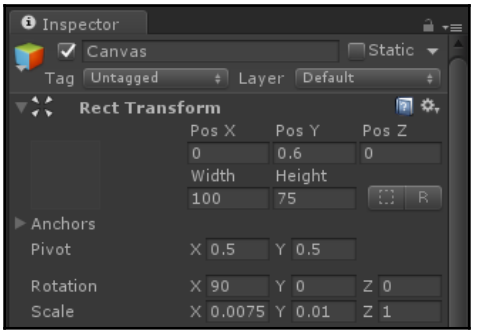


Now that they're down by default, we'll create a UI panel that rests on top of the wall base and displays how many orbs players will need to raise them for their team.

Create a new cube, name it `Counter`, and rescale it to `1`, `0.15`, and `0.75`. Create a standard dark gray material and apply it to the cube. Then, position it directly in the center of the top of the wall base, as shown in the following screenshot:

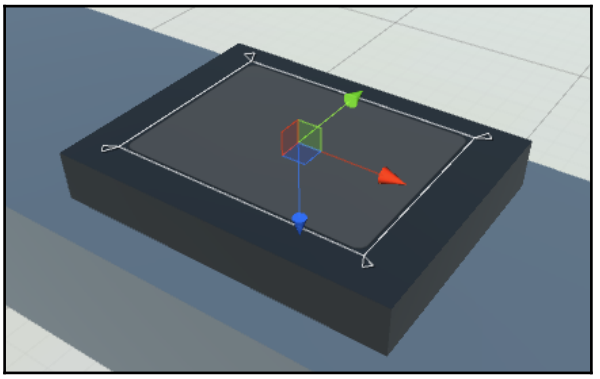


Now create a new **Canvas** object as a child of the **Counter** object, set the **Render Mode** to **World Space**, and configure its **Rect Transform** component, as shown in the following screenshot:

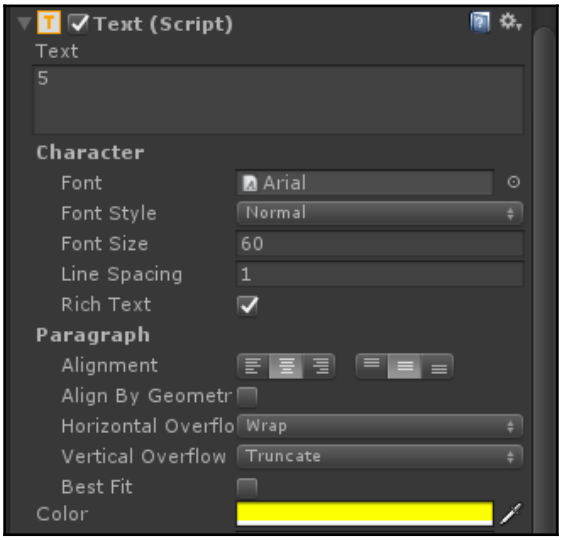


Giving it a large height and width with a small scale ensures that we maintain a high UI resolution, and setting the **X** rotation to `90` will make it lie flat on top of our base object.

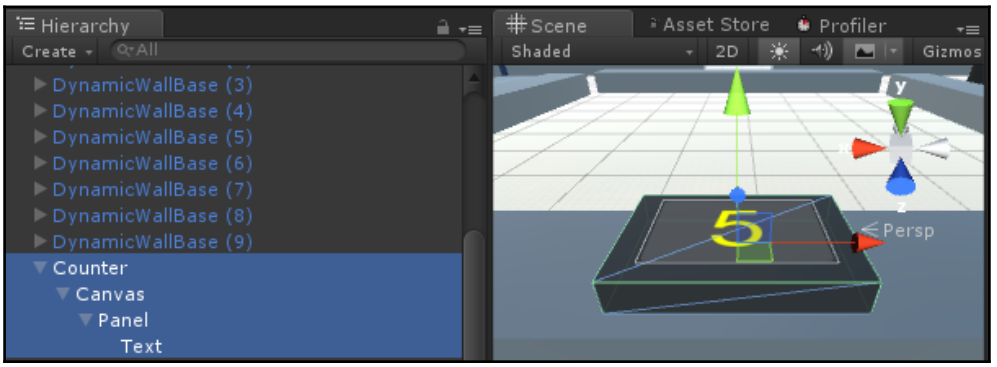
Next, create a **Panel** object within your **Canvas** object. Give it a light gray color and set its local **Pos Z** to `0.01` so that it's slightly above the base, as shown in the following screenshot:



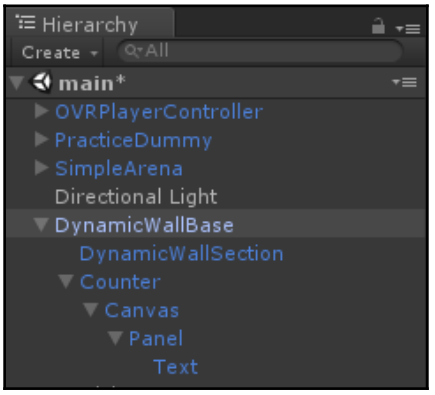
Create a new **Text** object within your **Panel** object, give it a yellow color, and drag the triangular anchors at the corners to expand it until it matches the size of your **Panel**. Configure the values of the **Text (Script)** component to match the following screenshot:



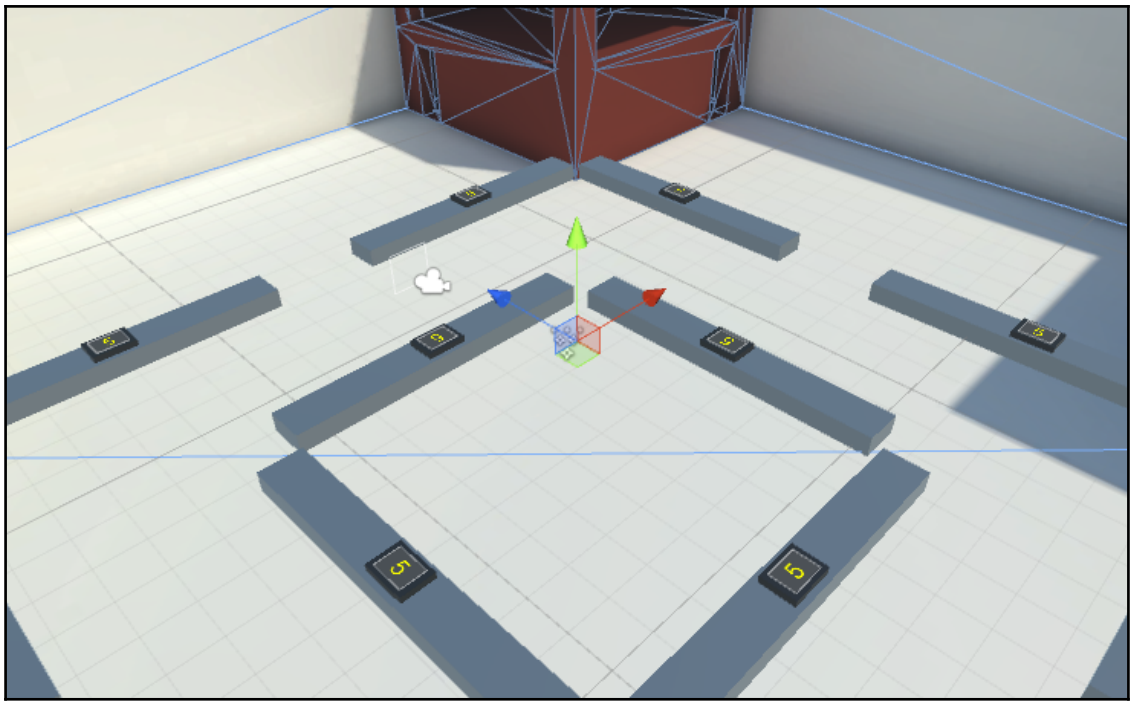
The default text value is **5** because that's how many resource orbs we'll require the player to deposit to activate the wall. Your completed counter should appear like the following screenshot, with the elements nested properly in the hierarchy:



Drag the **Counter** object onto the **DynamicWallBase** object that you positioned it over to add it as another child, and then click on **Apply** in the **Inspector** panel with the **DynamicWallBase** highlighted to add the counter as a part of the prefab definition, as shown in the following screenshot. The text in the hierarchy will turn blue to denote that the objects are part of a prefab:



The counter object will now appear on all of your wall instances, but in the case of the center crossed walls, the counters will overlap each other. Change the configuration from a cross to a box shape, as shown in the following screenshot:



Our counter object is complete and part of our wall prefab now, so next we'll build out the `DynamicWall` class to make use of its new component.

Open the `DynamicWall` script and declare a new function called `DepositOrbs` with an `int` parameter for the number of orbs being deposited:

```
public void DepositOrbs(int numOrbs)
{
}
```

Now declare two new variables, one to store the total cost of the wall and one to store the remaining cost:

```
public class DynamicWall : MonoBehaviour
{
  [SerializeField] private GameObject wallSection;
  public bool isRaised = true;
  private int totalCost = 5;
  public int remainingCost = 5;
  ...
}
```

We'll be accessing the `remainingCost` variable from other classes, so it needs to be declared as `public`, but we can keep `totalCost` declared as `private` because we'll only need to use it in the `DynamicWall` class.

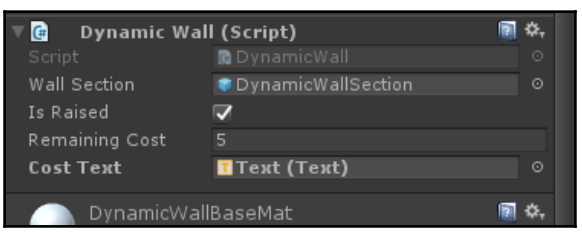
Add the following logic to your `DepositOrbs` function to subtract the passed in orbs from the remaining cost and call `MoveSectionUp` if the cost becomes 0:

```
public void DepositOrbs(int numOrbs)
{
  remainingCost -= numOrbs;
  if(remainingCost <= 0)
  {
    StartCoroutine(MoveSectionUp());
  }
}
```

Next, we'll link the UI text component to this script so that we can update it whenever the remaining cost changes. Add a new using line to include Unity's UI namespace and declare a new variable to store a reference to the **Text** object on our counter's canvas:

```
using UnityEngine.UI;
public class DynamicWall : MonoBehaviour
{
  [SerializeField] private GameObject wallSection;
  public bool isRaised = true;
  private int totalCost = 5;
  public int remainingCost = 5;
  [SerializeField] private Text costText;
  ...
}
```

Click on an instance of **DynamicWallBase** in the hierarchy to display it in the **Inspector** and drag that instance's **Text** child onto the new **Cost Text** field, as shown in the following screenshot:



Click on **Apply** in the **Inspector** to apply this change to all other instances of **DynamicWallBase**.

Declare a new function in the `DynamicWall` class called `UpdateText`:

```
public class DynamicWall : MonoBehaviour
{
  ...
  private void UpdateText()
  {
  }
}
```

In this function, assign the value of `remainingCost` to the `Text` reference so that the displayed number always matches the internal variable:

```
private void UpdateText()
{
  costText.text = remainingCost.ToString();
}
```

Add a call to your `DepositOrbs` function to update the text whenever orbs are deposited:

```
public void DepositOrbs(int numOrbs)
{
  remainingCost -= numOrbs;
  if(remainingCost <= 0)
  {
    StartCoroutine(MoveSectionUp());
  }
  UpdateText();
}
```

Next, we'll change how the wall is interacted with on the player's end so that it includes this new function. Open the `OrbManager` script and edit the name of the `ToggleWall` function to `ActivateWall`, as we'll be using a different function to bring it back down after it's been activated. You'll need to change the declaration and the call to it in the `Update` function:

```
private void ActivateWall()
{
  ...
}
```

In the `ActivateWall` function, take out all of the lines inside the `if` check for the `DynamicWall` tag except for the one that gets the `DynamicWall` component. It should now look as follows:

```
private void ActivateWall()
{
  if(totalOrbs == 0)
```

```
      return;
    Ray ray = new Ray(lookTransform.position, lookTransform.forward);
    RaycastHit hit;
    if(Physics.Raycast(ray, out hit, Mathf.Infinity))
    {
      if(hit.collider.tag == "DynamicWall")
      {
        DynamicWall wall = hit.collider.GetComponent<DynamicWall>();
      }
    }
  }
```

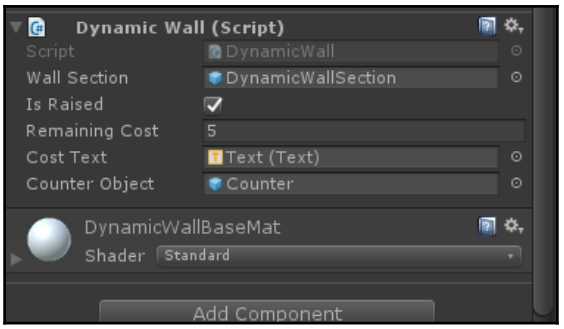
Add a call to the `DepositOrbs` function and pass in as many orbs as the player can, but not more than the total remaining amount:

```
  private void ActivateWall()
  {
    ...
    if(Physics.Raycast(ray, out hit, Mathf.Infinity))
    {
      if(hit.collider.tag == "DynamicWall")
      {
        DynamicWall wall = hit.collider.GetComponent<DynamicWall>();
        int orbsToDeposit = Mathf.Min(totalOrbs, wall.remainingCost);
        wall.DepositOrbs(orbsToDeposit);
        totalOrbs -= orbsToDeposit;
      }
    }
  }
```

Finally, we need to make sure that the counter is hidden when the wall activates. Create a new serialized variable to store a reference to the `Counter` object as follows:

```
  public class DynamicWall : MonoBehaviour
  {
    ...
    [SerializeField] private GameObject counterObject;
  }
```

Select an instance of **DynamicWallBase** in **Inspector** and drag the **Counter** object into the **Counter Object** field, and then click on **Apply** to apply it to all instances, as shown in the following screenshot:

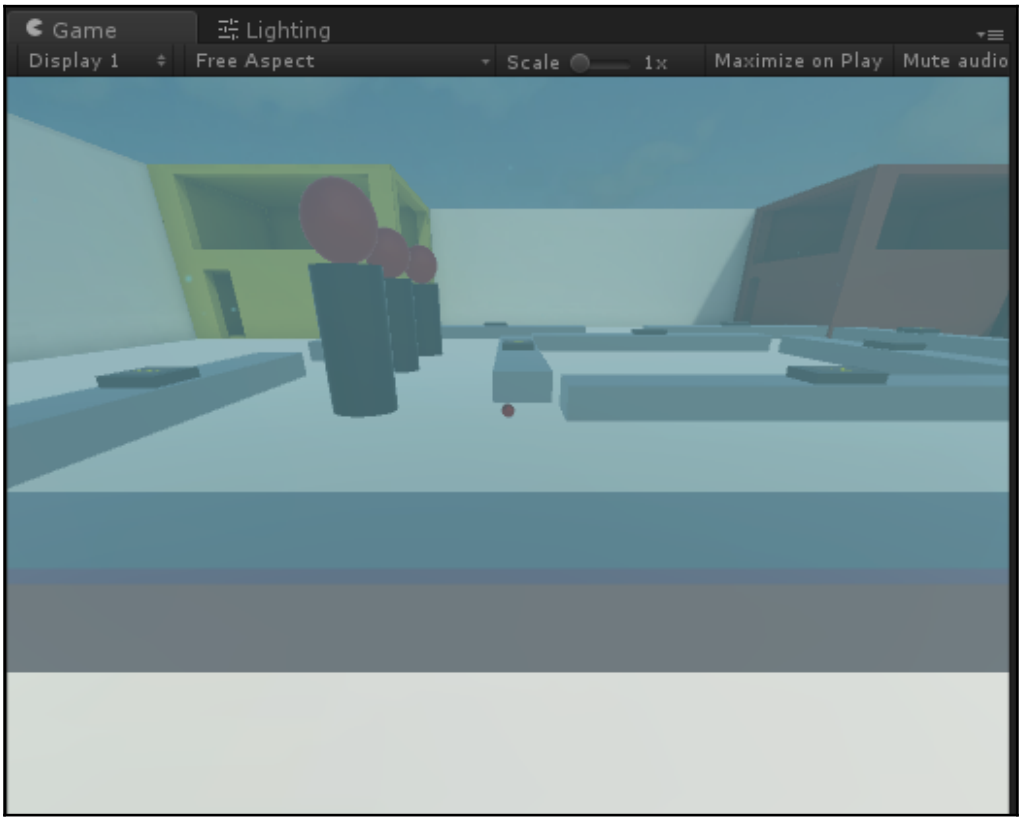


Add a line to the beginning of your `MoveSectionUp` function to hide the counter (we'll add a line to show it again later when we add the ability to move walls down) as follows:

```
public Ienumerator MoveSectionUp()
{
  counterObject.SetActive(false);
  ...
}
```

Now's a good time to test your feature and make sure you can successfully deposit energy orbs into the wall to make it activate. Place a few **PracticeDummy** instances around the scene, shoot them to collect some orbs, and then use the *F* key (or the Xbox controller) to activate whatever wall you're looking at:

After you deposit five orbs, you should see the counter disappear and the wall section come up:



Well done bringing your first simple dynamic UI element to the game world! In the next section, we'll introduce a team dynamic to the game and create another UI element, this time with an animation instead of a text display.

Adding teams to the game

We're going to make our walls activate according to the team of the player that activates them, so first we'll have to define the teams and assign one to our player. Since we already have four uniquely colored buildings in our map, let's have those four colors represent the teams.

Create a new script and name it `PlayerIdentity`. Open the script in your editor and create a new `enum` outside of the actual class definition to represent the four teams and then a variable inside the class to store the player's assigned team:

```
public enum Team
{
  Red,
  Blue,
  Yellow,
  Green
}
public class PlayerIdentity : MonoBehaviour
{
  public Team playerTeam;
  ...
}
```

We'll create a function to assign teams dynamically when we add handling for multiple players in a future chapter, but for our tests right now, let's assign the player to the `Green` team in the `Start` function:

```
void Start()
{
  playerTeam = Team.Green;
}
```

The `DynamicWall` instance will need to know what team the player activating it is on, so add an additional parameter to your `DepositOrbs` function of the `Team` type as follows:

```
public void DepositOrbs(int numOrbs, Team playerTeam )
{
  ...
}
```

Next, add a function called `SetSectionTeamColor` that you pass the same parameter to:

```
private void SetSectionTeamColor(Team playerTeam)
{
}
```

Add a `switch` statement to the function to assign the `wallSection` reference a color based on the player's team:

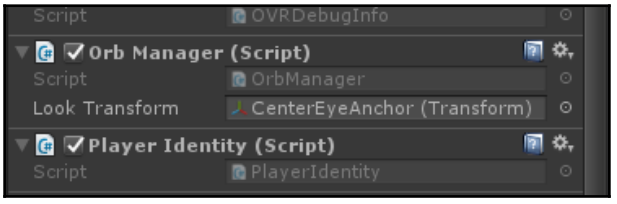
```
private void SetSectionTeamColor(Team playerTeam)
{
  Color teamColor = Color.black;
  switch(playerTeam)
```

```
    {
      case Team.Red: teamColor = new Color(1,0,0,0.5f);
      break;
      case Team.Blue: teamColor = new Color(0,0,1,0.5f);
      break;
      case Team.Yellow: teamColor = new Color(1,1,0,0.5f);
      break;
      case Team.Green: teamColor = new Color(0,0,1,0.5f);
      break;
    }
    wallSection.GetComponent<Renderer>().material.color = teamColor;
  }
```

Add a call to your `DepositOrbs` function to `SetSectionTeamColor` right before the section is moved up:

```
public void DepositOrbs(int numOrbs, Team playerTeam)
{
  remainingCost -= numOrbs;
  if(remainingCost <= 0)
  {
    SetSectionTeamColor(playerTeam);
    StartCoroutine(MoveSectionUp());
  }
  UpdateText();
}
```

Since we modified the `DepositOrbs` function parameters, we have to update the call to it in `OrbManager`. To get a reference to the `PlayerIdentity` class, we'll need it on our player prefab, so drag the **PlayerIdentity** script from your **Project** window onto the **OVRPlayerController** object to add it as an additional component, as shown in the following screenshot:

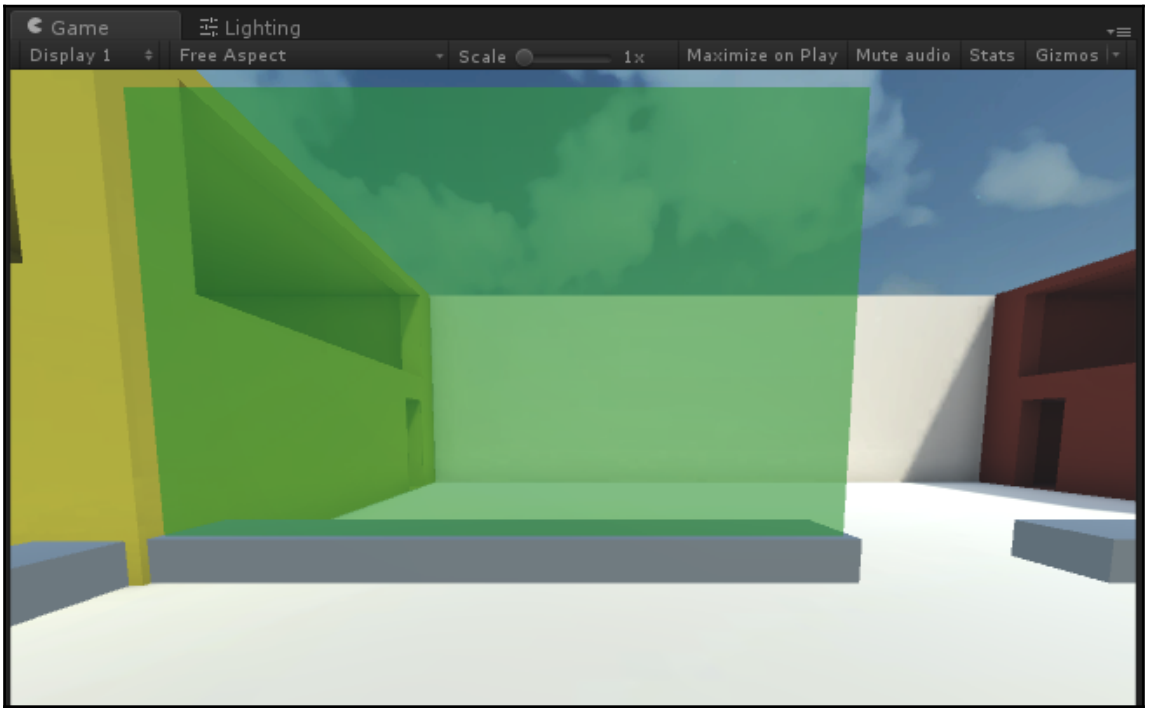


Open the `OrbManager` script and edit the call to `DepositOrbs` within the `ActivateWall` function to get a reference to the `PlayerIdentity` component and pass it in the stored team:

```
private void ActivateWall()
{
```

```
    ...
    if(hit.collider.tag == "DynamicWall")
    {
      DynamicWall wall = hit.collider.GetComponent<DynamicWall>();
      int orbsToDeposit = Mathf.Min(totalOrbs, wall.remainingCost);
      wall.DepositOrbs(orbsToDeposit,
      GetComponent<Playeridentity>().playerTeam);
      totalOrbs -= orbsToDeposit;
    }
    ...
  }
```

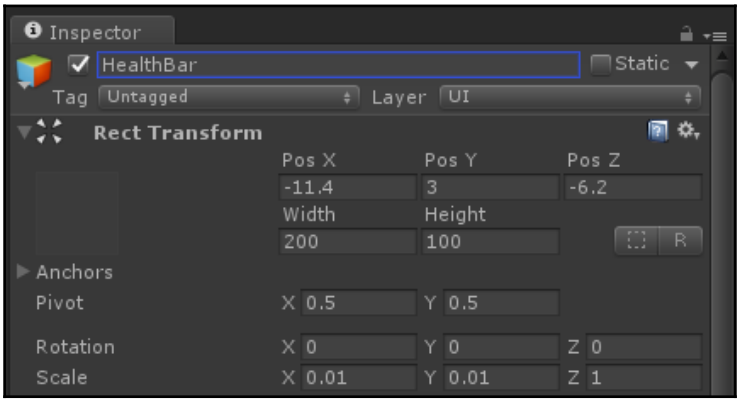
Now test your feature again. When you activate a wall, the section that comes up should be green to reflect your player's assigned team color:



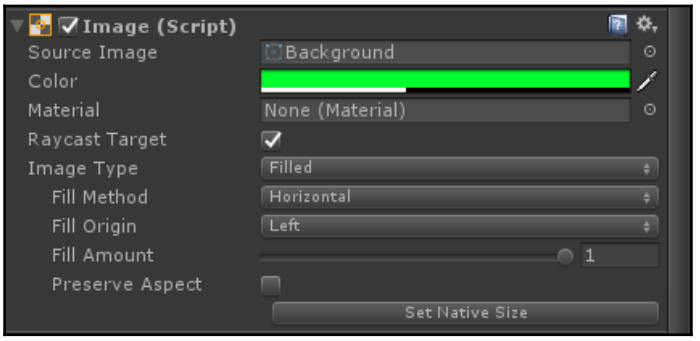
Adding a health bar to the dynamic wall

Now that walls can be activated for the benefit of a team, it makes sense to create a way for other teams to bring those walls back down. In this section, we'll be adding a health bar to the wall that shrinks every time it's hit with a bullet until eventually it goes back down.

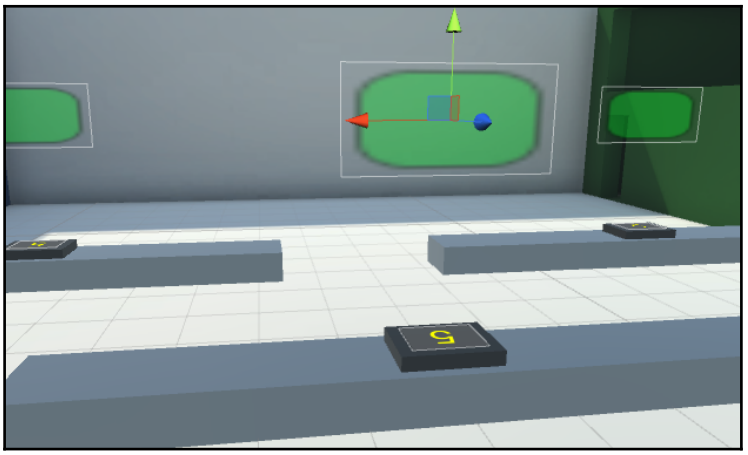
Create a new **Canvas** object and name it `HealthBar`, and set the **Canvas Mode** to **World Space**. Position it right in the middle of where the wall section will be when it's up. Configure the transform like the following (note that your position values may be different depending on which wall you're positioning it above, but the rest of the values should match exactly):



Create a new **Panel** object nested within your **HealthBar** canvas; you won't need to edit the **Rect Transform** component on this one. Instead, find the **Image (Script)** component on the **Panel** object and change the **Color** property to green. Change the **Image Type** to **Filled** and the **Fill Method** to **Horizontal**, which will make this panel able to shrink or grow based on a value between 0 and 1:



Make this **HealthBar** object yet another child of a **DynamicWallBase** object and apply the change to the prefab. Now every wall should have a visible health bar:



Next we'll add some functionality to these health bars in code. First, we'll need a reference to the panel itself. Create a new serialized reference in the `DynamicWall` class and two `int` variables to store total and remaining health:

```
public class DynamicWall : MonoBehaviour
{
  [SerializeField] private GameObject wallSection;
  public bool isRaised = true;
  private int totalCost = 5;
  public int remainingCost = 5;
  [SerializeField] private Text costText;
  [SerializeField] private GameObject counterObject;
  [SerializeField] private Image healthBar;
  private int totalHealth = 10;
  private int remainingHealth = 10;
  ...
}
```

Attach a reference to the new `healthBar` variable by dragging the new child of the wall object into the field labeled **Health Bar** in **Inspector**.

Now capture collisions on your wall by defining `OnCollisionEnter` as follows:

```
private void OnCollisionEnter(Collision collisionInfo)
{
}
```

Check if the colliding object is a projectile, and if it is, reduce the remaining health by one. Also, set the fill amount of the health bar to match the percentage of health remaining:

```
private void OnCollisionEnter(Collision collisionInfo)
{
  if(collisionInfo.collider.tag == "Projectile")
  {
    remainingHealth--;
  }
}
```

Check if the health has reached 0. If it has, move the wall section down and reset the counter, or else update the health bar:

```
private void OnCollisionEnter(Collision collisionInfo)
{
  if(collisionInfo.collider.tag == "Projectile")
  {
    remainingHealth--;
    if(remainingHealth <= 0)
    {
      healthBar.fillAmount = 0;
      counterObject.SetActive(true);
      remainingCost = totalCost;
      UpdateText();
      StartCoroutine(MoveSectionDown());
    }
    else
    {
      healthBar.fillAmount = (float)remainingHealth / totalHealth;
    }
  }
}
```

Now we just need to refill the health bar any time a wall is put back up. Add the following lines to your `DepositOrbs` function before the section is moved up:

```
public void DepositOrbs(int numOrbs, Team playerTeam)
{
  remainingCost -= numOrbs;
  if(remainingCost <= 0)
  {
    remainingHealth = totalHealth;
    healthBar.fillAmount = 1;
    SetSectionTeamColor(playerTeam);
    StartCoroutine(MoveSectionUp());
  }
  UpdateText();
}
```

That's it! You should now be able to raise walls with collected energy orbs and bring them back down by shooting them enough times to deplete your health bar.

Summary

This chapter covered the basics of Unity's UI system as it applies to virtual reality games and experiences. Right away we dove into the difficulties, particularly the need to use a custom input module to send messages to Unity's canvas-based UI system.

We were then able to explore the various uses of VR interfaces in both standard square menus and on dynamic objects places throughout the scene. In doing so, we added more complexity to our game mechanics.

In the next chapter, we'll be adding an entirely new feature to our game: audio. Using the Oculus audio tools and the tools built into Unity, you'll add sound effects with realistic spatialization that make the experience even more immersive.

7
Hearing and Believing with 3D Audio

Experiencing something is so much more than seeing it. We've been focusing on visual elements of VR, but for a truly immersive experience, we need to engage as many of the player's senses as we can, and this applies particularly to hearing.

A lot of the information we process based on our hearing is subconscious; for instance, if sound waves hit the back of our ears, what we hear is muffled by their conical shape. This muffling tells our brain that the sound is originating from behind us. If the sound waves bounce off our shoulders before entering our ears, that tells our brain that the sound is above us.

In typical screen-based games, this level of audio realism isn't possible, because the lack of head tracking makes our ears stay in the same static spot. However, with HMDs like the Oculus Rift, we can calculate the relative position and rotation of each ear, meaning the potential granularity of a realistic audio system is much greater. In this chapter, we'll develop an audio system for our Arena Combat game, starting with the simplest traditional implementations and moving into complex head-based audio that adds a new dimension to the sense of scale and space of virtual reality.

This chapter will cover the following topics:

- The science of how we hear
- Learning the basics of Unity audio
- Implementing 2D stereo audio
- Adding basic 3D spatialization
- Immersing your player with HRTFS
- Filling the game world with sound

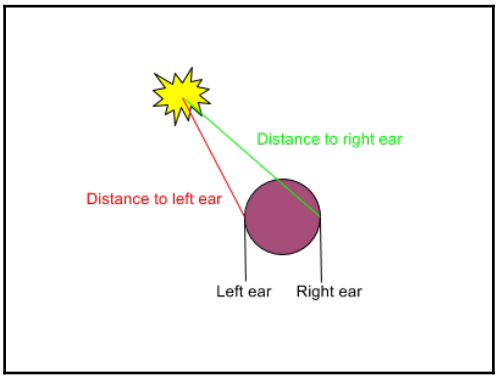
The science of how we hear

To understand realistic game audio, you first need to understand real-world audio. You may not think about your sense of hearing since you've been using it since you were born, but there are several subconscious assumptions being made by your brain as it receives audio signals.

Lateral localization of audio

Imagine trying to pinpoint the location of something moving in a perfect circle around you, like the rings around Saturn. Do you know what lets your brain estimate the precise location of an object just based on what each individual ear hears? You might be surprised to learn that it depends on whether the frequency is high or low.

When listening to low frequencies, our brains detect extremely small delays in the time it takes the sound to reach each ear and bases its directional estimate on that. This basic principle is shown in the following top-down diagram, where you can see the distance the sound needs to travel to the right ear is greater than the distance it needs to travel to reach the left ear:



However, when it comes to high frequencies, we listen for a difference in the level, or volume of the sound in each ear. **Attenuation** (or how audio falls off as it gets further away) is a great way for our brain to figure out which ear is closer to the sound, and thereby guess where the sound is coming from.

Vertical localization of audio

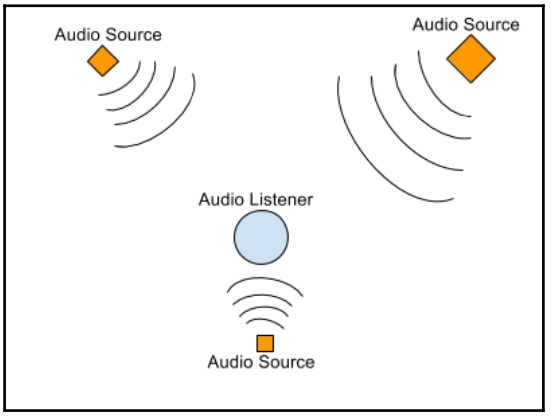
Have you ever cocked your head slightly to figure out if a sound was above or below you? This action can be completely subconscious, but putting one ear even slightly higher up than the other helps us determine which ear is receiving the sound first (think lateral localization, but with your head tilted so that its lateral axis is more closely aligned with the global vertical axis).

With the ability to track the player's head in virtual reality, we can replicate each method of localization and make audio sound more real than it ever has been in games before. Before we do that, though, we've got to cover the basics.

Learning the basics of Unity audio

At the most basic level, Unity's audio system functions with two components: an **audio source** and an **audio listener**. An audio source stores a sound file (known as a clip) and emits a sound at its location in space; the audio listener detects this event and plays the clip through the player's speakers or headphones based on the listener's distance from the source.

The following top-down diagram demonstrates this basic principle:



There should never be more than one audio listener per scene (that would be like having two pairs of ears) but you can spread audio sources all throughout the scene, or add them to existing objects, to create a multi-faceted soundscape for your game's environment.

By default, every **Camera** object created in Unity comes with an **Audio Listener** component. That means if you're using an extra camera for something (like our UI raycast implementation) it's important that an **Audio Listener** component only exists on the camera responsible for rendering what the user sees.

Spatial blending between 2D and 3D

On every **Audio Source** component, there is a property called **Spatial Blend** that lets you specify how much the sound from that source is modified by the spatial properties. With a value of **0**, the sound will be completely two-dimensional; in other words, it will be piped directly to the player's headphones or speakers at a static volume regardless of its position in space relative to the player.

As you might have guessed, with a **Spatial Blend** value of **1**, the audio is completely dependent on physical space and distance from the user. If the distance between the player and the source is greater than the **Max Distance** property on the **Audio Source**, they won't hear the effect at all.

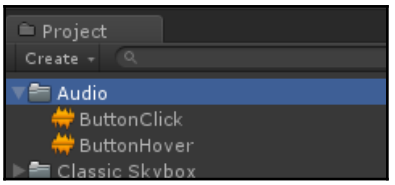
Sometimes, a value somewhere between 0 and 1 is preferable, and sometimes that value will change dynamically at runtime. Imagine a scenario in which you're talking to another player or NPC. You want to be able to hear where their voice is coming from when they're in your vicinity, but if they move to another room, you want to be able to hear them clearly, so you can pan the blend towards the 2D side to get a walkie-talkie effect.

In the next section, we'll implement some basic audio in our game, starting with 2D audio (**Spatial Blend** at 0), and later we'll explore the capabilities of 3D spatialized audio.

Implementing 2D stereo audio

We'll begin by implementing the simplest form of game audio: two-dimensional stereophonic audio. We'll start with menu sound effects and then move onto music and classic ambient soundscapes.

The first thing we need to do is import some audio files into the project. Find the `ButtonClick.wav` and the `ButtonHover.wav` files included with this chapter and drag them into a new folder in your **Project** window called `Audio`. You'll know that they're valid audio files from their yellow waveform icon:



Files with the `.wav` extension are one of the many supported audio file types in Unity. You can also import `.mp3`, `.ogg`, `.aif`, `.mod`, `.it`, `.s3m`, and `.xm`.

Next, we'll create a script that we can use to play these sounds when either of the buttons on the main menu are selected.

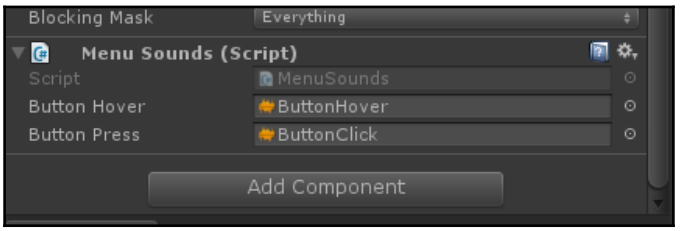
Playing a sound when a button is clicked on

Find the **MenuCanvas** object in your hierarchy and attach a new script named `MenuSounds`. Since both our **START** and **QUIT** buttons will make sounds, we're putting this script on the root canvas and we'll have both buttons point to their parent to call the audio functions.

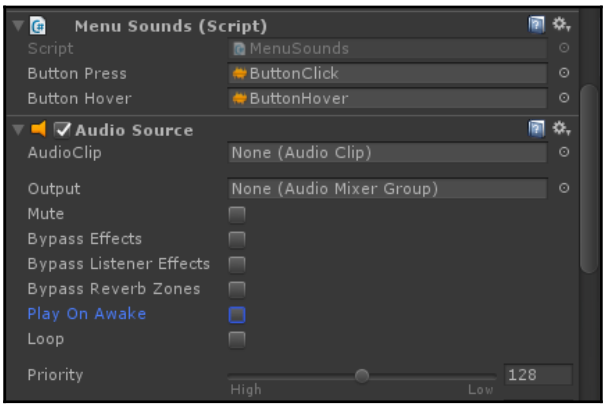
Add two new `AudioClip` variables, one to store a sound to play when the button is hovered over and one to play when the button is clicked:

```
public class MenuSounds : MonoBehaviour
{
   [SerializeField] private AudioClip buttonPress;
   [SerializeField] private AudioClip buttonHover;
}
```

Now link the imported audio files to these two variables by dragging them onto their respective fields on the **MenuSounds** component of **MenuCanvas**:



We'll also need an **Audio Source** component to play these clips. Click on **Add Component** and select **Audio Source**. Disable the **Play On Awake** property, which is enabled by default:



Play On Awake is best suited for objects that are expected to emit a sound as soon as they're instantiated, such as explosions or waterfalls.

Now define an `AudioSource` variable to store a reference to this component, an `Awake` function, and a new function in your `MenuSounds` class called `PlayPressSound`:

```
public class MenuSounds : MonoBehaviour
{
  [SerializeField] private AudioClip buttonPress;
  [SerializeField] private AudioClip buttonHover;
  private AudioSource audioSource;

  private void Awake()
  {

  }

  public void PlayPressSound()
  {

  }
}
```

Because the `Awake` function will run automatically as soon as the script is active in the scene, it's a good place to get a reference to the **Audio Source** component. Add the following line to your `Awake` function:

```
private void Awake()
{
   audioSource = GetComponent<AudioSource>();
}
```

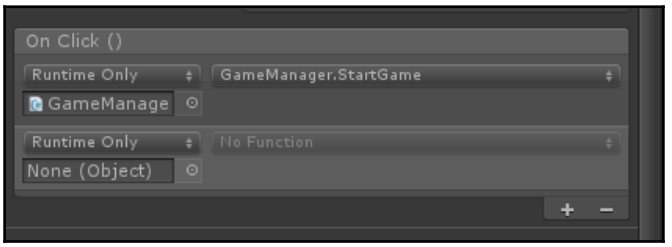
To reiterate a shortcut from a previous chapter, putting nothing in front of the `GetComponent` call means it will implicitly search the same game object that this script is attached to. This works in this case because our **Audio Source** component is on the same game object as our **MenuSounds** script.

Now, you can assign the clip to the source and play it in the `PlayPressSound` function:

```
public void PlayPressSound()
{
   audioSource.clip = buttonPress;
   audioSource.Play();
}
```

An **Audio Source** component can only hold one clip at a time, so with any source that is responsible for playing different clips regularly, it's important to make sure the clip we want to play is loaded before we call `Play`.

Highlight the **START** button in the **Inspector** window and scroll down to the **Button (Script)** component where your `StartGame` callback is. Click the + symbol to add a new callback:



Drag the **MenuCanvas** object into the field that reads **None (Object)** and then select **MenuSounds.PlayPressSound** from the menu on the right:

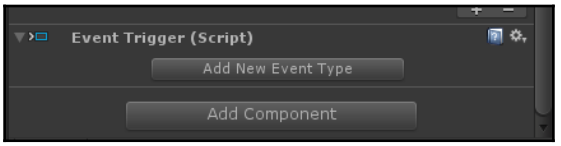
Repeat this process for the **QUIT** button so they both have the same **PlayPressSound** callback. Then, listen to your new feature by running your game and clicking one of the buttons on the main menu. Note that if you still have a call to `GameManager.StartGame` running on the first frame, this won't work because it instantly deactivates the menu (and therefore the audio source). Remove any calls to `StartGame` before testing.

Playing a sound when a button is hovered

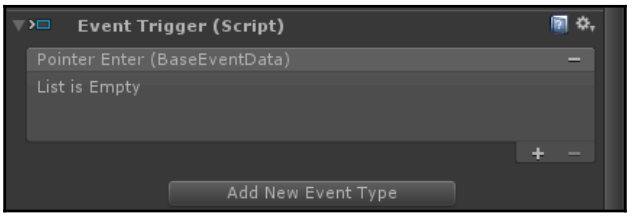
Playing a sound when a button is hovered will be slightly more complex, just because instead of using an existing callback list like **On Click ()**, we're going to add a new one to the object with an **Event Trigger** component. In the `MenuSounds` script, declare a new empty function called `PlayHoverSound:`

```
public void PlayHoverSound()
{
}
```

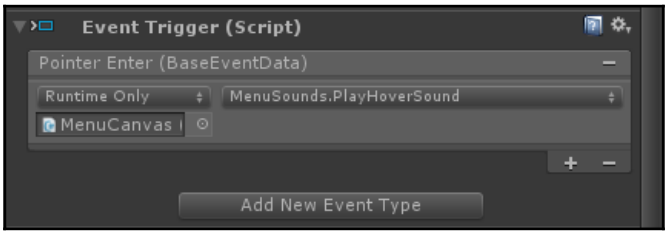
Highlight the **START** button in **Inspector**, click on **Add Component**, and select **Event Trigger**. You'll see the new **Event Trigger (Script)** component on the button, empty except for a button, **Add New Event Type**:



Click on **Add New Event Type** and select **PointerEnter** to create a new list of functions to be called when the player's gaze goes over the button:



Click the + button at the corner of the list to add a new function to the list. Drag the **MenuCanvas** object into the open slot and select **PlayHoverSound**:



Now add the logic to play the sound to the `PlayHoverSound` function in `MenuSounds`:

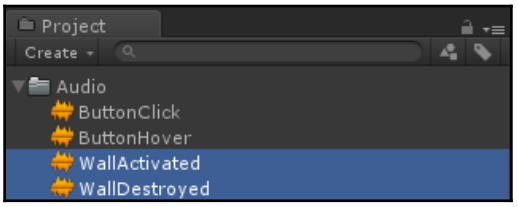
```
public void PlayHoverSound()
{
   audioSource.clip = buttonHover;
   audioSource.Play();
}
```

Now do the same with your **QUIT** button and then test your scene. You can quickly replicate the preceding steps by right-clicking the **Event Trigger (Script)** component and clicking on **Copy Component**. Then you can add it to the **QUIT** button by right-clicking on the **Add Component** button in the **Inspector** for it and selecting **Paste Component As New**. You'll now hear two unique 2D sounds when you hover and click either of your menu buttons. Menu buttons are the very tip of the iceberg of the audio we'll cover in this chapter, so next we'll move into the next dimension with some basic three-dimensional spatialization.

Adding basic 3D spatialization

In this section, we'll dive deeper into that **Spatial Blend** property we looked at earlier and create sound that has a discrete location in the level. We'll start by adding a few 3D sounds to our interactive walls: one to play when it's activated, and another to play when it's destroyed.

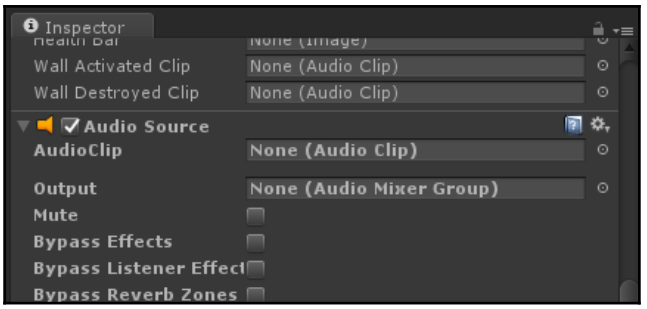
Find the audio files included with this chapter called `WallActivated.wav` and `WallDestroyed.wav` and drag them into your `Audio` folder:



Open your `DynamicWall` script and add two variables to store references to these sounds, as well as one for the **Audio Source** component:

```
public class DynamicWall : MonoBehaviour
{
    ...
    [SerializeField] private AudioClip wallActivatedClip;
    [SerializeField] private AudioClip wallDestroyedClip;
    private AudioSource audioSource;
    ...
}
```

We'll have to add an **Audio Source** component to the wall prefab before we can get a reference to it, so find an instance of **DynamicWallBase** to the scene and click on **Add Component** to attach a new **Audio Source**:



Drag the `WallActivated` and `WallDestroyed` audio files into the **Wall Activated Clip** and **Wall Destroyed Clip** fields, respectively. Click on **Apply** to save these changes to your **DynamicWallBase** prefab.

Go back to your `DynamicWall` script and add an `Awake` function that gets a reference to your newly-created **Audio Source** component:

```
public class DynamicWall : MonoBehaviour
{
   ...
   private AudioSource audioSource;

   private void Awake()
   {
         audioSource = GetComponent<AudioSource>();
   }
   ...
}
```

Next we'll add the code to actually play the clips, but we'll do it in a smarter way to minimize the bulk of this already complex script. Instead of making a separate function to play each sound, we'll make one function that takes an `AudioClip` parameter and handles both the assigning of the clip to the `AudioSource` and playing the clip. That way, we can play any sound with just one added line in any function.

Create a new function called `PlaySound`:

```
public class DynamicWall : MonoBehaviour
{
   ...
   private void PlaySound(AudioClip clip)
   {

   }
   ...
}
```

Add the same logic to this function as your other two sound functions, but use the `clip` parameter instead of either of your serialized variables:

```
private void PlaySound(AudioClip clip)
{
   audioSource.clip = clip;
   audioSource.Play();
}
```

Now add a line to the `MoveSectionUp` function to play `wallActivatedClip` when it's called. Also, add lines to enable or disable the counter object depending on whether the wall is active:

```
public IEnumerator MoveSectionUp()
{
   PlaySound(wallActivatedClip);
   counterObject.SetActive(false);
   ...
}
```

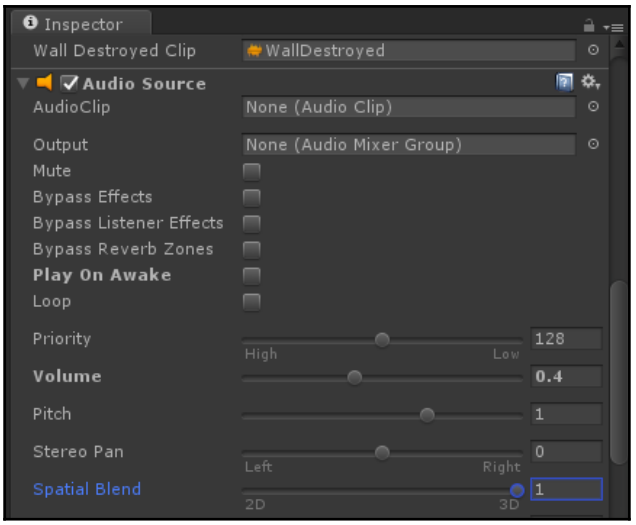
Now add another line to play `wallDestroyedClip` when the wall is moved down:

```
public IEnumerator MoveSectionDown()
{
   PlaySound(wallDestroyedClip);
   counterObject.SetActive(true);
   ...
}
```

Now the clips are ready to be played, and we just need to edit the **Audio Source** properties to make them 3D.

Making an audio source 3D

Click an instance of **DynamicWallBase**, find its **Audio Source** component in the **Inspector**, and set the **Spatial Blend** property to `1`. Also, set the **Volume** to `0.4`; we don't want to have too much noise if the player isn't very close to the wall. Don't forget to disable **Play On Awake** as well. Your **Audio Source** component should now look like this:



Finally, apply the changes to the prefab by clicking on the **Apply** button in the upper-right corner of **Inspector**. Test your level and activate a wall to hear the difference that spatialization makes. If you activate a wall on your right, the sound will be louder in your right ear. If you destroy a wall from far away, the sound will be very faint.

Immersing your player with HRTFs

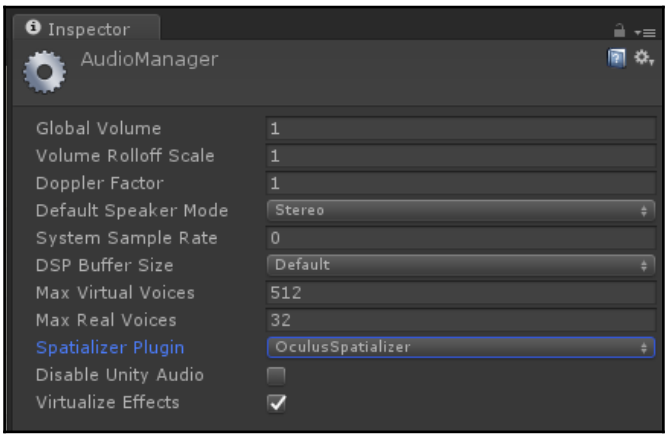
Stereophonic 3D is typically the most complex classic video game yet, but with VR, even more realistic audio is possible. Because of the data exposed through the constant tracking of the player's head, we can modify audio based on the exact position of both ears, instead of just the static head position. This modification is called a **head-related transfer function** (**HRTF**).

In this section, we'll modify our wall's audio source to be our first HRTF so we can demonstrate the capabilities and differences from 3D audio. We'll begin with Unity's simple first-party Oculus audio spatializer and then move onto Oculus's native spatializer plugin. Later, we'll create a soundscape for our level using HRTFs.

Sampling Unity's first-party HRTF

Unity features an extremely limited implementation of Oculus's HRTF spatializer that serves as a good place for getting started and hearing the subtle differences with HRTF audio.

To activate it, open the **Edit** menu, mouse over **Project Settings** and select **Audio**. The **AudioManager** settings will appear in the inspector. Find the **Spatializer Plugin** setting and set it to **OculusSpatializer**:



This will reveal a new checkbox on your **Audio Source** components labeled **Spatialize**; activate this on your **DynamicWallBase** prefab. Click on Play and try it out, tilting your head to hear the subtle differences between the reception of sound that each ear receives with the power of HRTF.

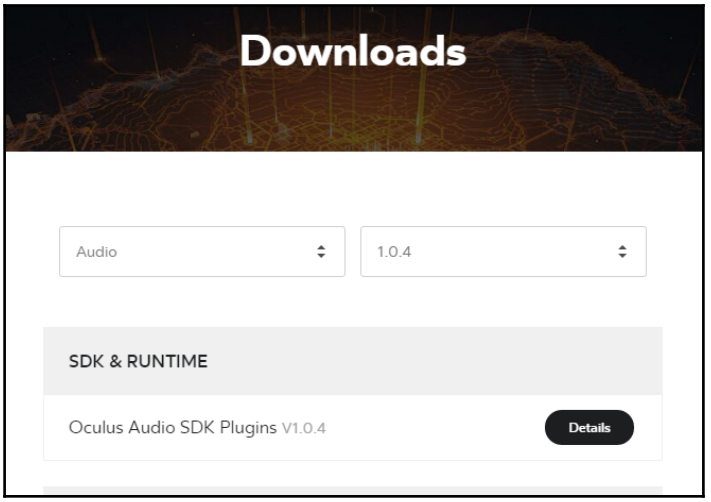
The blanket HRTF functionality that this setting enables is convenient but generic; in order to get a fully customizable HRTF plugin, we'll have to import the **Oculus Native Spatializer Plugin** (**ONSP**).

Importing the ONSP

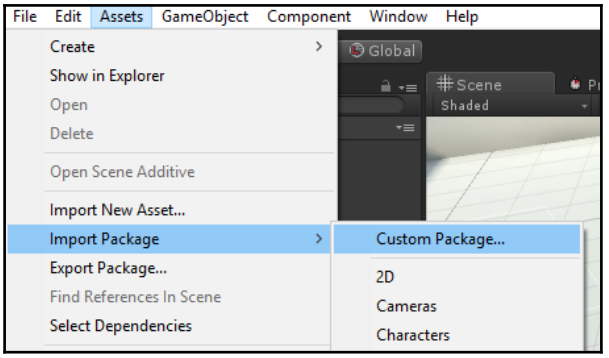
Before importing the OSNP files, we need to remove some files from our Unity installation that would clash with them. Don't worry, we'll save everything we remove, so you'll be able to revert to Unity's default state easily if you ever want to.

Open a file browser and navigate to your Unity installation directory (the default in Windows is `C:\Program Files\Unity\`). Within the installation directory, navigate to `\Editor\Data\VR\Unity\` and remove any files named `AudioPluginOculusSpatializer` (there may be one for each platform in subdirectories, that is, Win, OSX, Android, and many more). Because DLLs are loaded by Unity when the editor starts up, you may need to quit Unity before being able to delete them.

Once you've removed these files, you can import the ONSP into your project. In a web browser, go to `https://developer3.oculus.com/downloads/` and find the **Oculus Audio SDK Plugins** download. You can narrow down the list by selecting the **Audio** category from the drop-down list:



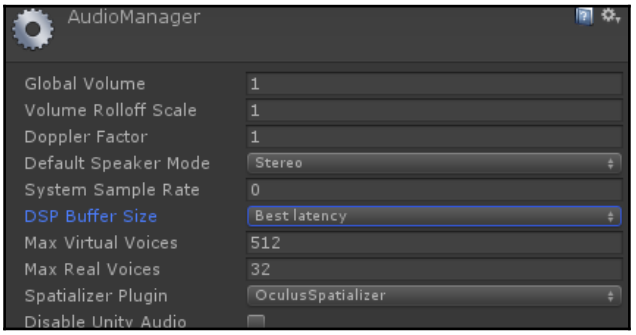
Click on **Details** to download the package and save it in an easily accessible folder. In Unity, open the **Assets** menu on the toolbar, mouse over **Import Package** and select **Custom Package…**:



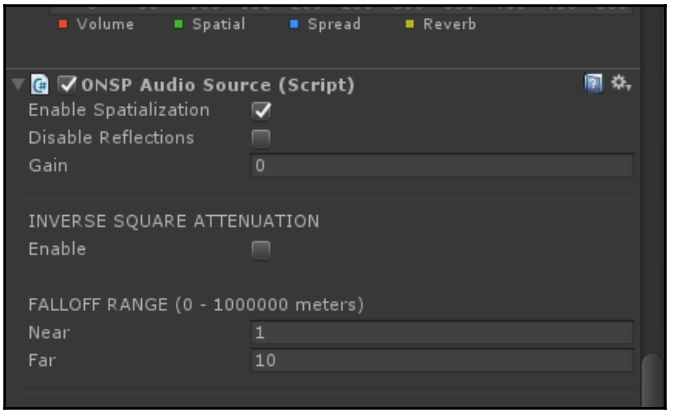
Navigate to the downloaded audio SDK folder and find the file called `OculusNativeSpatializer.unitypackage` in `\AudioSDK\Plugins\Unity\`. Import the package, which will add several files to a new folder called `OSPNative` and several new plugins to your `Plugins` folder which will replace the ones that you deleted from your installation directory.

Using the ONSP

Once you've imported the ONSP files, make sure the **Spatializer Plugin** field in your **AudioManager** is still set to **OculusSpatializer**. Also, change your **DSP Buffer Size** to **Best latency**:



Click one of your **DynamicWallBase** instances in the scene and add a new `ONSPAudioSource` script, which can be found in the `scripts` folder in `OSPNative`. This will reveal several new audio properties in **Inspector**:



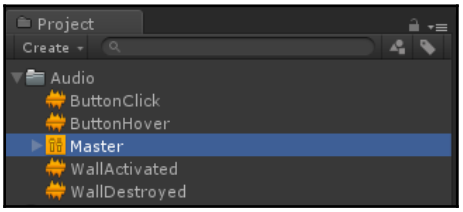
Check the box next to **Enable** under **INVERSE SQUARE ATTENUATION**. This will override the volume falloff curve in the **Audio Source** component and use an internal more realistic audio falloff instead.

This also enables the use of the **Near** and **Far** falloff range variables at the bottom of the component. The **Near** value dictates how far away in meters a sound needs to be before it begins to attenuate (or fall off), and the **Far** value is the maximum range of the attenuation.

Adding sound reflections to the scene

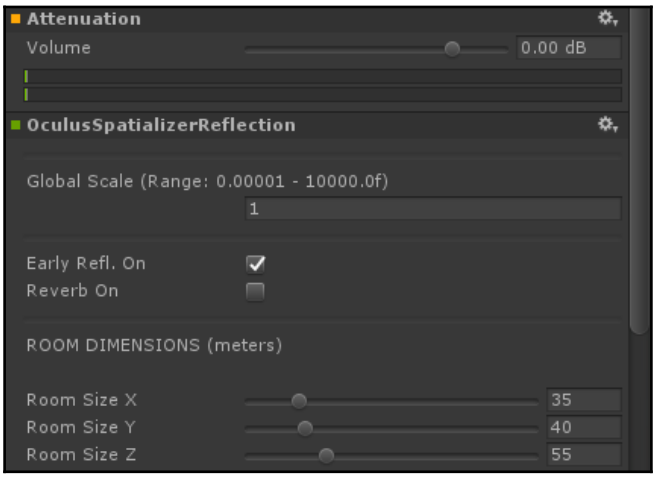
To make full use of the ONSP, we'll need to use one of Unity's advanced sound features: The **Audio Mixer**. The mixer will let us run all our sounds through filters and organize them by group/channel so that we can have dynamic, categorical control over all the audio in our game.

Right-click on your `Audio` folder in the **Project** window and select **Audio Mixer** in the **Create** submenu to create a new mixer. This will be the primary mixer for our scene for now, so name it `Master`:

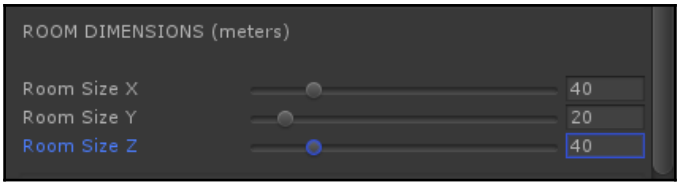


Click the arrow to the left of the mixer to expand it and view its children. Click the child that's also named **Master** to display it in **Inspector**. By default, it just has a **Pitch** and **Attenuation** value:

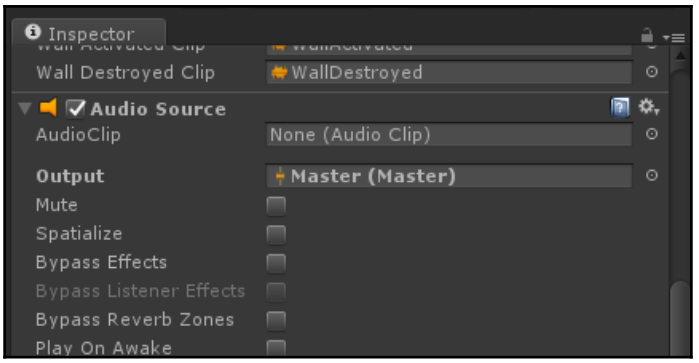
Click the **Add Effect** button at the bottom-right corner and select **OculusSpatializerReflection**. A new section of the mixer will appear in the **Inspector** window and you'll be able to set various values for our reflection effect:



Our level is approximately 40 meters by 40 meters, so set **Room Size X** and **Room Size Z** to match. Set **Room Size Y** to be `20` meters, slightly higher than the actual vertical limit of our level:



The last thing we'll need to do to add this new mixer to our audio pipeline is to link it to the applicable **Audio Source** components. Click the **DynamicWallBase** that you added the **ONSP Audio Source** to and drag the **Master** mixer asset into the **Output** field:



Click on **Apply** to make this change to the **DynamicWallBase** prefab definition and save your scene. A gizmo icon will appear at every new audio source location, and there will be a lot of them; if there are too many gizmos in your **Scene** view, you can control visibility options from the **Gizmos** menu along the top of the **Scene** window. At this point, take it for a test drive; as you walk through the scene triggering sound effects, take a note of how they sound when you activate a wall close to other geometry. You may need to adjust the volume of the components to account for differences between the plugins. The complexities of realistic spatialized audio are subtle but incredibly impactful if done right.

Filling the game world with sound

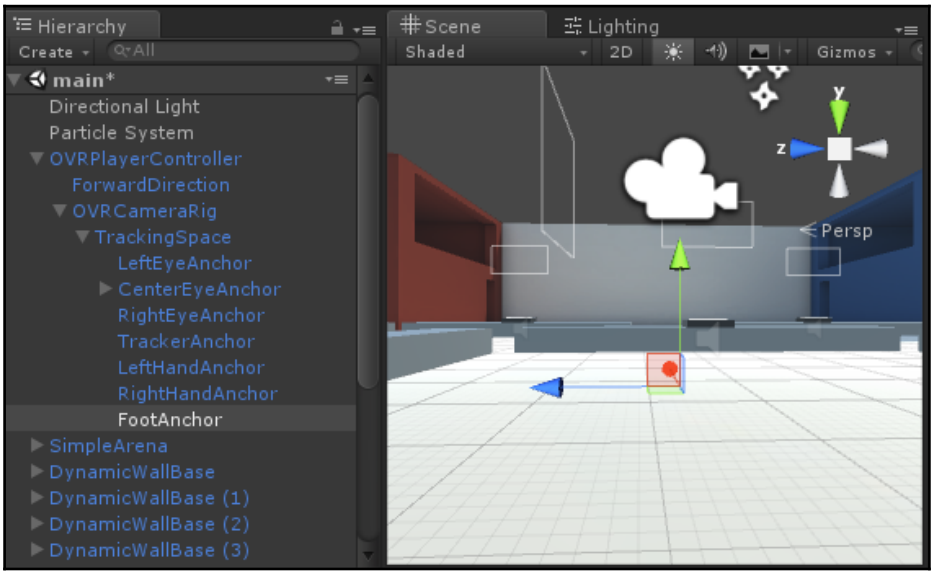
At this point, we've gone over all the basic methods of audio: stereophonic 2D, spatialized stereophonic 3D, and realistic HRTF 3D audio. With that foundational understanding, we'll use this section to add several more audio effects to the scene to make it seem truly complete and cohesive.

Adding footstep sounds

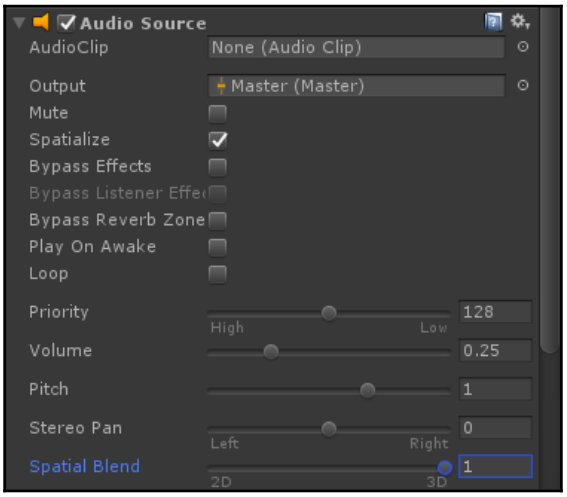
Footstep sounds are pretty common in shooters, especially shooters that don't have a radar mechanic. It's a great, subtle way to hear an enemy around a corner, and the more accurate the localization is, the more the sound becomes a useful part of the environment instead of just a noisy effect.

In this section, we'll create a footstep mechanic for the players in the game, and we'll use HRTF audio with reflections to create a realistic effect.

Drag the `Footstep1.wav` and `Footstep2.wav` files that were included with this chapter into your `Audio` folder. Create a new empty **GameObject**, name it `FootAnchor`, and make it a child of the **OVRCameraRig** so it sticks to the player's transform just like the other anchors:



Add an **Audio Source** component to the **FootAnchor** object as well, since it's position will be the position of origin for our footstep sounds. Configure it like so:



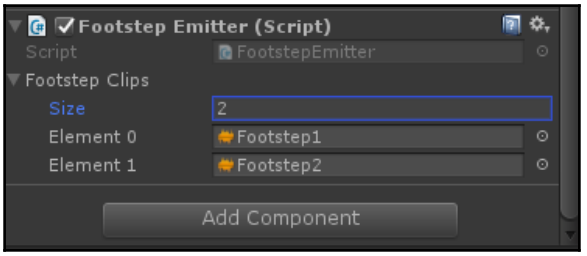
Also, add an **ONSPAudioSource** component to the **FootAnchor** object to make it use the native spatializer, and make sure you've dragged the **Master** mixer into the **Output** field so that it runs through the spatialized filter.

Create a new script called `FootstepEmitter` and attach it to your **FootAnchor** object. Open the script and define an array to hold your footstep sound effects and a reference to the **Audio Source** component:

```
public class FootstepEmitter : MonoBehaviour
{
  [SerializeField] private AudioClip[] footstepClips;
  private AudioSource audioSource;
}
```

You may be wondering why we're declaring an array instead of two discrete clip variables. While we only have two footstep sounds now, we could potentially add more at any phase in development, and by keeping an array we can add an extra clip without having to declare a new variable. When we go to play a footstep sound, we can pick a random one out of the array of all the loaded clips.

On the script component in the **Inspector** window, specify the size of the **Footstep Clips** array as `2` and drag the two imported footstep sounds into the open fields:



Initialize the `audioSource` variable in the `Awake` function:

```
public class FootstepEmitter : MonoBehaviour
{
  ...
  private void Awake()
  {
    audioSource = GetComponent<AudioSource>();
  }
}
```

Now define a new function called `PlayFootstepSound` that plays a random sound from the array:

```
private void PlayFootstepSound()
{
  int randomIndex = Random.Range(0, footstepClips.Length - 1);
  audioSource.clip = footstepClips[randomIndex];
  audioSource.Play();
}
```

The final step is to make these clips play at a rhythm as long as the player is moving. Create two new float variables, one called `timeSinceLastStep` and another called `timeBetweenSteps`:

```
public class FootstepEmitter : MonoBehaviour
{
  ...
  private float timeSinceLastStep = 0f;
  private float timeBetweenSteps = 0.5f;
  ...
}
```

Now define a new function called `UpdateStepTimer` that updates the time since the last step and plays a footstep sound if the new time exceeds the time between steps:

```
private void UpdateStepTimer()
{
  timeSinceLastStep += Time.deltaTime;
  if(timeSinceLastStep > timeBetweenSteps)
  {
    PlayFootstepSound();
    timeSinceLastStep = 0f;
  }
}
```

We don't want to update the step timer every frame unless we're actually moving, so define an `Update` function with a check to the input axes and only update it if they're not stagnant:

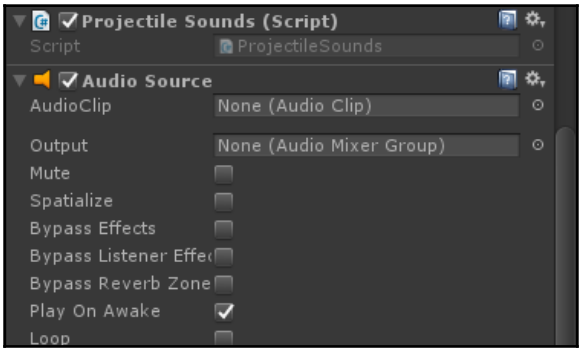
```
private void Update()
{
  if(Input.GetAxis("Horizontal") != 0 || Input.GetAxis("Vertical") != 0)
  {
    UpdateStepTimer();
  }
}
```

We'll leave this system as it is for now, so test it out and take your first audible steps in virtual reality. This isn't just for the player's benefit; when we implement multiplayer, you'll be able to hear these footsteps coming from anyone else in the game.

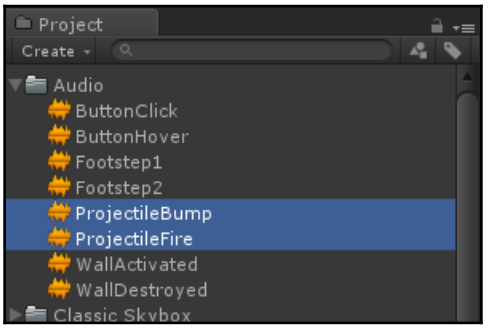
Adding projectile sounds

Of course, our projectiles wouldn't seem like projectiles at all without some nice punchy sound effects to complement them. In this section, we'll quickly add two sound effects to our projectile: one to play when it's fired, and one to play whenever it hits something.

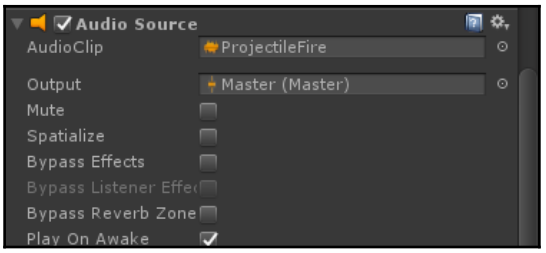
Highlight the **Projectile** prefab in the **Project** window to display it in **Inspector**. Create a new script called `ProjectileSounds` and add it as a component, as well as an **Audio Source** component. Make sure to keep **Play On Awake** enabled for this one, because we'll want to play our firing sound as soon as the projectile is instantiated:



Import the two files included with this chapter named `ProjectileFire.wav` and `ProjectileBump.wav`:



Drag the **ProjectileFire** clip directly onto the **AudioClip** field, along with the **Master** mixer asset:



Because we want to play this sound only once per projectile and we don't need to access it in code after that, we don't need to link it to a script, we can just leave it as the default clip. For the bump sound, however, we'll need some logic to play it whenever the projectile collides with something.

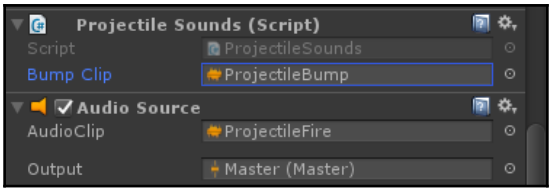
Open your `ProjectileSounds` script and add a reference to the bump sound effect and a reference to **Audio Source**:

```
public class ProjectileSounds : MonoBehaviour
{
   [SerializeField] private AudioClip bumpClip;
   private AudioSource audioSource;
}
```

Initialize the audio source in the `Awake` function:

```
public class ProjectileSounds : MonoBehaviour
{
   [SerializeField] private AudioClip bumpClip;
   private AudioSource audioSource;

   private void Awake()
   {
         audioSource = GetComponent<AudioSource>();
   }
}
```

Drag the **ProjectileBump** clip into the new **Bump Clip** field on the **Projectile Sounds (Script)** component:



On the **Audio Source** component, shift the spatial blending from 2D to 3D and check the **Spatialize** checkbox. Finally, add the `OnCollisionEnter` function to your `ProjectileSounds` script and have it play the bump clip whenever a collision is detected:

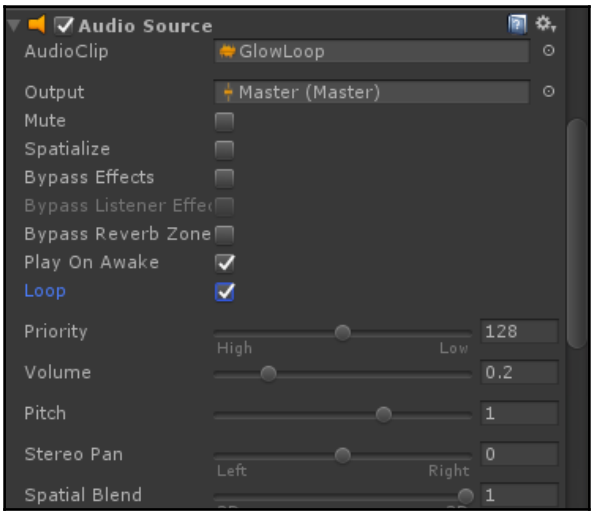
```
private void OnCollisionEnter(Collision collisionInfo)
{
   if(audioSource.clip != bumpClip)
   {
         audioSource.clip = bumpClip;
   }
   audioSource.Play();
}
```

Test your new feature by playing the game and firing some bullets against the wall. You should hear the firing sound as soon as the bullet is visible, and the bump will play whenever it collides with a new surface.

Adding ambient looping to energy orbs

For certain sounds in your game, you'll want audio to constantly loop instead of having to be played by a function call every time. Fortunately, this is as simple as checking another box in your **Audio Source** component. We'll create our first looping audio by attaching an ambient loop to our **EnergyOrb** prefab that will clue players into the location of an **EnergyOrb** even if it's out of their view.

Import the `GlowLoop.wav` file into your `Audio` folder. Select the **EnergyOrb** prefab and add a new **Audio Source** component to it, as well as an **ONSPAudioSource**. First, set the **SpatialBlend** to `1` like your other 3D sounds and move the **Volume** down to `0.2`; we want it to be a subtle effect. Then, drag the `GlowLoop` file into the **AudioClip** field, the **Master** mixer into the **Output** field, and check the **Loop** box:



Now, every **EnergyOrb** instantiated in your scene will have a light "glowing" tone, and you'll be able to instinctively locate **EnergyOrb** objects that are behind you thanks to the spatialization of the audio.

Summary

In this chapter, you brought sound to your game environment using a few different methods. Good audio is what holds VR environments together, and the importance of realistic sound in immersion is not to be underestimated.

We began with simple 2D audio, which is fast and easy to implement but really only good for audio scenarios in which location doesn't matter, such as menu sounds. We then modified Unity's **Audio Source** component in order to spatialize it and create basic stereophonic 3D audio.

Stereophonic audio can help players locate the origin of sounds around them, but it can't be used to determine whether the sound is coming from above or below the player because the "head" in the audio calculation is completely static and the ears are always in the same relative orientation.

We then added another dimension of realism to our simple 3D audio by implementing HRTFs, or head-related transfer functions. HRTFs are only possible with head-mounted devices because they take head/ear rotation into account; because we get all the head rotational data we need, we can easily support HRTFs using the Oculus Rift.

Using all three methods, we created a dynamic and vibrant soundscape for our game which will help our player believe what they hear as much as what they see.

8
Adding Tone and Realism with Graphics

Graphics haven't always been at the forefront of game development. In the early days, on systems such as the Atari 2800 and Magnavox Odyssey, avatars and items could only be represented by crude pixel estimations, and levels were, more often than not, simple 2D blocks. Older machines just didn't have the capability of rendering anything with even close to realistic detail, and the clunky CRT displays from back in the day wouldn't do fantastic renderings any justice anyway.

These days, the success and quality of a game depends heavily on how it looks. Rendering and lighting in games is as close to real life as it ever has been, and there are several graphical capabilities in modern 3D engines that mimic the actual physics of light. Virtual reality gives us the opportunity to immerse players in these realistically rendered environments, which demands a greater focus than ever on the graphical accuracy and quality of your game.

In this chapter, we'll cover some science around how objects are lit in real life, and how your eyes see them. We'll then transition into how these principles can be applied in game development using materials, maps, shaders, and image effects to get the best visual quality in your game without overloading your GPU.

This chapter will cover the following topics:

- A simple breakdown of the rendering pipeline
- Forward and deferred rendering
- Fundamentals of image effects
- Introduction to shader programs
- Adding tone to a scene with color grading

A simple breakdown of the rendering pipeline

Unsurprisingly, a lot goes into creating a virtual 3D world and then displaying it on a 2D screen. Unity obfuscates a lot of the low-level rendering process so we don't have to deal with it, but to master anything related to graphics it's important to have a fundamental understanding of what's happening behind the scenes.

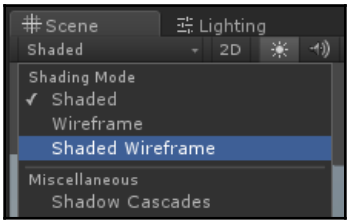
Every step of the rendering process is handled within something called the **rendering pipeline**. In this section, we'll go over each step in detail so you can gain a full understanding of what graphics are.

Defining the geometry

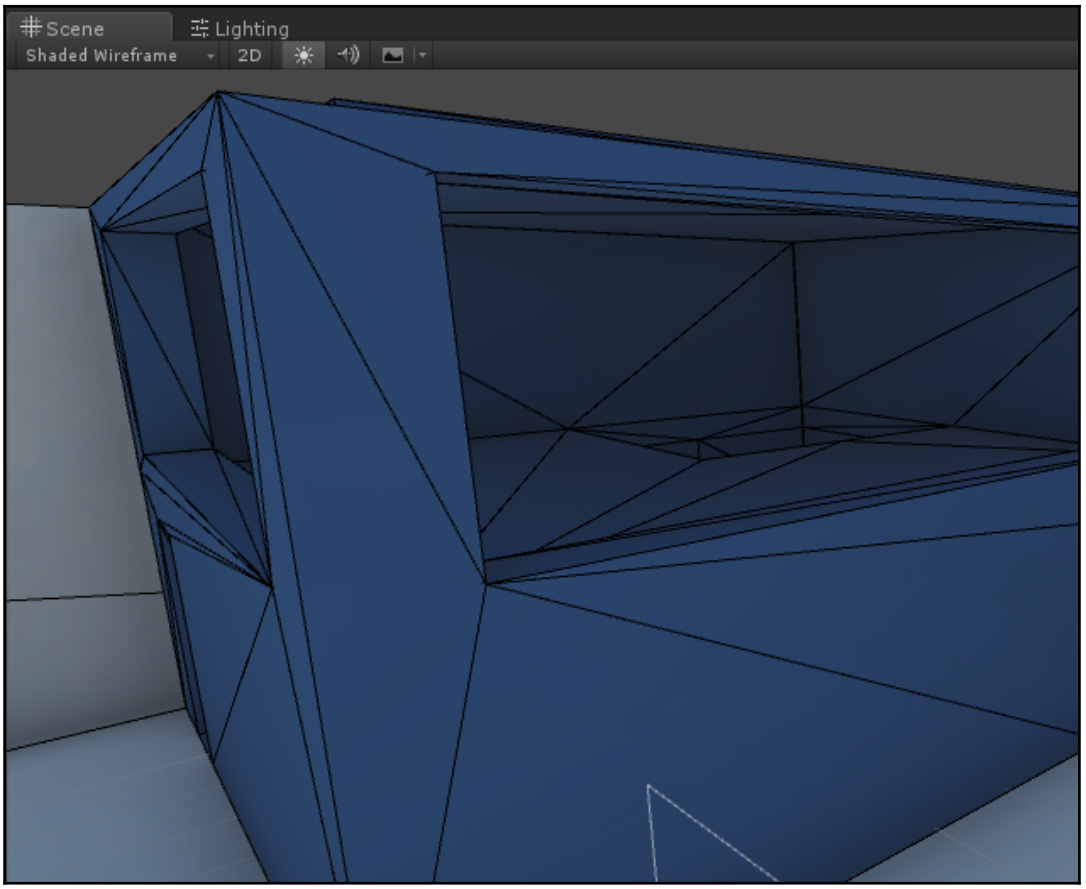
The first part of rendering is collecting the objects to be rendered. Every object in a scene is a collection of **vertices**, or the points of the object's geometry, and **indices**, which define the faces between the points.

Every set of three indices defines a triangular face of the object. Triangles are handy for defining complex surfaces because each individual triangle can only exist on one plane, so several two-dimensional triangles can define a single 3D object.

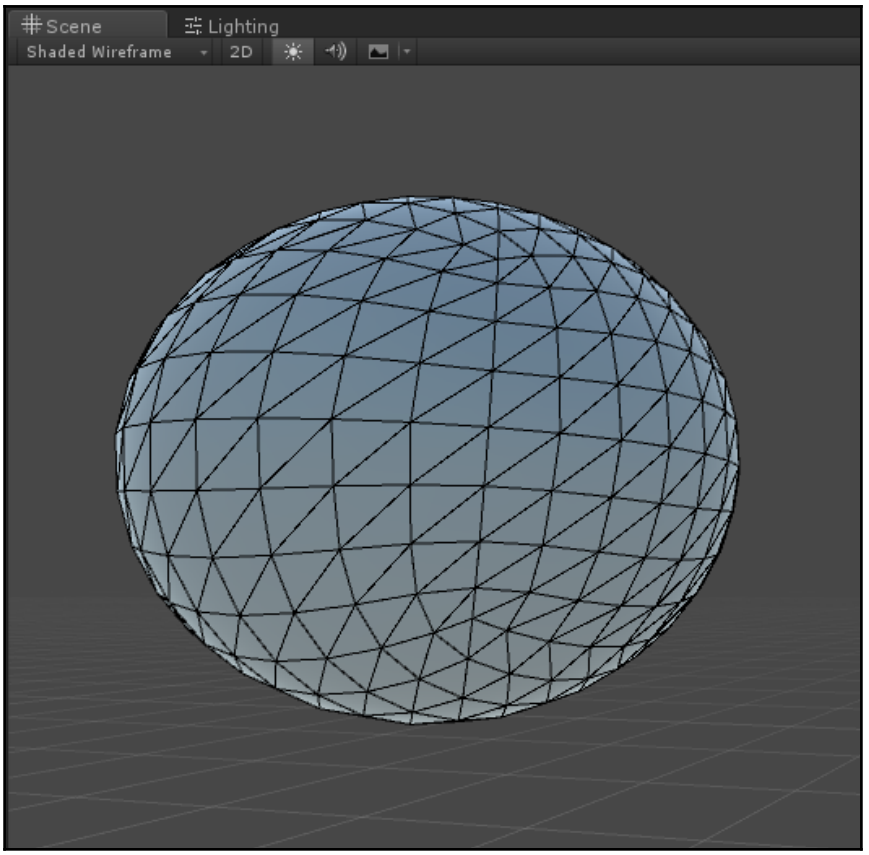
Let's look at the triangles that make up one of our objects in Unity. Open your Combat Arena project. In the **Scene** window, look for the drop-down menu in the upper-left that reads **Shaded**. Open this menu and select **Shaded Wireframe**:



This will render the outline of the object's triangles in the **Scene** view, showing the smaller parts that make up our building model:



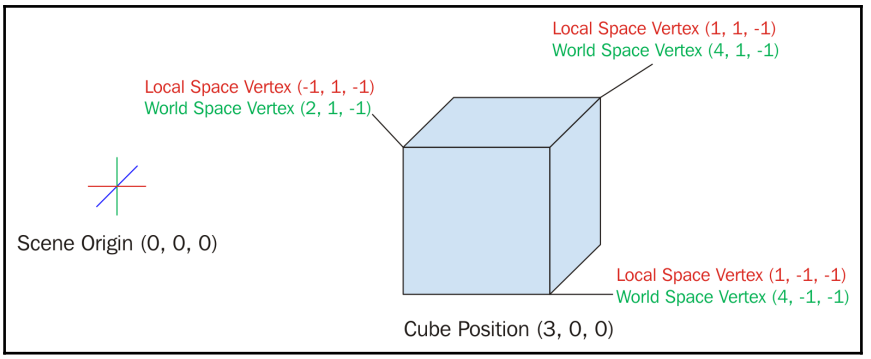
As you can see, these triangles aren't exactly clean or even; they were automatically generated by sketchup after the geometry had been created with simple extrusion tools. Some advanced modeling programs will let you explicitly arrange the triangles, allowing you to create an even distribution, like Unity's built-in sphere features:



Transforming the model into world space

When geometry is processed, the vertex positions of each object are represented in **local space**, meaning the coordinates are all relative from the origin of the model and they have nothing to do with where the object is supposed to be in the game world.

The next step involves transforming the coordinates based on the world space position of the model to place it in the environment; for instance, if an object was placed three meters to the right of the origin of a scene, we'd add three to every vertex's right (X) coordinate. This is demonstrated in the following diagram:



Transforming the world into camera space

The step immediately following the model transformation is the view transformation. This uses the same logic as transforming a model for the context of the world, but in this case we're transforming the world for the context of the camera. Imagine it like setting up a theatrical stage; the narrative world exists beyond the bounds of the stage, but everything important has been arranged to face the audience.

Lighting the objects in the scene

At this point, lighting is calculated for every object in the scene. In forward rendering, there's one lighting pass per light in the scene, so using several lights at the same time can really slow down your game during this phase in the pipeline.

Deriving a projection from the camera

The final transformation that occurs before your scene is displayed in the screen is **projection transformation**. This is what gives a sense of depth and perspective; basically, objects that are further from the camera are made smaller by dividing the *X* and *Y* scale by the *Z* distance from the camera.

Clipping geometry outside the camera's frustum

After the projection view is ascertained, all the geometry that lies fully outside the camera's projected frustum is discarded. This is sometimes called **frustum culling**, and it ensures the renderer isn't doing any more work than it needs to by processing elements that will never be seen by the player anyway. This isn't the same as **occlusion culling**, which prevents objects from being passed to the renderer at all based on their position.

Rasterization and texturing

The final step in the rendering pipeline is rasterization, or the pixel/fragment stage. This is when the geometry data—the vertices, indices, and accompanying metadata—are drawn per-pixel based on the resolution of the target display.

This phase is where textures are applied to your objects as well. Every vertex has an accompanying UV coordinate that corresponds to a location on a 2D texture file such as a PNG. For every triangle on the object, every pixel is colored by interpolating between the three UV coordinates on the texture. This process is called **texture mapping**.

Think of a texture map like a map of our globe. It can look warped when laid flat on a table, because it's a representation of a 3D object in 2D space. However, if you were to wrap it around a globe, it would fit perfectly. Think of UV coordinates as shared points between the map and the globe that help fit every part of a 2D texture onto a 3D surface.

Forward and deferred rendering

We mentioned forward rendering briefly in the last section, but what exactly does forward rendering mean? Forward rendering is one of the two main high-level rendering methods used by Unity; for the most part, you'll probably be using forward rendering for VR development, but it's still good to know the advantages and disadvantages of both methods.

Forward rendering

In forward rendering, lighting is calculated for each geometry **fragment**, or visible pixel of an object, regardless of whether another object will ultimately obscure any fragments that would otherwise be seen. Each individual light in the scene requires its own calculations, so if you have a complex scene with a lot of geometry and more than a few lights, you're going to hit a performance bottleneck pretty quickly.

The best way to get around this issue is to bake your lights instead of actually rendering them. By using the same light baking method we used in `Chapter 2`, *Stepping into Virtual Reality*, we can modify the texture to account for the light's intensity and then turn off the dynamic light itself.

Deferred rendering

The major strength of deferred rendering is the ability to have several dynamic lights in a scene without it necessarily costing too much in terms of performance. This is because it renders fragment data for every visible fragment within the screen resolution after performing a preliminary depth pass. That means two things: you won't waste any time processing fragments that are obscured by other objects in the scene, and you'll never process more fragments than you have pixels within your screen resolution.

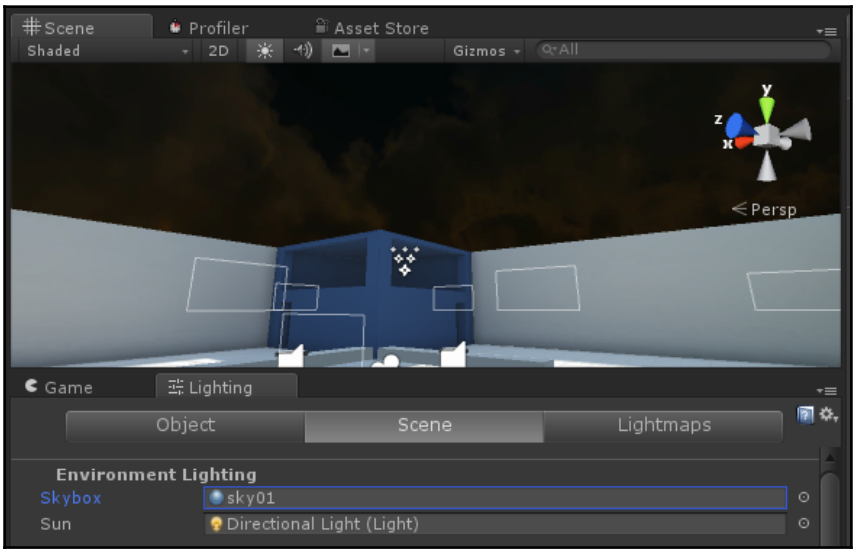
In the next section, we'll see the power of deferred rendering at work by creating a version of our scene lit by several dynamic lights.

Demonstrating the value of deferred rendering

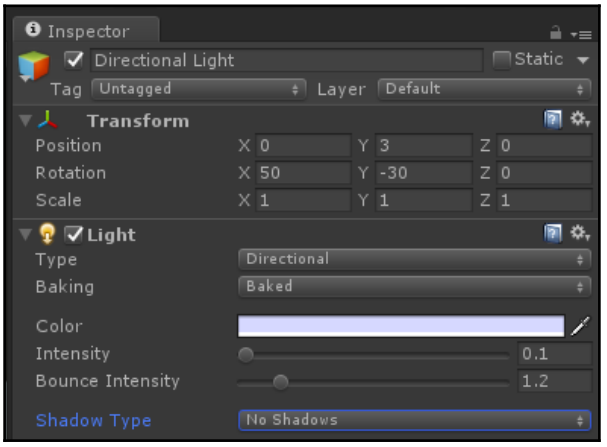
To see the true power of deferred rendering, we'll make a duplicate of our existing level but set it up to be a night scene instead of a day scene. Then, we'll add several point lights around the level, and switch to a deferred rendering technique to look at the performance differences in a scene where dynamic lights are abundant.

Re-name your scene file from `main.unity` to `Square_Day.unity` and then copy the file in your explorer, and name the copy `Square_Night.unity`. Also, feel free to rename the main folder that contains your light maps to `Square_Day` as well.

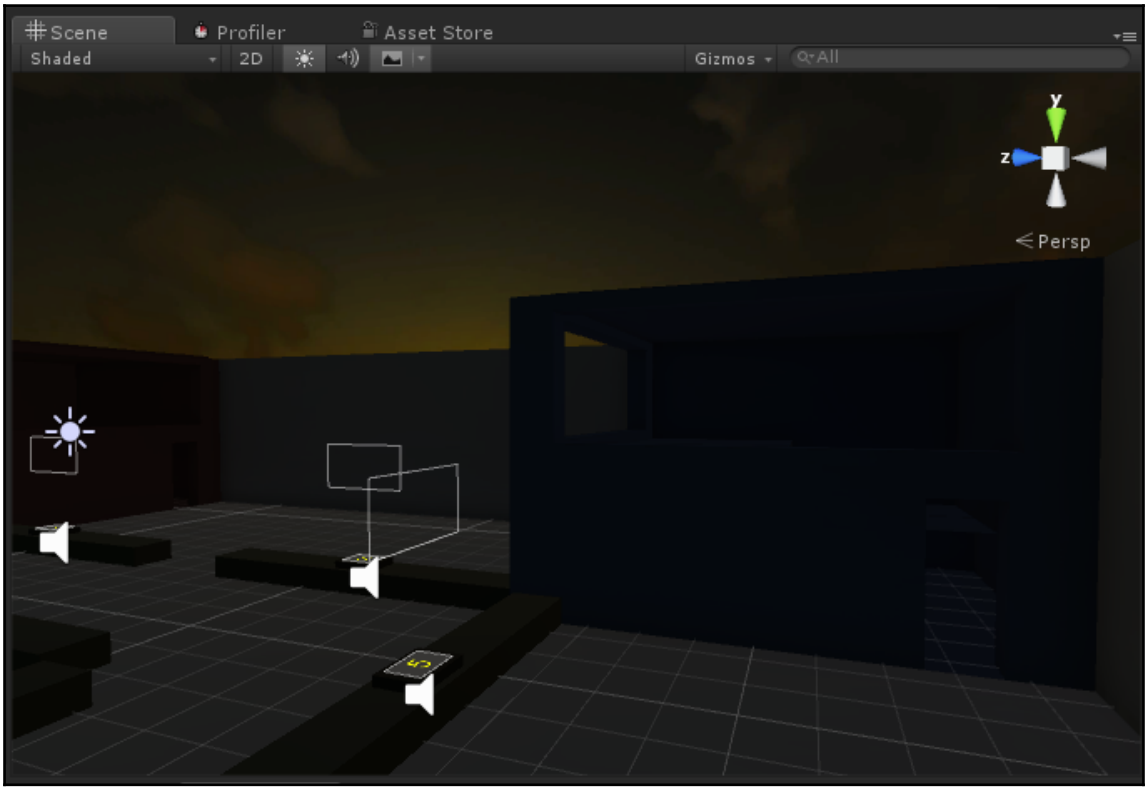
Open your **Square_Night** scene and navigate to the **Lighting** menu. Find the skybox called **sky01** in your `Classic Skybox` folder and drag it into the **Skybox** field in the **Lighting** window:



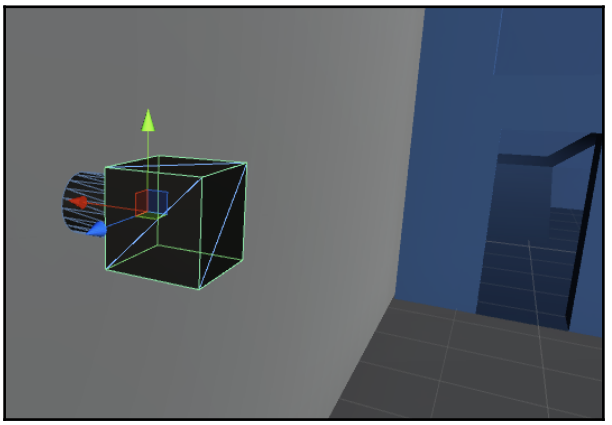
Obviously, our environment is still lit as though it's in daylight, so we'll need to tweak our light settings and re-bake it so it looks realistic. Highlight the **Directional Light** object in your scene in the **Inspector** window and change **Intensity** to `0.1`, **Color** to a light blue, and **Shadow Type** to **No Shadows**:



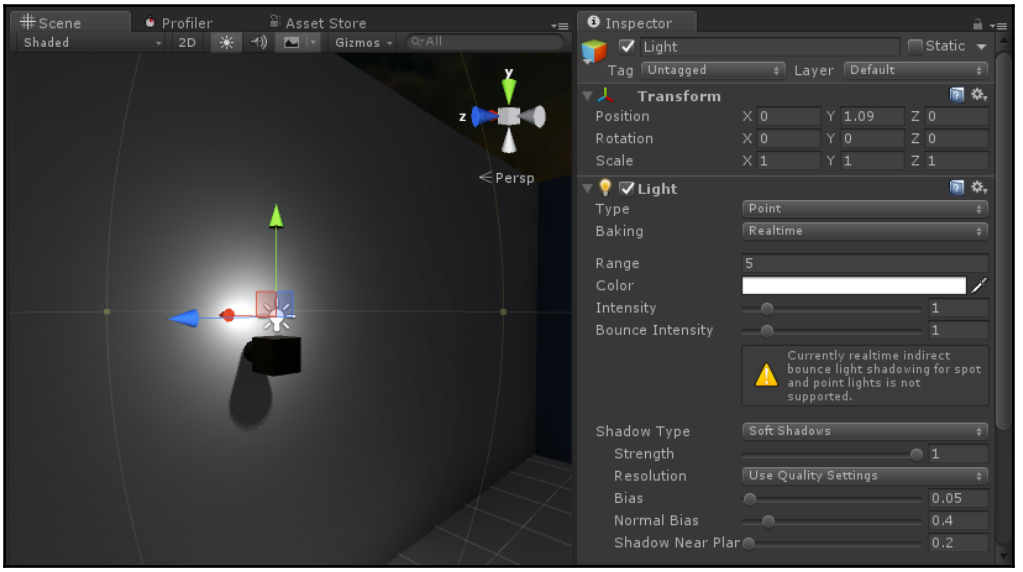
Now re-bake your scene to see the new lighting settings in action. Dropping the intensity and removing the shadows will give it a more natural "night-time" look; and changing the color from a slight yellow to a slight blue will mimic the darkness of night instead of the yellow brightness of the sun:



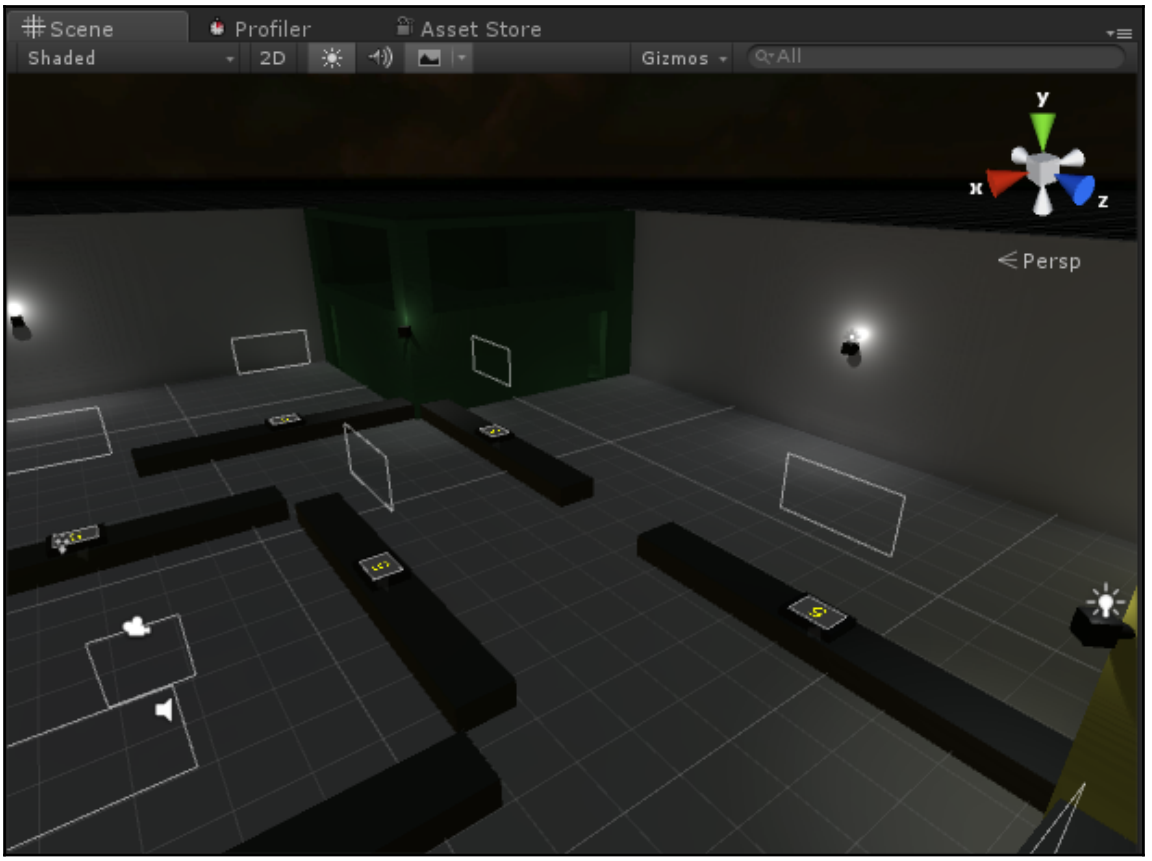
Next, we'll create a point light prefab so that we can scatter instances around the scene. Use primitive objects and a dark gray material to create a placeholder wall-mounted light asset, like so:



Make the cylinder a child of the cube and name the new combined object `WallLight`. Create a new empty game object as a child of the `WallLight` object and name it `Light`; the transform of this empty object will serve as the point of origin for the light itself. Position the empty game object slightly above the cube and add a new **Light** component to it in the **Inspector** window. Set **Type** to **Point**, **Shadow Type** to **Soft Shadows**, and **Range** to `5`. Your light should now look something like this:

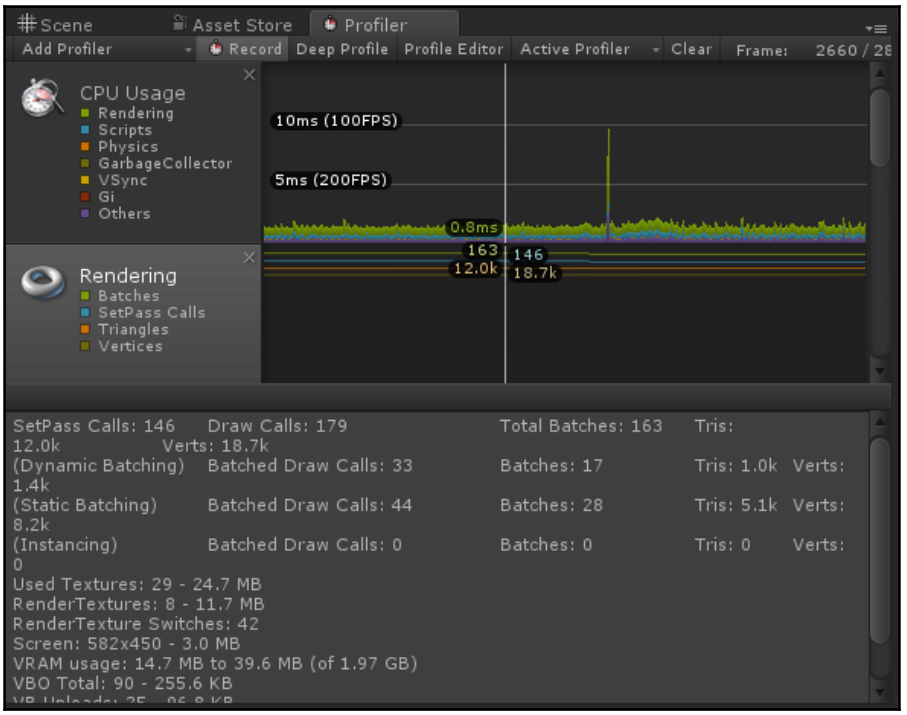


Now save the **WallPoint** object as a prefab and scatter ten or twelve of them around the scene to create a well-lit night scene. Don't forget to light the interior of your buildings too, particularly the bottom floors that don't have windows. Make your scene look something like this:

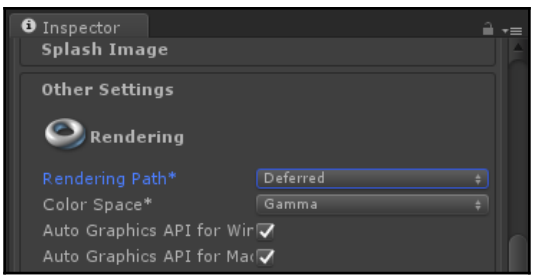


You may notice some lights turning off as you place more; the number of individual lights that can influence a single object is capped, but you can adjust this property in **Quality Settings** from the **Edit** menu. Try increasing the **Pixel Light Count** to something like `8` if you experience some flickering.

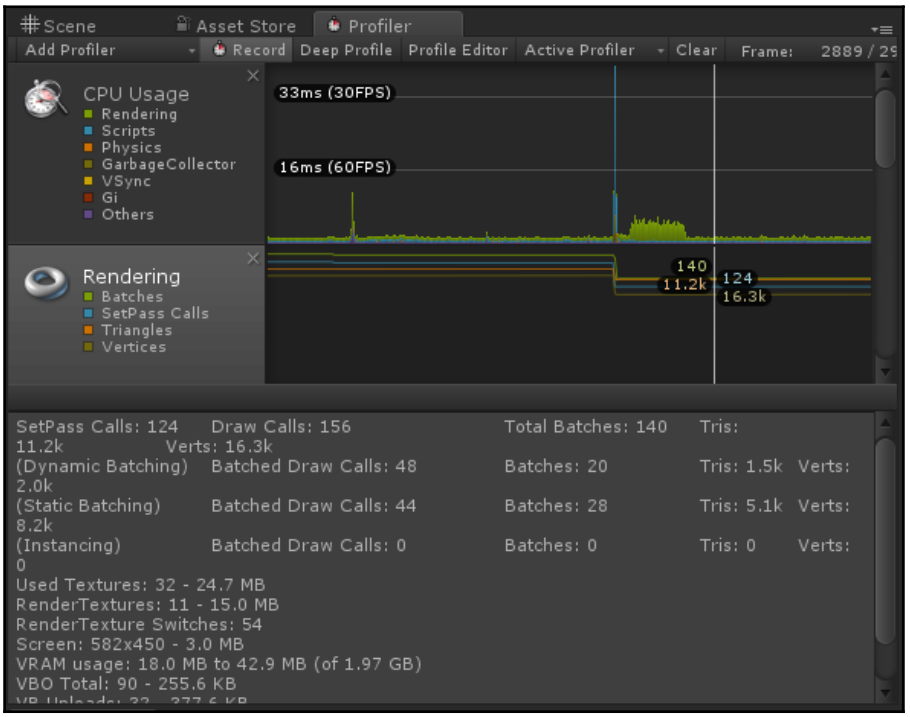
Of course, the **Pixel Light Count** default value is **4** for a reason. Rendering this many real-time lights is relatively rare in games. Try running your game now with the **Rendering** section of the profiler open and observe how intensive the rendering process is using the default forward rendering:



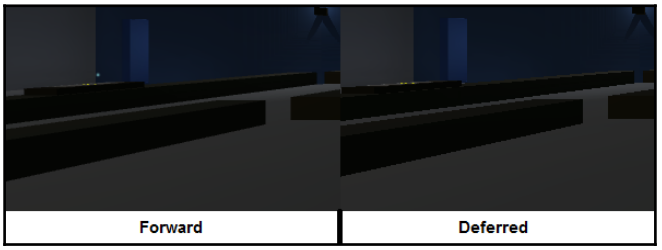
Next, we'll switch to the Deferred rendering path and measure the relative performance in the profiler. Open **Player** settings in the **Edit** menu and set the **Rendering Path** setting to **Deferred**:



If you don't run it for too long, your forward rendering frames and your deferred rendering frames will fit in the same profiler window. The drop in usage is clearly visible, and there are fewer draw calls:



You may have already noticed something strange when using a deferred rendering path: anti-aliasing is not being applied and the “stair-stepping” alias effect is much more obvious in geometry that runs diagonally along the screen. This is clear in the following side-by-side comparison, with the left screenshot from a forward renderer and the right screenshot from a deferred renderer:



This effect is so noticeable because the method of anti-aliasing we've been using, **multisample anti-aliasing** (**MSAA**), is not compatible with the deferred rendering process. This is especially unfortunate for VR experiences, because the distance of the screen from your eyes means aliasing and other pixel artifacts are quite obvious.

There are some anti-aliasing solutions that will still work with deferred rendering, such as **fast approximate anti-aliasing** (**FXAA**), but they're based on screen-space data with no knowledge of the actual geometry, and the effect is not as clean or efficient as MSAA with four or eight samples (we'll touch on this kind of image effect later).

It's cool to see what deferred rendering can do for our game in special cases, but most games use a forward rendering path just for the MSAA and the broader, more common support that it offers. This is why light baking is important: if we bake all of these lights instead, we can render the scene with a forward rendering path without having to make an extra pass for each light in the scene.

You might still find a reason to fall back on a deferred renderer later in development, perhaps for deferred-only features such as HDR, but for now forward is still the best method for us to use. In the **WallLight** prefab's **Light** component in the **Inspector** window, set it to **Baked** from **Realtime** and then bake the scene from your lighting menu. Now you can switch back to the forward rendering path and our night scene will perform just as well as our day scene.

Adding tone with color grading

Changing lighting isn't the only way to convey a certain mood or tone in a scene. Like many modern movies, many games also use color grading to give a certain hue to everything in a scene. For instance, in a desert scene, color grading could be used to make everything appear yellowed under the harsh sun, or a mild blue grading could make a frozen tundra seem even colder.

If you've ever used Instagram, you can think of color grading like filters; the source image goes in and is then combined with the colors of the filter to give it an overall tone. In this section, we'll apply a few different color grading effects to our scene to get an understanding of what it can do.

The basics of color grading

Color grading is a post-processing effect, which means it comes into play after your scene has been projected and rasterized. Each pixel's color value is treated like a coordinate on a 2D map of all colors. Typically this coordinate is then passed to a custom map of different colors that we provide and the pixel value from that map is used instead. This kind of map is called a **lookup texture** (**LUT**).

Generally, lookup textures can be created by starting with a default color map and then re-coloring it like you would tweak a standard image in Photoshop. The default color map looks like this:

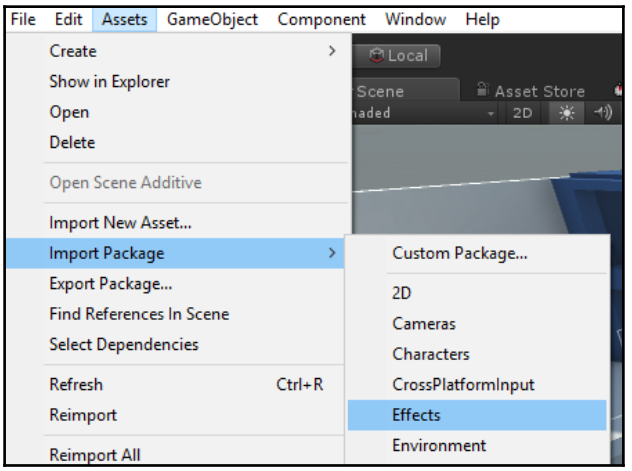
Every pixel in that map represents a possible color of a pixel in your render. Because this is the default map, applying this as a lookup texture to your scene wouldn't change its effect at all, because it's a direct mapping to the original colors. If we applied a high-contrast effect to this texture, however, it would look something like this:

If we used this modified map to color our scene, every time a pixel from the original map was rendered, it would be swapped out with the corresponding pixel on the modified map instead. In the next section, we'll see this effect for ourselves.

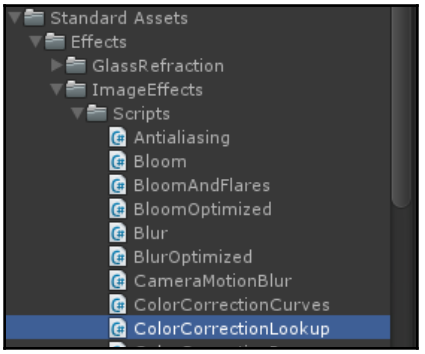
Adding a color grading script to the camera

The first step in adding color grading to a scene is to add a script to your main camera, which will perform the color swapping once the scene has been rendered. Fortunately, Unity includes a script that does this, so we don't need to write one ourselves. Because our night scene is pretty dark in some areas, switch back to the day scene so you can see the color changes clearly.

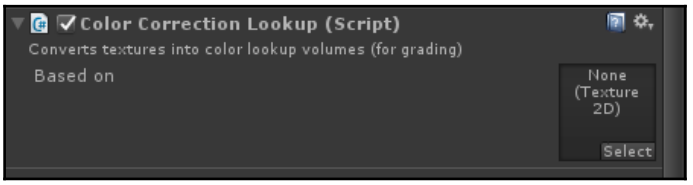
Open the **Assets** menu from the toolbar and import the **Effects** package:



Once all the files in the `Effects` package have been imported, open the `Effects` folder within **Standard Assets** in your **Project** window. Expand the `ImageEffects` folder and find `ColorCorrectionLookup` within `Scripts`:

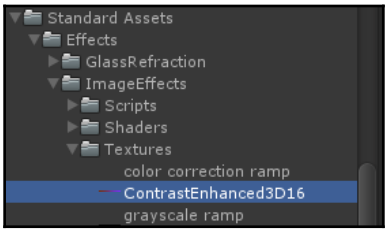


Drag this script onto the camera on **CenterEyeAnchor** within your **OVRPlayerController** object to add it as a component. You'll see an empty field labeled **Based on**; this is where we'll put the lookup texture that the script should use:

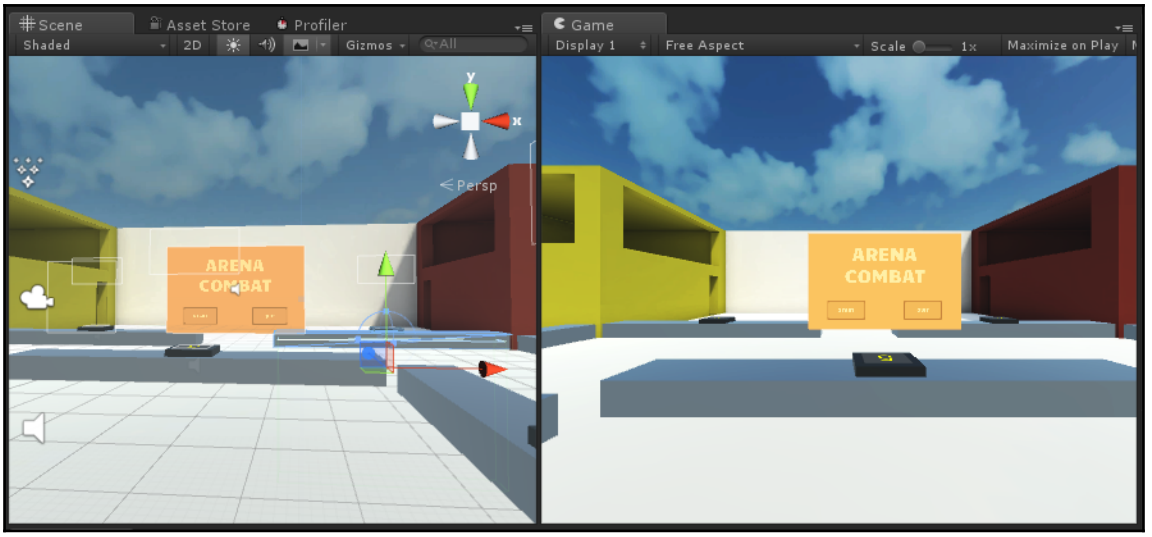


Sampling a lookup texture

To begin, let's add a simple contrast-enhancing lookup texture to this script. Go back to your `ImageEffects` folder and expand the `Textures` subfolder to find the **ContrastEnhanced3D16** texture:



Drag this texture into the open field on your **Color Correction Lookup (Script)** component and click on **Convert and Apply** to activate the effect; you'll immediately see the effect in the **Game** view. In the following screenshot, the same shot is framed in the **Scene** and the **Game** views so you can see the difference in contrast:

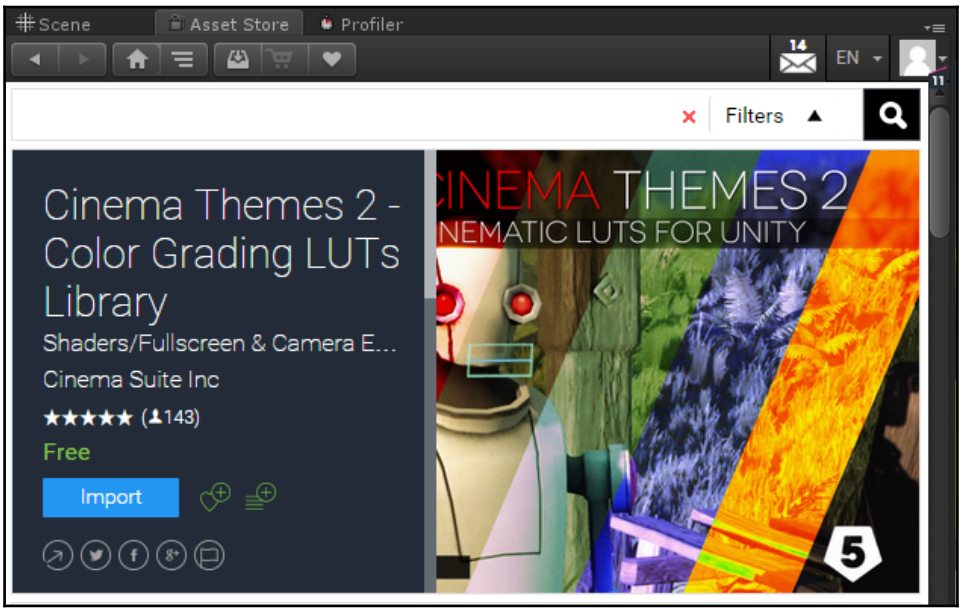


This effect isn't very intense, but it adds a nice amount of contrast and color to our scene. However, if an intense look is what you want, then textures such as this one are just the tip of the iceberg; in the next section, we'll look at some more extreme LUTs that can be used to add dramatic special effects to a scene.

Creating dramatic effects with vibrant LUTs

If you've ever seen the *Terminator* or *Predator* movies, you've seen the high-tech first person view that they give the eponymous characters. Sometimes everything has a red digital hue, and other times it looks like a thermal camera. These are all effects that we can create with image effects.

Open the **Asset Store** window and search for a package called **Cinema Themes 2**, which will look like this:



Import the package and expand the `Cinema Suite` folder in your **Project** window. In the `Cinema Themes` folder you'll find a folder called **LUTs**, which is full of different lookup textures that are all ready to be added to your camera. Drag the one called **Warm3D16** onto the LUT field on your camera and click on **Convert and Apply**. You'll notice the **Game** view turns **Scene** into a warm yellow environment:

Play around with some of the other effects to get a feel for the full capabilities of color grading. Some effects are as subtle as the enhanced contrast texture we used first, and other effects will be so extreme that the scene will be nearly unrecognizable. Some effects are good ones to use constantly throughout an entire level, while others may be best used sparingly for discrete effects only. When you've finished, switch back to the **ContrastEnhanced3D16** filter before moving on.

Changing the appearance of objects with shaders

So far, the graphical work that we've done has all been to do with how the entire scene is rendered. The material so far in this chapter has been relatively code-free and has largely depended on scene configuration. In this section, we'll delve more deeply into the code-oriented side of graphics programming and write a shader, which is essentially a custom set of instructions about how the GPU should render an individual object.

Before we start writing code, it will help to gain a basic understanding of what happens in a shader's execution, so we'll briefly go over the two main processing steps of a shader written in ShaderLab, Unity's shader syntax.

A quick overview of ShaderLab shaders

All ShaderLab shaders go through two main phases: the **vertex** phase and the **fragment** phase. All code in the vertex phase is run once on every vertex on an object, and all code in the fragment phase is performed for every pixel of the object rendered to the screen.

Shader code is unique in that it can't communicate directly with the main code. It's not written in C# and it executes on the GPU, while your other scripts all execute on the CPU. Therefore, it can get tricky to provide information to augment a shader's execution based on external data. It can be done, though, using something called **shader parameters**.

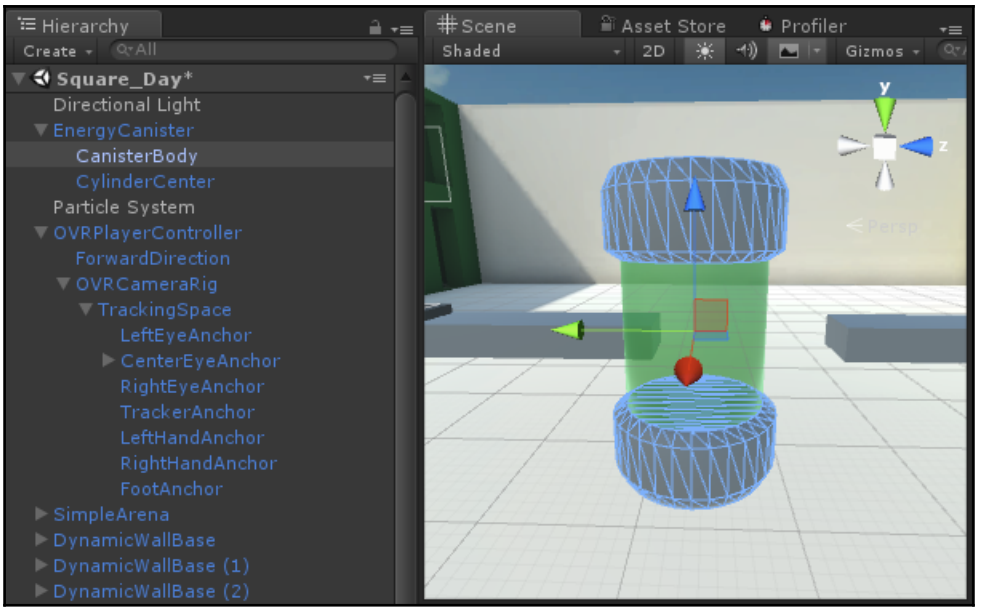
One of the most common kinds of shader parameter is a texture map. This is an image that colors the object by pixel according to the UV coordinates on each vertex. We'll start our first shader off by implementing one of these texture maps.

Importing a textured model

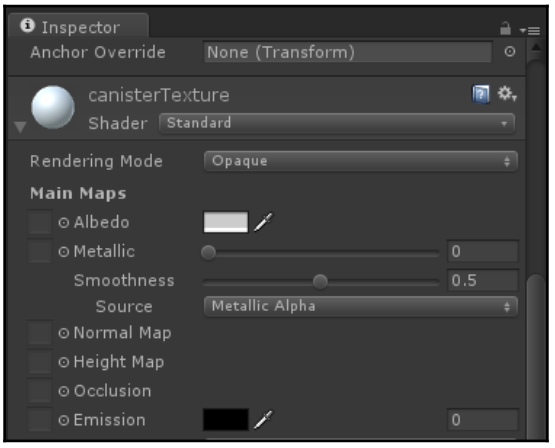
To test a texture shader, we'll need a textured model. Right now, the only thing that uses a texture in our game is the menu canvas, but objects in complete games have textures more often than not. Let's import our first object with a texture into the scene: an energy canister that can be destroyed to get energy orbs.

Import the `EnergyCanister.fbx` file (included with this chapter) into the `Models` folder in your **Project** window. Also, place the `EnergyCanister_Diffuse.png` texture in the `Textures` folder.

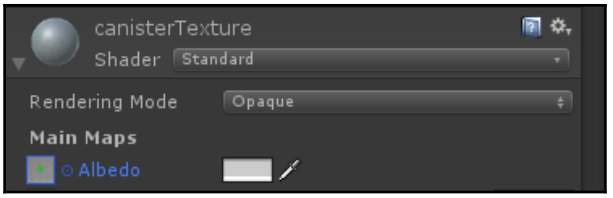
Drag the **EnergyCanister** model into the scene to add an instance of it. Expand the **EnergyCanister** object in the hierarchy and select the **CanisterBody** child by left-clicking it:



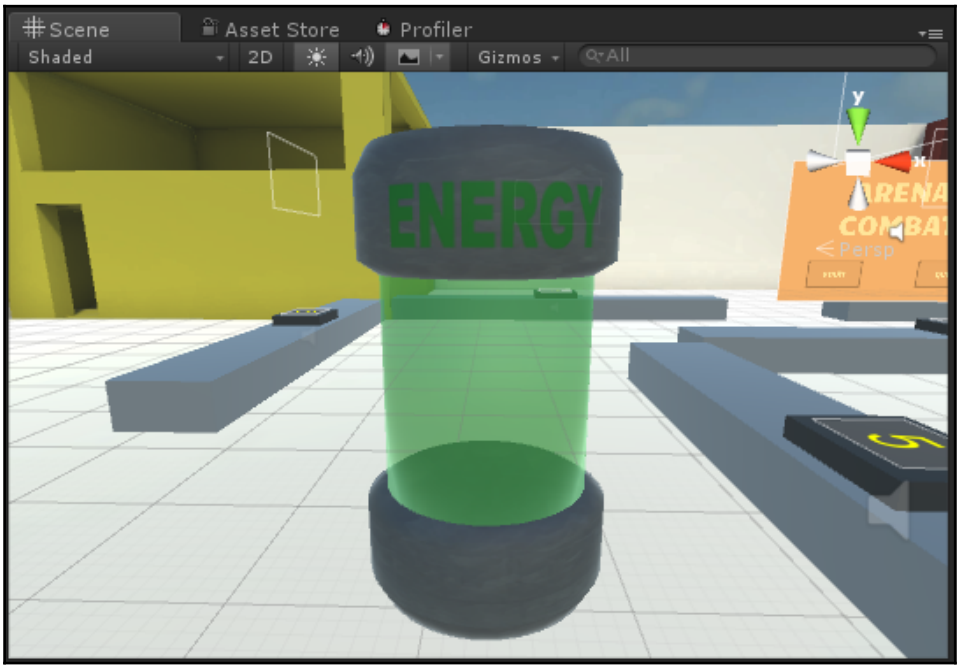
In the **Inspector** window, you'll see the **canisterTexture** material that was imported with the model, currently using the **Standard** shader. Click the arrow on the left side of the material to view its properties:



All the properties that you see here are shader parameters that affect how the canister is rendered. Not many of the properties are being used, though, as you can see by the empty square fields along the left side. Let's add our **EnergyCanister_Diffuse** texture as the **Albedo** map by dragging it from the **Project** window into the empty field to the left of the **Albedo** property:



The canister will now have a brushed metal texture and read **ENERGY**, but without any other maps it looks slightly dull:



To mitigate this dullness and make our energy canister look nicer, we'll add some maps to it. We could use the map slots provided by Unity's Standard shader, but to gain a better understanding of shaders at a fundamental level we'll write our own from scratch and add maps one-by-one.

Create a new folder in your **Project** window called `Shaders`. Right-click on it and select **Standard Surface Shader** in the **Create** list to add a new shader and name it `Basic`. When you open the shader in your code editor you'll notice some basic code already provided. Since we're covering the fundamentals of shaders, replace the default shader definition with the following pared-down version:

```
Shader "Custom/Basic"
{
  Properties
  {
  }
  SubShader
  {
    Tags { "RenderType"="Opaque" }
    CGPROGRAM
    #pragma surface surf Standard fullforwardshadows
    #pragma target 3.0
    struct Input
    {
      float2 uv_MainTex;
    };
    void surf(Input IN, inout SurfaceOutputStandard o)
    {
      o.Albedo = 1;
    }
    ENDCG
  }
  FallBack "Diffuse"
}
```

This is a good amount of code, so before we add any more, let's go over what we have already.

An overview of ShaderLab's fundamental syntax

The `Properties` block at the top of the shader, which is currently empty, will eventually contain definitions for all external information that is used by the shader, such as texture maps, user-defined colors, and other arbitrary numerical modifiers such as shininess.

The `SubShader` block contains the “meat” of the shader. Everything that affects the actual rendering of the shader goes in one of these blocks. It's possible to have multiple `SubShader` blocks to support older graphics cards that may not be able to handle everything you want to do in your shader; if multiple `SubShader` blocks are defined, the GPU will go through them from top to bottom until it reaches one that it can use.

The `Tags` section lets us define several directives for how Unity renders this particular object. Right now, the only tag is `RenderType`, which is a way to classify categories of objects that you might write a replacement shader for later (that is, a replacement shader that only replaces objects with the `Opaque` tag).

Within the `SubShader` block, there's a block encapsulated by the `CGPROGRAM` and `ENDCG` keywords. The ShaderLab syntax is a high-level abstraction of common low-level shader syntaxes such as Cg, so you can think of this block as a place to interact with low-level data directly. This is where the vertex and fragment phases are processed and contribute to the overall rendering pipeline that we covered earlier.

The pragma statements right beneath the `CGPROGRAM` keyword tell the Cg/HLSL code how it should run. The first line says we want this shader to use Unity's `Standard` lighting model with shadows supported from any light source. The second line tells this shader to target the 3.0 shader model, currently the highest supported shader model in Unity.

The `Input` struct and the `surf` function go hand-in-hand; the `Input` struct lets us define what information is passed into the `surf` function, which in turn applies the data to the rendered pixels. The `Input` struct can't be completely empty, so right now we have a UV coordinate for the main texture as the only member. We don't use this in the shader yet, but we will later. The `surf` function is where most of our custom shader code will exist, and where we'll apply texture maps to each pixel. Right now, it simply sets the output `Albedo` value to `1`, which will make any objects with this shader appear white.

Finally, the `FallBack` directive. This is rarely used, but if every `SubShader` block is unsupported by your video card, the shader name written here will be used instead. In our case, the standard basic `Diffuse` shader will be the fallback (but this is very unlikely to happen if you're using a GPU that supports VR in the first place).

Writing your first shader

Now that we've established the basic structure of a shader, let's start adding some features to it. First, we'll add a simple `Color` property so that we can use Unity's color picker to change the hue of an object with this shader.

Add the following line to your `Properties` block:

```
Shader "Custom/Basic"
{
  Properties
  {
    _Color ("Color", Color) = (1,1,1,1)
```

```
    }
    ...
}
```

The `_Color` keyword is the name of the property and the text in quotations is what it will be displayed as in the Unity **Inspector**. The second value in the parentheses is the property type, and the final set of parentheses is the default value in RGBA format.

Next, we need to declare a new variable in the `SubShader` block to store the value of our `_Color` property. Add the following definition before your `Input` struct:

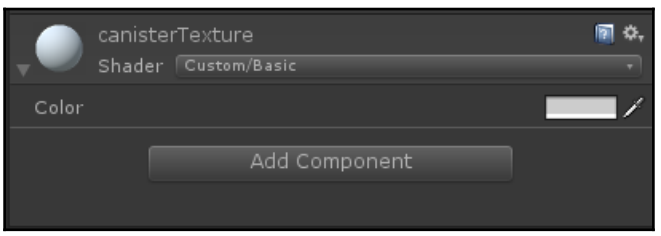
```
fixed4 _Color;
struct Input
{
  float2 uv_MainTex;
}
```

To make use of the new property, we'll have to assign its value in the `surf` function. Change the existing line to set the `Albedo` value to the value of the `_Color` property instead of `1`:

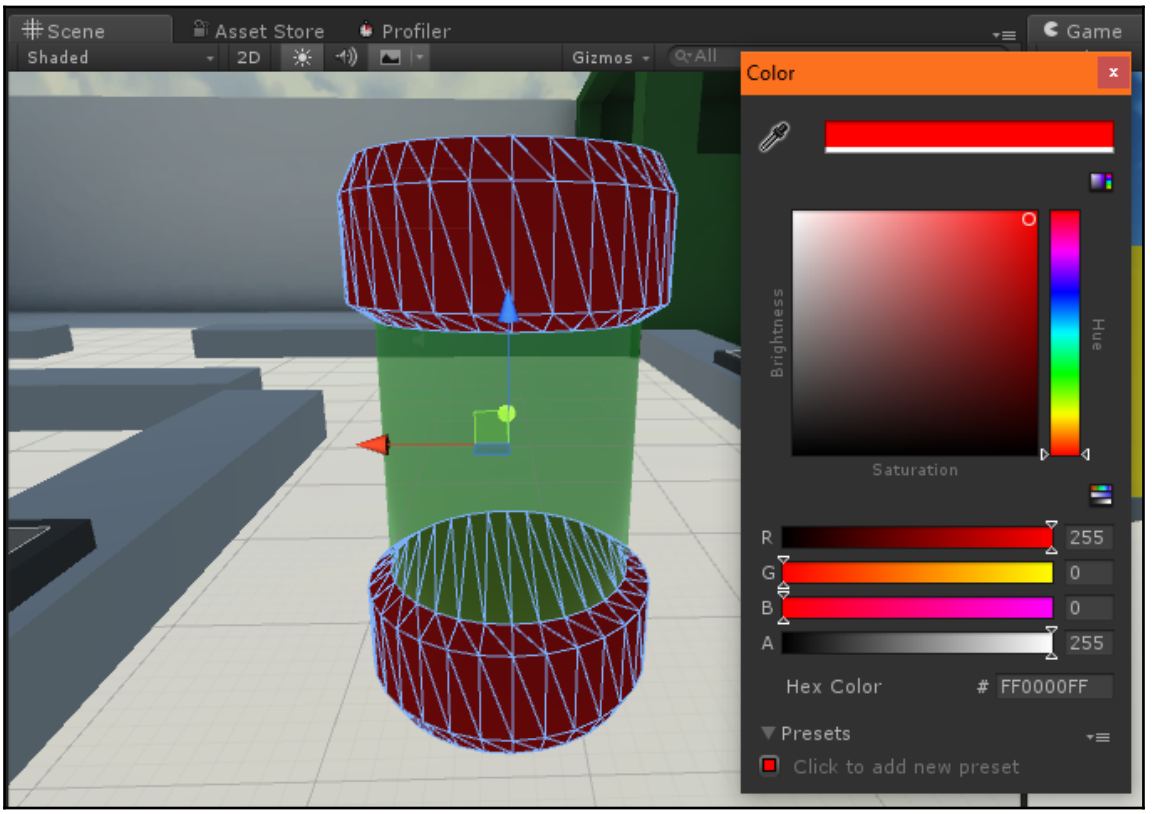
```
void surf(Input IN, inout SurfaceOutputStandard o)
{
  o.Albedo = _Color;
}
```

Now that we have enough code in the shader to see it in action, let's apply it to our canister so we can visualize our progress. Highlight the **CanisterBody** object that is a child of the **EnergyCanister** prefab.

In the **Inspector** window, scroll to the material component of **CanisterBody** and give it the new shader by selecting **Custom/Basic** from the **Shader** menu:



Open the color selector by clicking the box to the right of the **Color** property. As you change the color, you'll see the **EnergyCanister** appearance change in the **Scene** view (but without a texture, since we don't support textures in our basic shader yet):



Defining a diffuse texture property

Next, we'll create a property to store the diffuse texture of the object. A diffuse texture is typically meant for per-pixel color on an object, while other kinds of textures influence lighting or specularity.

Add the following definition to your `Properties` block:

```
Shader "Custom/Basic"
{
  Properties
  {
    _Color("Color", Color) = (1,1,1,1)
    _MainTex("Diffuse", 2D) = "white" {}
  }
  ...
}
```

The type for this property is `2D`, which means every UV will correspond to a two-dimensional coordinate on the texture.

Remember your `Input` struct, which has been in the shader since the beginning? The only property that comprises it currently, `uv_MainTex`, is what provides UV coordinates to the shader so that it can get the proper pixel from the texture during the fragment phase.

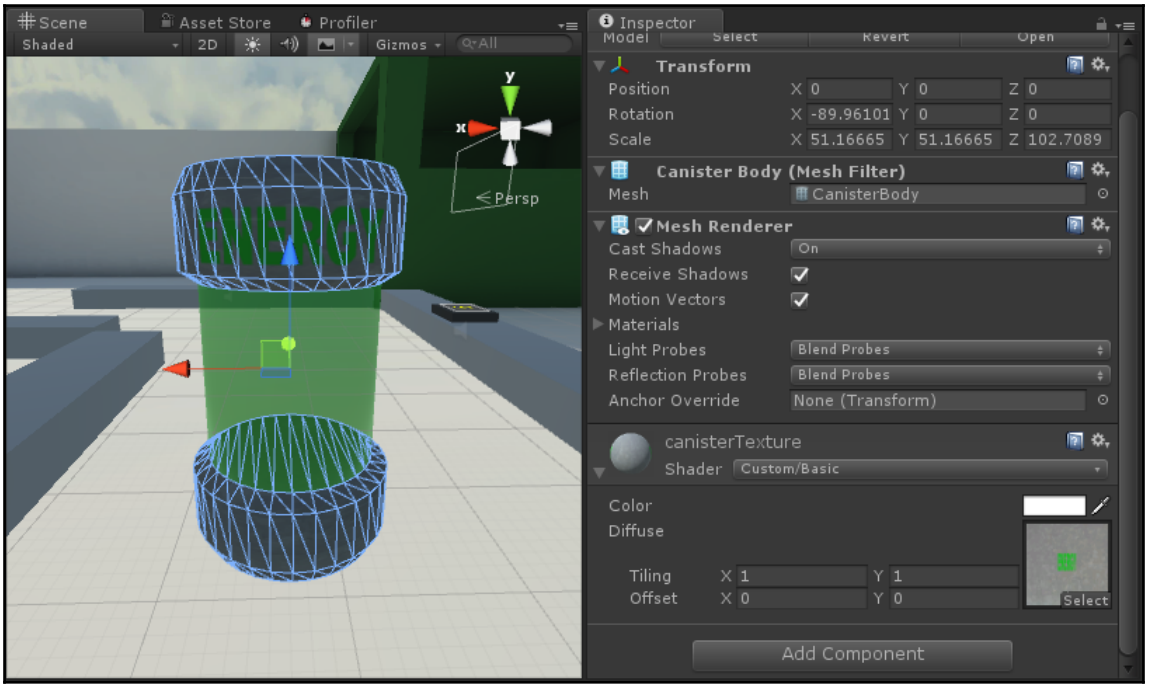
The only thing we still need is a reference to the texture to sample pixel values from the provided UVs. Add the following line before your `Input` struct:

```
fixed4 _Color;
sampler2D _MainTex;
struct Input
{
  float2 uv_MainTex;
}
```

Now modify the line in the `surf` function once again to multiply the `_Color` property by the color of the sampled pixel from the main texture:

```
void surf(Input IN, inout SurfaceOutputStandard o)
{
  o.Albedo = tex2D(_MainTex, IN.uv_MainTex) * _Color;
}
```

Return to the **CanisterBody** inspector window and make sure your **EnergyCanister_Diffuse** texture is in the newly-created **Diffuse** property slot. The object in your scene will now show the diffuse texture it was modeled with aligned properly according to its UVs:



Congratulations! You've just created your first shader, a basic lit texture/color shader. In the next section, we'll add some complexity to the shader by adding a secondary texture for lighting, called a **normal map**.

Adding a normal map property to a shader

Diffuse textures are simple and easy to apply, but objects that only have a diffuse texture without secondary supplementary textures run a high risk of appearing flat, unrealistic, or plastic.

In this section, we're going to add a new texture property to our basic shader called a **normal map**. Normal maps affect the angle at which light hits the pixel, and can create the illusion of small precise details in the model, such as cracks in wood grain or patterns in industrial steel.

The “colors” of a normal map texture aren't actually treated as colors at all; the red, blue, and green channels of each pixel are used to store *X*, *Y*, and *Z* coordinates of the normal vector, which defines the direction in which light is reflected. Obviously, these values can't be drawn or painted digitally in the way that diffuse textures can be, but there are a couple of methods for generating them.

The first method involves capturing the normals of a more detailed version of a mesh and then applying those normals to a mesh with less detailed geometry. In our case, we don't have a high-poly version of our canister that we can capture lighting data from, but this is a good technique to keep in mind if you need to cut down on the number of polygons rendered in your game without sacrificing too much quality.

The other method, which we'll be using in this section, is to algorithmically analyze a diffuse texture and make a “best guess” normal map by interpreting differences in color as small geometric details. This is the best method to use for details that you can easily see in a diffuse texture but can't easily model, such as natural details or small decorations.

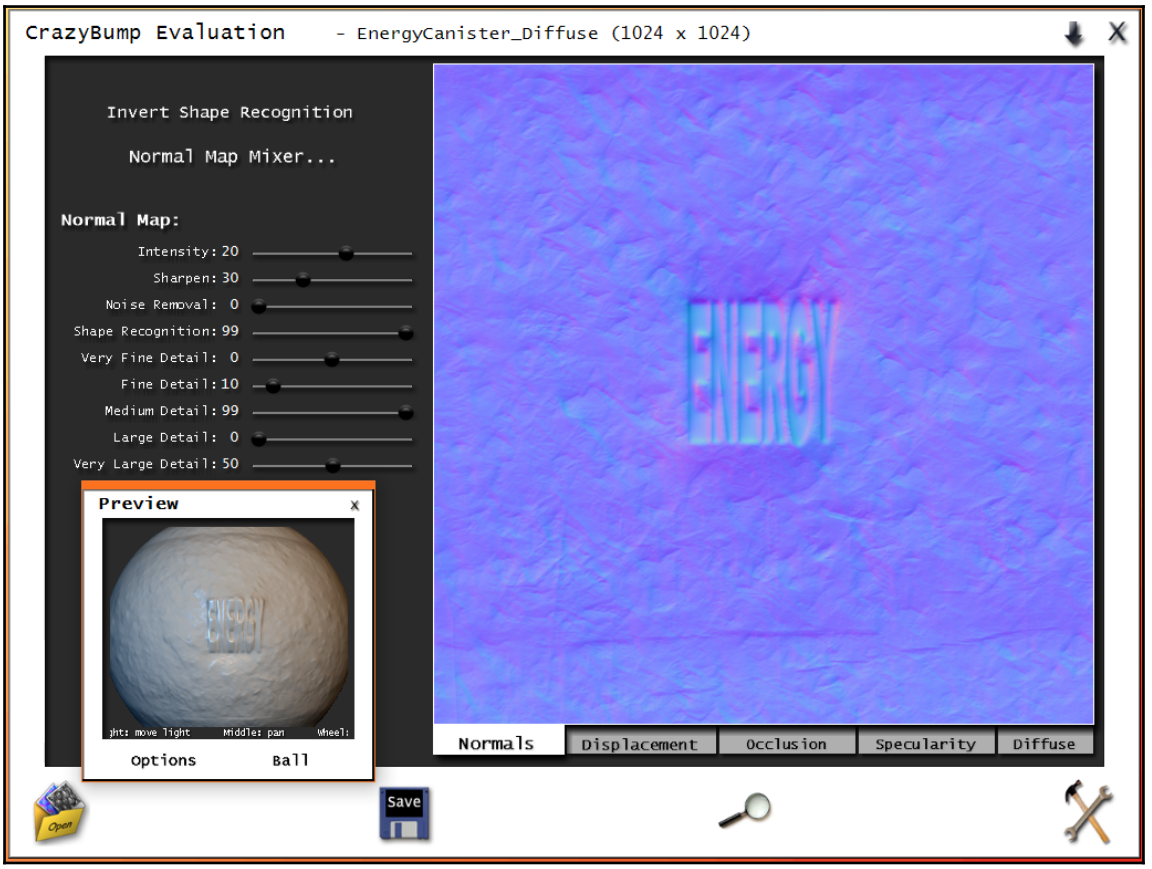
Generating a normal map

There are several programs available for generating many different texture maps, but the one we'll use is called CrazyBump. Download CrazyBump from `www.crazybump.com` and install it. This will begin a demo period of 30 days, after which a license is needed. After the installation completes and presents the main menu, click the folder icon in the lower-left area of the window, select **Open photograph from file**, and navigate to the **EnergyCanister_Diffuse** texture in your project.

CrazyBump will then ask you to pick one of two shapes to determine whether the normals are extruded or intruded:

Select the shape on the left; don't worry if it doesn't look quite right, this only represents the starting values of the map. We'll be able to tweak the effect on the next screen.

Modify the settings of the map to match the values in the following screenshot:



Click the **Save** icon at the bottom of the window and save the map to the same directory as the diffuse texture; name it `EnergyCanister_Normal`.

Now that we've got a normal map ready to apply, we just need to add a few lines to handle it in our shader. Open your `Basic` shader again and add a property at the top for a normal map:

```
Shader "Custom/Basic"
{
  Properties
  {
    _Color("Color", Color) = (1,1,1,1)
    _MainTex("Diffuse"), 2D) = "white" {}
    [Normal]_NormalTex("Normal", 2D) = "white" {}
```

```
    }
    ...
}
```

The `[Normal]` attribute before the property name doesn't change the functionality of the map, but it will warn you if you drag a texture that's not a normal map into the slot.

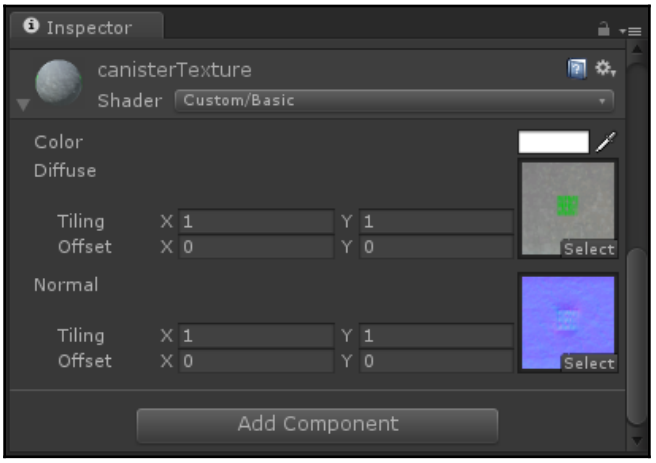
We don't need a new definition in our `Input` struct because the normal map will use the same UVs as the diffuse texture. However, we do need a new sampling reference. Add the following line underneath your other references:

```
fixed4 _Color;
sampler2D _MainTex;
sampler2D _NormalTex;
struct Input
{
  float2 uv_MainTex;
}
```

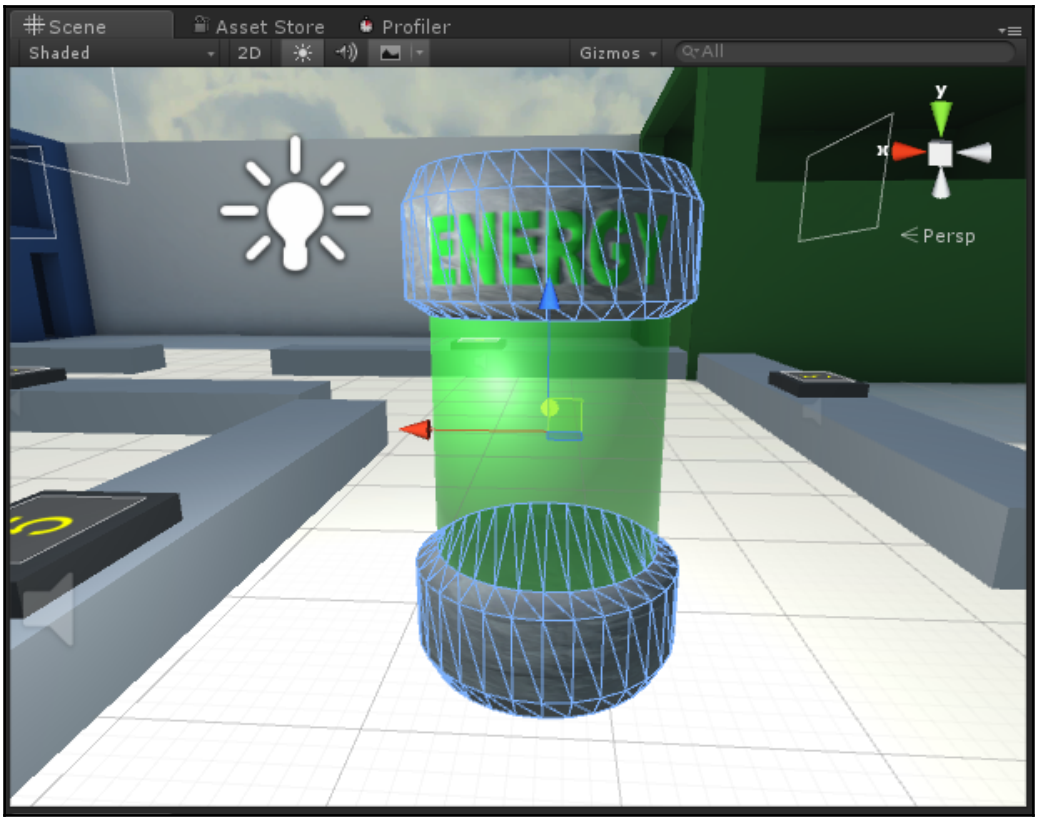
Finally, add a line to the `surf` function that sets each normal to the value sampled from our normal map:

```
void surf(Input IN, inout SurfaceOutputStandard o)
{
  o.Albedo = tex2D(_MainTex, IN.uv_MainTex) * _Color;
  o.Normal = UnpackNormal(tex2D(_NormalTex, IN.uv_MainTex));
}
```

Now go back to the **CanisterBody** material properties in **Inspector** and drag the new `EnergyCanister_Normal` map into the **Normal** slot:



Rotate your canister in the scene to see how the light hits it differently with this map applied. The text looks embossed and the steel shines along the grain. In this screenshot, we've positioned a point light directly next to an instance of **EnergyCanister** so you can see the detail clearly:



The effect that your new normal map has on the object is very obvious, but it's just the tip of the iceberg when it comes to what kinds of maps you can apply. Specular maps affect the shininess, or specularity, of pixels; height maps displace geometry per vertex based on their values. The world of shader programming is a vast expanse, so don't be afraid to experiment and try to replicate lighting effects that we see in real life with a combination of maps and surface functions.

Summary

In this chapter, we started off by diving into how a renderer works at a low level in order to glean a deep understanding of what's possible with graphics programming. After looking at the details, pros and cons of the forward and deferred rendering method, we created a scene that took advantage of the deferred rendering method by rendering several real-time lights without an additional rendering pass per light.

After covering the technical breakdowns, we took a brief look at several techniques you can use to give the graphics in your game a more unique, polished appearance. We started with color grading, an image effect that colors all rendered pixels uniformly based on a modified color matrix.

Finally, we looked at the broadest area of graphics programming: shaders. Unlike screen-space effects, shaders are assigned to specific meshes in a scene and can render them based on a combination of internal geometric data and parameters like textures. We created a basic shader that handles color tinting, coloring from a diffuse texture, and lighting based on a normal map. We generated the normal map using CrazyBump, a popular art tool for generating lighting detail based on diffuse patterns.

As we reach the end of the book, there's only one major feature left to implement, and arguably it's the largest of all: multiplayer networking. The last code we add to our game will enable your players to see other people and compete to claim victory over the other teams.

9
Bringing Players Together in VR

VR can be a lonely place. It might not seem like it at first, but it's more isolating than any other medium because it monopolizes your senses, attention, and ultimately your reality. That's why multiplayer experiences are so impactful in virtual reality, because you've been transported somewhere else with other real people that you can share the experience with.

In this chapter, we'll dive into the technology and techniques that make multiplayer experiences possible. In doing so, we'll bring other players into the arena and set the stage for a team-based competition that includes all of the features and mechanics you've designed throughout this book.

This chapter will cover the following topics:

- Creating a lobby space for joining matches
- Creating a networked game
- Using Unity's matchmaker system
- Synchronizing data in multiplayer matches

Creating a lobby space for joining matches

Up to this point, the player has existed in one of our two arena scenes perpetually. However, since we'll be hosting each match of the game in its own instance of one of these levels, we need a staging area where players can specify what match they join and at what level. We'll call this the **lobby**.

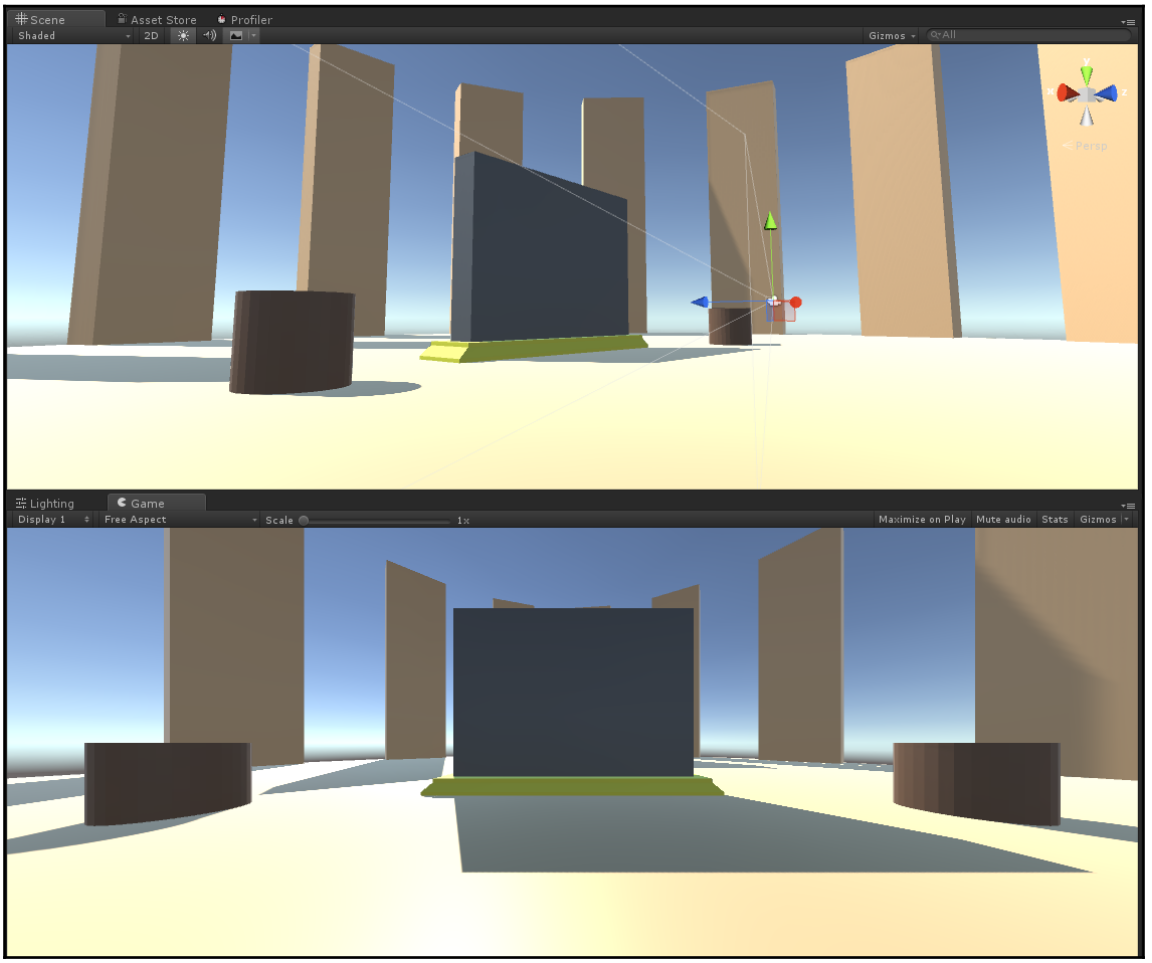
Our lobby scene will handle player interaction for hosting or joining a game; it will take specifications from the player and then load the proper scene before connecting them to the other players in the match.

Setting up the lobby scene

We'll want to move our menu to the lobby scene, so before doing anything else, create a prefab of the **MenuCanvas** object in one of your scenes and delete the instance from both. Create a new scene and name it `Lobby.unity`. Remove the default **Main Camera** object from the hierarchy and drag an instance of `OVRCameraRig` from the `Prefabs` folder within the `OVR` directory into the scene.

We're not using the **OVRPlayerController** prefab in this case because we don't actually need our player to move around the level; we can have them use a menu from a stationary location.

Import the `Lobby.fbx` model into the `Models` folder of your project and drag an instance of it into your scene. Adjust the position of the **OVRCameraRig** and the **Lobby** objects so that the player is looking at the center screen from 5 meters away (five cube lengths) and 1.7 meters above the floor. The view from the **OVRCameraRig** should look like the following in the **Game** and **Scene** windows:



Next, we'll put our menu on the lobby screen. Drag an instance of the **MenuCanvas** prefab into the scene, and position and scale it so that it fits inside the screen in the model:

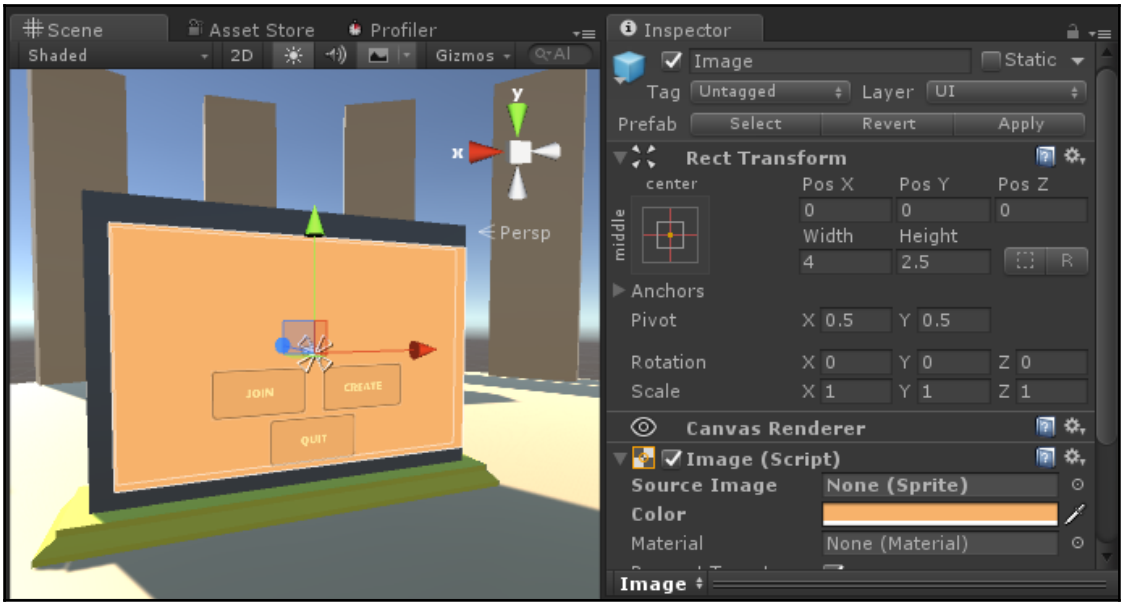
Of course, we'll have to edit this menu's interface and functionality before the lobby scene is usable; the menu will no longer simply enable player movement, and it will be responsible for finding or creating a game on the network and loading the proper scene for it.

Make two copies of the **START** button and change their text components to read **CREATE** and **JOIN**, and then delete the **START** button. Arrange the remaining buttons as shown in the following screenshot:

Because the buttons no longer have a reference to the player, the only callbacks that will be present on this instance of **MenuCanvas** are audio callbacks. Next, we'll create new callbacks for our **CREATE** and **JOIN** buttons. Before we do that, though, we need to create new menu canvases for creating and joining games.

Creating the Create menu

We'll focus on the menu for creating a new game on the server first. Duplicate the **MenuCanvas** to use as a starting object, position it slightly in front of the **MenuCanvas**, and name it `CreateCanvas`. Use the eyedropper tool next to the **Color** property to set the color to be the same as the texture's background and then remove the texture reference:

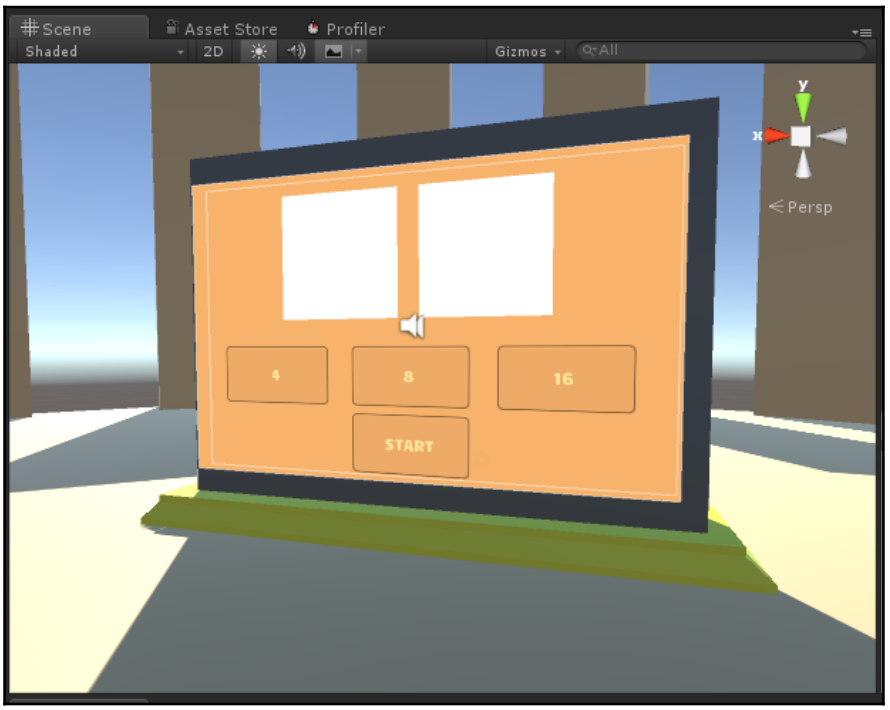


Then, position the three buttons on the same row, aligned with the **Y** position of the **JOIN** and **CREATE** buttons. Change their text components to `4`, `8`, and `16` from left to right. Also, add a fourth button at the bottom that says **START**. The menu should now look like the following:

These numbered buttons will represent the maximum number of players in the created game.

The other specification we'll let the player make is what map the match is played on. To do this, we'll create two new images that can be highlighted above the numbered buttons.

Create two new **Image** objects within the **CreateCanvas** object. Set their **Width** and **Height** to `256` and their **Scale** to `0.004`. Position them as shown in the following screenshot:



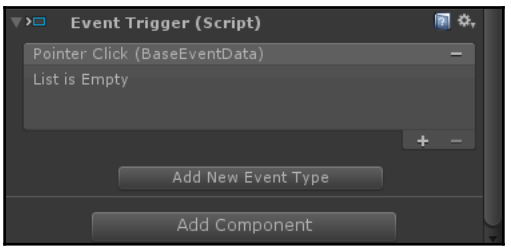
Import `SquareDayThumbnail.png` and `SquareNightThumbnail.png` (included with this chapter) into your project's `Textures` folder. Left-click on each once to display their import settings in the **Inspector**, and set the **Texture Type** of each to **Sprite (2D and UI)**. You can select both at the same time to batch the import options since they'll be the same for both.

Drag the **SquareDayThumbnail** texture onto the **Source Image** property of the left **Image** object and drag **SquareNightThumbnail** onto the right one:

Now that we've got our menu for creating network games, we'll need to write our script so that we can link our buttons to actions. We'll do this with a new script and several **Event Trigger** components.

Adding event triggers to the Lobby menu

Even though the **Image** objects don't come with a built-in UI action callback list, we can add callback functions to any UI element by adding an **Event Trigger** component. Left-click on the left **Image** object for the daytime map to display it in the **Inspector** panel, and then click on **Add Component** and select **Event Trigger**, as shown in the following screenshot:



This will create an empty callback list on this **Image**, which we can add different event listeners to. Repeat this process for the right-hand map image so that both have empty **Event Trigger** components (you can do this as a batch action too). There should already be **Event Trigger** components on the capacity buttons since we created them from a duplicate of the **START** button; we'll add additional functions to them later.

Before we add any functions to our triggers, we need to define them in code. Instead of modifying our `GameManager` script, let's leave it as-is and start from scratch with a new script called `NetworkGameManager`. Create the script and open it in your editor.

Create two new functions, `SetActiveMap` and `SetActiveCapacity`, both of which will take a `GameObject` parameter:

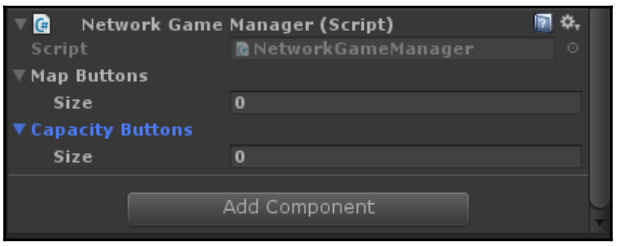
```
public void SetActiveMap(GameObject mapButton)
{
}
public void SetActiveCapacity(GameObject capacityButton)
{
}
```

The `GameObject` parameter will be passed by our UI callback function and will highlight the selected button while dimming the others. For this to work, though, we need references to the other buttons so that we can dim them.

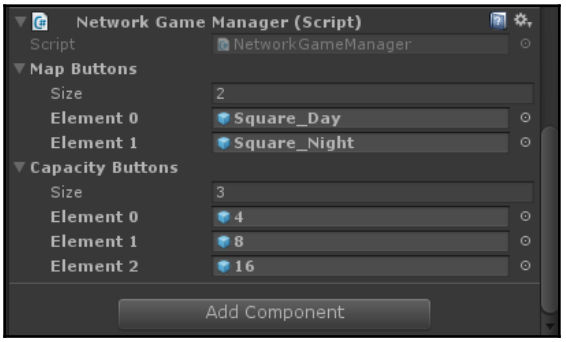
Declare two new lists at the top of the script. One of the new lists will hold the map buttons and the other list will hold the player capacity buttons:

```
public class NetworkGameManager : MonoBehaviour
{
  [SerializeField] private GameObject[] mapButtons;
  [SerializeField] private GameObject[] capacityButtons;
  ...
}
```

Drag the **NetworkGameManager** script to the **MenuCanvas** in the scene to add it as a component. Left-click on the **MenuCanvas** object to display it in **Inspector** and you'll see the two serialized arrays we just declared, both currently with a count of **0**:



Set the **Size** property of **Map** Buttons to `2`, and the **Size** property of **Capacity Buttons** to `3`. New fields will appear that you can drag each button into. Drag the map images into each map slot, and the capacity buttons into each capacity button slot from the `CreateCanvas` object, as shown in the following screenshot:



The last thing we'll need is two variables, one to store each active setting. Declare the following variables underneath your serialized arrays in the `NetworkGameManager` script:

```
public class NetworkGameManager : MonoBehaviour
{
  [SerializeField] private GameObject[] mapButtons;
  [SerializeField] private GameObject[] capacityButtons;
  private GameObject activeMap;
  private GameObject activeCapacity;
  ...
}
```

At this point, we have everything we need to fill in our `SetActiveMap` and `SetActiveCapacity` functions. Add a line to both to assign the passed in object as the active object:

```
public void SetActiveMap(GameObject mapButton)
{
  activeMap = mapButton;
}
public void SetActiveCapacity(GameObject capacityButton)
{
  activeCapacity = capacityButton;
}
```

We're going to denote inactive buttons by making their representative images transparent. To do this, we'll need to include a `UI` class in our `using` directives. Add the following line to the directives at the top of your script:

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
```

We'll also need a color definition for a neutral transparent color. Add the following variable `inactiveColor` underneath the other member variables:

```
public class NetworkGameManager : MonoBehaviour
{
  [SerializeField] private GameObject[] mapButtons;
  [SerializeField] private GameObject[] capacityButtons;
  private GameObject activeMap;
  private GameObject activeCapacity;
  private Color inactiveColor = new Color(1,1,1,0.5f);
  ...
}
```

Next, add a loop to the `SetActiveMap` function to set all nonactive map buttons in the array to use `inactiveColor`:

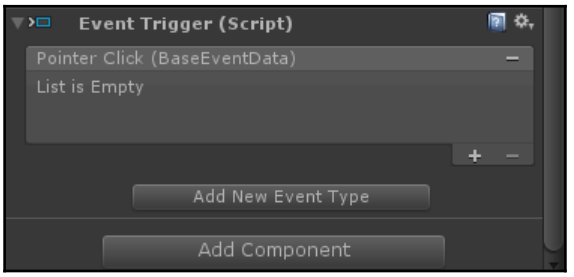
```
public void SetActiveMap(GameObject mapButton)
{
  activeMap = mapButton;
  for(int i = 0; i < mapButtons.Length; ++i)
  {
    if(mapButtons[i] != activeMap)
    {
      mapButtons[i].GetComponent<Image>().color = inactiveColor;
    }
    else
```

```
      {
        mapButtons[i].GetComponent<Image>().color = Color.white;
      }
    }
  }
```

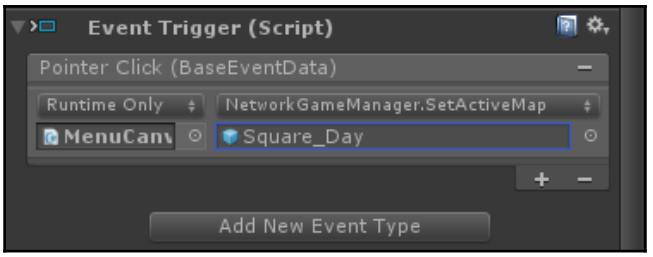
Implement a similar loop in the `SetActiveCapacity` function:

```
public void SetActiveCapacity(GameObject capacityButton)
{
  activeCapacity = capacityButton;
  for(int i = 0; i < capacityButtons.Length; ++i)
  {
    if(capacityButtons[i] != activeCapacity)
    {
      capacityButtons[i].GetComponent<Image>().color = inactiveColor;
    }
    else
    {
      capacityButtons[i].GetComponent<Image>().color = Color.white;
    }
  }
}
```

Now that our functions have some testable functionality, let's hook them up to the event triggers we added earlier and configure the UI camera and event system. Left-click on the **Square_Day** map image object and scroll to the **Event Trigger** component in the **Inspector**. Click on **Add New Event Type** and select **PointerClick**:



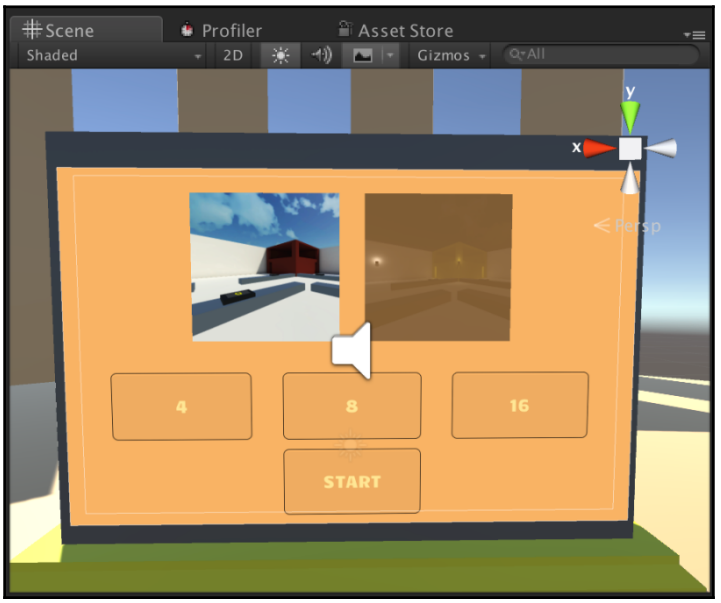
Click on the + button to add a new function call and drag the **MenuCanvas** object that holds the **NetworkGameManager** component into the object field. Select the **SetActiveMap** function from the list to the right of the object field and drag the **Square_Day** button object into the parameter field:



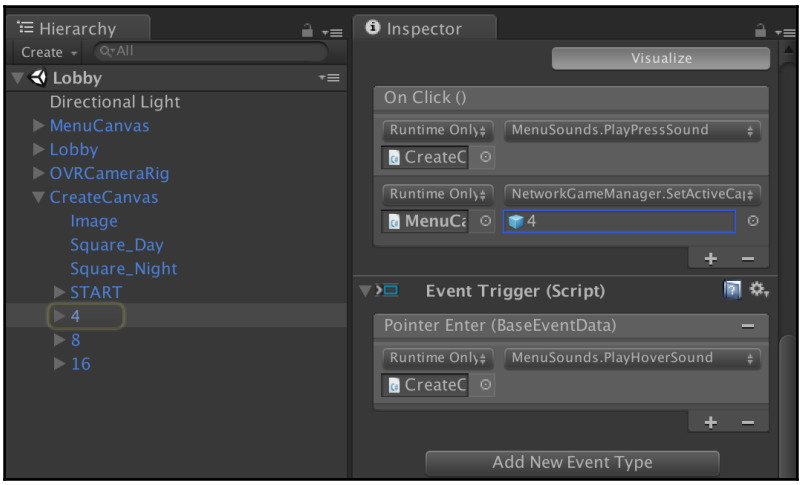
Repeat this process with the **Square_Night** button.

Lastly, we need to enable this UI for interaction in VR. Create a UI camera on the **OVRCameraRig** object, add a **BasicLookInputModule** to the scene's **EventSystem** object, and specify the newly created UI camera as the **Event Camera** on each canvas in the scene. You've done all of this earlier, but if you want a refresher, these same steps are explained in detail in `Chapter 6`, *Adding Depth and Intuition to a User Interface*.

Press Play to test the selecting logic. When you press the spacebar while looking at either of the maps, it should make the other one semitransparent, as shown in the following screenshot:



Now that we've finished map selection, let's move onto player capacity selection. Add a new callback to the **OnClick** events on each capacity button and drag the **MenuCanvas** object into the field so that you can select **SetActiveCapacity** and drag it onto the corresponding button objects:



Note that there may be empty **OnClick** calls from the references that were there when you saved it as a prefab in your day scene; feel free to add the new calls where those were or simply delete them.

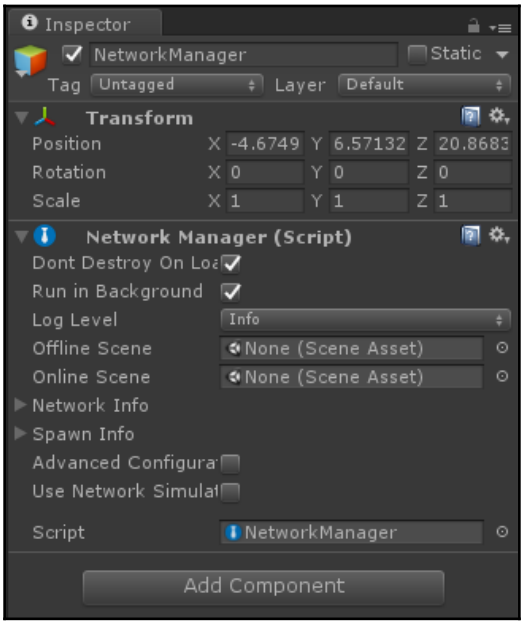
Once every capacity button has a **SetActiveCapacity** callback, the menu should be fully interactive. Test it and make sure you can select both a map and a player capacity:

Now that we've got a way for a player to configure a game, we'll write the functions that actually create it.

Creating a networked game

The first thing you'll need to facilitate your networked game is a **NetworkManager** component. This component handles most of the functionality associated with multiplayer settings and states, including connecting, disconnecting, hosting, joining, and spawning.

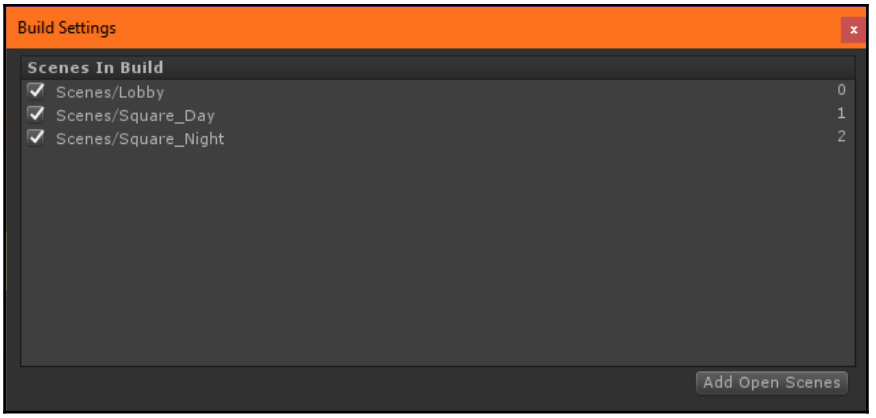
Create a new empty object in your lobby scene and name it `NetworkManager`. Add a **NetworkManager** component to it in **Inspector**:



Ensure that the first property, labeled **Dont Destroy On Load**, is checked. This property prevents the object that this component is attached to from being removed between scenes, so it will still be present in your game even if you switch to a completely different scene (that is, one of the maps). This will make it easier for us to enter a new scene and still have access to the **NetworkManager** component's information, and we can use it when the match is over to return to the original scene.

Notice the two fields labeled **Offline Scene** and **Online Scene**. The **Offline Scene** is the scene loaded whenever the player isn't engaged in a match, and the **Online Scene** is the scene the game loads as soon as a connection is established.

In order to add scenes to these fields, they must be manually added to the list of scenes in the **Build Settings** menu. Open **Build Settings** from the **File** menu on the toolbar and you'll see a list labeled **Scenes In Build** at the top. Drag each scene from your `Project` folder into this list to add them:



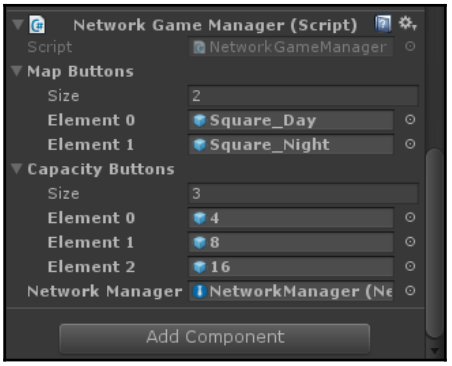
Now that you can use your scenes in the **Network Manager** component, drag the **Lobby** scene from your **Project** window into the **Offline Scene**, field and the **Square_Day** scene into the **Online Scene** field:

The **Square_Day** scene will serve as the default value for the **Online Scene** field but, because we'll be loading different scenes based on the map our player selects, our **Online Scene** property will need to be updated in code by our `NetworkGameManager`.

Open your `NetworkGameManager` class and create a new serialized field at the top to store a reference to our **NetworkManager** component:

```
using UnityEngine,Networking;
public class NetworkGameManager : MonoBehaviour
{
  [SerializeField] private GameObject[] mapButtons;
  [SerializeField] private GameObject[] capacityButtons;
  [SerializeField] private NetworkManager networkManager;
  private GameObject activeMap;
  private GameObject activeCapacity;
  private Color inactiveColor = new Color(1,1,1,0.5f);
  ...
}
```

In the Unity Editor, highlight the **MenuCanvas** in your **Inspector** and drag the newly created **NetworkManager** object into the **Network Manager** field:



Next, create a new function in your `NetworkGameManager` class called `StartHosting`:

```
public class NetworkGameManager : MonoBehaviour
{
  ...
  public void StartHosting()
  {
  }
}
```

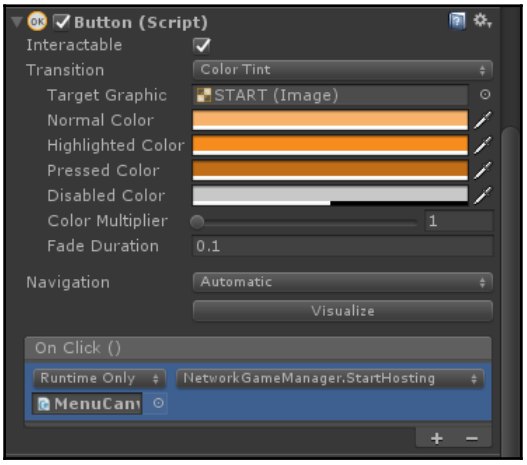
In the first line of the `StartHosting` function, we'll check if there's an active map, and if so, we'll set the `onlineScene` property to match the `activeMap` name:

```
public void StartHosting()
{
  if(activeMap != null)
  {
    networkManager.onlineScene = activeMap.name;
  }
}
```

Next, we'll add a line to make the player calling this function the host of a new game:

```
public void StartHosting()
{
  if(activeMap != null)
  {
    networkManager.onlineScene = activeMap.name;
  }
  networkManager.StartHost();
}
```

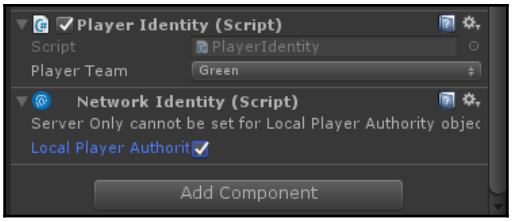
Finally, link your **START** button to the `StartHosting` function by adding it as an **On Click()** callback:



Before we test this, we need to change a few things about the map scenes. Open the **Square_Day** scene and remove the **GameManager** and **MenuCanvas**; all of the responsibilities that used to belong to these systems are now handled by the lobby, so they don't need to be in the map scene anymore.

Second, we need to remove the instance of the **Player** prefab from the scene. The **NetworkManager** component takes care of player spawning when a connection is established, so the prefab doesn't need to exist beforehand. However, we will need to specify the player prefab to the network manager so that it knows what to spawn.

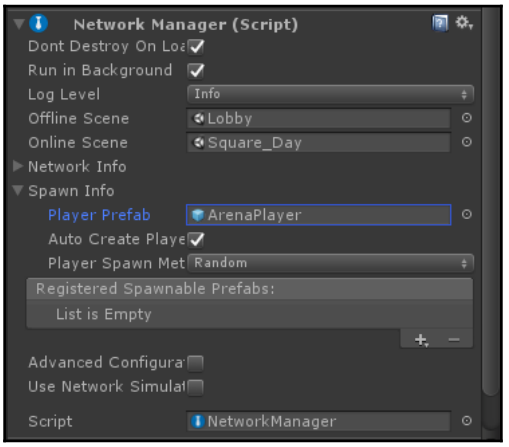
Enable the **OVRPlayerController** and **FiringSystem** scripts on the **Player** prefab so that they no longer start as disabled by default. You'll also need to add a component to make the prefab compatible with Unity's networking, so use the **Add Component** button to add a **NetworkIdentity** component. Check the box labeled **Local Player Authority**:



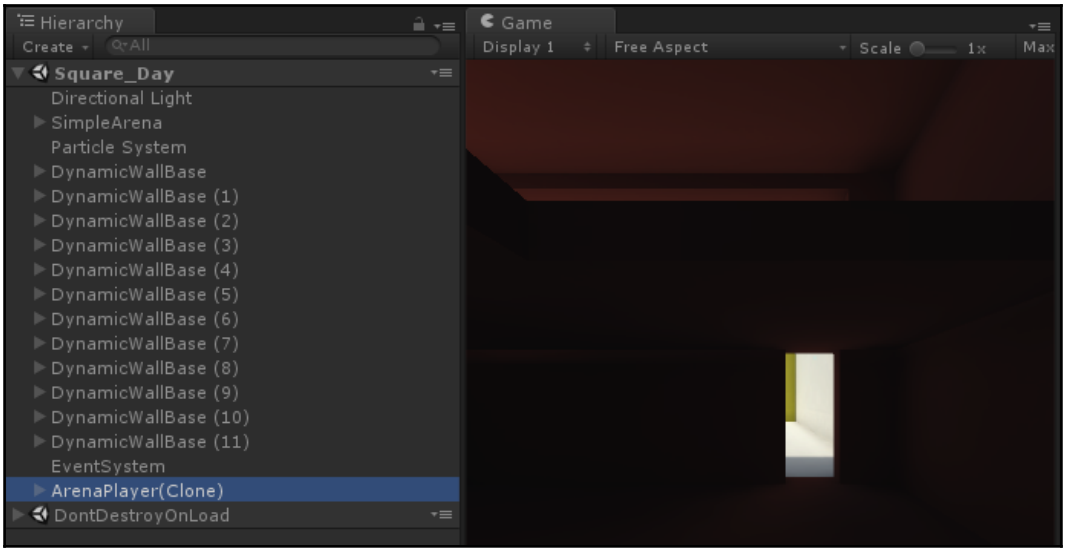
Rename the root object of the prefab to **ArenaPlayer** and save it in your `Prefabs` folder by dragging it from the hierarchy into your **Project** window, and then delete this instance from the scene.

Repeat this process with the **Square_Night** scene as well (but you don't need to save another player prefab; you can just remove the old prefab from the scene).

Now go back to your **Lobby** scene and expand the **Spawn Info** area of the **NetworkManager** component. Drag the **ArenaPlayer** prefab into the field labeled **Player Prefab**:



Finally, we're ready to test the `StartHosting` function. Start your **Lobby** scene and click on the **START** button on your **CreateCanvas** object. The **Square_Day** scene will be loaded and you'll instantly be able to control the **ArenaPlayer** prefab spawned by the **NetworkManager**:

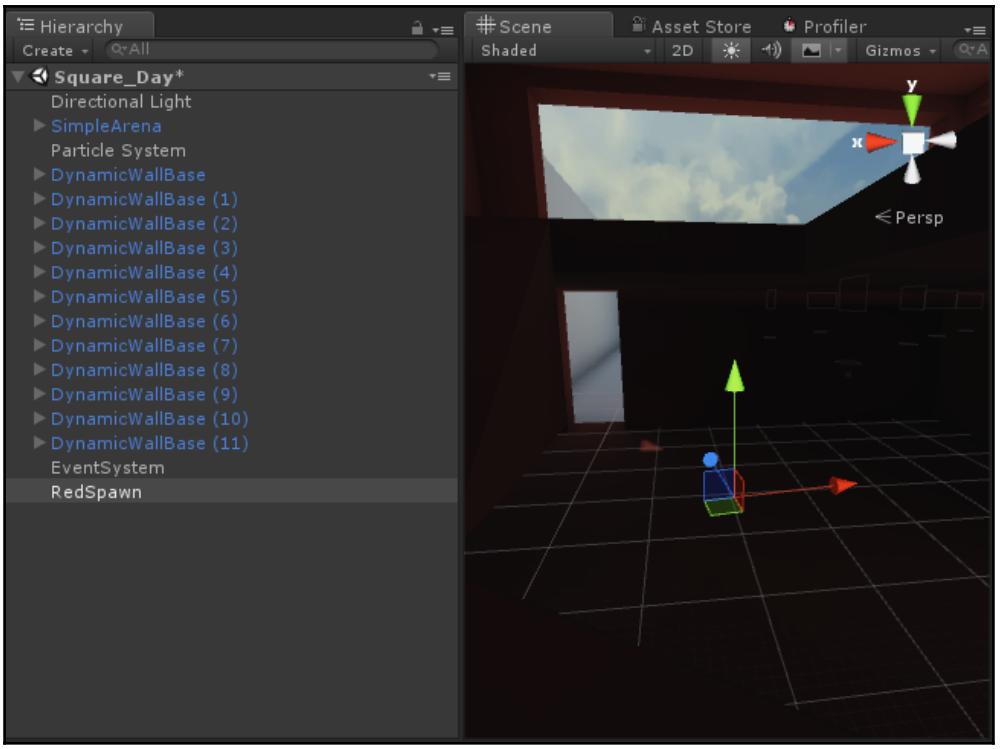


We know now that our network manager is capable of connecting us to a match, but at this point, it will always spawn us in the same place. To make sure that players are spawned fairly evenly, we'll define some spawn points next.

Defining player spawn points

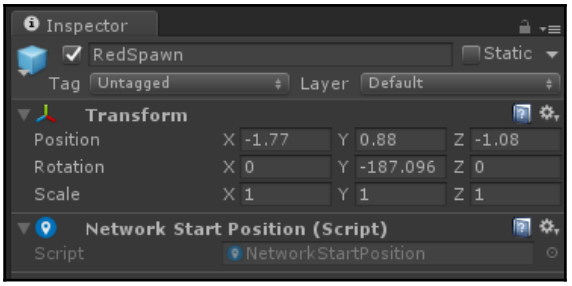
Spawn points are represented as prefabs in any scene that a networked player can connect to. For our purposes, we want four different kinds of spawn point: one for the red team, one for the blue team, one for the yellow team, and one for the green team.

Create an empty game object in your **Square_Day** scene and name it `RedSpawn`. Position it so that it's in the center of the bottom floor of the red building, facing in the general direction of the doors:

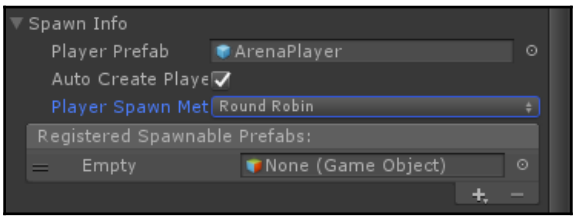


Save the **RedSpawn** object as a prefab by dragging it from the hierarchy into your `Prefabs` folder. Repeat this process by creating **GreenSpawn**, **YellowSpawn**, and **BlueSpawn** prefabs for the other buildings. Also, add all of these spawns to the **Square_Night** scene.

Once your prefabs have been created, highlight each in the **Inspector** and add a **NetworkStartPosition** component to it:



In order for our **NetworkManager** component to use these, we need to change the spawn method. Back in our lobby scene, highlight the **NetworkManager** object in the **Inspector** and change the **Spawn Method** field to **Round Robin**:



Test your `StartHosting` function again and notice that you'll now spawn in one of the four spawn points. When other players are able to join the game, they'll spawn at each point in a cyclical fashion. After a player has been spawned at each point, the cycle will repeat. What we'll do next is make sure that each player is assigned to the team that matches their spawn point's color.

Assigning players to teams

To set a player's team on the `PlayerIdentity` script, we'll add a function that checks for nearby spawn points as soon as the player is spawned and joins the team associated with whatever spawn is closest.

Open the `PlayerIdentity` script in your code editor and add a new function called `FindTeam`:

```
public class PlayerIdentity : MonoBehaviour
{
  ...
  private void FindTeam()
  {
  }
}
```

First, we'll get a list of every spawn point on the map. To do this, we'll need to add a `using` statement to add the `Generic` namespace that the `List` type is contained in. We'll also add a reference to the `Networking` namespace so that we can access properties of the network manager:

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

```
using UnityEngine.Networking;
```

Now we'll get the list from the network manager. This script doesn't have a reference to the **Network Manager** component, but fortunately it's accessible through a singleton property for convenience:

```
private void FindTeam()
{
  List<Transform> allSpawnPoints = NetworkManager.singleton.startPositions;
}
```

Now add a loop that will iterate through every spawn point in the list. Check the distance between the spawn point and the player; if it's less than 1 meter, we can assume that this is their spawn point. Call a new function called `JoinTeam` on the found spawn:

```
private void FindTeam()
{
  List<Transform> allSpawnPoints = NetworkManager.singleton.startPositions;
  for(int i = 0; i < allSpawnPoints.Count; ++i)
  {
    if(Vector3.Distance(allSpawnPoints[i].position, transform.position) <
1)
    {
      JoinTeam(allSpawnPoints[i]);
      return;
    }
  }
}
```

Now define the `JoinTeam` function. This function will consist of a `switch` statement that analyzes the name of the spawn point and assigns the `playerTeam` variable based on the result:

```
private void JoinTeam(Transform spawnPoint)
{
  switch(spawnPoint.name)
  {
    case "RedSpawn":
    playerTeam = Team.Red;
    break;
    case "BlueSpawn":
    playerTeam = Team.Blue;
    break;
    case "YellowSpawn":
    playerTeam = Team.Yellow;
    break;
    case "GreenSpawn":
```

```
        playerTeam = Team.Green;
        break;
      }
    }
```

Finally, remove the default team assignment from your `Start` function and call `FindTeam` instead:

```
void Start()
{
  FindTeam();
}
```

All players that join the game will now be added to every team in a cyclical fashion as they spawn, ensuring that each round collects a fair balance of players. However, there's one feature still clearly missing: the ability to actually join matches. We'll be tackling this next.

Using Unity's matchmaker system

Unity's network manager contains several functions used for interacting with a matchmaking system that allows hosts to list their games on the Internet and potential players to find them. This is what will let players join and play games with others without needing to know the host's IP address.

We'll build out a **JoinCanvas** menu in this section, allowing you to search for games instead of just creating them, but first we need to modify our hosting functions to post data to the matchmaking server.

Creating a matchmaker game

To use the matchmaker, we'll have to explicitly initialize it in code first. Add two `using` statements to the top of your `NetworkGameManager` script to include the `Match` and `Types` namespaces:

```
unity UnityEngine;
using System.Collections;
using UnityEngine.Networking;
using UnityEngine.Networking.Match;
using UnityEngine.Networking.Types;
using UnityEngine.UI;
```

Next, add the following line to `Start` in the `NetworkGameManager` class:

```
public class NetworkGameManager : MonoBehaviour
{
  ...
  private void Start()
  {
    networkManager.StartMatchMaker();
  }
  ...
}
```

Now that the matchmaker has been initialized, we can access it through the `networkManager.matchMaker` reference. Erase the contents of the `StartHosting` function and declare a new string to store the map name, using `Square_Day` as the default if the player hasn't selected a map:

```
public void StartHosting()
{
  string mapName = activeMap != null ? activeMap.name : "Square_Day";
}
```

Declare another variable to hold the maximum number of players, using a default value of `8` if the player doesn't select one manually:

```
public void StartHosting()
{
  string mapName = activeMap != null ? activeMap.name : "Square_Day";
  uint gameSize = activeCapacity != null ? uint.Parse(activeCapacity.name)
: 8;
}
```

Next, we'll make a request to `CreateMatch`. This request will store all of the specifications of a single match, including the map name and the maximum players that we've just set:

```
public void StartHosting()
{
  string mapName = activeMap != null ? activeMap.name : "Square_Day";
  uint gameSize = activeCapacity != null ? uint.Parse(activeCapacity.name)
: 8;
  networkManager.matchMaker.CreateMatch(mapName, gameSize, true,
  "", "", "", 0, 0, OnMatchCreate);
}
```

This function has more parameters than we've configured variables for, but a lot of them are fine as empty defaults for our purposes. The only other parameter you should notice is `OnMatchCreate`, which is the name of the function to be called when the request returns. The callback function can have any name, but to make it a viable callback it must take the `bool`, `string`, and `MatchInfo` parameters.

Define this callback function now right below your `StartHosting` function:

```
private void OnMatchCreate(bool successful, string details, MatchInfo info)
{
}
```

Create a conditional check of the `successful` value, which will be true as long as the match was successfully created by the matchmaking service. Throw a warning if it was unsuccessful for whatever reason:

```
private void OnMatchCreate(bool successful, string details, MatchInfo info)
{
  if(successful)
  {
  }
  else
  {
    Debug.LogWarning("There was an error creating an internet match.
Details:
    " +   details);
  }
}
```

Add the following lines within the successful block to begin hosting the created match, which will look very similar to our original `StartHosting` function:

```
private void OnMatchCreate(bool successful, string details, MatchInfo info)
{
  if(successful)
  {
    MatchInfo hostInfo = info;
    NetworkServer.Listen(hostInfo, 9000);
    networkManager.StartHost(hostInfo);
  }
  else
  {
    Debug.LogWarning("There was an error creating an internet match.");
  }
}
```

This completes the ability to create and host matches once again, and you can test it by clicking on the **START** button on the **CreateCanvas** as always. The difference is that the match is now registered with the matchmaker any time it's created, which opens the door into the next section: getting a list of these registered matches and joining them as a client.

Before you can successfully host a match for your game, you need to activate the **Multiplayer** service for your game. Open the **Services** window from the **Windows** menu, log in with your Unity ID, and select **Multiplayer** to set up the service step-by-step.

Joining a matchmaker game

We'll begin this section by creating all the functions we'll need to search for active games and connect to them. Open your `NetworkGameManager` script and create a new function called `FindMatches`:

```
public void FindMatches()
{
}
```

This function will only contain one line, a call to the matchmaker's `ListMatches` function, which has similar parameters to the `CreateMatch` function in the last section:

```
public void FindMatches()
{
  networkManager.matchMaker.ListMatches(0, 4, "", true, 0, 0,
OnListMatches);
}
```

The first parameter represents the current *page* of returned matches, and the second number represents the number of matches per page. This is helpful for only downloading a certain number of matches at a time when there could potentially be hundreds. For now, we're only going to view the first four available matches. The final parameter is another callback, which we'll implement now. First, add a using statement at the beginning of the `NetworkGameManager` script to include the `Generic` namespace, like we did with the `PlayerIdentity` script.

Create a new function called `OnListMatches` with the following parameters:

```
private void OnListMatches(bool successful, string details,
List<MatchInfoSnapshot> matches)
{
}
```

The final parameter of the `OnListMatches` function is the list of matches returned by the matchmaking server, represented by the `MatchInfoSnapshot` class. Each snapshot contains information about the match necessary to connect to it, including the network ID.

Add a conditional check of the `successful` parameter to match the other callbacks you've implemented so far:

```
private void OnListMatches(bool successful, string details,
List<MatchInfoSnapshot> matches)
{
  if(successful)
  {
  }
  else
  {
    Debug.LogWarning("There was an error searching for internet matches.
    Details: " + details);
  }
}
```

Add a nested conditional check to make sure the list of matches returned is larger than 0:

```
private void OnListMatches(bool successful, string details,
List<MatchInfoSnapshot> matches)
{
  if(successful)
  {
    if(matches.Count != 0)
    {
    }
    else
    {
      Debug.Log("There are no matches available.");
    }
  }
  else
  {
    Debug.LogWarning("There was an error searching for internet matches.
    Details: " + details);
  }
}
```

Add one last line to join the first match found if at least one match has been returned. In the next section, we'll link menu buttons to this function so that the player can select from the returned matches:

```
private void OnListMatches(bool successful, string details,
List<MatchInfoSnapshot> matches)
{
  if(successful)
  {
    if(matches.Count != 0)
    {
      NetworkManager.singleton.matchMaker.JoinMatch(matches[0].networkId,
      "", "", "", 0, 0, OnJoinMatch);
    }
    else
    {
      Debug.Log("There are no matches available.");
    }
  }
  else
  {
    Debug.LogWarning("There was an error searching for internet matches.
    Details: " + details);
  }
}
```

This function has a callback as well, so declare a new function called `OnJoinMatch`:

```
private void OnJoinMatch(bool successful, string details, MatchInfo info)
{
}
```

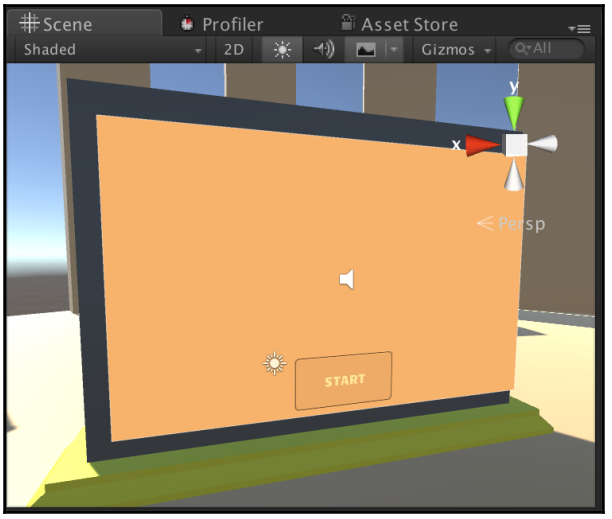
Perform one more success check before joining the selected match with the `StartClient` function:

```
private void OnJoinMatch(bool successful, string details, MatchInfo info)
{
  if(successful)
  {
    MatchInfo hostInfo = info;
    NetworkManager.singleton.StartClient(hostInfo);
  }
  else
  {
    Debug.LogWarning("There was an error joining the selected match.
    Details: " + details);
  }
}
```

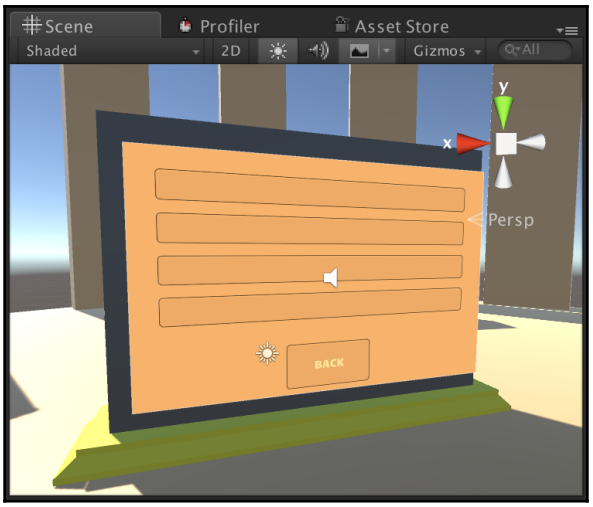
The code for hosting and joining matches is finished, so now all we need to do is create a menu for joining games (based on our menu for creating games) and tie all of the menus together to have a fully functional lobby scene.

Tying together the multiplayer lobby

Create a copy of your **CreateCanvas** and disable the original in the **Inspector** to hide it for now. Remove all of the buttons and images from the canvas, except the **START** button, so that you're left with a mostly empty menu, as shown in the following screenshot:



Rename the **START** button to `BACK` and change its text to **BACK** to match. Create a new **Button** object within the **JoinCanvas** and set **Width** to `350`, **Height** to `30`, and **Scale** to `0.01`. Give it the same styling and font as the other buttons, but remove the text for now. Duplicate the button three times and arrange the four in a list, naming them `Match1`, `Match2`, `Match3`, and `Match4` in **Inspector**. They should look something like the following on the canvas:

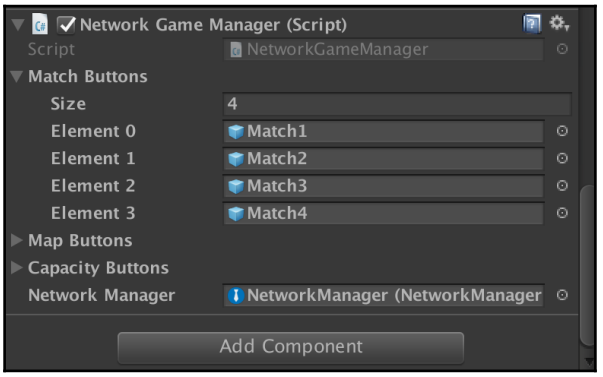


These four buttons will represent up to four matches returned by the matchmaking system. They'll display the level name and the current number of players in the match using the button's text field, which the `NetworkGameManager` script will update.

We'll need a reference to all of these buttons, so declare a new serialized array for the buttons at the top of your `NetworkGameManager` class:

```
public class NetworkGameManager : MonoBehaviour
{
  [SerializeField] private GameObject[] matchButtons;
  ...
}
```

Highlight the **NetworkGameManager** component on the **MenuCanvas** in **Inspector** and drag each match button into four new fields for this array:



Declare a new function called `PopulateMatchButtons` to populate these buttons with information gathered from the `FindMatches` function:

```
private void PopulateMatchButtons(List<MatchInfoSnapshot> matches)
{
}
```

Create a loop to iterate over each one of the match buttons, setting the text to display match information or disabling the button if there isn't a match available for that slot:

```
private void PopulateMatchButtons(List<MatchInfoSnapshot> matches)
{
  for(int i = 0; i < matchButtons.Length; ++i)
  {
    if(!matchButtons[i].activeSelf)
    matchButtons[i].SetActive(true);
    if(i < networkManager.matches.Count)
    {
      Text buttonText = matchButtons[i].GetComponentInChildren<Text>();
      buttonText.text = matches[i].name + "\t" + matches[i].currentSize;
    }
    else
    {
      matchButtons[i].SetActive(false);
    }
  }
}
```

The first lines of our loop check whether the current button is inactive; if so, it reactivates it. Buttons that are made inactive when a match isn't found need to be reset in order to access their text components if they get reused.

Add a call to `PopulateMatchButtons` to `OnListMatches` right after we verify that a list of matches greater than 0 has been received. Remove the previous function call to `JoinMatch`, as this will become a function called by our new menu:

```
private void OnListMatches(bool successful, string details,
List<MatchInfoSnapshot> matches)
{
  if(successful)
  {
    if(matches.Count != 0)
    {
      PopulateMatchButtons(matches);
    }
    else
```

```
        {
          Debug.Log("There are no matches available.");
        }
      }
      ...
    }
```

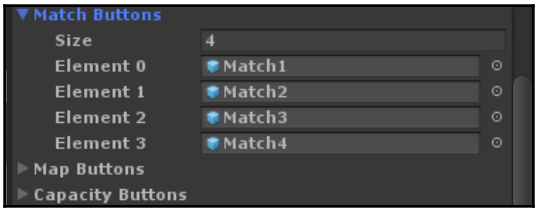
We want to make sure the menu searches for matches as soon as it has been activated, so add a call to `FindMatches` to the `Start` function:

```
private void Start()
{
  networkManager.StartMatchMaker();
  FindMatches();
}
```

Now we can write the function that our UI buttons will call when the player selects one to join a match. Declare a new function called `JoinMatch` with an `int` parameter for the match index:

```
public void JoinMatch(int matchIndex)
{
  networkManager.matchMaker.JoinMatch(networkManager.matches[matchIndex]
  .networkId, "", "", "", 0, 0, OnJoinMatch);
}
```

Now display each button object's **Button** component in the **Inspector** and add a call to `JoinMatch` to each. For the parameter of the function on the **Match1** button, enter **0**, for the parameter on the **Match2** button enter `1`, and so on, so that the buttons from top to bottom call `JoinMatch` with incrementing array indices:



Now every button on the **JoinCanvas** menu should have a callback except for the **BACK** button. In the next section, we'll link all of our menus together, starting with that **BACK** button.

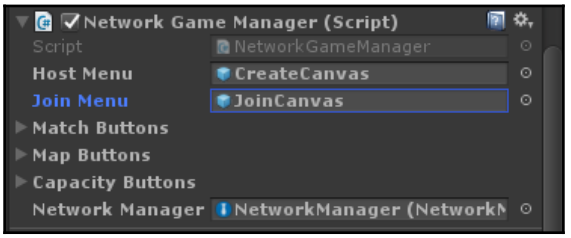
Linking the menus together

The last thing we need to add to our **Lobby** scene before it's complete is a fully navigable menu system. We have three disconnected menus, but there's no way to get between them. In this section, we'll add code to our `NetworkGameManager` script to show and hide our two submenus.

Declare two new serialized variables at the top of the `NetworkGameManager` class: one for the match creation menu and another for the join match menu:

```
public class NetworkGameManager : MonoBehaviour
{
  [SerializeField] private GameObject hostMenu;
  [SerializeField] private GameObject joinMenu;
  ...
}
```

Drag the **CreateCanvas** object onto the **Host Menu** field on the **NetworkGameManager**, and the **JoinCanvas** object onto the **Join Menu** field:



Now declare three new functions: `ShowHostMenu`, `ShowJoinMenu`, and `ShowMainMenu`, each responsible for activating either of the submenus or disabling both:

```
public void ShowHostMenu()
{
  hostMenu.SetActive(true);
}
public void ShowJoinMenu()
{
  joinMenu.SetActive(true);
}
public void ShowMainMenu()
{
  hostMenu.SetActive(false);
  joinMenu.SetActive(false);
}
```

Remember to disable both menus in the hierarchy so that the game begins with only the main menu visible.

The final step to tie your lobby together is to link all of your menu buttons to each other. Add calls to `ShowHostMenu`, `ShowJoinMenu`, and `ShowMainMenu` on each of their respective buttons. After that, your **Lobby** scene is as good as done; you can create matches from the main menu or conjure a list of matches to join. If you'd like, take a moment to test it out; if you don't have two computers to run it on, you can create a build, run it in windowed mode, and play it simultaneously with the Unity Editor to serve as the host and client, respectively. What you may notice, though, is that nothing in the game is synchronized between the two players; this is because we have yet to apply the final touch, which is data synchronization in matches.

Synchronizing data in multiplayer matches

Everything that you've done up to this point has been about getting players in the same match, but we haven't covered anything about keeping in-game events synced between the server and client. Fortunately, Unity's networking API makes this very easy; we can use most of the code we've already written with some slight modifications to make it network-aware.

Syncing player movement

The first thing we'll tackle is player movement, so you can see other people walking around the map. We'll build upon our existing player prefab and the **NetworkIdentity** component, as well as a new kind of component: **NetworkTransform**.

Select the **ArenaPlayer** prefab in your **Project** window and add a **NetworkTransform** component in the **Inspector**. This alone is enough to sync the position across the network but, if you tested now, you'd run into an interesting problem: input from any player would be reflected on every other player instance in the game. This is because there's one instance of **ArenaPlayer** for every connected player, and each has a component that checks for keyboard input.

To remedy this, we'll make our `OVRPlayerController` script network-aware so that each player can only control the local **ArenaPlayer** object that they're responsible for.

Open the `OVRPlayerController` script from the `OVR` folder in your project.

You may want to make a copy of the `OVRPlayerController` script before you modify it; after all, this is the first time we'll be modifying code that we originally got from the Oculus Utilities package. However, even if you don't save a copy, you'll always be able to download a new one.

Add a `using` statement to the top of the script to include the `Networking` namespace:

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

Now change the class `OVRPlayerController` inherits from to `NetworkBehaviour` instead of `MonoBehaviour`:

```
public class OVRPlayerController : NetworkBehaviour
{
  ...
}
```

Classes that inherit from `NetworkBehaviour` behave almost exactly like classes that inherit from `MonoBehaviour`, but with some extra properties and functions for network awareness. The first of these new properties we'll use is the `isLocalPlayer` variable.

Scroll to the `Update` function of `OVRPlayerController` and add an early return to the very top if the object the script is running on is not the local player:

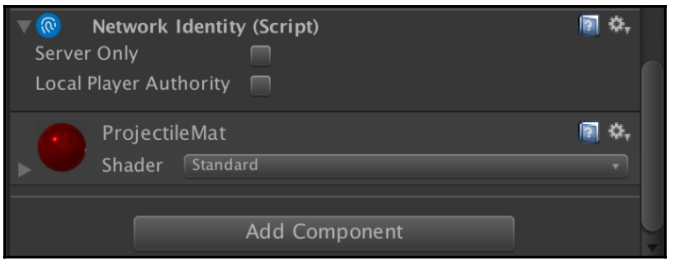
```
protected virtual void Update()
{
  if(!isLocalPlayer)
    return;
  ...
}
```

Now each player will only control themselves. If you test this now, you'll be able to see individual players move around the map independently.

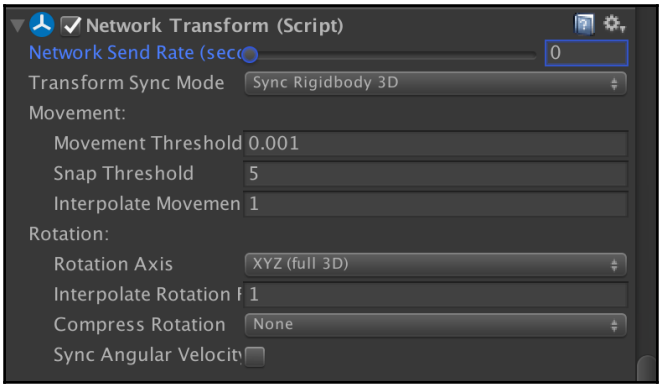
Player navigation is now synced, but if you fire any bullets, you'll only see them on the client they were fired from; bullets need to be instantiated and updated by the network just like the players themselves. In the next section, we'll make our bullets and firing system network-aware.

Handling object spawning on the network

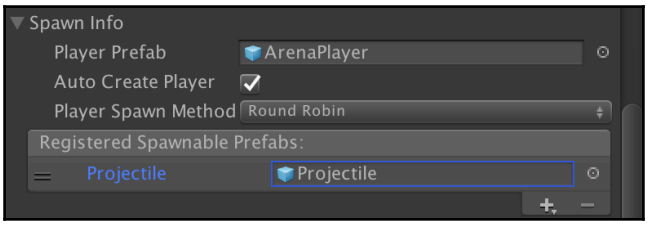
The first thing we'll need to do to our **Projectile** prefab to make it compatible with our networking system is to add a **Networkidentity** component to it. Select your **Projectile** prefab in the **Project** window and find **NetworkIdentity** in the **Add Component** menu:



Next, add a **NetworkTransform** component to it so that each client will know exactly where it spawns. You can set the **Network Send Rate** to **0** because the network won't need to synchronize it after applying the initial velocity; the trajectory will be the same across all clients.



Next, we'll need to register our projectile as a spawnable prefab. The network manager needs to know every prefab that could potentially be spawned on the network, so open your **Lobby** scene and display the **NetworkManager** in the **Inspector**. Add an entry to the **Registered Spawnable Prefabs** list and drag the **Projectile** prefab into it:



Finally, we'll need to edit our `FireProjectile` function to tell the server to fire a bullet for the client instead of having the client fire the bullet; this ensures consistency among all clients because the server sends the same action to every connected player. This concept is called **server authority**.

Unity contains a handy attribute for turning functions into server commands. All command functions must be marked with the `[Command]` attribute, and the function name must begin with `Cmd`. Modify the `FireProjectile` function in the `FiringSystem` script to match these requirements, while also setting the `FiringSystem` class to inherit from `NetworkBehaviour`:

```
using UnityEngine.Networking;
public class FiringSystem : NetworkBehaviour
{
  ...
  [Command]
  private void CmdFireProjectile()
  {
    ...
  }
}
```

Make sure to change the call in `Update` from `FireProjectile` to `CmdFireProjectile` as well. Within the `CmdFireProjectile` function, add a line at the very end to call the `NetworkServer.Spawn` function:

```
private void CmdFireProjectile()
{
  GameObject newProjectile = Instantiate(projectilePrefab);
  newProjectile.transform.position = lookTransform.position +
    lookTransform.forward;
  newProjectile.transform.rotation = lookTransform.rotation;
```

```
    Rigidbody rigidbody = newProjectile.AddComponent<Rigidbody>();
    rigidbody.AddForce(newProjectile.transform.forward * projectileForce,
      ForceMode.Impulse);
    Destroy(newProjectile, 10);
    NetworkServer.Spawn(newProjectile);
  }
```

Now try testing your game once again and firing a projectile as usual. You'll be able to see the bullet from the perspective of the player firing it and from the perspective of every other client in the game. The bullet still needs to do something though, so the next sections will be about making our walls network-aware and adding some consequence to a collision of a bullet with a player.

Detecting bullet collisions on the network

We'll begin by making our dynamic wall health system compatible with the network, so that any walls that are reduced to zero health will fall down in a synchronized fashion across every client. The first step to this is to add a **NetworkIdentity** component to the **DynamicWallBase** prefab, which is a requirement for any `NetworkBehaviour` scripts.

Open your `DynamicWall` script, include the `Networking` namespace, and change the class to inherit from `NetworkBehaviour`:

```
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;
using System.Collections;
using UnityEngine.Networking;
public class DynamicWall : NetworkBehaviour
{
  ...
}
```

To synchronize a specific variable across the network, it needs to be marked with a `[SyncVar]` attribute. Add the following attribute above your `remainingHealth` variable:

```
public class DynamicWall : NetworkBehaviour
{
  ...
  [SyncVar] private int remainingHealth = 10;
  ...
}
```

We're going to want to call the function that reduces the wall health conditionally, so let's move most of it out of `OnCollisionEnter` into its own function. Declare a new function called `ReduceHealth` and move the following lines currently in `OnCollisionEnter` into it:

```
private void ReduceHealth()
{
  remainingHealth--;
  if(remainingHealth <= 0)
  {
    healthBar.fillAmount = 0;
    counterObject.SetActive(true);
    remainingCost = totalCost;
    UpdateText();
    StartCoroutine(MoveSectionDown());
  }
  else
  {
    healthBar.fillAmount = (float)remainingHealth/totalHealth;
  }
}
```

Add a call to this new function within the check for the `Projectile` tag in `OnCollisionEnter`, adding an extra check to only perform this function on the server, so it isn't performed once for each client every time the event occurs:

```
private void OnCollisionEnter(Collision collisionInfo)
{
  if(!isRaised)
    return;
  if(collisionInfo.collider.tag == "Projectile" && isServer)
  {
    ReduceHealth();
  }
}
```

There's one problem with this functionality currently: the health will be properly updated, but everything else after the subtraction operation isn't part of any network function or variable. This means that those actions won't be updated on the clients, so even if the health reaches 0, they won't be able to see the change.

There's a mechanism built into Unity to solve this problem, too: `SyncVar` hook functions. Each `SyncVar` can optionally have a function that it calls whenever the value is changed, enabling us to call all of the other code currently in the `ReduceHealth` function. The only requirement is that the hook function takes a parameter of the same value as the `SyncVar`.

Create a function to use as the hook called `OnHealthReduced` and move all of the code after the first line of `ReduceHealth` into this new function:

```
private void OnHealthReduced(int health)
{
  if(remainingHealth <= 0)
  {
    healthBar.fillAmount = 0;
    counterObject.SetActive(true);
    remainingCost = totalCost;
    UpdateText();
    StartCoroutine(MoveSectionDown());
  }
  else
  {
    healthBar.fillAmount = (float)remainingHealth/totalHealth;
  }
}
```

Now associate this function with the `remainingHealth` variable by adding the following parameter to the `SyncVar` attribute:

```
public class DynamicWall : NetworkBehaviour
{
  ...
  [SyncVar(hook = "OnHealthReduced")] private int remainingHealth = 10;
  ...
}
```

Now the `OnHealthReduced` function will be called anytime the `remainingHealth` variable is updated on the server, and it will propagate to every client connected, meaning all of the actions that follow the health subtraction will stay in sync.

We don't want walls to be the only thing affected by bullet collisions; we also want our bullets to have an effect when they hit other players. For now, we'll add a simple server function to send players back to their spawn whenever they're hit with a bullet.

Go back to your `PlayerIdentity` script and add a function called `Respawn`. Before we fill out the `Respawn` function, we'll need a reference to the player's original spawn point, so create a new variable called `spawnPoint` and set the class to inherit from `NetworkBehaviour`:

```
public class PlayerIdentity : NetworkBehaviour
{
  public Team playerTeam;
  private Transform spawnPoint;
```

```
}
```

Capture this value in the `FindTeam` function right after a spawn is found:

```
private void FindTeam()
{
  if(Vector3.Distance(allSpawnPoints[i].position, transform.position) < 1)
  {
    spawnPoint = allSpawnPoints[i];
    JoinTeam(allSpawnPoints[i]);
    return;
  }
}
```

Now fill out your `Respawn` function to return the player to their original spawn location:

```
private void Respawn()
{
  gameObject.transform.position = spawnPoint.position;
  gameObject.transform.rotation = spawnPoint.rotation;
}
```

Finally, add an `OnCollisionEnter` function that calls the `Respawn` function on the server if the player is hit with a projectile:

```
private void OnCollisionEnter(Collision collisionInfo)
{
  if(!isServer)
    return;
  if(collisionInfo.collider.tag == "Projectile")
    Respawn();
}
```

You've now got a simple but complete respawn function that will reset players whenever their enemies manage to hit them with a projectile, motivating them to avoid bullets as they run around the map.

Ending a match after a set time

Every part of our basic game flow is complete, apart from a way to end the match, so we'll add a timer that lets the game run for 5 minutes before bringing all players back to the main menu. We'll do this by creating a **GameTimer** component and adding it to the **NetworkManager** object as soon as a host starts a game.

Create a new script called `GameTimer` and add it as a component to the **NetworkManager** object in your **Lobby** scene. Open the script, include the `Networking` namespace, and have it inherit from `NetworkBehaviour` as usual:

```
using UnityEngine;
using System.Collections;
using UnityEngine.Networking;
public class GameTimer : NetworkBehaviour
{
  ...
}
```

Declare a new variable `secondsRemaining` and initialize it to `300` in the `Start` function:

```
public class GameTimer : NetworkBehaviour
{
  private float secondsRemaining;
  private void Start()
  {
    secondsRemaining = 300;
  }
}
```

In the `Update` function, add an early return if the instance running this script isn't the server. If it is the server, subtract the seconds that have passed since the last frame and end the game if the value reaches 0:

```
private void Update()
{
  if(!isServer)
    return;
  secondsRemaining -= Time.deltaTime;
  if(secondsRemaining <= 0)
  {
    NetworkManager.singleton.StopHost();
    Destroy(this);
  }
}
```

Add a call to add a **GameTimer** component to the **NetworkManager** object in the `OnMatchCreate` function of `NetworkGameManager`:

```
private void OnMatchCreate(bool successful, string details, MatchInfo info)
{
  if(successful)
  {
    MatchInfo hostInfo = info;
```

```
        NetworkServer.Listen(hostInfo, 9000);
        networkManager.StartHost(hostInfo);
        GameObject timer = new GameObject("Timer");
        timer.AddComponent<GameTimer>();
      }
      else
      {
        Debug.LogWarning("There was an error creating an internet match.");
      }
    }
```

You've implemented the simplest way to end a networked game; when 5 minutes have passed, all players will be returned to the main menu to create or join another match. Of course, this isn't generally how multiplayer games are ended; usually, it occurs after a team accomplishes a specific goal or gets a certain number of kills. If you're feeling adventurous, try coming up with your own win conditions and coding new, more complex flows.

Summary

In this section, we tackled the massive task of adding networked game support to our project. While a lot of features were simplified or pared down for the purposes of technicality, the project now contains a sample of several different techniques that work together to make up a full networked game flow.

We began by exploring the network manager and tying it to a lobby space by allowing players to create or join games with a UI interface. Once we did that, we moved onto syncing data within the actual games.

Syncing data can be as simple as **NetworkTransform** components, which constantly sync the transform of an object across every player on the network, or as complex as server-only or client-only functions that propagate messages to other players to ensure consistency in the world; as you build out your own games, you'll develop an instinct for when to use which method when.

In the next and final section, we'll go through the final motions of every game project, from packaging it up to shipping it and publishing it on the Oculus Store.

10
Publishing on the Oculus Store

Throughout the course of this book, you've implemented the basic features of virtually every pillar of VR development. You've created methods of interaction and navigation, given your game a distinct look and feel through a deeper understanding of graphics and rendering, and enabled connecting players worldwide through networked matchmaking.

As you bring these skills into your own creations and build outwards and upwards from the fundamentals demonstrated in our Arena Combat game, you'll need to know how to package and prepare your work so that it's ready to be deployed to the public. In this chapter, we'll cover the process of preparing your build in Unity, as well as the requirements it will need to meet in order to be successfully published on the Oculus Store.

This chapter will cover the following topics:

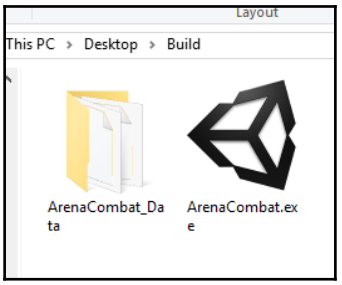
- Packaging a final Unity build
- Meeting the Oculus submission guidelines
- Uploading your first build to Oculus

Packaging a final Unity build

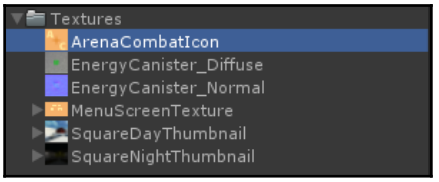
Typically, running the Unity Editor or making quick builds with default settings is enough for testing purposes, but before you upload any public builds, you should take care to configure a few build details that ensure a clean, polished final product.

Adding a game icon

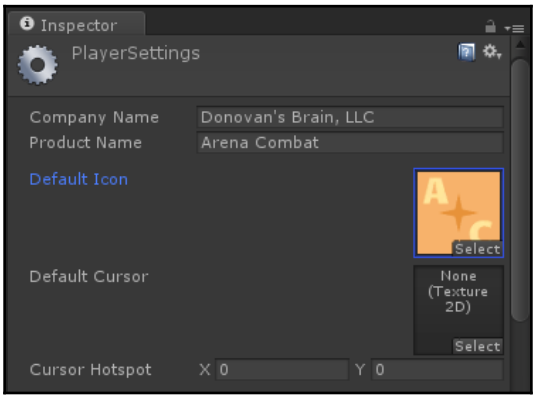
All of the builds you've created so far have had the default Unity icon, shown in the following screenshot:



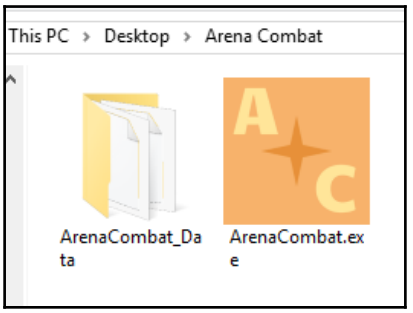
In this section, we'll go through the quick and easy steps of adding a custom icon to your build. Open your Arena Combat project and drag `ArenaCombatIcon.png` (included with this chapter) into the `Textures` folder:



Open the **Player** settings from the **Edit** menu and notice the field labeled **Default Icon** in **Inspector**. Drag **ArenaCombatIcon** onto this field, and update the **Product Name** and **Company Name** as well (get creative with the company name if you want!):



When you next make a build, this new icon will appear instead of the Unity icon:



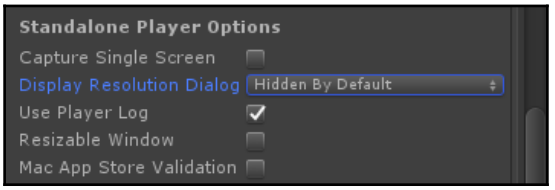
Before we make our final build, though, we'll finalize our **Quality** and **Player** settings.

Configuring final player settings

The player settings handle how the game is presented and run once it's started. This includes the options window that comes up right after you start a build; it lets the player toggle between fullscreen and window modes, and set the quality level and resolution. This is called the **display resolution dialog**.

The display resolution dialog is a handy menu, but since it has a static appearance and looks quite generic, we'll hide it by default in favor of implementing a more contextual settings menu in-game.

Within the **Player** settings menu, where you set the default icon, scroll down until you find the **Display Resolution Dialog** field. Change this setting from **Enabled** to **Hidden By Default**:



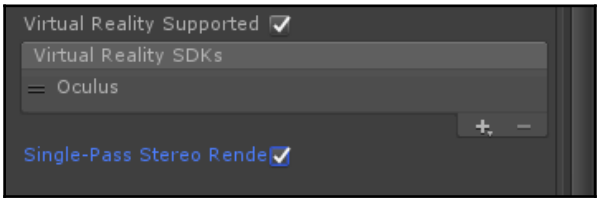
The resolution display dialog will now only appear if you're holding down the *alt* key while the game is starting; otherwise, it will go straight to VR.

Next, scroll down to the section labeled **Splash Image**. In this area, you can specify a custom texture to display while the game is initialized. You may have already noticed Unity's default splash screen when you started testing builds.

Feel free to drag your own splash screen into this field, or simply disable Unity's.

You need a paid Unity license to disable Unity's default splash screen.

The last setting in the **Player** settings menu, in the **Other Settings** section, that we'll look at is the **Single-Pass Stereo Rendering** feature. This is a relatively new feature that optimizes rendering for stereoscopic displays such as the Oculus Rift; it will make your game run generally faster at a baseline, though it may clash with advanced graphical features such as screen-space effects. As a standard rule, it's always a good idea to use single-pass stereo rendering unless you see it cause problems or artifacts in your game's rendering:



Now that we've taken care of the relevant settings in the **Player** menu, let's move on to another very important configuration for production builds: the **Quality** menu.

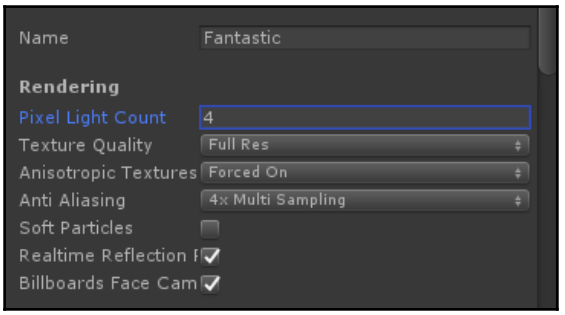
Configuring final quality settings

Open the **Quality** settings from the **Edit** menu. At the very top of the **Inspector** window, you'll see the quality matrix; this lets you configure different levels of quality for each different platform you're deploying your game on. Since we're only building for the Oculus Rift right now, you'll see only one platform column, with six default quality level rows (you'll see more columns if you chose to install other platforms, such as iOS or Android):

Ensure the **Fantastic** quality level is the default by clicking the down arrow to the right of **Default** and selecting **Fantastic**.

You can click the checkboxes next to each quality setting to disable it, but leave them all on for now. These lower quality levels can still be accessed through the resolution display dialog, and can be handy for identifying whether performance issues lie in the quality setting or something else.

Scroll down to the **Pixel Light Count** field. You probably remember this setting from our graphics work earlier; we changed this value around when playing with real-time lights, but because the lighting in both of our scenes is now baked, move this value to `4`:



Finally, scroll down to the **Anti Aliasing** setting, which you may also remember from the chapter about graphics. Change this value to **4x Multi Sampling.** This could potentially go up to 8x, but because anti-aliasing is so expensive, it's a good idea to keep the default value at 4x unless you see aliasing artifacts very frequently. 2x is also a viable setting if you really need to increase the frame rate of your game, but is not recommended, since anti-aliasing can drastically reduce immersion, especially on current generation VR headsets that run at around a 2k resolution.

Getting to know the output log

As a last focus of Unity-specific features, we'll take a look at something that's been a part of our builds since the beginning: the data folder. Every time you create a build for your project, you'll find a folder next to the executable that has the same name as the executable itself, but with `_Data` at the end.

Create a build now and find the data folder for it. If you run the build once, you can open the data folder and you'll notice a file called `output_log.txt`. This is an immensely helpful tool for debugging builds outside the Unity Editor, as it essentially routes all information that you'd typically be able to see in the editor's **Console** window into this single text file. Open it in a normal text editor now to see a sample of the output information.

If you use Unity's `Debug.Log` and `Debug.LogWarning` functions, every time they're called they'll write out to this log file, so it's a good idea to use them in any case where the game might run into trouble. That way, if a player reports an issue, you can tell them to look in the output log and try to find out exactly where the issue came from.

This concludes our final settings within Unity. Next, we'll look at requirements that all games (not just Unity games) need to meet in order to be accepted by Oculus.

Meeting the Oculus submission guidelines

In order for your build to be eligible for publication on the Oculus Store, it needs to meet two sets of guidelines: the content policy and the technical requirements. In this section, we'll go over both sets, but keep in mind that these requirements may change and are always kept up-to-date at `https://developer3.oculus.com/documentation/publish/latest/concepts/publish-prep-app/`.

Meeting the Oculus content policy

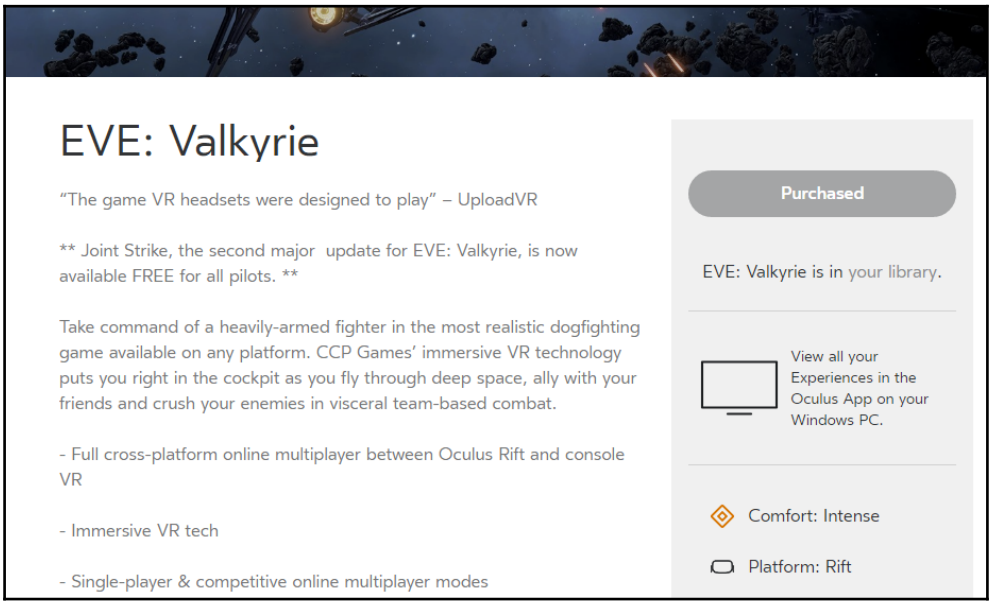
In its content policy, Oculus does a good job of summarizing the basic guidelines in one simple sentence: "…what we are likely to accept generally falls within the threshold of an R-rating for a movie."

This means no pornographic content, and nothing especially gory, sexual, or abusive. Additionally, your game can't allow players to gamble with real money. If you want to create a poker game in VR, you're obviously free to do so, but if your game allows players to put real money at stake, then you'll have to host it somewhere other than the Oculus Store.

When you publish on the Oculus Store, your game's page will include a **comfort rating**. There are three comfort ratings: **Comfortable**, **Moderate**, and **Intense**. Comfortable experiences generally keep the player in one spot observing the world around them; lack of player motion eliminates a lot of the opportunity for nausea.

Moderate experiences, by contrast, might require the player to move around the game world. Our Arena Combat game requires the player to move around in order to play, but the movement isn't very extreme and there aren't any sudden changes in elevation, so it would probably fall under the Moderate category.

Intense experiences are typically reserved for players who are naturally comfortable in virtual reality and can handle extreme motion without feeling sick. Intense experiences typically have free-flying cameras, frequent acceleration, or fast-paced action. For example, *EVE: Valkyrie*, a launch title for the Rift, received an Intense rating for acceleration and rotation in a zero-gravity environment:



The last part of the Oculus content policy is the age rating. When you publish your game, you'll be able to assign it one of two age ratings: 13+ and 17+. Since players must be at least 13 years old to have an Oculus account anyway, you don't need to worry about tailoring your content around small children, but if your game deals with more mature themes you might consider limiting it to the 17+ category only.

Meeting the Oculus minimum technical requirements

Once you've set your content assessment, you'll need to be sure your game meets the minimum technical requirements. Put simply, this means your experience must typically run at 90 frames per second on the hardware configuration that Oculus has set as the low end of the VR-capable PC spectrum.

At the time of writing, the minimum configuration includes:

- An NVIDIA GTX 970 or AMD 290 graphics card
- An Intel i5-6400 processor
- 8 gigabytes of RAM
- Windows 7 (64-bit)

As is the nature of hardware, though, these specifications are bound to change, so ensure you check the latest hardware requirements on the Oculus website before testing your minimum-spec build.

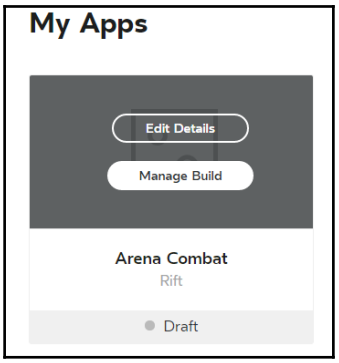
What about Asynchronous Timewarp and Spacewarp?

If you recall from the first chapter of the book, the Oculus runtime includes Asynchronous Timewarp and Spacewarp, two features that keep VR experiences smooth even when they drop below 90 frames per second. It's okay to rely on this feature if you might drop five or ten frames during an infrequent moment of heavy processor load, but it's not meant to be a silver bullet for optimization; make sure your build only uses Asynchronous Timewarp when absolutely necessary on your minimum-spec build.

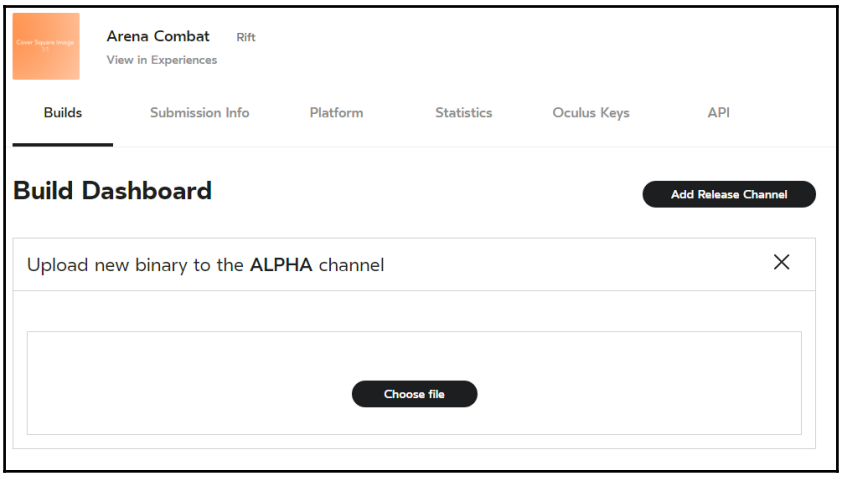
Uploading your first build to Oculus

In this section, we'll go through the process of uploading a build to the Oculus Dashboard and configuring it before it's ready to be submitted to Oculus for review. We'll be using the Arena Combat game as an example in this section, but you don't need to follow along with the project you've made with this book; feel free to follow these steps with your own project when you're ready to begin prepping it for publication.

Go to `https://dashboard.oculus.com/` and login with your Oculus ID (the same one you used when you first configured your Rift). Click the **My Apps** tab and then select **Create New App**. Follow the prompts to name your app and then it will appear in the **My Apps** screen as a draft. Mouse over it and click the button labeled **Manage Build**:



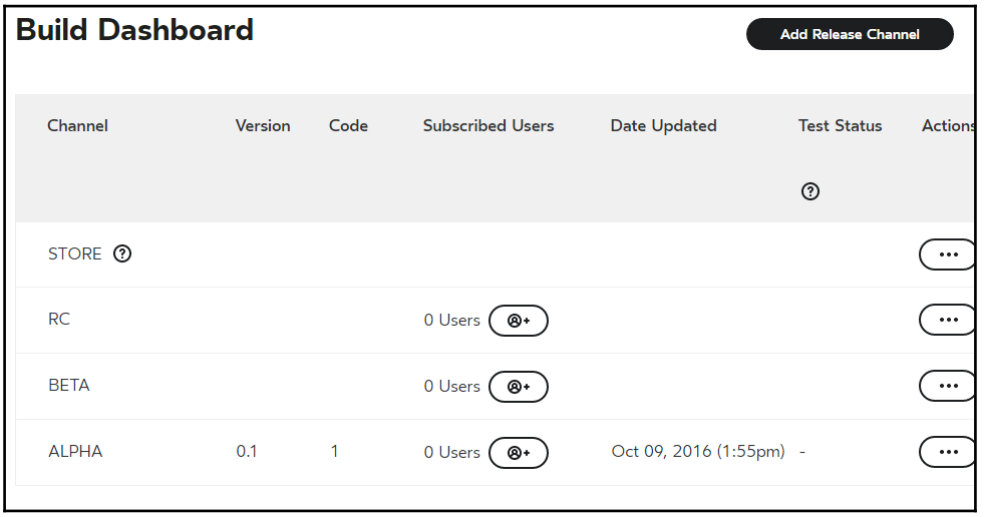
Clicking this will bring you to the **Build Dashboard**, which lets you upload a build to the **ALPHA** channel by default:



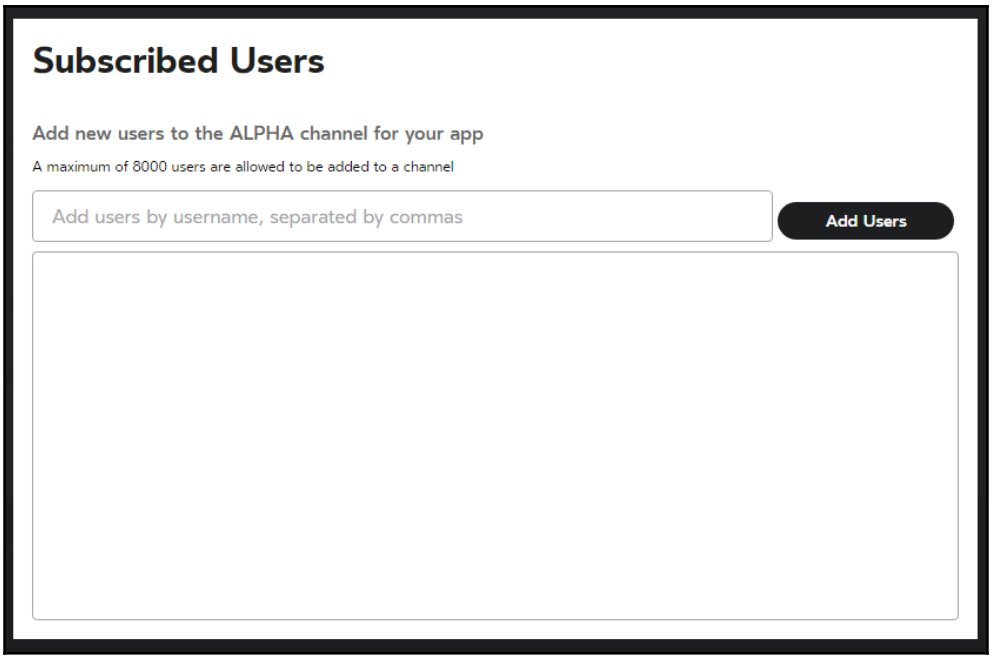
Before you upload a build, you'll need to specify some options, including the version number and path to the launch file. There's also an opportunity to include any redistributables your app may need, but you don't have to worry about this with a standard Unity project. Once you click on **Upload**, your build will be validated and processed.

Managing release channels

Once your build finishes processing, you'll see it appear in a list of your release channels at the bottom of the page. There are four release channels by default: **STORE**, RC, **BETA**, and **ALPHA**:



These channels can each feature different builds and different groups of users, allowing you to isolate use cases in your production preparation and manage multiple pools of players at once. Users can be added to each channel by pressing the button in the **Subscribed Users** column:

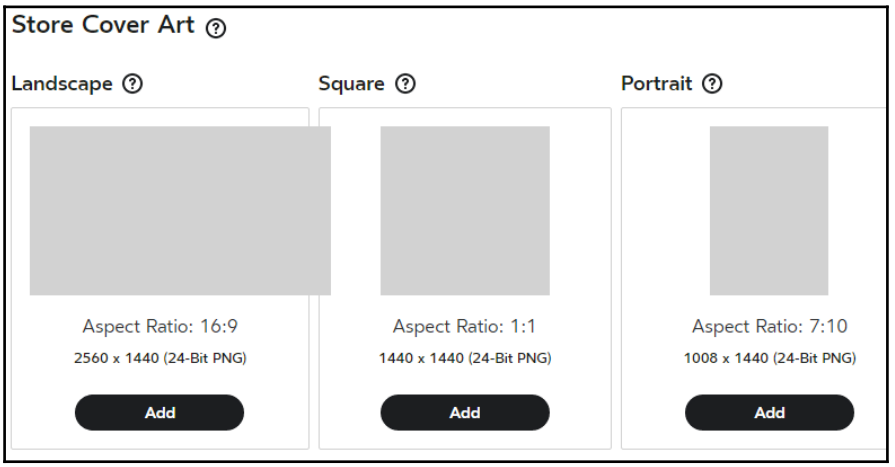


You can also add entirely new channels by clicking the **Add Release Channel** button in the upper-right of the page.

Uploading submission information

At the top of the page, click the **Submission Info** tab directly to the right of the **Builds** tab that you're currently in. This is the section of the dashboard where you specify all the forward-facing information about your build, from technical specifications to in-store banner assets and pricing information.

Click the **Assets** tab on the left. Here, you'll see an empty slot for every logo, icon, and banner that will be used to represent your game when it's on the store, including some basic specifications for each image:



The **Submission Info** section is also where you'll submit to the store when you're ready for Oculus review; once you're completely satisfied with the build and the assets you've uploaded with it, you can click the **Submit** tab on the left to finalize and submit the game.

Other sections of the dashboard

There are a few other sections of the dashboard that might be useful in your development. The **Platform** tab provides everything you need to integrate Oculus services with your app; these services include in-app purchases, achievements, and leaderboards.

The **Statistics** tab displays information about how your game is being received by the public. You can see various analytics about revenue and how often players are using your app; this is a good tab to help gauge the success of your game after it goes live.

The **Oculus Keys** window will be available when your app has been approved by Oculus. You can generate keys that activate full versions of your app and spread them to promote your game or, if you'd rather sell your game outside the Oculus Store, you can offer your app exclusively through generated keys.

Finally, the **API** tab provides documentation and information you'll need to connect your app to the services offered in the **Platform** tab. This is a great place to find everything you need to connect your app tightly to the Oculus ecosystem.

Summary

In this chapter, we quickly covered all the final steps to package your game and submit it to Oculus. We began by going over some final build settings within Unity, and then covered a brief summary of the Oculus content policy and minimum technical requirements.

We then looked at the Oculus Dashboard at a glance, getting acquainted with the different windows that you can use to configure your app information before and after submission. By uploading builds to different release channels and then adding users to them, you can test different improvements on several discrete audiences before finally submitting your game from the **Submission Info** tab.

This is the end of this book, but just the beginning of your journey as a virtual reality developer. You've got all the basic skills you need to create a VR experience for the Oculus Rift; the next step is to come up with an idea, form a team of designers, artists, and programmers, and bring your idea to life so that you can share your imaginative vision with the world.

Now get out there and create some new realities!

Index

E

F

G

H

P

R

S

T

U

V

X

Z