O'REILLY®

Early Release

RAW & UNEDITED

Learning Virtual Reality

DEVELOPING IMMERSIVE EXPERIENCES AND APPLICATIONS FOR DESKTOP, WEB, AND MOBILE

Tony Parisi

Learning Virtual Reality

Developing Immersive Experiences and Applications for Desktop, Web and Mobile

Tony Parisi

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Learning Virtual Reality
by Tony Parisi

Printed in the United States of America.

Published by O'Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editor: Meg Foley
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January -4712: First Edition

Revision History for the First Edition
2015-07-06: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=0636920038467* for release details.

063-6-920-03846-7

[FILL IN]

Table of Contents

CHAPTER 1
Introduction

Virtual Reality is a medium with tremendous potential. The ability to be transported to other places, to be fully immersed in experiences, and to feel like you're really there-- *present*-- opens up unimagined ways to interact and communicate. Until recently, virtual reality was out of reach for the average consumer due to cost and other factors. However, advances in the technology over the last few years have set the stage for a mass market revolution that could be as influential as the introduction of television, the Internet, or the smartphone.

Virtual reality-- VR for short-- comprises a collection of technologies: 3D displays, motion tracking hardware, input devices, software frameworks, and development tools. While consumer-grade VR hardware is young and evolving, a handful of platforms have emerged as go-to choices, including Oculus Rift, Samsung Gear VR, and Google Cardboard. Each delivers a different level of VR experience, at a different price point, with varying degrees of in-your-hands portability.

Software to create and display consumer virtual reality is also coming together rapidly. The Unity3D and Unreal game engines, popular for making desktop and mobile games, have become tools of choice for native VR development. And the web is not far behind: WebGL and 3D JavaScript frameworks like Three.js and Babylon.js are providing a path for creating open source, browser-based virtual reality experiences for desktop and mobile operating systems.

It's an exciting time! With so much energy going into development, and so much consumer interest, VR just might be the next big wave of computer technology. In this book, we explore the hardware, software, application techniques and interface design challenges encountered by today's virtual reality creator. Virtual reality is still early. It's a lot like the wild west, and you are a pioneer. The landscape may be fraught with unknowns, even dangers-- but we push on, driven by the promise of a better life. Let's take a peek at this new frontier.

Figure 1-1 shows a screen shot of the now-famous Tuscany VR demo, created by the team at Oculus VR to show off their hardware. Put on the Oculus Rift and launch the demo. You are on the grounds of a Tuscan estate, looking at a beautiful villa. Clouds drift lazily across the sky. You hear birds chirping, and the sound of waves lapping gently against a shore.

You move through the scene, video game-style using the w, a, s and d keys on your keyboard (known to gamers as the "WASD keys"). If you play a lot of PC games, this is nothing new. But now, turn your head: looking up, down, and behind, you can see the entire estate. You are there, immersed in a virtual world that completely surrounds you. Walk forward, into the villa, and take a look around. Walk out, up to the edge of the property and see the lake below. For a few moments at least, you forget that you are not actually in this other place. You're *present*.

This feeling of total immersion, of being somewhere else, experiencing something else entirely, is what we are striving for with virtual reality. And this is where our journey begins.

Figure 1-1. Tuscany VR Demo by the Oculus VR Team

What is Virtual Reality?

Reality is merely an illusion, albeit a very persistent one.
—Albert Einstein

Virtual Reality has one goal: to convince you that you are somewhere else. It does this by tricking the human brain-- in particular the visual cortex and parts of the brain that perceive motion. A variety of technologies conspire to create this illusion, including:

- **Stereoscopic Displays.** Also known as *3D displays*, or *head mounted displays* (HMDs). These displays use a combination of multiple images, realistic optical distortion, and special lenses to produce a stereo image that our eyes interpret as having three-dimensional depth.
- **Motion Tracking Hardware.** Gyroscopes, accelerometers and other low-cost components are used in virtual reality hardware to sense when our bodies move and our heads turn, so that the application can update our view into the 3D scene.
- **Input Devices.** Virtual reality is creating the need for new types of input devices beyond the keyboard and mouse, including game controllers and hand- and body-tracking sensors that can recognize motion and gestures.
- **Desktop and Mobile Platforms.** This includes the computer hardware, operating systems, software to interface to the devices, frameworks and engines that run applications, and software tools for building them.

Without all four of the above components, it is hard to achieve a fully immersive virtual reality experience. We will dive into the details throughout the book; for now let's take a quick look at each.

Stereoscopic Displays

The main ingredient in virtual reality is a persistent 3D visual representation of the experience that conveys a sense of depth. To create this depth, virtual reality hardware systems employ a 3D display, also known as a *stereoscopic display* or *head mounted display*.

For years, one of the biggest impediments to consumer-grade virtual reality was an affordable stereoscopic display that is light and comfortable enough to be worn for an extended period. This situation changed dramatically when the team from Oculus VR created the *Oculus Rift*. First introduced in 2012, the Rift was a breakthrough in VR hardware featuring a stereoscopic display and a head-tracking sensor built into a lightweight headset that could be purchased as a development kit for a few hundred

dollars. While the original development kit, known as the DK-1, was fairly low-resolution, it was enough to get the entire industry excited and unleash a storm of VR development. Newer Rift development kit versions, such as the DK-2 depicted in Figure 1-2, feature higher display resolution, position as well as orientation tracking, and better performance.

Figure 1-2. The Oculus Rift Head-Mounted Display, Development Kit 2

So what does the Oculus Rift actually do? To create the illusion of depth, we need to generate a separate image for each eye, one slightly offset from the other, to simulate *parallax*-- the visual phenomenon where our brains perceive depth based on the difference in the apparent position of objects (due to our eyes being slightly apart from each other). To create a really good illusion, we also want to distort the image to better emulate the spherical shape of the eye, using a technique know as *barrel distortion*. The Oculus Rift does both.

Putting these techniques together, a more accurate screen shot of the Tuscany VR demo, a capture of the entire display that renders this scene in virtual reality, would look like Figure 1-3. This shows the rendering as it appears on a computer screen connected to the Oculus Rift's head mounted display.

Figure 1-3.
Tuscany VR Demo Rendered in Stereo for the Oculus Rift

From a software point of view, an Oculus Rift application's job is to render an image like the one in Figure 1-3 a minimum of 60 and ideally 120 times per second, to avoid any perceived lag, or *latency*, that might break the illusion, or worse, lead to nausea that is often associated with poorly-performing VR. Exactly how we do this in our applications is the subject of another chapter.

Note that the Oculus Rift is not the only game in town. As we will see in the next chapter, there are several head mounted displays to choose from. Some of these work only with desktop computers, others are just for smartphones, and still others can only be used with game consoles. The HMDs come in a variety of styles and a range of prices. But as consumer-ready virtual reality displays go, Oculus is the first and-- as of this writing-- still the best.

Motion Tracking Hardware

The second essential trick for making the brain believe it is in another place is to track movements of the head and update the rendered scene in real time. This mimics what happens when we look around in the real world.

One of the innovations in the Oculus Rift is its rapid head motion tracking using a high speed *inertial measurement unit* (IMU). Head-tracking IMUs combine gyroscope, accelerometer and/or magnetometer hardware, similar to the ones found in

smartphones today, to precisely measure changes in rotation. The VR hardware systems covered in the next chapter employ a variety of IMU configurations.

Motion tracking of the head is as important, perhaps more important, than quality stereo rendering. Our perceptual systems are very sensitive to motion, and as is the case with lag in stereo rendering, high latency in head tracking can break the feeling of immersion and/or cause nausea. Virtual reality IMU hardware must track head movement as rapidly as possible, and the software must keep up. When stereo rendering and head motion tracking are properly combined, and updated with enough frequency, we can achieve a true feeling of being immersed in the experience.

Input Devices

To create a convincing feeling of immersion, head mounted displays completely enclose the user's eyes, cutting them off from seeing the outside world. This makes for an interesting situation with respect to input: users have to "fly blind," not seeing their mouse or keyboard when using them. Anyone who has tried a demo of the Oculus Rift knows all too well the feeling of your friend guiding your hands to the WASD or arrow keys so that you can use them to navigate your body through a virtual scene.

Most of the time, blind operation of the computer keyboard and mouse does not offer a good experience from a human factors point of view. For that reason, a lot of experimentation is happening, with the following types of input devices being used to provide a greater feeling of immersion:

- **Game controllers**, such as for the Microsoft Xbox One and Sony PS4 consoles. These controllers can be connected to desktop computers to interact with virtual reality scenes.
- **Hand-tracking motion input sensors**. Over the last few years, low-cost motion input devices have been available, including the Leap Motion controller and NimbleVR. These devices track hand motions and recognize gestures, but require no hand contact or touching, similar to the Xbox Kinect.
- **Wireless hand and body trackers,** such as the full body motion STEM system by Sixense and the Hydra by Razer. These devices combine wand-like hand motion sensing with control buttons similar to those found in game controllers.

At this point in time, there is no one way to interact in VR using these new input devices. Much depends on the type of application being created, and the system to which the devices are connected. While it is too early to say what the "mouse of virtual reality" is going to be, it will likely emerge from products in the above set, or devices like them. In the meantime, all this research is a big part of the fun. Expect to see a lot of innovation in this area over the next few years.

Computing Platforms

Many VR applications will run on a majority of existing computers and mobile phones. A relatively modern desktop or high-powered laptop can do the trick with Oculus Rift; smartphones can also offer a good VR experience, provided they have enough CPU and graphics power. For most of us, this means that our existing computers and devices can become virtual reality boxes simply by adding a few peripherals. But for those seeking super-high production value experiences, the latest desktop PC with a best-of-breed graphics processor and fastest CPU might be the top item on the wish list for next Christmas.

As VR matures and gains in popularity, we may also begin to see dedicated computers, phones and consoles-- dedicated, that is, to the sole purpose of enabling amazing virtual reality.

Software to create VR applications comes in several flavors: native software development kits, game engines and frameworks, and even the latest versions of modern web browsers.

Native Software Development Kits (SDKs)

These are the device drivers and software libraries used in conjunction with the computer's host operating system. On Windows they would be Win32 libraries used in C ++ applications; on Android, they would be Java libraries; and so on. One can build a native application simply using the SDKs, and "roll your own" with respect to everything else, such as 3D graphics and game behaviors. But most developers use engines or frameworks.

Game Engines and Frameworks

Unless you are a game engine developer, you probably won't want to deal with the Native SDKs directly; you will more than likely turn to a game engine such as Unity3D (described in detail in Chapter 3). Libraries like Unity3D, also known as *middleware*, take care of low-level details of 3D rendering, physics, game behaviors and interfacing to devices. Most VR developers today build their apps using game middleware like Unity3D.

Many middleware engines have strong cross-platform support, allowing you to write your code once (or most of it at least) and target multiple platforms, including desktop and mobile. They also usually come with a powerful set of tools, known as *level editors* or *integrated development environments* (IDEs).

Web Browsers

Much in the way HTML5 added mobile features over a short number of years, nearly achieving parity with native mobile capability, browser makers are fast-following the

development of virtual reality. In the case of VR, the adoption of features into web browsers is looking like it will only take one or two years, not four to five.

The upshot of this is twofold: first, it means that we can use web technologies like HTML5, WebGL and JavaScript to create our applications, making it potentially faster to code and more cross-platform; second, it affords us access to all of the existing infrastructure the web has to offer, such as hyperlinking between VR experiences, hosting content in the cloud, developing multi-user shared experiences and integrating web data directly into our virtual reality applications.

Video Players

Stereoscopic video represents a whole class of virtual reality technology by itself. Unlike with game engines, where the application's graphics are completely synthetic, based on hand-crafted 3D models, animations, backgrounds, and so on, stereo video is captured from the real world. This makes for truly realistic and often stunning experiences: imagine taking a virtual helicopter tour of the grand canyon in VR, being able to look around as you fly over the canyon. Video is not fully interactive in the way that a 3D virtual environment can be, so the use of this media type has some limitations.

Stereo video recording requires multiple cameras-- at a minimum two. But if we also want the video to be *panoramic*, that is, capture a 360 view of an entire scene for use in virtual reality, then we need even more cameras. Pioneers in the field such as California-based Jaunt VR are experimenting with setups that use dozens of cameras, for making the first VR feature-length films.

The capture and production of VR video is a nascent field. Several companies and research projects are devoting their efforts to it. There are also a variety of VR video players in development. Some players work only on native environments, others only for mobile or for the web; while some developers are creating full cross-platform players. One of the biggest issues in this young endeavor is that there are as yet no standard formats for storing and playing back of the videos, so if you want to produce video content, you may have to choose a single vendor for the hardware, the production tools and the playback software.

Virtual Reality Applications

Even though consumer virtual reality is just a few years old, we are already seeing a staggering range of applications. To say that VR has captured peoples' imaginations would be an understatement. Developers are trying to build virtually *everything* in VR, understandably with mixed success so far.

While it is way too early to pick winners, and identify "killer apps" for VR, there are several domains that show great promise, including:

- **Video Games.** This is an obvious candidate, the one that most people immediately imagine when you tell them about virtual reality. The potential for deep immersion, higher production value and stickier engagement has developers, console makers and peripheral manufacturers salivating in anticipation. It is fair to assume that the majority of skilled, independent and large development shops creating VR are currently doing so for games.
- **Virtual Worlds.** Social, user-generated persistent virtual worlds could be a powerful combination with virtual reality immersion. Companies like High Fidelity, created by *Second Life* founder Philip Rosedale, and AltSpace VR, a new San Francisco bay area startup, are leading this charge.
- **Education.** For years, 3D visualization has been a great tool for interactive learning; VR immersion could make learning even more approachable and effective.
- **Productivity.** Some researchers and small companies are exploring using VR as a replacement for the decades-old desktop computer metaphor. Imagine a virtual reality, 360-degree workspace that holds your personal information, contacts, work projects, etc.
- **Tourism.** Stereoscopic 360 panoramas in VR are really compelling. They represent a simple, effective way to convey the experience of being elsewhere without having to get on a plane.
- **Architecture and Real Estate.** Architecture and real estate firms are already experimenting with virtual reality, using both video and interactive graphics. Video can be great for showing existing properties; interactive graphics works well for visualization buildings and complexes in the planning stages i.e. using 3D CAD models.
- **Live Events.** VR video is promising to be quite popular for concerts, news and other live events. Musicians Paul McCartney and Jack White are among several rock stars who have already broadcast virtual reality versions of their live shows.
- **Web Browsing.** Mozilla is leading the charge in experimenting with adding VR support to their browser, and Google is not far behind creating the same features for Chrome. Beyond building the technology plumbing, the research team at Mozilla is exploring visual and interface designs for how to navigate a universe of information in virtual reality.
- **Enterprise Applications.** There are countless potential VR applications for the enterprise, including simulation and training for military use, medical diagnostics and training, and engineering and design.

The above list is just a sampling. We have no way of knowing which applications will be most successful in a new medium like virtual reality, or what will become popular. We also can't predict what other ideas will spring from the minds of clever folks. But if the last few decades of technology have taught us anything, it's that whatever people come up with, it will probably be something we never could have imagined before.

Chapter Summary

Consumer virtual reality is upon us. As we have seen in this chapter, VR brings together several technologies, including 3D stereoscopic displays, motion tracking hardware, new input devices, computers and mobile phones. The key innovations that enable virtual reality are stereoscopic rendering and motion tracking. When these two are properly combined, we feel immersed, or *present*, and the illusion of VR is compelling enough to transport us to another place.

Developing for virtual reality can take on various forms: native SDKs that access raw platform features; game engines that give us more power and cross-platform reach; web browsers, for creating shared, connected virtual reality apps; and video players and tools for creating panoramic stereo videos based on the real world.

Though the technology is still immature, virtual reality is already being used in a number of applications, from games to education, from real estate to rock concerts. Developers are also creating productivity and enterprise applications, looking to the power of VR immersion to enhance business value.

In the next chapter we will explore the wide variety of virtual reality hardware systems in use today. Let's get to it.

CHAPTER 2

Virtual Reality Hardware

In this chapter we will take a look at popular consumer virtual reality hardware in use today, including VR headsets and the computers or phones they run on.

Even though this is a rapidly evolving industry, some manufacturers are already becoming established as leaders, most notably Facebook with the Oculus Rift. But there are several other head mounted displays (HMDs) to choose from. Some HMDs require using desktop computers, others are just for smartphones, and still others are for use with game consoles. To further muddy the picture, new HMDs are coming out all the time.

Let's examine three headsets that represent a wide range of HMDs on the market today, and that will form the focus of the rest of the chapters in the book: the Oculus Rift for desktop virtual reality; Samsung's Gear VR (based on Oculus technology) for a high-end mobile VR experience; and Google's Cardboard VR, a simple, low-cost way of adapting your existing smart phone to become a VR device.

The Oculus Rift

While there have been several attempts to market consumer virtual reality headsets over the past few decades, nothing ever pushed it over the top until the Oculus Rift came along. Buoyed by an extremely successful Kickstarter campaign to fund its original development, Oculus VR raised significant venture funding and ultimately sold the company to Facebook in a blockbuster $2B acquisition that the industry is still

talking about. Without that watershed event, it is unlikely virtual reality would have gotten as popular as quickly, or captured the public's interest in the way it has. But it did, and it has, and here we are.

The Oculus Rift has spawned an entire VR industry. This includes companies large and small creating applications, tools, peripheral hardware and development services. Numerous applications are being built for the Oculus Rift, spanning gaming, architecture, medicine, real estate, tourism, entertainment and education. , With its voracious appetite for CPU and graphics processor cycles, the Rift has even given a shot in the arm to the declining desktop PC industry.

The Oculus Rift is a stereoscopic display with built-in head motion tracking sensors. It straps to the head, allowing hands-free operation. The Rift is a peripheral: it attaches to a computer: Mac, Linux or Windows; desktop or laptop. The Rift is tethered, with a cable running to the computer. At the moment the Rift is quite bulky-- but that will most certainly change with the newer models being designed as we speak.
The following is a brief history of the Oculus Rift, by way of exploring the evolution of this groundbreaking VR headset.

The Gold Standard of Virtual Reality

By any reckoning, as of this writing the Oculus Rift is the virtual reality headset against which all others will be measured in coming years. While the current development kit versions still feel clunky in some ways, each iteration gets noticeably better than the last. Oculus VR has publicly announced plans for a retail consumer headset release in late 2015/early 2016 that promises to be affordable, comfortable and amazing.

The DK1

The original Oculus Rift developer kit, known as the DK1, is a big affair-- weighing in at over 13 ounces (.33kg) and measuring 7 inches diagonal-- and therefore not very comfortable for extended use. It has an external control box in addition to two sets of wires, one for the USB connection to the head tracker, and one for the video signal coming from the computer. The DK1 is depicted in Figure 2-1.

Figure 2-1. The Original Oculus Developer Kit, or DK1

The DK1 has a resolution of 1280 x 800 pixels. This screen real estate is divided between the two eyes, so the effective resolution is 640 x 800; in other words, the DK1 is a low-resolution device by today's standards for computer monitors. The display has a horizontal field of view of well over 90 degrees, which is important for emulating the visual field of view when experience in the real world. The head tracking of the inertial measurement unit (IMU) is quite fast at 250Hz; this is critical to enable a true sense of immersion and to reduce motion sickness.

Despite its low visual resolution, the wide field of view and rapid head tracking response time of the DK1 made it the first practical low-cost VR headset, at least for short sessions. And it was promising enough to get the Oculus VR off the ground as a company. Oculus made DK1 units available initially to its Kickstarter backers in the summer of 2012, then for sale online to the general public in late 2012. Then, infused with a large round of venture capital, the company went straight to work on the next generation, the DK2.

The DK2

In the spring of 2014, Oculus released the second major incarnation of its headset, code-named DK2. In contrast with the unwieldy DK1, the Oculus Rift DK2 is a sleek device. It is still quite large, covering a good portion of the user's face. But it is much lighter and features a more pleasing form factor than its predecessor. It also has only

one cable coming out of the HMD, which then splits into the HDMI and USB cables that connect to the computer's video and USB ports, respectively.

The DK2 improved upon its predecessor with lower-persistence, 1920 x 1080 display (960 x 1080 per eye). Beyond the display improvements, the DK2 incorporated a major advancement: positional head tracking. This allows the user to not only look around, but move around within a virtual scene - forward, back, side to side, and up and down. However, use of the positional tracker requires being in front of a tracking camera mounted on the computer, which effectively means the user needs stay within a small zone near the computer. The positional tracking also only works when the user is facing toward the camera.
The DK2 is depicted in Figure 1-2 (see Chapter 1). The tracking camera for the DK2 is pictured in Figure 2-2.

Figure 2-2. Position Tracking Camera for the Oculus Rift DK2

DK2 and the Examples in This Book

As of this writing, the DK2 is the primary Oculus Rift device being used in development. The examples throughout the book have been developed for DK2, using the associated software development kit (SDK) versions.

"Crescent Bay"

The third major Oculus Rift developer version is code-named *Crescent Bay*. Crescent Bay features greater capabilities and big improvements in design. It has a higher resolution display and 360-degree head tracking-- meaning that you don't have to be looking at the tracking camera; there are position sensors mounted at the rear of the device as well. Crescent Bay is depicted in Figure 2-3.

Figure 2-3. The Newest Oculus Rift, Code-Named "Crescent Bay"

As of this writing, Crescent Bay is only available as a demo. It is presumed that it, or a model quite close to it, will ship as a developer kit version sometime in 2015, in preparation for the first Oculus retail headset release in late 2015/early 2016.

Really Real Virtual Reality

Anyone who has been privileged to try a demo of Crescent Bay will attest that it represents a major leap in capability over the DK2. In my opinion, Crescent Bay is the difference between experiencing the *promise* of virtual reality and actually experiencing it. The level of realism and feeling of presence while in the demos is unmatched. There were many moments when I completely forgot that I was in a simulation... which I believe is the entire point. Long story short: if you're ever offered a chance to check it out, don't pass up the chance.

As you can see, Oculus VR is continually striving to improve on its creation. The three major development kits each represent a big step forward in features and comfort. If this trajectory continues, then something truly astounding is in store for us when the company ships its first commercial version.

Setting Up Your Oculus Rift

If you have decided to take the plunge and order an Oculus Developer kit, then this section can help you get going with your new device.

If you are ready to take the plunge and want to order a developer kit, go to the Oculus Products tab and select DK2 to order it online:

https://www.oculus.com/dk2/

What Computer Do I Need?

To have the best experience with the Oculus Rift, you will want to connect it to a powerful desktop computer or laptop. The following table lists a few models known to work well with the DK2.

[to do: add info for each configuration]

Computer	Specifications
Dell Laptop	processor, GPU etc.
Alienware Laptop	processor, GPU etc.
MSI Laptop	processor, GPU etc.
Macbook Pro 15" Retina	processor, GPU etc.

Downloading the Oculus Runtime, Examples and SDK

The Oculus Rift is not yet a consumer product: setting up the hardware and software can be tricky and requires a modicum of technical knowledge and a lot of patience. In order to experience content using the Rift, you must at a minimum download the Oculus Runtime for your platform. The runtime provides the software necessary for the Rift to communicate with your computer's operating system.

Oculus runtime installers can be found on the Oculus developer site at

http://developer.oculus.com

Once you have the runtime installed, you can try out the Oculus Rift by downloading and installing applications from a variety of sources. Appendix X contains a list of several sites that feature Oculus content. To get started, try out the official portal from Oculus, *Oculus Share*.

https://share.oculus.com/

If you also wish to develop for the Oculus Rift, you will need to become an Oculus Developer and download the software development kit (SDK). If you don't have one already, go to Oculus developer site to sign up for a free developer account. Then, log in and follow the instructions for your particular platform. Once you are set up with the SDK, you can develop desktop Oculus applications as described in Chapters 3 through 5.

Samsung Gear VR: Deluxe, Portable Virtual Reality

The Oculus Rift may be the flagship VR headset, but it is not without its issues. First, it requires a very powerful computer with a fast graphics processor. If you try to run Rift applications with a garden variety PC laptop, Macbook Air, or older desktop computer, you won't have much fun. The frame rate will be slow, which is not only dissatisfying, but can actually produce nausea. Second, the Rift is bulky and connected to the computer with a wire. If you want to run applications that use DK2 positional tracking, you will need to install the position tracking camera that is shipped with the DK2 and sit yourself down right in front of it for the tracking to work. Finally, it is not very mobile. Yes, you can pack up a six pound laptop and your Rift to take to a demo party, but it's not the kind of thing you will want to bring to the Starbucks (believe me, I've seen it done and it's not pretty), or on your train commute. This all makes for a cumbersome, stationary experience-- and a potentially expensive one if you also need to buy a new computer.

Anticipating these issues, Oculus has also produced a much lighter-weight mobile solution. Through a partnership with Samsung, Oculus technology has been incorporated into *Gear VR*, a revolutionary headset that combines Oculus optics (those barrel distortion lenses) with new head tracking technology, placed in a custom headset that houses a mobile phone with a high-resolution display.

Samsung released the Gear VR as an "Innovator Edition" in late 2014. The Gear VR unit costs US$199 and can be purchased through Samsung's online store. Gear VR currently only works with the Samsung Galaxy Note 4, a US$800 phone [footnote here: the Note 4 can be purchased for as low as $299 if with a new two-year carrier plan], or the Samsung S6, priced at US$600, so this is a pricey option. But it is very high quality, and may be the killer mobile option for the next few years. The original Gear VR Innovator Edition is depicted in Figure 2-4.

Figure 2-4. The Samsung Gear VR

The Gear VR's graphics are stunning, thanks to the Galaxy Note 4's display resolution of 2560 x 1440 pixels (1280 x 1440 per eye). The headset contains a custom IMU based on Oculus technology. The Gear VR IMU tracks much faster and with less latency (delay) than the IMU built into the phone itself.

The Gear VR housing has several intuitive input controls, including an eye adjustment wheel (for matching the device's inter-pupillary distance to your own), as well as controls that let you access functions of the phone within: a volume control, a headphone jack, and most importantly, a trackpad for touch input.

The Gear VR is a fantastic consumer VR headset. A quick trip through the available demos and apps might even convince you that it's the best one on the market today. It offers a better experience than DK2, with a much nicer form factor and amazing display resolution. If money were no object, this could be it. However, at a eight hundred to a thousand dollars including a newly purchased phone, the price is still prohibitive. This situation could change if Samsung can figure out how to accommodate other phones, drop the total cost of ownership, or otherwise change the equation.

The Oculus Mobile SDK

Developing applications for Gear VR's Android-based operating system requires a different Oculus SDK, called the Mobile SDK, which is also available on the Oculus developer website:

http://developer.oculus.com

We will cover the basics of Gear VR development in Chapter 4.

Google Cardboard: Low-Cost VR for Smart Phones

If they were automobiles, the Oculus Rift and Gear VR would be the Tesla and Lamborghini of VR headsets: best-of-breed but out of reach for the average consumer. The rough user experience, aggressive machine specs and price tag in the hundreds of dollars mean that using an Oculus is a major commitment. Gear VR provides a sublime experience, but for now it is priced out of reach for anyone but a serious enthusiast or early adopter. Over time, the hope is that these higher-end consumer VR systems will become more affordable and accessible, and truly mass-consumer.

In the meantime, there is Google's *Cardboard VR*: a simple, low-cost way of adapting your existing smart phone to become a VR device. In 2014 Google introduced Cardboard VR to enable using pretty much any mobile phone without needing new hardware. Cardboard requires just a box with two lenses-- costing approximately US$2 in parts-- into which you place your phone.

Google's original Cardboard VR unit, debuted at the I/O conference in May 2014, is pictured in Figure 2-6.

Figure 2-5. The Original Google Cardboard VR Viewer, Shown at Google I/O in 2014

To experience Cardboard VR, simply launch a Cardboard-ready application and place your mobile phone into the box. You will be immersed in a VR experience: a stereo-rendered 3D scene which you can look around and move around in by turning your head. Not bad for 2 bucks!

Cardboard is actually a reference specification. Google doesn't offer it as a product; you can get the specifications from them and build one of your own. The Cardboard specifications can be found at

https://www.google.com/get/cardboard/manufacturers.html

If you don't have the time or inclination to build a cardboard from scratch, you can also purchase a ready-to-assemble kit from one of several manufacturers, including DODOcase, I Am Cardboard, Knox Labs, and Unofficial Cardboard. In addition to selling kits, each of these manufacturers also provides a mobile app, downloadable from the Google Play store and/or the iTunes store, that provides a handy list of Cardboard-aware VR applications.

According to Google, over one million Cardboard viewers had shipped as of the first quarter of 2015. This number dwarfs the installed bases of Oculus Rift and Gear VR, and on that basis alone, Cardboard is a force to be reckoned with. There are already hundreds of applications available for Cardboard, including games, 360-degree video and photo viewers, and educational simulations. Cardboard has powered VR experiences with big-name entertainers like Sir Paul McCartney and Jack White, and brands such as Volvo. So while insiders will tell you that the Cardboard VR experi-

ence isn't as deep or immersive as Oculus Rift, for many people, it's going to be their first VR experience.

But Does it Have to be Made of *Cardboard*?

No...

to do: URLs to a few plastic and foam makers

Stereo Rendering and Head Tracking with Cardboard VR

The Cardboard VR approach to stereo rendering is simpler than Oculus: it is an undistorted, 90 degree horizontal field of view. This allows applications to do simple side-by-side rendering in two viewports, one for each eye. An example of a 3D scene rendered side-by-side for Cardboard, from the game *Dive City Rollercoaster*, is shown in Figure 2-7.

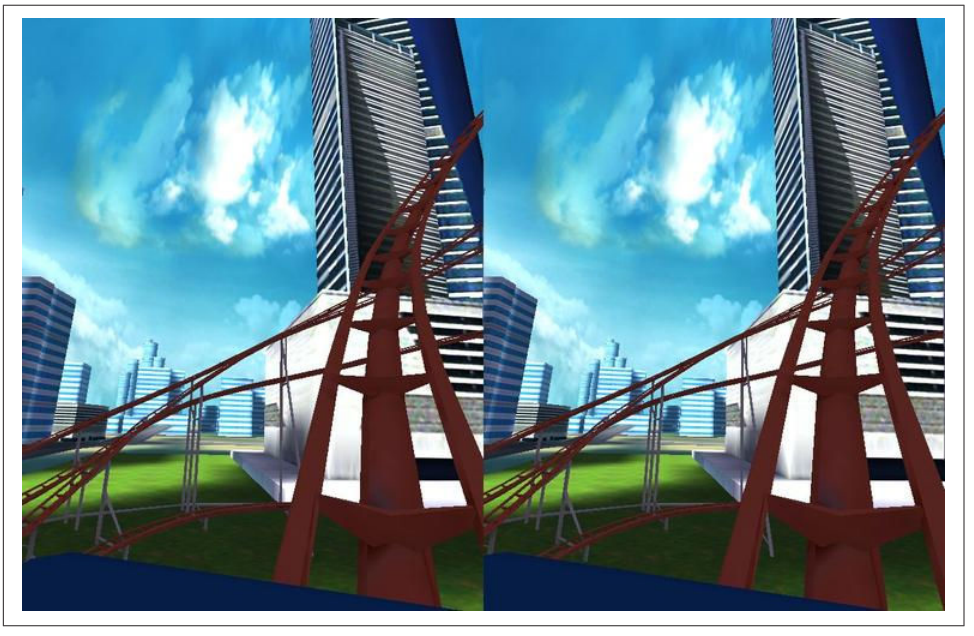


Figure 2-6.
Dive City Rollercoaster for Google Cardboard. Developed by Dive Games

https://play.google.com/store/apps/details?id=com.divegames.divecitycoaster&hl=en

Head tracking for Cardboard is also straightforward. It simply uses the existing operating system orientation events generated by the phone's compass and accelerometer.

Cardboard VR Input

For Cardboard VR, input is still a work in progress. Because the phone is encased in the box, you can't touch the screen to tap or swipe. To get around this, the original Google unit featured a magnet mounted on the outside of the box that will trigger compass events that can be detected by an application. However, that combination works only with certain phone models.

Other makers of Cardboard units are developing innovations that will enable more universal input and work with most phones. For example, DODOcase, a San Francisco-based company that manufacturers tablet and phone cases, is now making a Cardboard unit that uses a wooden lever to trigger a piece of plastic that touches the surface of the phone, emulating the touch of a finger. With this mechanism, developers can use standard touch events in their VR applications. Several cardboard makers are experimenting with creating peripherals, such as Bluetooth-connected devices that provide simple click or swipe input.

Developing for Google Cardboard

Google provides SDKs for Cardboard developers in two forms: an Android native SDK, and an add-in for the popular Unity3D game development system. Those can be downloaded from the Android developer site at

https://developers.google.com/cardboard/overview

In Chapter 6, we will cover the details of building a Cardboard app using the Android SDK.

Note that you don't need to be an Android developer to write for Cardboard. Because most Android operating systems now support WebGL, you can actually write mobile web VR applications too! This is an exciting new area for VR development that we will cover in Chapter 5.

Chapter Summary

Consumer VR hardware is a young an evolving field. The Oculus Rift leads this evolution with breakthrough display and head tracking technology. New developer models are being released each year, on the way toward a broad consumer release in late 2015/early 2016. Each version of the Rift gets better, more ergonomic and more consumer-friendly.

While the Oculus Rift is the standard-bearer for virtual reality, it's not the only game in town. Gear VR, based on Oculus technology, is an outstanding product. Its high-

resolution display, lightweight headset with intuitive controls, and the Oculus Store user interface, arguably provide the highest-quality VR experience to date. However, at a total cost of ownership of US$1,000, it is currently out of reach for most consumers.

On the other end of the spectrum, Cardboard VR turns your existing smart phone into a VR device using just a few dollars worth of parts-- a cardboard box and two plastic lenses. While it may not be the high-end experience of a Gear VR or the ultimate Oculus Rift retail release, Cardboard represents a low-cost alternative that could provide a first taste of VR for many.

Well, that's it for our nickel tour of VR hardware. It's time to go build something.

CHAPTER 3

Going Native: Developing for Oculus Rift on the Desktop

The next several chapters cover the essentials of virtual reality development. Let's start by creating a native application for desktop computers using the Oculus Rift. While the information in this chapter is specific to the Rift and the Oculus SDK, the same techniques will apply when developing for the HTC Vive and other headsets.

In programming our first VR application, we will explore the following core concepts:

- **Constructing a 3D scene.** Creating the visual, interactive and scripted elements that represent the virtual reality environment.
- **Rendering the 3D scene in stereo.** Rendering the scene from each of two cameras representing the user's eyes. Via the lenses in the Oculus Rift headset, the two rendered images are combined into a single, coherent visual field to create a stereoscopic view of the environment.
- **Head tracking to provide presence.** Using the position and orientation of the Oculus Rift headset to change the virtual camera's position and orientation within the environment.

To illustrate these ideas, we need to write a lot of 3D code. Programming directly to a 3D rendering API like OpenGL or DirectX is a lot of work, and is beyond the scope of this book. Instead, we are going to use the popular *Unity3D* game engine. Unity provides enough power to build games and VR quickly without too steep of a learning curve.

But before we delve into Unity, let's make sure we understand the basics of 3D graphics. If you are already familiar with 3D programming concepts, feel free to skip to the next section.

3D Graphics Basics

3D: A Definition

Given that you picked up this book, chances are you have at least an informal idea about what we are talking about when we use the term *3D graphics*. But to make sure we are clear, we are going to get formal and examine a definition. Here is the Wikipedia entry (from http://en.wikipedia.org/wiki/3D_computer_graphics).

> 3D computer graphics (in contrast to 2D computer graphics) are graphics that use a three-dimensional representation of geometric data (often Cartesian) that is stored in the computer for the purposes of performing calculations and rendering 2D images. Such images may be stored for viewing later or displayed in real-time.

Let's break this down into its components: 1) the data is represented in a 3D coordinate system; 2) it is ultimately drawn ("rendered") as a 2D image, for example, on your computer monitor; or in the case of VR, it is rendered as two separate 2D images, one per eye; and 3) it can be displayed in real-time: when the 3D data changes as it is being animated or manipulated by the user, the rendered image is updated without a perceivable delay. This last part is key for creating interactive applications. In fact, it is so important that it has spawned a multi-billion dollar industry dedicated to specialized graphics hardware supporting real-time 3D rendering, with several companies you have probably heard of such NVIDIA, ATI, and Qualcomm leading the charge.

3D programming requires new skills and knowledge beyond that of the typical app developer. However, armed with a little starter knowledge and the right tools, we can get going fairly quickly.

3D Coordinate Systems

If you are familiar with 2D Cartesian coordinate systems such as the window coordinates used in a Windows desktop application or iOS mobile app, then you know about x and y values. These 2D coordinates define where child windows and UI controls are located within a window, or the where virtual 'pen' or 'brush' draws pixels in the window when using a graphics drawing API. Similarly, 3D drawing takes place (not surprisingly) in a 3D coordinate system, where the additional coordinate, z, describes depth, i.e. how far into or out of the screen an object is drawn. If you are already comfortable with the concept of the 2D coordinate system, the transition to a 3D coordinate system should be straightforward.

The coordinate systems we will work with in this book tend to be arranged as depicted in Figure 3-1, with x running horizontally left to right, y running vertically from bottom to top, and z going in and out of the screen. The orientation of these axes is

completely arbitrary, done by convention; in some systems, for example, the z axis is the vertical axis, while y runs in and out of the screen.

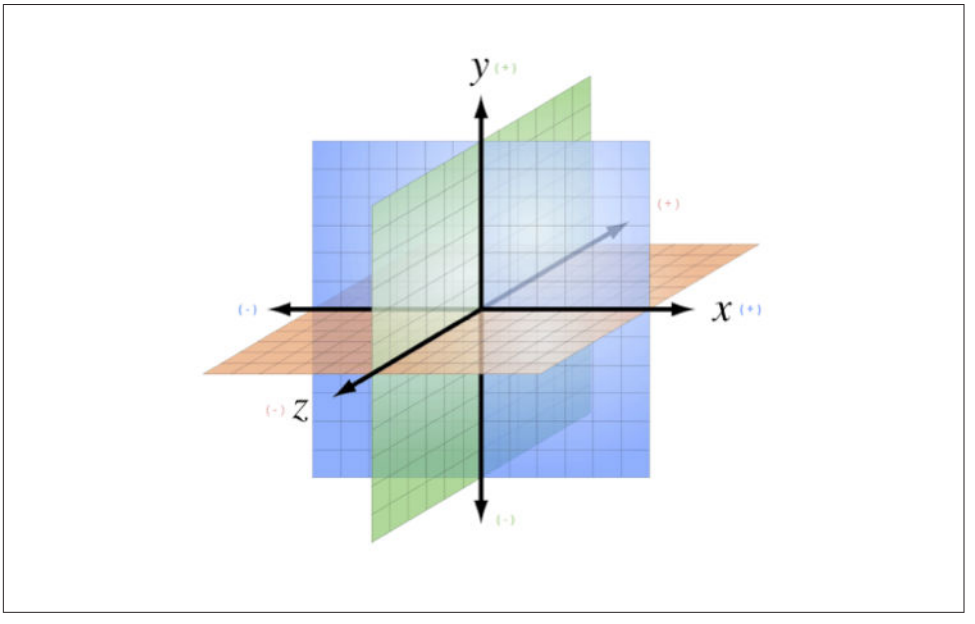


Figure 3-1. 3D Coordinate System

Unity3D, the tool we will use for the examples in this chapter, follows the coordinate system depicted in Figure 3-1: z is the in/out axis, with positive z going into the screen. This is known as a *left-handed coordinate system*, vs. the *right-handed coordinate system* often used in OpenGL applications, where positive z comes out of the screen.

Left Hand/Right Hand?

Remember, the orientation of the axes in 3D coordinate systems are arbitrary conventions. We will see examples of right-handed coordinates when we cover WebVR in a later chapter. Don't worry over it; it is fairly easy to make the mental transition from one system to another once you're comfortable with the general idea.

Meshes, Polygons and Vertices

While there are several ways to draw 3D graphics, by far the most common is to use a mesh. A mesh is an object composed of one or more polygonal shapes, constructed out of vertices (x, y, z triples) defining coordinate positions in 3D space. The poly-

gons most typically used in meshes are triangles (groups of three vertices) and quads (groups of four vertices). 3D meshes are often referred to as models.

Figure 3-2 illustrates a 3D mesh. The dark lines outline the quads that comprise the mesh, defining the shape of the face. (You would not see these lines in the final rendered image; they are included for reference.) The x, y and z components of the mesh's vertices define the shape *only*; surface properties of the mesh, such as the color and shading, are defined using additional attributes, as we will discuss below.

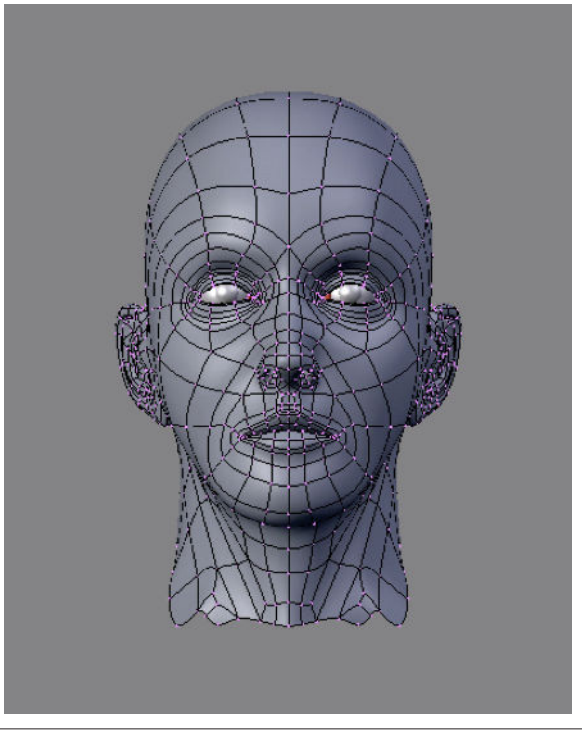


Figure 3-2. A 3D Polygonal Mesh

Materials, Textures and Lights

The surface of a mesh is defined using additional attributes beyond the x, y, and z vertex positions. Surface attributes can be as simple as a single solid color, or they can be complex, comprising several pieces of information that define, for example, how light reflects off the object or how shiny the object looks. Surface information can also be represented using one or more bitmaps, known as *texture maps* (or simply textures). Textures can define the literal surface look (such as an image printed on a T-

shirt), or they can be combined with other textures to achieve sophisticated effects such as bumpiness or iridescence. In most graphics systems, the surface properties of a mesh are referred to collectively as *materials*. Materials typically rely on the presence of one or more lights, which (as you may have guessed) define how a scene is illuminated.

The head depicted in Figure 3-2 above has a material with a purple color and shading defined by a light source emanating from the left of the model. Note the shadows on the right side of the face.

Transforms and Matrices

3D meshes are defined by the positions of their vertices. It would get really tedious to change a mesh's vertex positions every time you want to move it to a different part of the view, especially if the mesh were continually animating. For this reason, most 3D systems support transforms, operations that move the mesh by a relative amount without having to loop through every vertex, explicitly changing its position. Transforms allow a rendered mesh to be scaled, rotated and translated (moved), without actually changing any values in its vertices.

Figure 3-3 depicts 3D transforms in action. In this scene we see three cubes. Each of these objects is a cube mesh that contains the same values for its vertices. To move, rotate or scale the mesh, we do not modify the vertices; rather, we apply transforms. The red cube on the left has been translated 4 units to the left (-4 on the x-axis), and rotated about its x- and y-axes. (Note that rotation values are specified in *radians*—units that represent the length of a circular arc on a unit circle, with 360 degrees being equal to 2 * PI.) The blue cube on the right has been translated 4 units to the right, and scaled to be 1.5 times larger in all three dimensions. The green cube in the center has not been transformed.

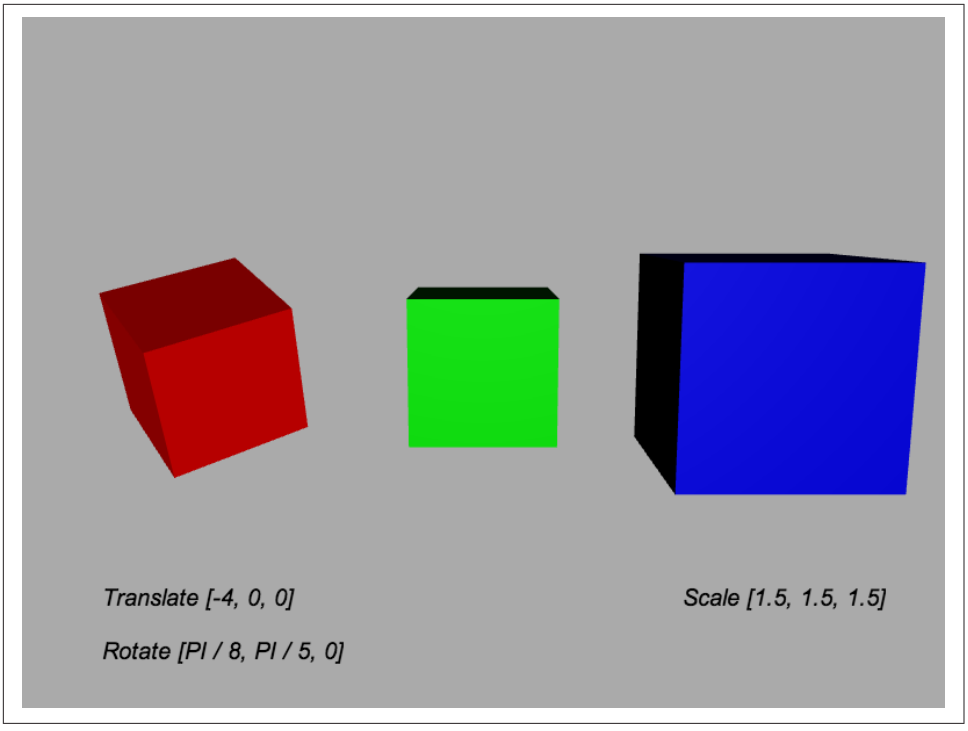


Figure 3-3. 3D Transforms: Translation, Rotation and Scale

A 3D transform is typically represented by a transformation matrix, a mathematical entity containing an array of values used to compute the transformed positions of vertices. Most 3D transforms use a 4x4 matrix, that is, an array of 16 numbers organized into four rows and four columns. Figure 3-4 shows the layout of a 4x4 matrix. The translation is stored in elements m12, m13 and m14, corresponding to the x, y and z translation values. x, y and z scale values are stored in elements m0, m5 and m10 (known as the diagonal of the matrix). Rotation values are stored in the elements m1 and m2 (x-axis), m4 and m6 (y-axis), and m8 and m9 (z-axis). Multiplying a 3D vector by this matrix results in the transformed value.

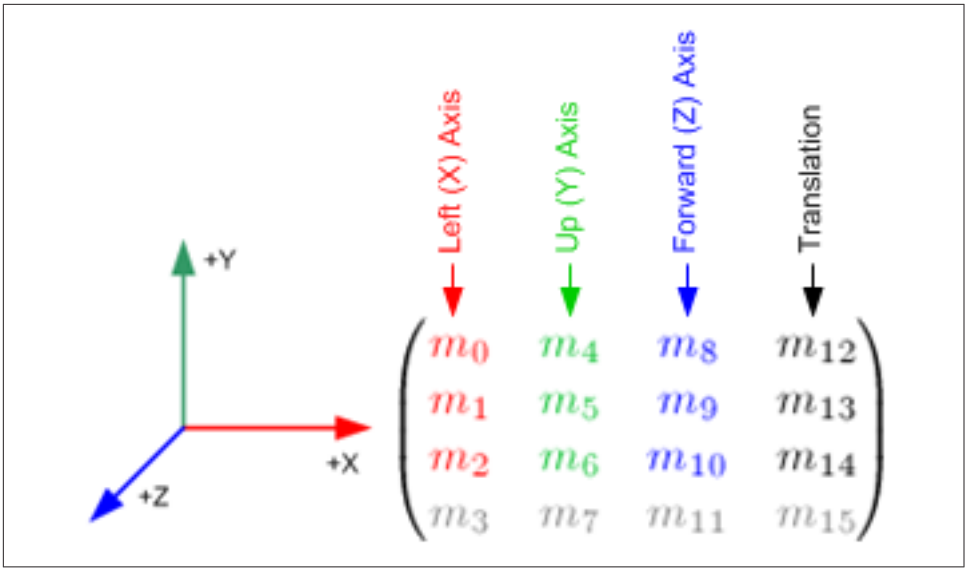


Figure 3-4. A 4x4 Transformation Matrix Defining Translation, Rotation and Scale

If you are a linear algebra geek like me, you probably feel comfortable with this idea. If not, please don't break into a cold sweat. Unity3D and the other tools we are using in this book allow us to treat matrices like black boxes: we just say translate, rotate or scale and the right thing happens. But for inquiring minds, it's good to know what is happening under the covers.

Cameras, Perspective, Viewports and Projections

Every rendered scene requires a point of view from which the user will be viewing it. 3D systems typically use a camera, an object that defines where (relative to the scene) the user is positioned and oriented, as well as other real-world camera properties such as the size of the field of view, which defines perspective (i.e. objects farther away appearing smaller). The camera's properties combine to deliver the final rendered image of a 3D scene into a 2D viewport defined by the window or canvas.

Cameras are almost always represented using a couple of matrices. The first matrix defines the position and orientation of the camera, much like the matrix used for transforms (see above). The second matrix is a specialized one that represents the translation from the 3D coordinates of the camera into the 2D drawing space of the viewport. It is called the projection matrix. I know: more math. But the details of camera matrices are nicely hidden in Unity3D, so you usually can just point, shoot and render.

Figure 3-5 depicts the core concepts of the camera, viewport and projection. At the lower left we see an icon of an eye; this represents the location of the camera. The red

vector pointing to the right (in this diagram labeled as the X axis) represents the direction in which the camera is pointing. The blue cubes are the objects in the 3D scene. The green and red rectangles are, respectively, the near and far clipping planes. These two planes define the boundaries of a subset of the 3D space, known as the view volume or view frustum. Only objects within the view volume are actually rendered to the screen. The near clipping plane is equivalent to the viewport, where we will see the final rendered image.

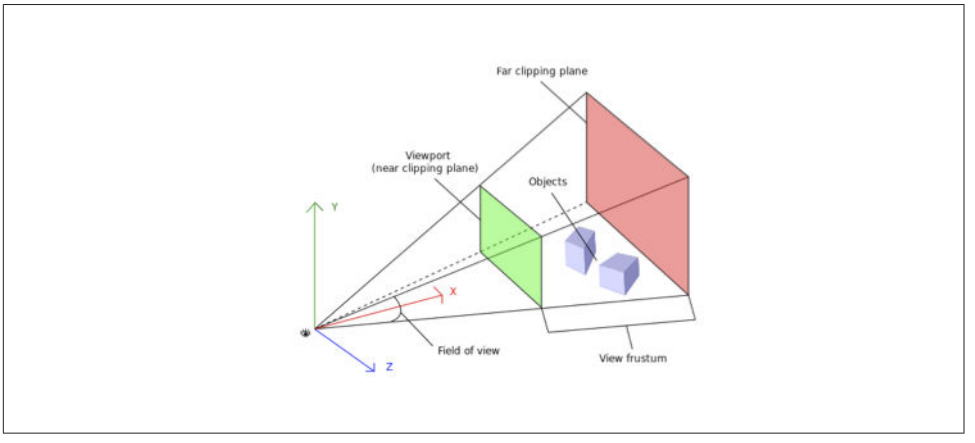


Figure 3-5. Camera, Viewport and Perspective Projection

Cameras are extremely powerful, as they ultimately define the viewer's relationship to a 3D scene and provide a sense of realism. They also provide another weapon in the animator's arsenal: by dynamically moving around the camera you can create cinematic effects and control the narrative experience. Of course, with VR, camera control has to be a balance between moving the user around within the scene and allowing them the freedom of movement necessary to convey a sense of presence. We will explore this idea more in later chapters.

Stereoscopic Rendering

Rendering 3D for virtual reality adds a wrinkle to how we deal with cameras, viewports and projection matrices. In essence we have to do the whole thing twice, once per eye. Thankfully, this is easier than it sounds. There are a few approaches to doing it, but the simplest is as follows:

- **Define one main camera for the simulation.** The application maintains a single main camera, as if it were being rendered in mono. All animations and behaviors that affect the camera, such as terrain following, collisions, or VR head tracking, are performed on this one main camera. This provides a simple and consistent way to deal with the camera as an object, separate from the concerns of render-

ing. It also allows the developer to easily create both stereo and mono versions of the same application, when that is appropriate.
- **Render from two cameras.** The application maintains two additional cameras, which are used only to render the scene. These cameras always follow the position and orientation of the main camera, but are slightly offset to the left and right of it, to mimic the user's interpupillary distance.
- **Render to two viewports.** The application creates a separate viewport for the left and right rendering cameras. This viewport is half the width of the full screen, and the full height. The graphics for each camera are rendered to the respective viewport using a projection matrix set up specifically for each eye, using the optical distortion parameters of the device (provided by the Oculus SDK).

And there you have it: the basics of 3D graphics for VR, in a few pages. There is a lot of stuff to get right, and even the simplest details can take time to master. And as you scale up your development from simple to more complex applications, you will want to do more without have to get mired in low level coding. For these reasons, it is best to use a game engine. You can write an engine of your own, if you are so included. But if you're like me, you would rather spend your time working on the application itself and just use an existing engine. Thankfully, there are several good ones, including the subject of the next section, Unity3D.

Unity3D: The Game Engine for the Common Man

Not all virtual reality applications are games. However, professional game engines have become a tool of choice for developing virtual reality, because of the close fit between the capabilities of the engines and the requirements for creating good VR. Game engines provide a host of features, including high-quality rendering, physics simulations, real-time lighting, scripting, and powerful WYSIWYG editors.

While there are several great game engines on the market, the go-to solution for VR seems to be Unity Technologies' *Unity3D*. Unity has emerged as the preferred tool for indie and hobbyist game development, due to its combination of power and accessibility at a great price. Highlights of Unity3D include:

- **Powerful**. The Unity player runtime provides many important graphics features, such as a rich material system with physically-based rendering, real-time lighting, physics, and scripted behaviors.
- **Extensible**. Unity is based on an entities-and-components model that makes the system flexible, extensible and configurable via user scripts; even the editor's features can be overriden, allowing for the creation of custom editing tools.
- **WYSIWYG editor**. The Unity editor can be daunting at first, but once you are familiar with it, you will find it easy to use. The editor supports many productivity features and work flows, and can import models and scenes from professional tools like Autodesk Maya and 3ds Max.

- **Portable**. The player runtime supports native desktop (OS X, Windows, Linux), popular games consoles such as Xbox and PlayStation, mobile operating systems including iOS and Android, and the Web via both a player plugin and experimental support for WebGL. This means that developers can invest the time to learn and master Unity and be assured that they can port their work to other environments.
- **Affordable**. The free version of Unity is fully featured. For commercial use, the company offers reasonable licensing terms that include an affordable monthly fee and a modest royalty - but only if you are charging for your application.
- **Rich ecosystem.** Unity has an online Asset Store featuring countless 3D models, animations, code packages and utilities. The Asset Store is in a large way responsible for Unity having one of the most vibrant developer communities on the planet.
- **Ready for VR.** The Oculus SDK comes with a Unity integration package that provides support for key virtual reality programming constructs, and sample scenes and code to get started.

To work with the samples in this chapter, you will need to download and install Unity on your computer. Go to the download page and follow the instructions:

http://unity3d.com/get-unity

Once installed, launch Unity, create a new empty 3D project, and explore the product. Figure 3-6 shows an OS X screen shot of the Unity interface. Here are some of the basics: There is a main scene view, which can be configured into a four-view layout (left, top, right, and perspective), or other types of layouts. The **Hierarchy** pane lets you browse through the objects in the current scene. The **Project** pane shows you a view of all the assets currently in your project-- these may or may not be loaded into the current scene. The **Inspector** pane provides detailed property information, and the ability to edit properties, for the currently selected object.

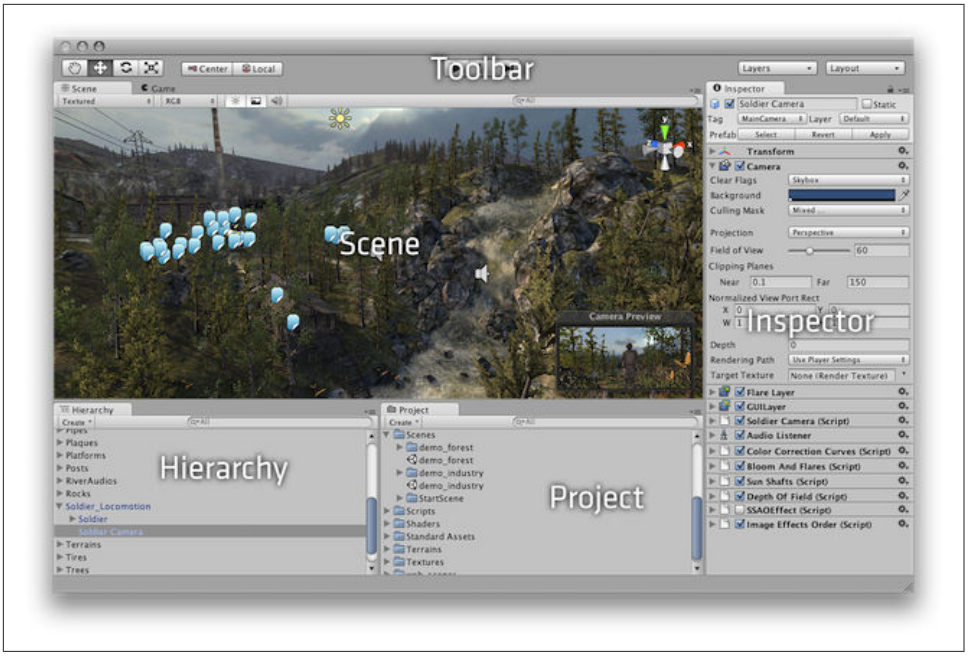


Figure 3-6.
Unity3D Editor Interface

From the official product documentation (http://docs.unity3d.com/Manual/LearningtheInterface.html)

Is Unity the Only Way to Make VR?

You may be wondering if Unity is the only game in town when it comes to creating VR applications. The short answer is: no. You can always use the capabilities of the Oculus SDK to write your own native application in C++, straight to OpenGL or DirectX. Or, you can use any of a number of other great commercial engines such as the Unreal Engine or Crytek's CryENGINE. For information on these products, see Appendix X.

Setting up the Oculus SDK

Before you can use a tool like Unity3D to develop for the Oculus Rift, you also need to install the Oculus SDK, which is available on the Oculus developer website:

http://developer.oculus.com

If you don't have an account, sign up; it's free. Once you are logged in to the site, do the following:

1. Select **Downloads** from the top navigation bar.
2. Select **Platform: PC** from the first pull-down.
3. Under the **SDK & RUNTIME** section, download the *Oculus Runtime* by clicking the **Details** button. Agree to the terms of the end-user license agreement (EULA) and click **Download**.
4. Run the installer for the Oculus Runtime on your PC.
5. Under the **SDK & RUNTIME** section, download the *Oculus SDK* by clicking the **Details** button. Agree to the terms of the end-user license agreement (EULA) and click **Download**.
6. Download and unzip the SDK to your hard drive, in a location of your choice.
7. While you're here, grab the Unity package for Oculus. Under the **ENGINE INTEGRATION** section, download the Unity 4 Integration by clicking the **Details** button. Agree to the terms of the end-user license agreement (EULA) and click **Download**. Note that we'll be using Unity 5 for our development, but this version of the package is OK.
8. Download and unzip the Unity Integration package to your hard drive, in a location of your choice.

With the Oculus SDK installation complete, we can now move on to setting up Unity3D for use with the SDK.

The Oculus SDK On Non-Windows Platforms

The Oculus SDK, and therefore the Unity support for Oculus, currently only runs on Windows-- the OS X and Linux support were mothballed in early 2015. This situation will potentially change, but as of this writing, we can only use Unity to develop for Oculus on Windows. If you have a Mac or Linux machine you can follow along, but unless you had already installed the Oculus SDK prior to the spring of 2015, for those systems you won't be able to run the examples.

Setting Up Your Unity Environment for Oculus Development

Now that you have downloaded the Oculus runtime, the SDK, and the Unity Integration package, you are ready to start developing for Unity. You do that by importing the package into a Unity project. Let's start by creating an empty Unity project. Launch Unity3D, or if it is already running, select **File->New Project...**

Name your new project *UnityOculusTest*. Now that the project has been created, we will import the SDK into it and build a simple application.

Figure 3-7 shows a screenshot of the package importing process. To import the Unity Integration package into the new project, follow these steps:

1. Find the **Assets** pane of the **Project** tab in the Unity IDE. **Choose Assets -> Import Package ->Custom Package...**
2. You should see a file dialog box. Use it to navigate to the location of the downloaded Unity Integration package.
3. Select the file *OculusUnityIntegration.unitypackage*.
4. Once you have clicked **Open**, Unity will scan the file and present you with a list of package contents to import. For now, let's just bring them all into the project: make sure that all of the objects in the list are checked, and click the **Import** button. You will now see assets present in the **Assets** pane, where there were none before. In particular, you should see folders named *OVR* and *Plugins*.

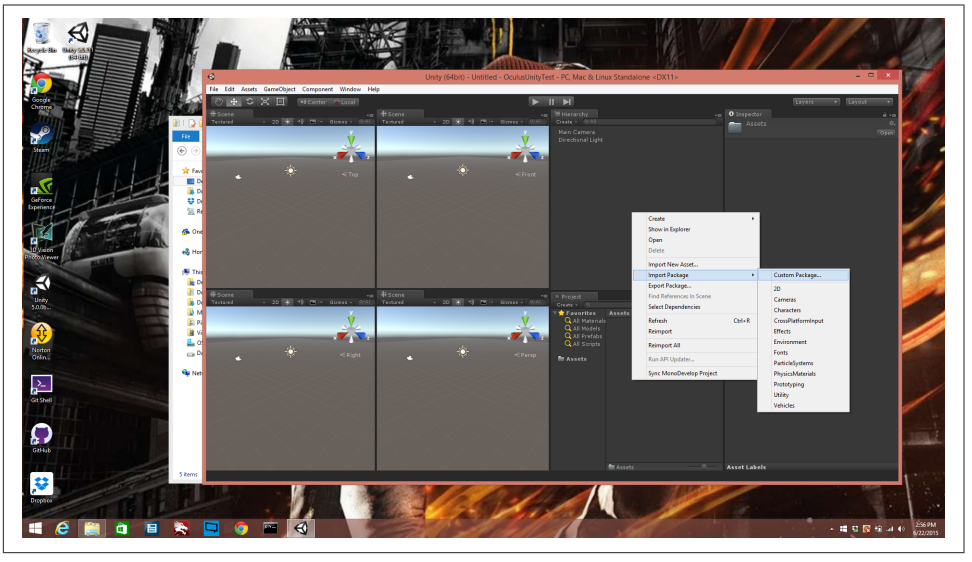


Figure 3-7. Importing the Oculus Integration package into a Unity Project

You now have a Unity project ready to go for building an Oculus application. Let's get into it.

Your First VR Application

Once you have imported the Unity Integration package into your project, you are ready to build your first desktop VR sample. The package comes with example content, so after importing, it is now in your project. In just a few steps you can have it running on your computer. The demo we are going to build is pictured in Figure 3-8, a screen shot from my PC laptop. It is a simple scene with hundreds of cubes floating in space. When you attach the Oculus Rift to your computer, you will be immersed in a view of this scene. Turn your head left, right, up and down; there are cubes everywhere you look!

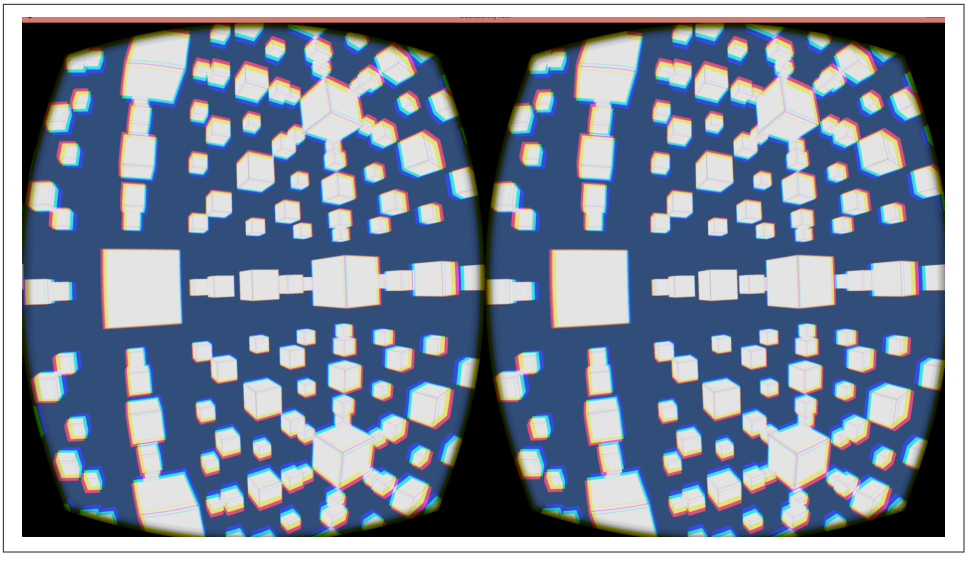


Figure 3-8. Screen shot of Unity Cubes Example Running in Oculus Rift

Let's build and run this example.

To get the assets for the cube world built into your new project, select the folder ***Assets->OVR->Scenes*** in the **Project** pane of the Unity interface. In the detail pane you will see an icon for a Unity scene named *Cubes*; double-click that icon. You should now see the scene in the main editor views. Figure 3-9 shows the Unity editor with a four-view layout of the *Cubes* scene (left, right, top and perspective).

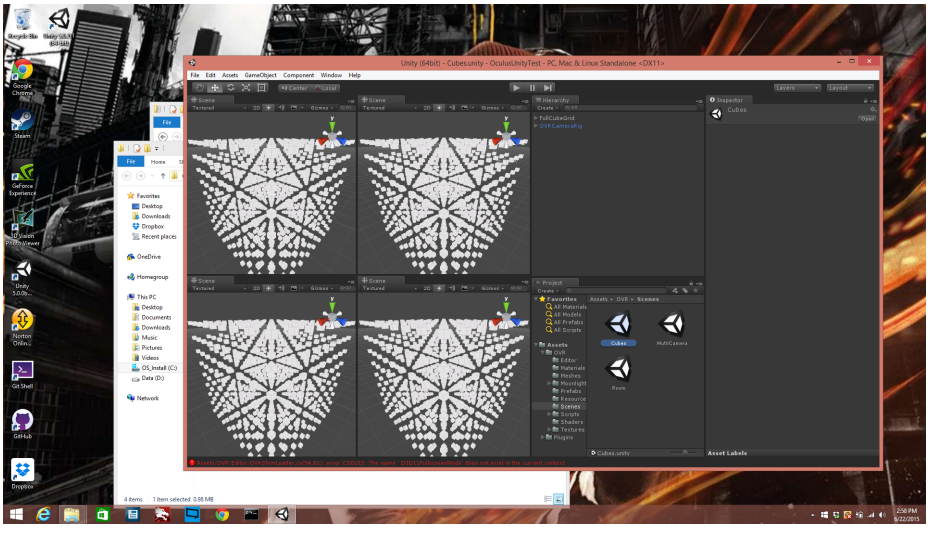


Figure 3-9. Cubes Scene Viewed in the Unity3D Editor

You can hit the Play button at the top of the Unity window to get a quick preview on your main monitor. Make sure you have **Maximize on Play** selected in the **Game** window (bottom left of the four panes in the four-view layout). Then, hit the play icon in the control bar on the middle top of the Unity interface. You should see something like the screen shot in Figure 3-10.

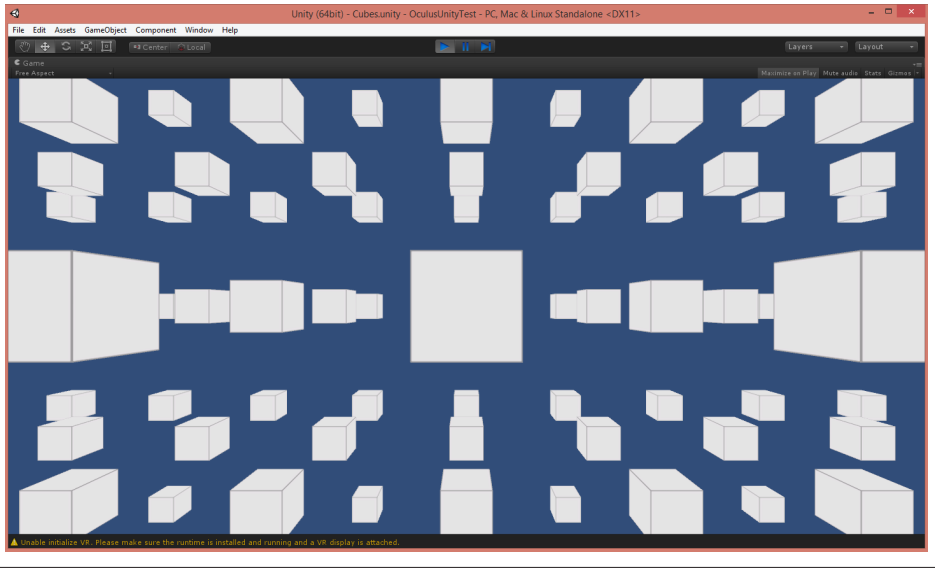


Figure 3-10. Unity Player Preview of Cubes World

If the Oculus Rift is not attached, you will see a warning to that effect in the status bar at the bottom of the window. If the Rift is attached, you may see a warning in the status bar that says the following:

```
VR direct mode rendering is not supported in the editor.
Please use extended mode or build a stand-alone player.
```

That's OK. We are going to build a stand-alone player anyway. Let's do that now.

Building and Running the Application

Unity supports creating games for a variety of target environments, including native desktop, web (using their proprietary player plugin), WebGL (experimental), mobile platforms like iOS and Android, and game consoles such as Xbox and PlayStation.

To build and run the Cubes World application for the desktop, first select File->Build Settings... you will see a dialog like the one depicted in Figure 3-11.

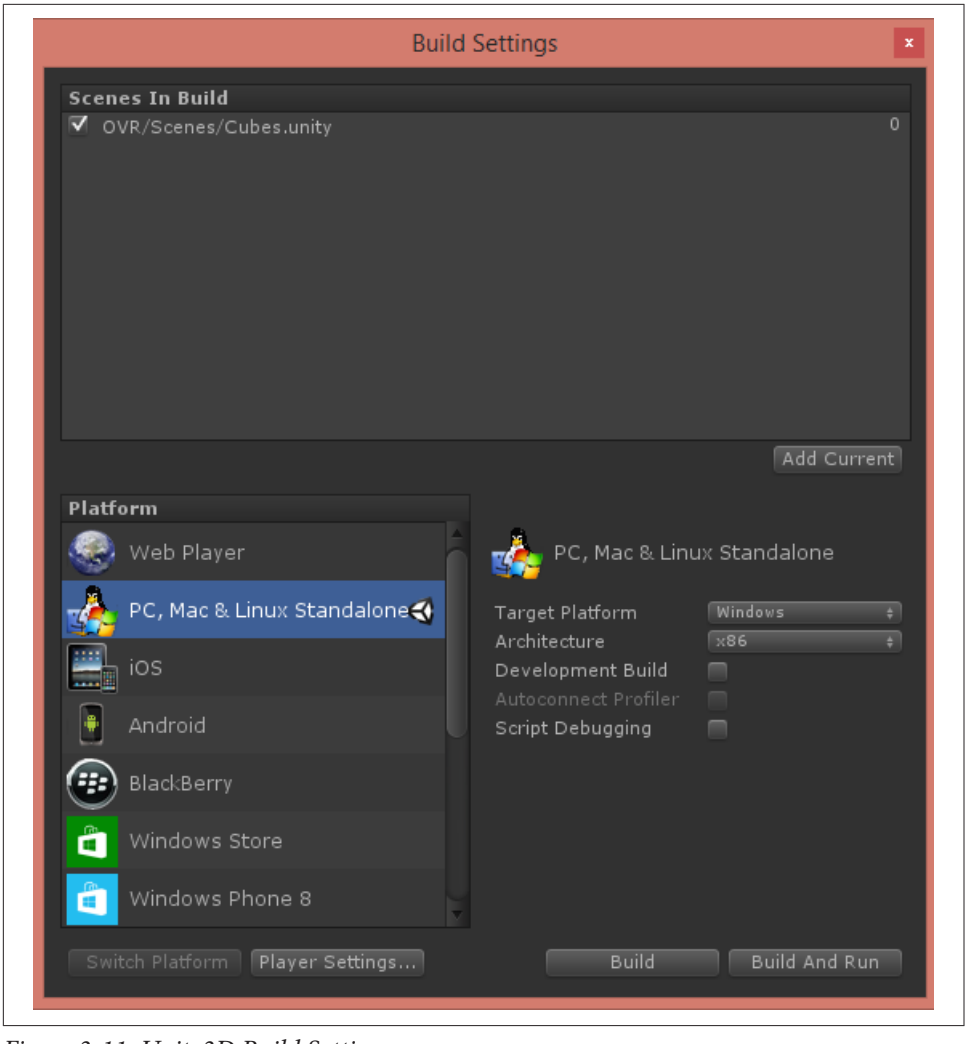


Figure 3-11. Unity3D Build Settings

Now perform the following steps - and make sure remember these steps for setting up subsequent Unity Oculus Rift projects:

1. Select PC, Mac & Linux Standalone as your platform. Click on that item in the list and then click the **Switch Platform** button.
2. Make sure **Target Platform** is set to **Windows** in the combo box on the right.
3. Add the demo scene to your build. The **Scenes in Build** list at the top will be empty to start; you need to add the current scene. You should already be viewing it in the Editor view, so if you click the **Add Current** button, it should be added to the list, and you will see a checked item named *OVR/Scenes/Cubes.unity*.

4. Select **Build and Run**. You will be prompted for a file name under which to save the executable. (I named mine *OculusUnityTest.exe*, but this is up to you.)

Assuming you didn't get any build errors, the application should launch. If you haven't yet connected the Rift, you will see the cube world rendering full-screen in monoscopic mode, and it won't be responsive to your mouse or any other input. If this is the case, press Alt+F4 to exit the app, and let's connect the Rift and try again.

If your Rift is already connected, you should see the application launch full screen, with the standard Oculus VR health and safety startup screen as depicted in Figure 3-12. Press a key, pop the Rift on your head and go.

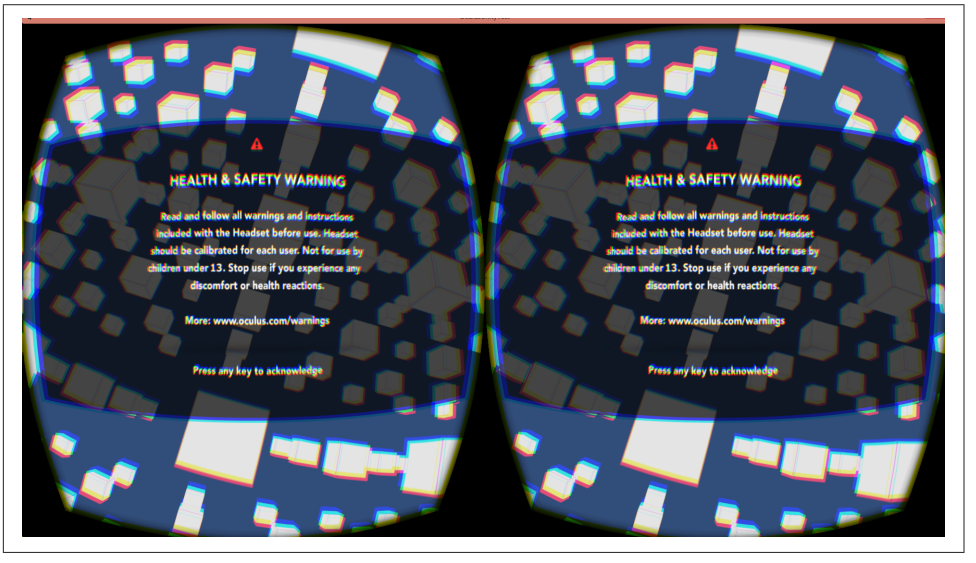


Figure 3-12. Cubes World Startup Screen, Successfully Launched on the PC

Congratulations. You've just built your first VR application! Now let's have a look at how it works.

Walking Through the Code

So, how does the Unity Oculus VR support work? First, let's see how the Unity scene is set up. The Unity Integration package comes with a camera rig prefab. (*Prefab* is a Unity term for a set of pre-built objects that can be used together as a unit.) This prefab provides the Oculus stereo rendering and head tracking required for basic VR support. Let's take a look at how it is constructed.

In the **Hierarchy** pane, locate the *OVRCameraRig* object, and expand it by clicking the down arrow icon to the left of the name. The camera rig contains a child object, *TrackingSpace*. Expand that, and you'll see four children: *LeftEyeAnchor*, *Center-*

EyeAnchor, *RightEyeAnchor* and *TrackerAnchor*. The left and right anchor are the keys; they contain the cameras used to render the view from the left and right eyes, respectively. You can see in the **Inspector** pane that by selecting either of the those objects that they contain camera components. The camera components contain bland default values, which will be updated in code each time through the application's update cycle. Let's look at that code now.

In the **Hierarchy** pane, select the object *OVRCameraRig* again. You will see that it contains a few script components. We want to look at the properties for the component labeled OVR Camera Rig (Script). Its **Script** property has the value *OVRCameraRig*. Double-click on that value and *MonoDevelop*, Unity's C# programmer's editor, will open the source file *OVRCameraRig.cs* from the project sources.

About the C# Language

I think we might need a note here generally describing C# for the uninitiated, and point to a few good resources for learning. Thoughts?

In the MonoDevelop source code window, scroll through or search until you find the definition of the method `LateUpdate()`, show in the listing below.

```
#if !UNITY_ANDROID || UNITY_EDITOR
    private void LateUpdate()
#else
    private void Update()
#endif
    {
        EnsureGameObjectIntegrity();

        if (!Application.isPlaying)
            return;

        UpdateCameras();
        UpdateAnchors();
    }
```

The #if statement evaluates to true-- we're not building for Android-- so this code defines the method named `LateUpdate()`. Unity3D script objects can define several methods to receive updates each time through the simulation loop, including `Update()` and `LateUpdate()`. `LateUpdate()` is the preferred method for updating cameras, because the engine ensures that it is called after all other update operations, for example collecting external input such as the Oculus SDK's head-tracking data.

Our late update method first calls a helper, `EnsureGameObjectIntegrity()`, to make sure that the scene contains all of the required objects, that is, the objects defined in the *OVRCameraRig* prefab. Why do this? Because the script component might have

been included in a project without the rest of the prefab. In this case it wasn't, so the method call isn't really necessary. But the script has been designed with that robustness in mind.

After a check to see whether the application is actually running-- it may not be, either because it hasn't be set into run mode yet, or because the object is being viewed in the Unity editor and not in play mode-- we do the real work. First, we update the camera parameters. `UpdateCameras()` retrieves the camera parameters for each eye:

```
        private void UpdateCameras()
        {
            if (needsCameraConfigure)
            {
                leftEyeCamera = ConfigureCamera(OVREye.Left);
                rightEyeCamera = ConfigureCamera(OVREye.Right);

    #if !UNITY_ANDROID || UNITY_EDITOR
                needsCameraConfigure = false;
    #endif
            }
        }
```

The method does a one-time initialization (for the desktop version), retrieving the camera parameters that were defined in the Oculus Configuration utility for the desktop, and setting a flag to say that the camera values have been configured.

Now to retrieve the parameters for each eye. This is done in the method `ConfigureCamera()`.

```
        private Camera ConfigureCamera(OVREye eye)
        {
            Transform anchor = (eye == OVREye.Left) ? leftEyeAnchor : rightEyeAnchor;
            Camera cam = anchor.GetComponent<Camera>();

            OVRDisplay.EyeRenderDesc eyeDesc = OVRManager.display.GetEyeRenderDesc(eye);

            cam.fieldOfView = eyeDesc.fov.y;
            cam.aspect = eyeDesc.resolution.x / eyeDesc.resolution.y;
            cam.rect = new Rect(0f, 0f, OVRManager.instance.virtualTextureScale, OVRManager.instance.v
            cam.targetTexture = OVRManager.display.GetEyeTexture(eye);
            cam.hdr = OVRManager.instance.hdr;

            ...

            return cam;
        }
```

Depending on which eye is passed in as the parameter, we retrieve either the left or right camera object, and its camera component from the camera rig prefab. Then, we adjust various values in the camera. Note the line

```
    OVRManager.display.GetEyeRenderDesc(eye)
```

The `OVRManager` script class is the main interface to the Oculus Mobile SDK. It is responsible for doing a lot, including interfacing with native code in the Oculus SDK. If you're curious about what's in that script, you can go back to the Unity editor and select the *OVRCameraRig* object from the hierarchy pane; you will see that it has a second script component, OVR Manager (Script). Double-click the **Script** property's value to open the C# source in MonoDevelop (file *OVRManager.cs*). Take a browse through that if you like. For now, we are going to treat it as a black box, copying various values into the left or right camera, including the field of view, aspect ratio, viewport, render target, and whether the camera should support high dynamic range (HDR).

Our basic camera parameters are set up, but we still need to position and orient the cameras based on where the Oculus Rift HMD is looking. This is done in method `UpdateAnchors()`.

```
private void UpdateAnchors()
{
    bool monoscopic = OVRManager.instance.monoscopic;

    OVRPose tracker = OVRManager.tracker.GetPose();
    OVRPose hmdLeftEye = OVRManager.display.GetEyePose(OVREye.Left);
    OVRPose hmdRightEye = OVRManager.display.GetEyePose(OVREye.Right);

    trackerAnchor.localRotation = tracker.orientation;
    centerEyeAnchor.localRotation = hmdLeftEye.orientation; // using left eye for now
    leftEyeAnchor.localRotation = monoscopic ? centerEyeAnchor.localRotation : hmdLeftEye.orie
    rightEyeAnchor.localRotation = monoscopic ? centerEyeAnchor.localRotation : hmdRightEye.or

    trackerAnchor.localPosition = tracker.position;
    centerEyeAnchor.localPosition = 0.5f * (hmdLeftEye.position + hmdRightEye.position);
    leftEyeAnchor.localPosition = monoscopic ? centerEyeAnchor.localPosition : hmdLeftEye.posi
    rightEyeAnchor.localPosition = monoscopic ? centerEyeAnchor.localPosition : hmdRightEye.po

    if (UpdatedAnchors != null)
    {
        UpdatedAnchors(this);
    }
}
```

`UpdateAnchors()` uses the `OVRManager`'s helper objects, `OVRTracker` and `OVRDisplay`, to obtain the positional tracker's current position and the HMDs current orientation, which it then sets into the `localPosition` and `localRotation` properties of the respective objects. The left and right eye anchor objects are used for rendering; the center object is for bookkeeping in the application, so that it knows where the mono camera is without having to recalculate it from the stereo cameras; and the tracker is used to save the position tracker value.

And that's it. By adding a single prefab to a basic Unity project, your app can have Oculus Rift stereo rendering and head tracking. The prefab is a little complex, and if

you dig deep through the Unity sources you will discover a lot of arcana. But it works, and this simple example illustrates how to create desktop VR using Unity3D.

Chapter Summary

In this chapter we created our first virtual reality application for desktop PCs using the Oculus Rift. We explored the tools and techniques for developing VR, including constructing a 3D scene, rendering in stereo, and head tracking to convey presence. We also got a brief primer on 3D graphics so that we can have an understanding of general 3D programming concepts, as well as those specific to virtual reality.

We explored using Unity3D, the powerful, affordable game engine that is becoming a de facto choice for developing VR applications. Unity saves us time and hassle over writing our own native 3D code, and provides several productivity and work flow features. The Oculus desktop SDK comes with a ready-to-go Unity package that includes code for handling common VR tasks such as stereo rendering and head tracking, and pre-built VR scenes to get us going quickly.

Armed with a basic understanding of 3D, the core concepts of VR development, and the power of Unity, we were able to get started programming virtual reality. Much of the rest of the book is variations on these themes. From here, we will explore other platforms and tools, before diving headlong into building a full, real VR application.

CHAPTER 4

Going Mobile: Developing for Gear VR

In this chapter, we will learn to develop virtual reality applications for the flagship mobile device on the market today, Samsung's *Gear VR*. The ideas we covered in the previous chapter on desktop VR translate directly into programming for mobile, but the Android platform used to develop for Gear VR also has unique concerns. Once again, we will use the Unity3D engine to build our examples. But before we get into the details of coding, let's explore this revolutionary new device.

In partnership with Oculus, Samsung has created a mobile VR solution that combines Oculus optics (the barrel distortion lenses) with new head tracking technology, placed in a custom headset that houses a smartphone with a high-resolution display. Samsung released the Gear VR as "Innovator Edition" in late 2014 for the Note 4 phone, and again in early 2015 for the S6 line. The Gear VR unit costs US$199 and can be purchased through Samsung's online store. Gear VR only works with those phones, so this is not only potentially pricey option (the phones can cost from US $600-$800 without a two-year plan), but it is also tied to those particular phone models-- at least for now. But Gear VR is the highest-quality mobile VR on the market and provides a wonderful experience.

The Gear VR Innovator edition for S6 phones is depicted in Figure 4-1.

Figure 4-1. Samsung GearVR Innovator Edition for S6 and S6 Edge Phones

Gear VR features high resolution graphics with a display resolution of 2560 x 1440 pixels (1280 x 1440 per eye) in both Note 4 and S6 phones. The headset contains a custom IMU based on Oculus technology that tracks much faster and with less latency than the IMU built into the phone.

The Gear VR housing has several intuitive input controls, including an eye adjustment wheel (for matching the devices inter-pupillary distance to your own), and several controls that let you access the phone within: a volume control, a headphone jack, and most importantly, a trackpad for touch input, known as the *touchpad*.

The Gear VR User Interface and Oculus Home

Beyond the big breakthroughs in ergonomics and resolution, the Gear VR features an innovative user interface for discovering, launching and installing mobile VR applications called *Oculus Home*. Oculus Home is an app, but it is also a set of improvements to Samsung's mobile operating system.

Oculus Home provides a completely immersive interface for browsing VR applications, so that you don't have to remove the headset to launch a new app. While inside the store interface, you can also see system alerts such as for new emails, text messages, notifications and so on. The net result is that you can have an uninterrupted VR experience, going between applications, and stay in there as long as you like-- all the while not being cut off from the other important goings-on of your phone. This is a major improvement over the desktop Oculus Rift experience, where you are typically taking the headset off and putting it back on between applications. A screenshot of Oculus Home is shown in Figure 4-2, with the Store interface selected, allowing you to browse for new apps to install.

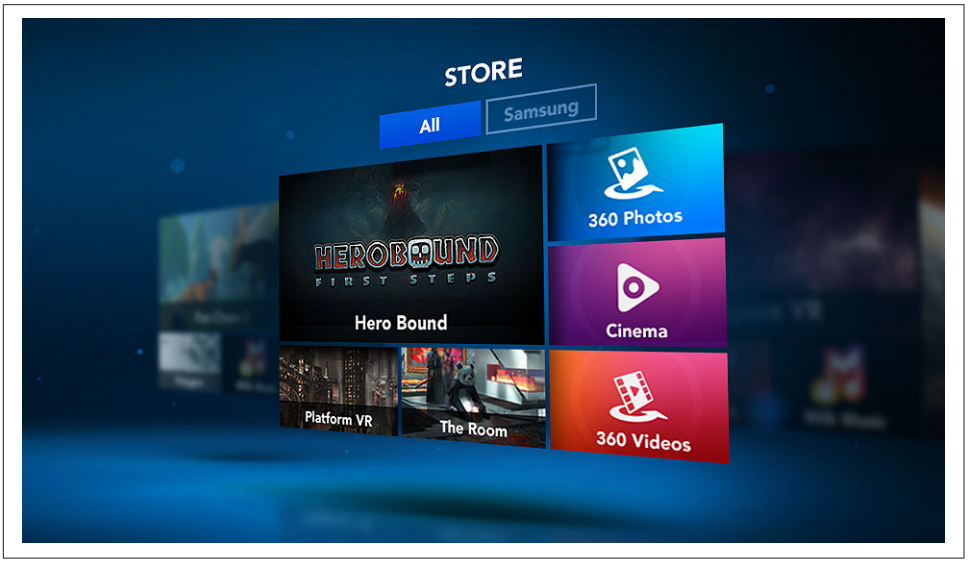


Figure 4-2. Oculus Home for Gear VR

Using the Oculus Mobile SDK

Developing applications for Gear VR's Android-based operating system requires the Oculus Mobile SDK, which is available on the Oculus developer website:

http://developer.oculus.com

If you don't have an account, sign up; it's free. Once you are logged in to the site, do the following:

1. Select **Downloads** from the top navigation bar.
2. Select **SDK: MOBILE** from the **SDK** pull-down.
3. Click the link to the latest mobile SDK.
4. Accept the end-user license agreement (EULA).
5. Download and unzip the SDK to your hard drive, in a location of your choice. I put mine in the */Applications* folder on my Macbook.

Setting up the Android SDK

To work on Gear VR, the Oculus Mobile SDK must be used in conjunction with the Android SDK. To set up the Android SDK, go to

https://developer.android.com/sdk/installing/index.htm

You can install the Stand-alone SDK Tools, or you can get Android Studio. This is your choice. We will be using Android Studio for our Java work in Chapter 6. But you can always set that up later and just go with the Stand-alone SDK Tools for now.
You will also want to learn about the Android debugging tools, because you may need to use them on the command line. There is a good writeup of the Android debugging tools here:

http://developer.android.com/tools/debugging/index.html

Generating an Oculus Signature File

You will need a signature file to run applications on your device. Applications that access the VR functionality of the Oculus mobile SDK must be built with a unique signature in order to access low-level device functions through APIs provided by the SDK. This is not how you will deploy applications in practice-- only properly signed Android package files (APKs) are needed as part of the Oculus Store deployment process-- but it is required for deploying on your development devices during testing.

The Oculus developer site has an online signature file generating tool at

https://developer.oculus.com/osig/

It contains detailed instructions for how to obtain your device ID and use that to generate a signature file. Essentially, you have to run the following command in a terminal (shell) window:

```
adb devices
```

This command will print a unique ID for your device. Copy that ID, enter it into the field at the top of the signature tool page, and you will get a file to download. Once you have downloaded that file, squirrel it away in a safe place on your computer; you will need it for all Gear VR development regardless of which development tool you decide to use.

Setting Up Your Device for USB Debugging

You will also need to set up USB debugging on your device in order to build and deploy applications to it. There's a trick to it, essentially an Easter egg in the Note 4 or S6 phones' settings. There is an article describing it here:

http://forums.androidcentral.com/samsung-galaxy-tab-pro-8-4/371491-enabling-developer-mode.html

Once you have done all the above steps, you're ready to use the Oculus Mobile SDK to develop Gear VR applications. Please be patient; Android setup can be tricky, and the Oculus Mobile SDK adds a few extra wrinkles.

Developing for Gear VR Using Unity3D

In the last chapter we learned about creating desktop Oculus Rift applications using Unity3D, the powerful, affordable game engine from Unity Technologies. We can also use Unity3D to create applications for Gear VR, using the Oculus Mobile SDK.

Do I Have to Use Unity?

You may be wondering if Unity3D is the only way to build for Gear VR. The short answer is: no. Gear VR development is, ultimately, Android development. Unity has several features that makes Android development easier, especially for creating games and virtual reality. But if you are not a Unity developer, or if you prefer writing native Android code over using middleware, you have other options. One such option is *GearVRf*, a native Java framework created by Samsung and recently released as an open source library. Appendix X contains more information on GearVRf.

Setting Up Your Unity3D Environment

Now that you have downloaded the Oculus Mobile SDK, set up your Android environment, and created a signature file, you are ready to start using the Unity Integration package that comes with the Oculus Mobile SDK. You do that by importing the package into a Unity project. Let's start by creating an empty Unity project. Launch Unity3D, or if it is already running, select **File->New Project...**

Name your new project *UnityGearVRTest*. Now that the project has been created, we will import the SDK into it and build a simple application.

Figure 4-2 shows a screenshot of the package importing process. To import the Unity Integration package into the new project, follow these steps:

1. Find the **Assets** pane of the **Project** tab in the Unity IDE. **Choose Assets -> Import Package ->Custom Package...**
2. You should see a file dialog box. Use it to navigate to the location of the downloaded Unity Integration package (for me on my Macbook it's *Applications/ovr_mobile_sdk/VrUnity/UnityIntegration/*).
3. Select the file *UnityIntegration.unitypackage*.
4. Once you have clicked **Open**, Unity will scan the file and present you with a list of package contents to import. For now, let's just bring them all into the project: make sure that all of the objects in the list are checked, and click the **Import** button. You will now see assets present in the **Assets** pane, where there were none before. In particular, you should see folders named *OVR* and *Plugins*.

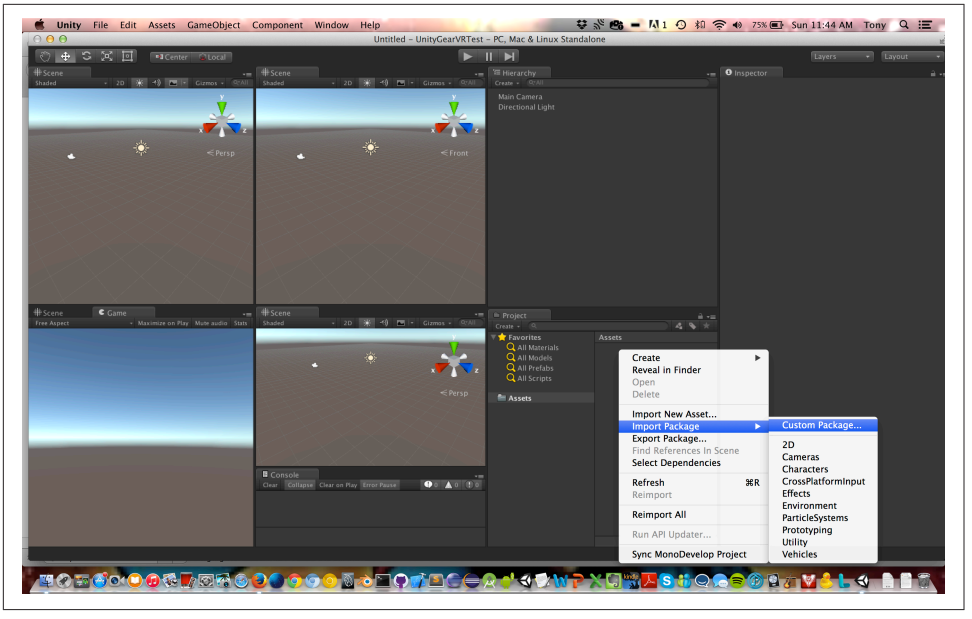


Figure 4-3. Importing the Unity Integration for Gear VR Package Into a Unity Project

You're good to go! You can now build Gear VR applications using Unity.

A Simple Unity3D Sample

Once you have imported the Unity Integration package into your project, you are ready to build your first Gear VR sample. The package comes with ready example content, so it's already in your project. In just a few steps you can have it running on your phone. The demo we are going to build is pictured in Figure 4-3, a screen shot from my Note 4 phone before inserting it into the Gear VR headset.

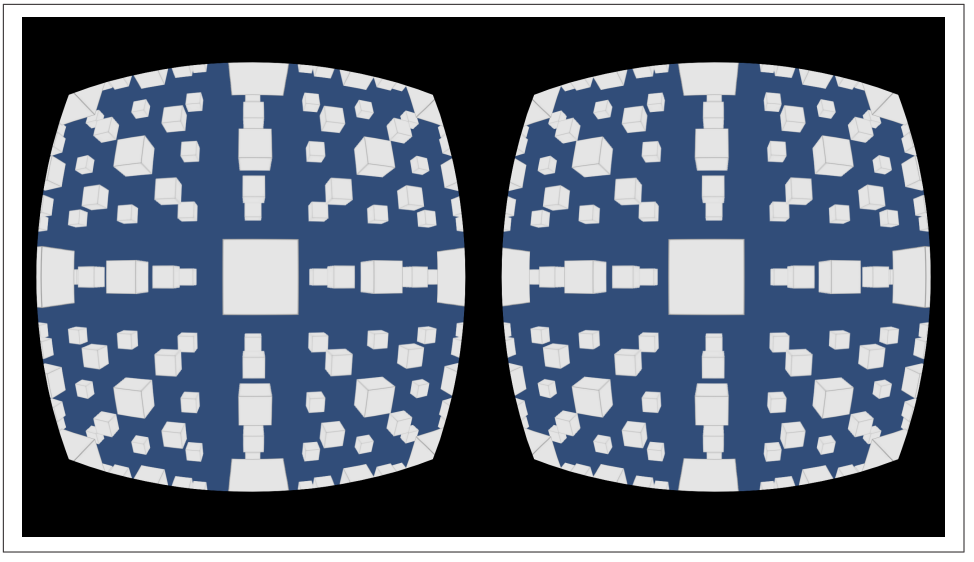


Figure 4-4. "Cube World" Simple Gear VR Application Built with Unity3D

Let's build and run the example on our phone.

To get the assets for the cube world built into your new project, select the folder ***Assets->OVR->Scenes*** in the **Project** pane of the Unity interface. In the detail pane you will see an icon for a Unity scene named *Cubes*; double-click that icon. You should now see the scene in the main editor views. You can hit the Play button at the top of the Unity window to get a preview on your computer.

Now, to build the app for your phone, you will need to adjust some settings. First, open the Build Settings dialog by selecting **File->Build Settings...** from the main menu. You will see a dialog that resembles the screen shot in Figure 4-4.

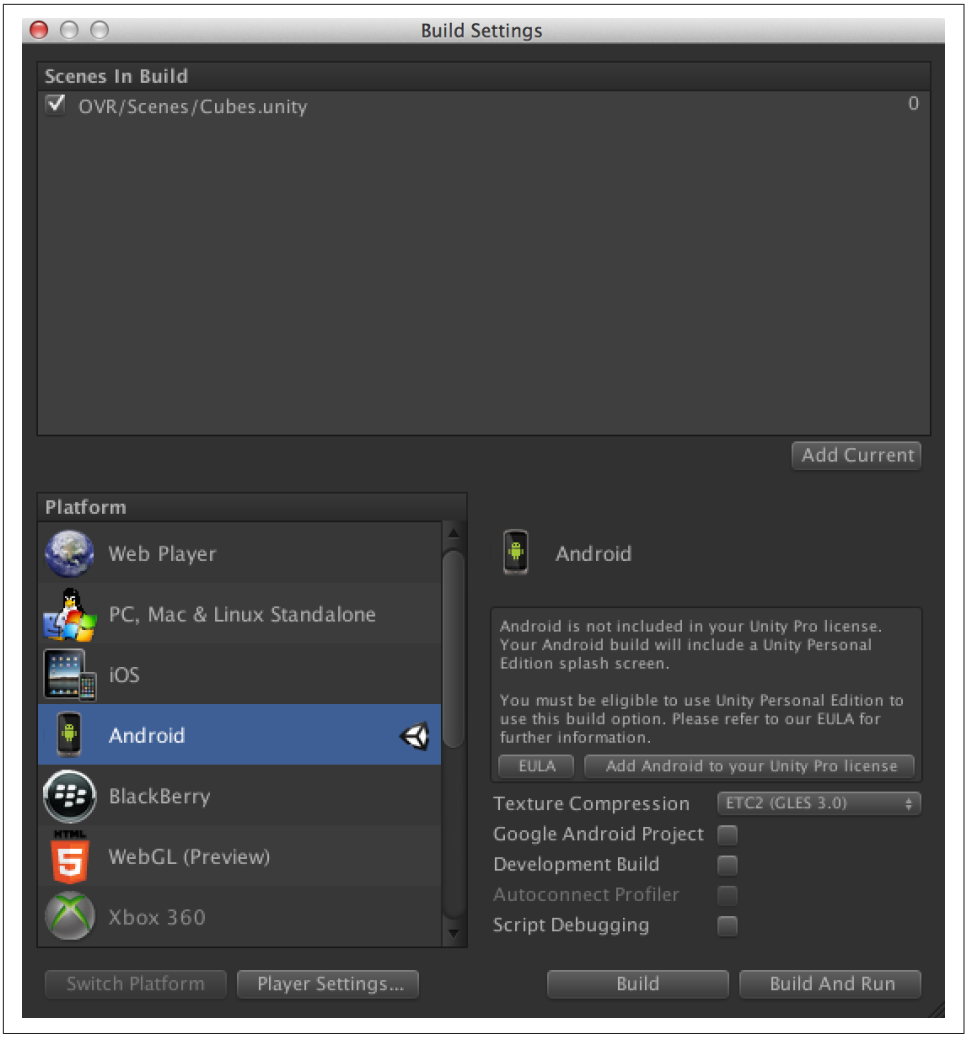


Figure 4-5. Build Settings for the Cube World Android App

Now perform the following steps - and make sure remember these steps for setting up subsequent Unity Gear VR projects:

1. Select Android as your platform. Click on the **Android** item in the list and then click the **Switch Platform** button.
2. Add the demo scene to your build. The Scenes in Build list at the top will be empty to start; you need to add the current scene. You should already be viewing it in the Editor view, so if you click the **Add Current** button, it should be added to the list, and you will see a checked item named *OVR/Scenes/Cubes.unity*.
3. Make sure the **Texture Compression** setting says ETC2 (GLES 3.0).

4. Now click the **Player Settings...** button on the bottom left. This will open the Inspector to the Player Settings panel. (You should see it on the far right of the Unity interface.) Below a few general settings you will see tabs with icons for the different platforms. Click the Android icon to see the Android settings. Select the **Other Settings** sub-pane and make sure that **Rendering Path** is set to Forward Rendering, that the **Multithreaded Rendering** check box is checked, and that **Graphics Level** is set to Force Open GL ES 2.0.

Almost done. Now we have to tweak a few other project settings using **Edit->Project Settings** from the main menu. Each of these will bring up a pane in the Inspector (by default, on the far right of the Unity interface).

1. Select **Edit->Project Settings->Time** and make sure that both **Fixed Timestep** and **Maximum Allowed Timestep** are set to 0.0166666 (milliseconds, i.e. 60 frames per second update).
2. Select E**dit->Project Settings->Quality**. Make sure **Pixel Light Count** is 0, and **V Sync Count** says Don't Sync.

There is one final thing we need to do so that we can run the project on our phone. To run Gear VR applications built with the Oculus SDK, you will need a signature file. If you haven't already, follow the instructions earlier in the chapter and generate a signature file for your device. Once you have that, squirrel it away in a safe place; you will need it for all Gear VR development.
Each Unity3D project needs a copy of the signature file put in a known place in the build, specifically *Assets->Plugins->Android->assets* (note lowercase 'a' in that last folder name). You can put the file in your project either by drag-and-drop from your desktop folder into the folder within the **Project** pane within the Unity interface, or you can copy the files using your operating system interface and locating the folder where your Unity project is stored. (On my Macbook, that is in the path *Tony/Unity Projects/UnityGearVRTest/Assets->Plugins->Android->assets*.)

OK! If you got through all that, you should be ready to build and run the app for your phone. Make sure to connect your computer to your device with a USB cable, and that the device is set up for USB debugging (described earlier in the chapter). Now, hit Build and Run in the **Build Settings** dialog. You will be prompted for a file name under which to save the .apk (Android package) file. I chose the name *UnityGearVRTest.apk*. That's the name that will show up on the phone in your list of apps. Hit **Save** to save the .apk. The app should now launch, showing the split-screen Oculus distortion view from Figure 4-3. Disconnect the cable, pop the phone into your Gear VR headset, and go. You should be inside a world of cubes, and when you move your head, you will be looking around the scene. Welcome to mobile virtual reality!

A Note About Unity and Oculus Mobile SDK Versions

Mileage may vary, and depending on which version of the various tools you're using you may have to change more, or fewer, options than described above. I used Unity 5 (build 5.0.1f1) on my Macbook on OS X, and version 0.5.1 of the Oculus Mobile SDK.

Now it's time to explore the inner workings of the Unity Integration package for Gear VR. This is more or less the same code as in the desktop example from Chapter 3, but there are also a few subtle differences, so you may want to follow along anyway.

First, let's see how the Unity scene is set up. The Unity Integration package comes with a camera rig prefab. This prefab provides the Oculus stereo rendering and head tracking required for Gear VR support. Let's take a look at how it is constructed.

In the **Hierarchy** pane, locate the *OVRCameraRig* object, and expand it by clicking the down arrow icon to the left of the name. The camera rig contains a child object, *TrackingSpace*. Expand that, and you'll see four children: *LeftEyeAnchor*, *CenterEyeAnchor*, *RightEyeAnchor* and *TrackerAnchor*. The left and right anchor are the keys; they contain the cameras used to render the view from the left and right eyes, respectively. You can see in the **Inspector** pane that by selecting either of the those objects that they contain camera components. The camera components contain bland default values, which will be updated in code each time through the application's update cycle. Let's look at that code now.

In the **Hierarchy** pane, select the object *OVRCameraRig* again. You will see that it contains a few script components. We want to look at the properties for the component labeled OVR Camera Rig (Script). Its **Script** property has the value *OVRCameraRig*. Double-click on that value and *MonoDevelop*, Unity's C# programmer's editor, will open the source file *OVRCameraRig.cs* from the project sources. Scroll through or search in the editor window until you find the definition of the method `Update()`, show in the listing below.

```
#if !UNITY_ANDROID || UNITY_EDITOR
    private void LateUpdate()
#else
    private void Update()
#endif
    {
        EnsureGameObjectIntegrity();

        if (!Application.isPlaying)
            return;

        UpdateCameras();
        UpdateAnchors();
    }
```

The #if statement evaluates to false-- we're building for Android-- so this code defines the method named `Update()`. Every Unity3D script object defines this method to receive updates every time through the simulation loop. Our update method first calls a helper, `EnsureGameObjectIntegrity()`, to make sure that the scene contains all of the required objects, that is, the objects defined in the *OVRCameraRig* prefab. Why do this? Because the script component might have been included in a project without the rest of the prefab. In this case it wasn't, so the method call isn't really necessary. But the script has been designed with that robustness in mind.

After a check to see whether the application is actually running-- it may not be, either because it hasn't be set into run mode yet, or because the object is being viewed in the Unity editor and not in play mode-- we do the real work. First, we update the camera parameters. `UpdateCameras()` retrieves the camera parameters for each eye:

```
        private void UpdateCameras()
        {
            if (needsCameraConfigure)
            {
                leftEyeCamera = ConfigureCamera(OVREye.Left);
                rightEyeCamera = ConfigureCamera(OVREye.Right);

    #if !UNITY_ANDROID || UNITY_EDITOR
                needsCameraConfigure = false;
    #endif
            }
        }
```

The method does this each time through the loop for Android, because parameters might have changed between times through the run loop.

Now to retrieve the parameters for each eye. This is done in the method `ConfigureCamera()`.

```
        private Camera ConfigureCamera(OVREye eye)
        {
            Transform anchor = (eye == OVREye.Left) ? leftEyeAnchor : rightEyeAnchor;
            Camera cam = anchor.GetComponent<Camera>();

            OVRDisplay.EyeRenderDesc eyeDesc = OVRManager.display.GetEyeRenderDesc(eye);

            cam.fieldOfView = eyeDesc.fov.y;
            cam.aspect = eyeDesc.resolution.x / eyeDesc.resolution.y;
            cam.rect = new Rect(0f, 0f, OVRManager.instance.virtualTextureScale, OVRManager.instance.v
            cam.targetTexture = OVRManager.display.GetEyeTexture(eye);
            cam.hdr = OVRManager.instance.hdr;

            ...

            return cam;
        }
```

Depending on which eye is passed in as the parameter, we retrieve either the left or right camera object, and its camera component from the camera rig prefab. Then, we adjust various values in the camera. Note the line

```
OVRManager.display.GetEyeRenderDesc(eye)
```

The `OVRManager` script class is the main interface to the Oculus Mobile SDK. It is responsible for doing a lot, including interfacing with native Android code in the SDK. If you're curious about what's in that script, you can go back to the Unity editor and select the *OVRCameraRig* object from the hierarchy pane; you will see that it has a second script component, OVR Manager (Script). Double-click the **Script** property's value to open the C# source in MonoDevelop (file *OVRManager.cs*). Take a browse through that if you like. For now, we are going to treat it as a black box, copying various values into the left or right camera, including the field of view, aspect ratio, viewport, render target, and whether the camera should support high dynamic range (HDR).

Our basic camera parameters are set up, but we still need to position and orient the cameras based on where the Gear VR HMD is looking each frame. This is done in method `UpdateAnchors()`.

```
private void UpdateAnchors()
{
    bool monoscopic = OVRManager.instance.monoscopic;

    OVRPose tracker = OVRManager.tracker.GetPose();
    OVRPose hmdLeftEye = OVRManager.display.GetEyePose(OVREye.Left);
    OVRPose hmdRightEye = OVRManager.display.GetEyePose(OVREye.Right);

    trackerAnchor.localRotation = tracker.orientation;
    centerEyeAnchor.localRotation = hmdLeftEye.orientation; // using left eye for now
    leftEyeAnchor.localRotation = monoscopic ? centerEyeAnchor.localRotation : hmdLeftEye.orie
    rightEyeAnchor.localRotation = monoscopic ? centerEyeAnchor.localRotation : hmdRightEye.or

    trackerAnchor.localPosition = tracker.position;
    centerEyeAnchor.localPosition = 0.5f * (hmdLeftEye.position + hmdRightEye.position);
    leftEyeAnchor.localPosition = monoscopic ? centerEyeAnchor.localPosition : hmdLeftEye.posi
    rightEyeAnchor.localPosition = monoscopic ? centerEyeAnchor.localPosition : hmdRightEye.po

    if (UpdatedAnchors != null)
    {
        UpdatedAnchors(this);
    }
}
```

`UpdateAnchors()` uses the `OVRManager`'s helper objects, `OVRTracker` and `OVRDisplay`, to obtain the positional tracker's current position and the HMDs current orientation, which it then sets into the `localPosition` and `localRotation` properties of the respective objects. The left and right eye anchor objects are used for rendering; the

center object is for bookkeeping in the application, so that it knows where the mono camera is without having to recalculate it from the stereo cameras; and the tracker is used to save the position tracker value. (Remember that Gear VR only tracks orientation, not position; but this script is also used in the Oculus desktop SDK for Unity so it also tracks position.)

And those are the basics. We're now up and running on Gear VR.

Handling Touchpad Events

Virtual reality isn't only about rendering and camera tracking. We also need to process user input. The Gear VR comes with the touchpad, a great input device built into the side of the HMD. This is going to be our main source of input. Let's get a feel for how to use it.

Create a new Unity project, and call it *UnityGearVRInput*. As you did for the first example, import the Unity Integration package into the new project. In the **Project** pane, open the folder *Assets->OVR->Scenes* and double-click the scene named *Room*.

This loads a very simple scene, just a gray box that will be the inside of the room. This example uses the touchpad to rotate the camera within the room. You should be able to see it in the Game view, located on the bottom left of the Unity interface, as highlighted by the red circle in Figure 4-5.

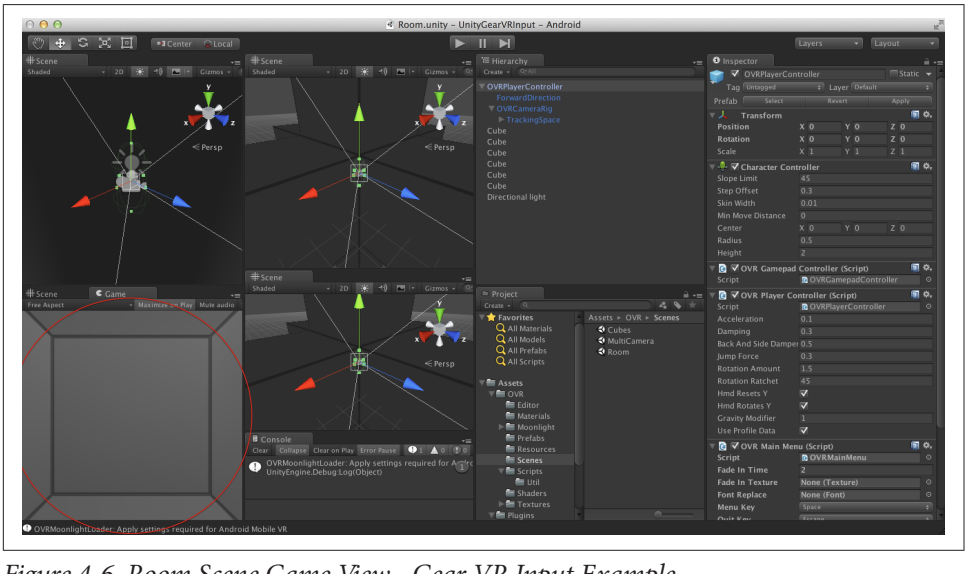


Figure 4-6. Room Scene Game View - Gear VR Input Example

In the **Hierarchy** pane you should see several objects: an object named *OVRPlayerController*, six cubes, and a Directional light. *OVRPlayerController* is another SDK prefab; this one contains a camera rig as in the previous example, but it uses touchpad input instead of head tracking to move the camera. If you expand the prefab by clicking on the down-arrow icon you will see object's contents, including the camera rig and several script components in the **Inspector** pane.

Now let's run the example. As before, you need to set up your build settings. Select **Android** and then press the **Switch Platform** button. Click **Add Current** to add the room sample to the build. Set up your other settings as you did above - remember **Player Settings...** (Rendering Path, Multithreaded Rendering, Graphics Level) and **Edit->Project Settings** (time steps and quality settings). Finally, remember to copy the signature file for your device. Once this is all done, you should be able to build and run the sample.

Assuming you were successful, you will be inside a gray box. Swipe the touchpad to rotate your view within the box. Let's look at the code. Select the object OVRPlayerController in the **Hierarchy** pane. You'll see its properties in the Inspector, including several scripts. Find the script named OVR Player Controller (Script), and double-click the value for its **Script** property. This will open the source file *OVRPlayerController.cs* in MonoDevelop.

The script's `Update()` method is quite involved. We are most concerned with a helper method, `UpdateMovement()`. This method, which also works on the desktop, uses the current mouse X position to calculate a rotation about the Y (up) axis for the camera. Scroll to or search for that method, and within it, the lines of code that check for mouse input:

```
if (!SkipMouseRotation)
    euler.y += Input.GetAxis("Mouse X") * rotateInfluence * 3.25f;
```

We can use this same approach to handle touchpad input, because the Unity Integration package automatically converts touchpad input to mouse input as follows:

- Forward/back swipes on the touchpad are converted to mouse movement in X.
- Up/down swipes are converted to mouse movement in Y.
- Taps on the touchpad are converted to mouse button presses, with mouse button code 0.
- Taps on the back button next to the touchpad are converted to mouse button presses, with mouse button code 1.

The mouse X and Y values are obtained by calling `Input.GetAxis()`, a function that is built into the Unity input system. The variable `euler` is an *euler angle*, that is, an x,y,z triple that represents rotation about each individual axis. The method continues for a while, potentially also getting input from a gamepad controller, but we are not using one in this example. The last line in the method converts the computed euler value to a quaternion

and saves it into the player controller's transform, thereby rotating the controller, which ultimately results in the rotation of its descendant camera rig.

```
transform.rotation = Quaternion.Euler(euler);
```

Implementing Gaze Tracking

Now that we know how to stereo render, track HMD motion and use the touchpad for input, we only need one more thing to be "doing VR," and that is gaze tracking. If we combine gaze tracking with touchpad input, we can highlight and select objects by looking at them and tapping.

The *SDKExamples* Unity package that comes with the Oculus Mobile SDK contains a simple example of gaze tracking in action: a small scene with three objects and a crosshair that changes size when you gaze over one of the objects. The example is depicted in Figure 4-6.

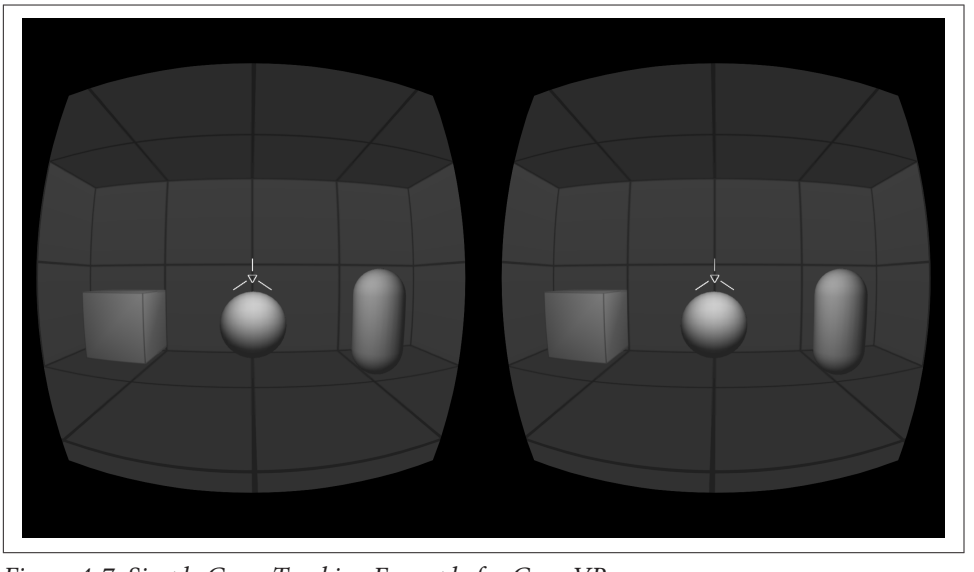


Figure 4-7. Simple Gaze-Tracking Example for Gear VR

Let's build and run this example and look through the code. To build it, create a new project *UnityGearVRGaze*, and import the *SDKExamples* package from the Oculus Mobile SDK directory on your hard drive (on my Macbook it's under *Applications/ovr_mobile_sdk/VrUnity/SDKExamples/*).

As in the previous examples, we need to set up the project's build settings. Select **Android** and then press the **Switch Platform** button. (NOTE: this is going to take a

long time for some reason! ... grab a coffee or check your email. It's a known bug with importing all the assets in the Unity Integration package, and hasn't been fixed as of this writing.)

Now, open the crosshair example scene: navigate to *Assets->Scene*s in the **Project** pane, and select *Crosshair_Sample.*

Go back to the **Build Settings** dialog, and click **Add Current** to get the crosshair sample into the build.

Set up your settings as you did above - remember **Player Settings...** (Rendering Path, Multithreaded Rendering, Graphics Level) and **Edit->Project Settings** (time steps and quality settings). Finally, remember to copy the signature file for your device. Once this is all done, you should be able to build and run the sample.

Now build and run the application. If that all went well, the app will launch on your phone. Put it in your headset and you will see that the cursor gets bigger when it is over one of the three objects.

The code to implement the cursor update is in source file *Crosshair3D.cs*. To open it, select the object named *Crosshair3D* in the **Hierarchy** pane. Then in the Inspector, find the component named Crosshair 3D (Script), and double-click on the value of its **Script** property to open the file in MonoDevelop. The following listing shows the code for `LateUpdate()`, a method called by Unity3D on all objects to perform last-ditch updates each time through the simulation.

```
    void LateUpdate()
    {
#if CROSSHAIR_TESTING
        ...
#endif
        Ray ray;
        RaycastHit hit;

        if (OVRManager.display.isPresent)
        {
            // get the camera forward vector and position
            Vector3 cameraPosition = cameraController.centerEyeAnchor.position;
            Vector3 cameraForward = cameraController.centerEyeAnchor.forward;

            GetComponent<Renderer>().enabled = true;

            //*************************
            // position the cursor based on the mode
            //*************************
            switch (mode)
            {
            case CrosshairMode.Dynamic:
                // cursor positions itself in 3D based on raycasts into the scene
                // trace to the spot that the player is looking at
```

```
                ray = new Ray(cameraPosition, cameraForward);
                if ( Physics.Raycast(ray, out hit))
                {
                    thisTransform.position = hit.point + (-cameraForward * offsetFromObjects);
                    thisTransform.forward = -cameraForward;
                }
                break;
            case CrosshairMode.DynamicObjects:
                // similar to Dynamic but cursor is only visible for objects in a specific layer
                ray = new Ray(cameraPosition, cameraForward);
                if (Physics.Raycast(ray, out hit))
                {
                    if (hit.transform.gameObject.layer != objectLayer)
                    {
                        GetComponent<Renderer>().enabled = false;
                    }
                    else
                    {
                        thisTransform.position = hit.point + (-cameraForward * offsetFromObjects);
                        thisTransform.forward = -cameraForward;
                    }
                }
                break;
            case CrosshairMode.FixedDepth:
                // cursor positions itself based on camera forward and draws at a fixed depth
                thisTransform.position = cameraPosition + (cameraForward * fixedDepth);
                thisTransform.forward = -cameraForward;
                break;
            }

            if (Input.GetButtonDown(OVRGamepadController.ButtonNames[(int)OVRGamepadController.But
            {
                ray = new Ray(cameraPosition, cameraForward);
                if (Physics.Raycast(ray, out hit))
                {
                    hit.transform.gameObject.BroadcastMessage("OnClick", SendMessageOptions.DontRe
                }
            }
        }
        else
        {
            GetComponent<Renderer>().enabled = false;
        }
    }
```

The method makes sure that we have a running HMD display and, if so, performs a ray cast. In this example, the crosshair script is in Dynamic mode, so the code positions the cursor a bit in front of the object, making it look larger than it it was positioned away from the camera at the normal fixed distance.

We can combine gaze with the touchpad to select objects or trigger other behaviors. After positioning the cursor, the code checks to see if a button was pressed using the

gamepad controller; if so, then it broadcasts an `OnClick` message to the application. Under the covers, the Unity Integration package translates touchpad input to gamepad input. In particular a tap on the touchpad acts like pressing the **A** button on the gamepad. So, taken together, touchpad taps with gaze over can act like a mouse click on an object in the scene.

Deploying Applications for Gear VR

OK so you've developed and tested your application and it's working. Now you want to publish it to the Oculus Store for the world to experience. How do you do that?

A full treatment of the topic is out of scope for this book, but here are a few basics. First, you need to sign your application. You must replace the temporary Oculus signature file used in development with a valid Android application signature file. The Android SDK contains information on how to do this; go to http://developer.android.com/tools/publishing/app-signing.html. Unity has integrated support for signing your application, which can be found in **Edit->Project Settings->Player->Publishing Options**. Use that interface to insert a newly created Android keystore generated by the Android SDK.
Once you have generated an Android Package file (APK) with a valid signature, you can submit it to the Oculus Store. This is a bit of a process, involving packaging image files in known directories, writing application manifests, exchanging files via Dropbox, email back-and-forth with the Oculus Store online service and, presumably, humans on the other end making the final approvals.

Full instructions for publishing to the Oculus store are online at

http://static.oculus.com/sdk-downloads/documents/GearVR_SubmissionGuidelines.pdf

Chapter Summary

In this chapter, we learned how to develop applications for Gear VR, Samsung's revolutionary mobile VR headset powered by Oculus technology. We explored the current editions of the hardware and supported phones, and took a quick look at the wonderfully designed user interface known as the Oculus Store.

Unity3D comes with excellent support for developing Gear VR applications. We created sample projects to stereo render, track HMD motion, use the touchpad for input, and track gaze to provide cursor feedback and select objects.

This chapter also took a quick look at the full cycle of developing, testing, debugging and deploying for Gear VR, including using the Android SDK, the Oculus Mobile SDK and its Unity Integration package, the Oculus code signing tool, and submitting applications to the Oculus Store.

While the material covered in this chapter is simple in concept, the devil is in the details. Developing for Android is no picnic. Unity3D is powerful but takes some getting used to. And the additional quirks in dealing with Gear VR and Oculus Store deployment make everything that much more... *interesting*. But hopefully you will find that it's worth the trouble. Gear VR is a great device, and a major step on the road to consumer-grade, mass-market virtual reality.

CHAPTER 5

WebVR: Browser-Based Virtual Reality in HTML5

In previous chapters we saw how to use native platform SDKs to create virtual reality applications with great graphics, high performance, and a sense of presence. If we are developing a single-user experience for a specific desktop or mobile platform, and don't mind that it requires an app download and install, then this approach may be all that we ever need.

But many of us creating VR would prefer to build web applications. Why? because integrating VR with the web offers the following advantages over native applications:

- **Instant access.** No download and app install required; just type a URL or click on a hyperlink to launch virtual reality experiences.
- **Easy integration of web data** like Wikipedia, YouTube and social networks, as well as open APIs to thousands of web services.
- **Cross-platform.** HTML5 runs on all desktop and mobile devices. On mobile platforms, HTML5 code can be delivered either via browsers or embedded in apps.
- **Faster, cheaper development.** HTML5 and JavaScript are arguably the easiest cross-platform system for creating apps ever devised, and we can use our choice of open source development tools, most of which are free of charge.
- **Easier deployment.** The cloud is set up to deliver web applications, and updates just happen without having to go through app stores.

There is such a strong belief in this idea that work is already taking place to make it happen. The latest generation of web browsers includes support for virtual reality, using new APIs and related technologies dubbed "WebVR."

While WebVR is in its early stages, still only running in developer builds of browsers, it is already showing promising results. Figure 5-1 shows a screen shot of a demo created by Brandon Jones of Google: a version of id Software's *Quake 3*, ported to WebGL and running in the browser. Note the Oculus distortion: this version renders to an Oculus Rift and tracks head movements to update the camera, using the WebVR API.

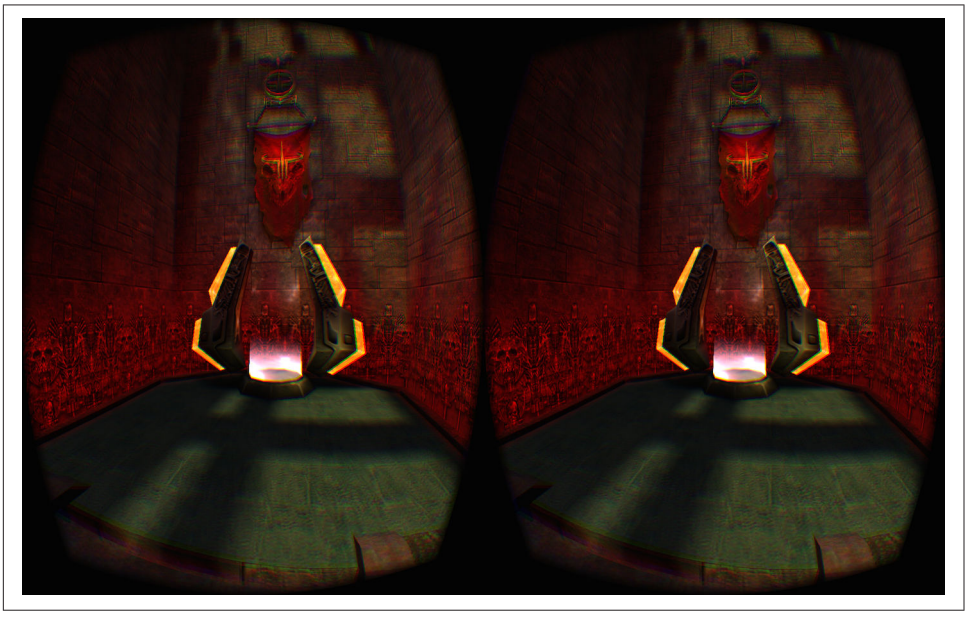


Figure 5-1. Quake 3 WebVR demo, developed by Brandon Jones of Google http://media.tojicode.com/q3bsp/

This chapter explores how to develop using WebVR. We'll study the details of the API currently available in nightly developer builds of Firefox and Chrome. Then we will take a peek at rendering 3D in the browser with WebGL using the popular Three.js library, and survey open source tools for putting together VR applications for the web. But first, let's look at how this exciting new development in web browser technology came to be.

The Story of WebVR

The roots of WebVR can be traced to the spring of 2014. In the wake of the historic acquisition of Oculus VR by Facebook, Mozilla engineer Vlad Vukićević and his colleague, VR Researcher Josh Carpenter, began work to add Oculus Rift support to Firefox. There had been previous attempts to integrate the Rift with browsers as far back as the DK1, using either web sockets or browser plugins to communicate with

the head tracking IMU hardware. But the performance of these solutions was relatively slow, resulting in motion lag and the associated queasiness.

Vukićević and Carpenter decided to see how far they could get with a native implementation built into the browser, free of the performance issues that came with plugins. By summer, Mozilla had released a first, experimental version of the code in a developer-only build of Firefox available for download on Vlad's personal blog site, along with a few pieces of example content.

In June 2014, just prior to the first public release, the Mozilla team presented their work at the San Francisco WebGL Developers Meetup, a group 800 strong that meets to discuss 3D rendering in browsers. Josh Carpenter came to present, and brought along Brandon Jones, an engineer on the Chrome team, to demo his own VR prototype. It turns out the two teams had already been collaborating to create a common API between the two browsers, and not two months after beginning their projects, they had achieved it: using this new API, programmers could write their VR code once and run it in both browsers. At that moment, WebVR was born.

Since that June 2014 meeting there has been a flurry of activity, including revisions to the API, showcase web sites, interface design research, and local meetup groups and whole web sites devoted to learning about WebVR. For more information, Appendix X contains a list of WebVR resources.

The Web and the Power of Collaboration

The collaboration between Mozilla and Google on WebVR is a testament to the power of web standards and open community development. Mozilla were the prime movers, and could have run with their own implementation for a long time before bringing in other browser makers, but instead they decided to include Google in the discussion early on. This provided instant validation of the idea and also gave outside developers a high level of confidence that they could build cross-browser VR applications. This kind of collaboration is something you would never see with a closed-system VR platform.

Note that as of this writing, WebVR is not a *standard.* Rather, it is a set of extensions built into the two most popular browsers. But that's OK; this actually follows a typical pattern for modern browser development, where new capabilities are added as experimental features first, and then standardized after the features have been evaluated and tested by developers. Someday, we could see the WebVR API added as a W3C recommendation and become a true standard.

The WebVR API

Browsers already have the capability to render stunning real-time 3D using WebGL, the API standard for rendering 3D graphics using JavaScript. So as it turns out-- and this is not to take anything away from the great efforts of the engineers involved-- adding VR support to browsers actually required very few modifications. The WebVR API comprises these key browser innovations:

- **Discovering/connecting to VR devices.** The browser provides an API for the developer to discover and connect to the VR device(s) attached to the computer.
- **VR fullscreen mode.** Web browsers already have a fullscreen mode to support game development. VR fullscreen mode extends this to automatically put the VR device, such as a Rift, into the correct video driver mode on the computer. It also performs the Oculus-style barrel distortion in native code rather than the programmer having to implement it in their application.
- **Head tracking.** A new JavaScript API to track head position and orientation so that we can adjust the camera each frame before we render.

With these three features, a well-designed JavaScript application can use WebGL to render virtual reality scenes. Let's have a look at how to use them.

Supported Browsers and Devices

WebVR is currently implemented in development builds of Firefox and Chrome. To try them out, download the Firefox Nightly using the download links and instructions at http://mozvr.com/downloads/ or the Chromium WebVR build from https://drive.google.com/folderview?id=0BzudLt22BqGRbW9WTHMtOWMzNjQ.

These are still experimental builds, but the hope is that someday WebVR features will make it into retail versions of the browsers. There are a few serious limitations-- for example, browsers have historically throttled refresh rates at 60 frames per second, whereas good VR requires 75-90 FPS to provide convincing immersion. Also, the installation process is still fairly manual, as it requires installing the Oculus SDK separately.

WebVR, Mobile Browsers and Google Cardboard

Note that the API and techniques described in this chapter are not for use with Google's *Cardboard*. Cardboard is a specification for rendering 3D scenes with a mobile phone dropped into an inexpensive side-by-side head-mounted stereo viewer, using the phone's IMU to track head movements. It is possible to build Cardboard apps using a mobile browser; we'll cover this, along with general Cardboard development topics, in the next chapter.

Querying for VR Devices

The first thing a WebVR application needs to do is query the browser for which VR devices are connected to the computer. WebVR currently supports Oculus DK1 and DK2 devices, but eventually it could support others; also multiple VR devices may be connected the computer at once. Example 5-1 shows how to use the browser function `getVRDevices()` to get a list of connected devices.

About the JavaScript Language

I think we might need a note here generally describing JavaScript for the uninitiated, and point to a few good resources for learning. Thoughts?

Example 5-1. Querying for VR Devices

```
myVRApp.prototype.queryVRDevices = function() {

        // Polyfill - hide FF/Webkit differences
        var getVRDevices = navigator.mozGetVRDevices /* FF */ ||
                           navigator.getVRDevices; /* webkit */

        if (!getVRDevices) {
             // handle error here, either via throwing an exception or
             // calling an error callback
        }

        var self = this;
        getVRDevices().then( gotVRDevices );
        function gotVRDevices( devices ) {
             // Look for HMDVRDevice (display) first
             var vrHMD;
             var error;
             for ( var i = 0; i < devices.length; ++i ) {
                 if ( devices[i] instanceof HMDVRDevice ) {
                     vrHMD = devices[i];
                     self._vrHMD = vrHMD;
                     self.leftEyeTranslation = vrHMD.getEyeTranslation( "left" );
                     self.rightEyeTranslation = vrHMD.getEyeTranslation( "right" );
                     self.leftEyeFOV = vrHMD.getRecommendedEyeFieldOfView( "left" );
                     self.rightEyeFOV = vrHMD.getRecommendedEyeFieldOfView( "right" );
                     break; // We keep the first we encounter
                 }
             }

             // Now look for PositionSensorVRDevice (head tracking)
             var vrInput;
             var error;
             for ( var i = 0; i < devices.length; ++i ) {
                 if ( devices[i] instanceof PositionSensorVRDevice ) {
```

```
                    vrInput = devices[i]
                    self._vrInput = vrInput;
                    break; // We keep the first we encounter
                }
            }
        }
    }
```

In the examples in this section, we have created a JavaScript class named `myVRApp`. `myVRApp` defines the method `queryVRDevices()` to do the query. First, we need find the correct version of the query function in a cross-browser way. For both Firefox and Chrome, this function is a method on the `navigator` object; but Firefox names it `mozGetVRDevices()`, while the Chrome name is `getVRDevices()`. Note that this the only place in WebVR where we need conditional code based on which browser is running; from here, the API is identical across browsers.

We save the correct version of the function into a local variable `getVRDevices`, and use that to make the query. Functions like this use JavaScript *promises*, objects which have a `then()` method used to supply a callback function. The callback function, `gotVRDevices()`, iterates through the list of connected VR devices, looking for two kinds of devices. `HMDVRDevice` objects represent the connected display hardware, and `PositionSensorVRDevice` objects represent the head tracking hardware. (While the Oculus Rift has those two types of hardware integrated into a single device, that may not be the case with all VR hardware in the future, so the API designers thought it best to separate the two types in order to keep the design general enough for future use cases.) If found, the devices are tucked away into the object's `_vrHMD` and `_vrInput` properties for use in the application.

About `'self'`

Explain how we have to save 'this' into a separate variable...

Note the line of code near the beginning of the method that checks whether `getVRDevices` is non-null:

```
if (!getVRDevices) {
```

This test is important; it's our way of determining whether the WebVR API exists at all. In this way we can write robust code that also works in browsers that don't support WebVR, for example by reporting to the user that the application requires a WebVR-enabled browser.

Setting Up VR Fullscreen Mode

VR applications must run fullscreen to create a feeling of immersion. Web browsers have supported fullscreen mode for several years in order to run video games, but VR fullscreen requires additional treatment: the browser may have to use special display modes, depending on the VR hardware, and it may also need to perform additional visual processing, such as Oculus Rift barrel distortion. For WebVR, the browser method to go into fullscreen mode has been extended so that we can pass in the `HMDVRDevice` obtained from `getVRDevices()` above. Example 5-2 shows us how.

Example 5-2. VR Fullscreen Mode

```
myVRApp.prototype.goFullScreen = function() {

    var vrHMD = this._vrHMD;

    // this._canvas is an HTML5 canvas element
    var canvas = this._canvas;

    // Polyfill - hide FF/Webkit differences
    function requestFullScreen() {
        if ( canvas.mozRequestFullScreen ) {
         canvas.mozRequestFullScreen( { vrDisplay: vrHMD } );
        } else {
         canvas.webkitRequestFullscreen( { vrDisplay: vrHMD } );
     }
    }

    requestFullScreen();
}
```

The application object defines a helper method `goFullScreen()` that hides the details, but there is actually not that much to it. We just need to ask the browser to put the graphics window into VR fullscreen mode using the HMD. The graphics window in this case is an HTML5 canvas, the one used to render the application with WebGL (more on this below). Previously, during initialization, we saved the WebGL canvas in the `_canvas` property of the object. Now, we call a method of the canvas to go into fullscreen mode. This is another one of those places where different browsers use a different method name-- unfortunate, but this seems to be a fairly common practice for somewhat new features like fullscreen mode-- `mozRequestFullScreen` for Firefox, and `webkitRequestFullscreen` for Chrome. Though the method names are different, their signatures are the same. We pass the HMD device to this method, indicating that we want VR fullscreen mode instead of regular fullscreen mode.

Head Tracking

The final piece of the WebVR API implements head tracking. We need to be able to obtain an accurate position and orientation from the device, and reflect that in the current position and orientation of the camera at all times.

The code in Example 5-3 shows WebVR head tracking in action. Our application defines a method, `update()`, that is called each time it is ready to render the scene. `update()` uses the head tracking device that the application saved into the property `_vrInput` when it originally queried for VR devices. The head tracking device has a method, `getState()`, which returns an object with the current position and orientation. Those values, if present, are copied into the position and orientation of the camera being used to render the scene.

Example 5-3. Tracking the VR Device's Position and Orientation

```
myVRApp.prototype.update = function() {
        var vrInput = this._vrInput;
        var vrState = vrInput.getState();

        if ( !vrState ) {
            return;
        }

        // Update the camera's position and orientation
        if ( vrState.position !== null ) {
            this.setCameraPosition( vrState.position );
        }

        if ( vrState.orientation !== null ) {
            this.setCameraOrientation( vrState.orientation );
        }

}
```

These are the basics of the WebVR API. In the next section, we'll put them into practice in the context of a full application.

Creating a WebVR Application

Now that have a sense of how the API works, let's shift gears and look at how to put WebVR applications together. Web applications are different beasts from native desktop and mobile apps, and developing for HTML5 comes with its own set of challenges and trade-offs. The web offers many choices in open source tools and libraries, but most of the tools aren't as polished as systems like Unity3D and Unreal; we have cross-platform development capability, but we still need to mindful about the differ-

ences between browsers; and while it's easy to get web applications online, deploying them professionally can be a black art.

In this section we will build a full, though quite simple, WebVR application to illustrate the issues. We will look at how put a web "page" together that is the WebVR app; how to create and render 3D scenes using Three.js, a popular JavaScript library for programming WebGL; how to use Three.js objects that implement Oculus Rift stereoscopic rendering and head tracking; and finally, how to implement an HTML5-based user interface suitable for launching into VR fullscreen mode. Let's do it.

Three.js: A JavaScript 3D Engine

As WebVR developers, we'll need to do 3D rendering. The API of choice for creating 3D in a web browser is WebGL. WebGL runs in all browsers and on all platforms and devices. WebGL allows access to the full capabilities of the GPU to create beautiful real-time 3D renderings and animations in web pages for all types of applications, and it's perfectly suited to rendering virtual reality. But to do anything more than the most basic tasks using the API out-of-the-box requires serious effort and literally hundreds of lines of code. This is not a recipe for rapidly building applications in the modern world. So unless you feel like creating a 3D game engine from scratch, you will probably find a library to help ease the burden.

While there are many choices for getting started with your WebGL development, the undisputed leader in this category is **Three.js** (http://threejs.org/). Three.js provides an easy, intuitive set of objects that are commonly found in 3D graphics. It is fast, using many best-practice graphics engine techniques. It is powerful, with several built in object types and handy utilities. It is open source, hosted on GitHub, and well maintained, with several authors contributing to it.

Three.js has become a de facto choice for WebGL development. Most of the great WebGL content you can view online has been built with it, and there are several rich and highly innovative works live on the web today. The WebVR team at Mozilla developed a set of objects for extending Three.js with VR capability and contributed them to the project, and now many developers are using those extensions to create the new generation of Three.js applications for WebVR.

A Full Example

Before we dig into the code, let's run the example, depicted in Figure 5-2. Here we see the Chromium browser running windowed, displaying the stereo rendering for a cube texture-mapped with the WebVR logo.

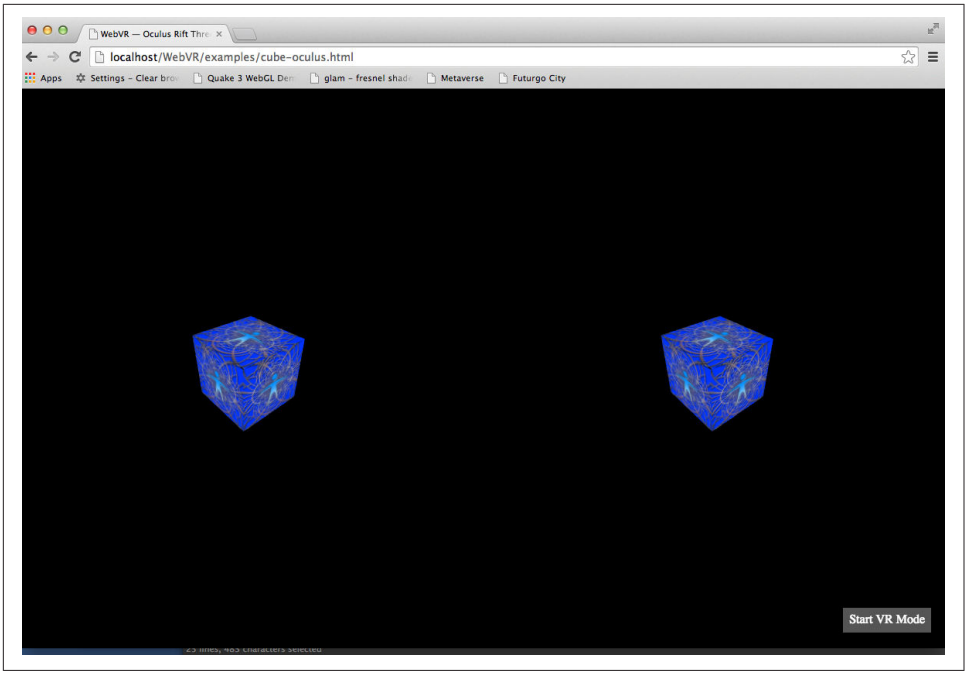


Figure 5-2.
A simple WebVR example running in the Chromium WebVR-enabled build

http://tparisi.github.io/WebVR/examples/cube-oculus.html

WebVR logo via Pixabay http://pixabay.com/en/network-networking-networked-365945/

If you have your Oculus Rift attached to the computer, you can tilt the headset and you will see the cube move around the screen. (Actually, the camera is moving in response to the changes in the headset's orientation.)

Now, click the **Start VR Mode** button. This will put the browser in fullscreen mode. It's time to put on your Rift and look around. You should now be immersed in the experience. And there it is: welcome to WebVR!

Setting Up the Project

To look through the code, go clone the repository from Github at

https://github.com/tparisi/WebVR

Running Web Examples on Your Local Machine

If you plan to run this and other web examples from your local machine, you will need to run them using a web server.

I run a local version of a standard LAMP stack on my MacBook… but all you really need is the 'A' part of LAMP, i.e. a web server such as Apache. Or if you have Python installed, another option is the SimpleHTTPServer module, which you can run by going to the root of the examples directory and typing

```
python -m SimpleHTTPServer
```

and then pointing your web browser at http://localhost:8000/. There is great tech tip on this feature at the Linux Journal web site at

http://www.linuxjournal.com/content/tech-tip-really-simple-http-server-python.

The Web Page

All web applications start with a web page, and WebVR is no exception. The listing in example 5-4 shows the HTML5 markup for the web page. After a bit of header information that includes the title and an embedded CSS style to style the **Start VR Mode** button and 3D canvas, we see the body of the page. It's pretty simple, just two elements: an element named `button`, for the button, and one named `container`, that will contain a WebGL canvas for drawing. We will initialize the 3D contents of `container` in a moment, using JavaScript. And that's it for the page markup.

Example 5-4. The Web Page Markup

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>WebVR &mdash; Oculus Rift Three.js Cube</title>
      <style>

            .button {
                position: absolute;
                bottom: 20px;
                right: 20px;
                padding: 8px;
                color: #FFF;
                background-color: #555;
                z-index:1;
            }

            #container {
                position:absolute;
```

```
                left:0;
                top:0;
                width:100%;
                height:100%;
            }
        </style>


</head>
<body>

    <div class="button">Start VR Mode</div>
    <div id="container"></div>

</body>
```

The JavaScript Code

The next few lines on the page are script elements that include JavaScript code for various web libraries. The important ones here are the script elements for Three.js and the two Three.js extensions for WebVR, the classes `THREE.VREffect` and `THREE.VRControls`. These extensions were developed by Diego Marcos of Mozilla and contributed to the Three.js repository on Github. Like many Three.js extensions, they are not built into the main build of Three.js, but they are included in the examples that come with the project on Github.

```
<script src="../libs/jquery-1.9.1/jquery-1.9.1.js"></script>
<script src="../libs/three.js.r68/three.js"></script>
<script src="../libs/three.js.r68/effects/VREffect.js"></script>
<script src="../libs/three.js.r68/controls/VRControls.js"></script>
<script src="../libs/requestAnimationFrame/RequestAnimationFrame.js"></script>
```

Following that, we have a script element with several lines of JavaScript defined inline, shown in listing 5-5. This is the code for the application. It starts with the main program, a jQuery callback function that will be called once the page is loaded and ready. The function calls several helper functions, which we will go through in detail.

Example 5-5. The Main JavaScript Program

```
<script type="text/javascript">
    var container = null,
    renderer = null,
    effect = null,
    controls = null,
    scene = null,
    camera = null,
    cube = null;
```

```
$(document).ready(
        function() {

            // Set up Three.js
            initThreeJS();

            // Set up VR rendering
            initVREffect();

            // Create the scene content
            initScene();

            // Set up VR camera controls
            initVRControls();

            // Run the run loop
            run();
        }
);
```

First, we have to initialize our use of the Three.js library. Let's look at the code for this, shown in Example 5-6.

Example 5-6. Initializing Three.js to Render in WebGL

```
function initThreeJS() {
    container = document.getElementById("container");
    // Create the Three.js renderer and attach it to our canvas
    renderer = new THREE.WebGLRenderer( { antialias: true } );

    // Set the viewport size
    renderer.setSize(window.innerWidth, window.innerHeight);
    container.appendChild(renderer.domElement);

    window.addEventListener( 'resize', function(event) {
        renderer.setSize(window.innerWidth, window.innerHeight);
    }, false );

}
```

We ask Three.js to create a new renderer object, asking it to render in WebGL using antialiasing. Antialiasing is a rendering technique that eliminates jagged lines. It's not turned on by default so we must specify that by passing the `antialias` property to the constructor. We then size the renderer object to match the size of the entire window--remember, VR applications are full screen affairs. Under the covers, the Three.js renderer object creates a DOM `Canvas` element, which is required for rendering WebGL. We add the canvas to our page, and we are now ready to render graphics. As a final flourish we add a resize handler, so that when the window is resized the Three.js renderer object is, too. Three.js initialization is now complete.

Next, we call the helper function `initVREffect()` to set up stereo rendering and full-screen mode handling. The code for this is shown in Example 5-7.

Example 5-7. Setting up Three.js Stereo Rendering and Browser Fullscreen Mode

```
function initVREffect() {
    // Set up Oculus renderer
    effect = new THREE.VREffect(renderer, function(err) {
        if (err) {
            console.log("Error creating VREffect: ", err);
        }
        else {
            console.log("Created VREffect: ", effect);
        }
    });

    // Set up fullscreen mode handling
    var fullScreenButton = document.querySelector( '.button' );
    fullScreenButton.onclick = function() {
        effect.setFullScreen(true);
    };
}
```

`initVREffect()` creates an instance of `THREE.VREffect`, which will be used in our run loop below to render the scene in stereo from two cameras. The function also sets up fullscreen mode by adding a callback to the button object defined in the markup. The callback calls the effect's `setFullScreen()` method to go into fullscreen when the button is clicked.

Now, we need to create the 3D content that we will render in the scene. With Three.js, we can create content by loading models in a variety of 3D formats such as Wavefront OBJ and COLLADA, or we can create it by writing code. We'll do the latter, as shown in Example 5-8.

Example 5-8. Initializing the Three.js Scene

```
function initScene() {
        // Create a new Three.js scene
        scene = new THREE.Scene();

        // Add  a camera so we can view the scene
        // Note that this camera's FOV is ignored in favor of the
        // Oculus-supplied FOV for each used inside VREffect.
        // See VREffect.js h/t Michael Blix
        camera = new THREE.PerspectiveCamera( 90,
          window.innerWidth / window.innerHeight, 1, 4000 );
        scene.add(camera);

        // Create a texture-mapped cube and add it to the scene
        // First, create the texture map
        var mapUrl = "../images/webvr-logo-512.jpeg";
        var map = THREE.ImageUtils.loadTexture(mapUrl);
```

```
        // Now, create a Basic material; pass in the map
        var material = new THREE.MeshBasicMaterial({ map: map });

        // Create the cube geometry
        var geometry = new THREE.BoxGeometry(2, 2, 2);

        // And put the geometry and material together into a mesh
        cube = new THREE.Mesh(geometry, material);

        // Move the mesh back from the camera and tilt it toward the viewer
        cube.position.z = -6;
        cube.rotation.x = Math.PI / 5;
        cube.rotation.y = Math.PI / 5;

        // Finally, add the mesh to our scene
        scene.add( cube );

    }
```

Three.js has a simple, easy to use paradigm for creating graphics. The scene consists of a hierarchy of objects. At the root is a THREE.Scene, and each object is added as a child of that scene, or a descendant thereof. This particular scene consists of just the cube, and a camera from which to view it. We create the camera first, a `THREE.PerspectiveCamera`. The parameters to the camera's constructor are field of view, aspect ratio, and the distance from the camera to the front and back clipping planes. Under the covers, the `VREffect` object used by WebVR will ignore the field of view, using instead a set of values provided in the `HMDVRDevice` device we discovered when we initialized the `VREffect`.

The remainder of the scene creation code concerns building the cube. The cube object is a Three.js mesh, of type `THREE.Mesh`. To create a mesh, we need a piece of geometry and a material. The geometry is of type `THREE.BoxGeometry`, and the material is a `THREE.MeshBasicMaterial`, that is, an unlit object which in this case has a texture map. We supply the texture map as the `map` property to the material's constructor. After we create the mesh, we move it back a little bit from the camera--which is positioned at the origin by default-- and rotate it a bit around the `x` and `y` axes so that we can see it's a real 3D object. We add it to the scene as a child, and with that, we have a visual scene ready to render. We're almost there!

The last major piece of setting up the VR support is head tracking. The way this works with Three.js is to create a *camera controller*, that is, an object which will changes the position and orientation of the camera based on user input. For non-VR applications, Three.js comes with first person controllers, orbit controllers and other ways to move the camera based on the mouse and keyboard. For WebVR, we need a controller that uses the WebVR API for head tracking. Diego's WebVR extension comes with a class, `THREE.VRControls`, for this purpose. As we can see from the code

listing in Example 5-9, it's really only one line of code to set up, though in this example we are also supplying an error handler for the case when the app can't create the controller for some reason, e.g. the browser is not VR-ready.

Example 5-9. Setting up the VR Controls in Three.js

```
function initVRControls() {

    // Set up VR camera controls
    controls = new THREE.VRControls(camera, function(err) {
        if (err) {
            console.log("Error creating VRControls: ", err);
        }
        else {
            console.log("Created VRControls: ", controls);
        }
    });
}
```

That's it for setup and initialization. The only thing left is to implement the *run loop*, which is the heart of our application. The run loop drives continuous animation and rendering of 3D scenes, by handing the browser a callback function that it will call every time it is ready to render the contents of the page again. Let's walk through the listing in Example 5-10.

Example 5-10. The Run Loop

```
var duration = 10000; // ms
var currentTime = Date.now();
function animate() {

    var now = Date.now();
    var deltat = now - currentTime;
    currentTime = now;
    var fract = deltat / duration;
    var angle = Math.PI * 2 * fract;
    cube.rotation.y += angle;
}

function run() {
    requestAnimationFrame(function() { run(); });

        // Render the scene
        effect.render( scene, camera );

        // Update the VR camera controls
        controls.update();

        // Spin the cube for next frame
        animate();

}
```

The helper function `animate()` is responsible for animating the scene by rotating the cube a little about the y-axis each frame. It does this by calculating how much to add to the `y` rotation as a function of time. Each time through, we ask the browser for the current time (in milliseconds) and subtract a previously saved value to obtain `deltat`; `deltat` is then divided by our duration value, in this case 10 seconds, to obtain a fractional amount; and the fraction is used to calculate `angle`, a rotation that is a fraction of 2 * PI radians (or 360 degrees).

But you may be wondering how `animate()` gets called. Modern browsers support a function, `requestAnimationFrame()`, which an application can use to request the browser to call it when it's ready to redraw the page. We pass our function run() to that, and thus we have our run loop. Every time through the run loop, run() performs the following actions:

1. Render the scene, using the `THREE.VREffect` object, which renders in stereo suitable for WebVR;
2. Update the THREE.VRControls object, which polls the connected WebVR devices for the latest position and updates the main camera;
3. Call `animate()` to drive the animation of the rotating cube.

And that's it. This example is contrived, sure, but it shows the basics of building an end-to-end WebVR application. With the tools covered in the next section, we can go far beyond spinning cubes; we can create full VR applications with high production value and awesome performance.

If you are curious to know about the inner working's of Mozilla's WebVR extensions to Three.js, you can look through the code in the latest repository on Github:

https://github.com/mrdoob/three.js/
Also, the source files for the WebVR extensions are included in the Github project for this example at:

https://github.com/tparisi/WebVR

Tools and Techniques for Creating Web VR

Now that we have seen how to build a basic WebVR application using the new browser APIs and the Three.js library, it's time to take a step back. While Three.js is a great, easy way to get started, it is by no means our only way to create and deploy WebVR. In this section we'll take a quick tour of the tools and frameworks at our disposal.

WebVR Engines and Development Tools

Over the last few years, several tools have emerged to create 3D content and applications for the web using WebGL. Unlike tools like Unity3D and Unreal, these products are web applications; there is no need to download packaged software. Developers use a web-based interface to create or import art, and then add behaviors, scripts and interaction, all within a browser. Several such tools have been enhanced recently to support virtual reality via WebVR, including:

playcanvas (http://www.playcanvas.com/) - London-based playcanvas has developed a rich 3D engine and cloud-based authoring tool. The authoring tool features real-time collaborative scene editing to support team development; Github and Bitbucket integration; and one-button publishing to social media networks. The playcanvas engine is open source and can be downloaded from Github at https://github.com/playcanvas

One playcanvas developer has built an Oculus-enabled version of *AfterGlow*, a downhill ski racing game created to promote Philips' Ambilight lighting product (see http://afterglowskigame.com/). There is also a playcanvas add-in on Github at https://github. com/playcanvas/oculus-vr. Figure 5-3 shows a screenshot of the playcanvas development environment.

Goo Create (http://www.gootechnologies.com/) - Goo is oriented toward digital marketing, but the engine boasts a list of traditional game engine features. The company is building apps and experiences both for the Oculus and more lightweight mobile systems such as Moggles (http://moggles.com/).

Goo Create now features a VR camera script in the asset library; this can be dragged and dropped inside the viewport onto a camera and voilá - instant VR.

Verold Studio (http://www.verold.com/) is a lightweight publishing platform for 3d interactive content developed by Toronto-based Verold, Inc. It is a no-plugin, extensible system with a simple JavaScript API, so that hobbyists, students, educators, visual communication specialists and web marketers can integrate 3D animated content easily into their web properties. Verold customers are now moving their 3D creations into VR, including in education, architecture and product design. Mozilla is also using it to build out its MozVR showcase site (see below).

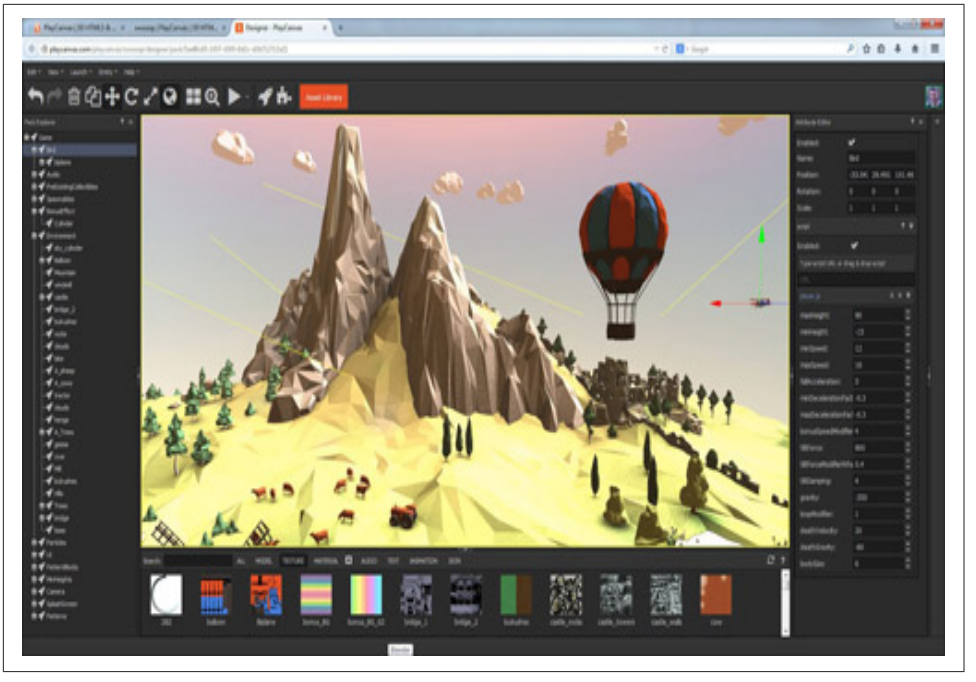


Figure 5-3.
PlayCanvas, a Web-Hosted Game Engine and Development Tool

https://playcanvas.com/

The tools surveyed above are very promising. However, they are still not that mature. Also, the usage fees and source code licensing for some of them are still in flux. Depending on your exact project needs and budget, your mileage may vary. Make sure to check the websites for the latest information on each tool.

Using Unity3D and Unreal for WebVR Development

If you are already developing desktop or mobile native and using Unity3D or Unreal, you probably don't want to learn another tool. The good news is that you don't have to: both Unity and Unreal have added WebGL export. As of this writing, the WebGL export feature in Unity is free to use, while the Unreal WebGL export is free to use but requires a royalty on games/apps sold. Consult the web sites for these tools to get the most up-to-date licensing information.

Figure 5-4 shows the Unity3D WebGL exporter in action: a screen shot of the WebGL version of *Dead Trigger 2* by developer Madfinger Games.

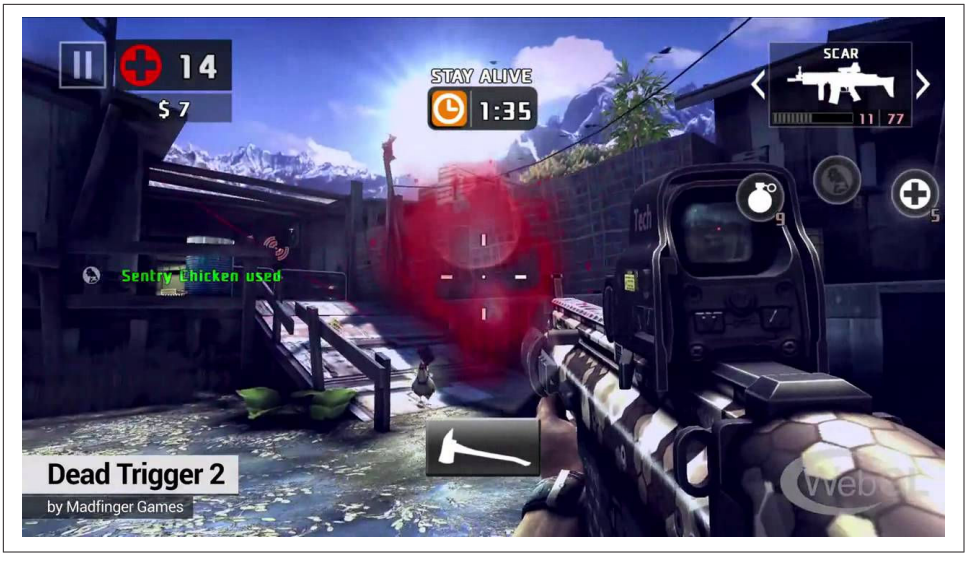


Figure 5-4.
WebGL Version of Dead Trigger 2, Using Unity3D WebGL Exporter

http://www.madfingergames.com/deadtrigger2/

For existing developers of native applications that want to try developing for the web, this would seem like the perfect solution. But it's not without its issues. These export paths rely on a rather new technology called *Emscripten* (https://github.com/kripken/emscripten/wiki). Emscripten cross-compiles the actual code from a game engine like Unity to a low-level JavaScript called asm.js. The result is a game or experience with very high frame rate, comparable to a native game engine. But the cross-compiled engine code itself can consist of several megabytes of JavaScript code that has to get downloaded into the browser-- before the game itself downloads. So expect to see lengthy load screens and progress bars before your VR experience comes over the wire. If this is acceptable, then no worries; if it's not, then maybe you want to look into using one of the engines discussed in the previous section.

Open Source Libraries and Frameworks

When it comes to JavaScript libraries for 3D development, Three.js isn't the only game in town. There are other great libraries worthy of a look, including:

BabylonJS (http://www.babylonjs.com/) is a fully featured open source JavaScript library created by David Catuhe of Microsoft. BabylonJS has added a `BABYLON.WebVRCamera` object, which not only renders in stereo but also implements the camera con-

troller, so it is like a combination of the two VR extensions to Three.js described above.

GLAM (for GL and Markup) (http://www.glamjs.org/) is a declarative language for 3D web content, which I designed. GLAM combines the power of WebGL with a set of easy-to-use markup tags and style properties. To create 3D, you define page elements, set their attributes, add event listeners, and define styles in CSS - just like authoring a web page. GLAM is intended for general web 3D development, but also includes features for developing WebVR in Oculus and Cardboard VR. To get a taste of what authoring 3D markup might look like, refer to Example 5-11 below. The document defines a WebVR scene with a cube. Rather than writing dozens of lines of setup code in Three.js, you just author the content, and the GLAM library takes care of the rest.

SceneVR (http://www.scenevr.com/) is another declarative language, created by Ben Nolan. SceneVR has been designed specifically for creating virtual reality scenes but is actually quite similar to GLAM. Ben and I are collaborating to create a unified tag set, though this work is preliminary.

Example 5-11. A WebVR Scene Defined in the GLAM Language

```
<glam>
  <renderer type="rift"></renderer>
  <scene>
    <controller type="rift"></controller>
    <cube id="photocube"></cube>
  </scene>
</glam>
```

WebVR and the Future of Web Browsing

WebVR has huge promise. The ability to access rich VR experiences without a download or install, and to author VR content using simple, affordable tools, could be revolutionary. But it is still very early, and there is going to be a lot of experimentation.

User interface is a very interesting area for investigation. The design of web-connected VR applications raises interesting questions, such as: once we create connected, fullscreen immersive VR experiences on the web, how do we navigate between them? What is the equivalent of a bookmark? A cursor? The **Back** button? With these in mind, the Mozilla team, led by VR researcher Josh Carpenter, has launched MozVR (http://www.mozvr.com/), a combination showcase site and playground for experimenting.

MozVR features several demos, an integrated user interface for navigating between VR experiences, and a blog with reports on the latest research and projects. See Figure 5-5.

Figure 5-5.
Sechelt, a Virtual Reality Landscape Experience featured on MozVR, Mozilla's Showcase Site

http://mozvr.com/

MozVR is still a work in progress. But Josh and team are dedicated to advancing the state of the art and are keen to collaborate. If you are building something cool with WebVR, they would love to hear from you.

Chapter Summary

In this chapter we delved into the exciting world of WebVR. Today's leading browsers support virtual reality, allowing us to develop VR experiences using open source tools, and deliver them to end users without a download or app install. WebVR provides device discovery, fullscreen stereoscopic rendering and head tracking in a consistent API that allows us to write our apps once and deliver them in all browsers.

Developing for WebVR is largely about developing in WebGL. We took a tour of a complete WebVR application written in the popular Three.js library, using WebVR extensions developed by the Mozilla VR research team. We also surveyed several existing WebGL tools and engines that are suitable for creating WebVR, including tools we've seen in previous chapters like Unity3D and Unreal, which now have WebGL export capability.

WebVR is new, experimental, and not quite ready for prime time. But by tapping into the Web's power of open collaboration, and harnessing the talents of countless committed individuals, it has tremendous potential. WebVR could not only change the playing field for virtual reality development; it could transform the face of web browsing forever.

CHAPTER 6

VR Everywhere: Google Cardboard for Low-Cost Mobile Virtual Reality

So far we have focused on developing consumer VR for high end hardware such as Oculus Rift, HTC Vive, and Samsung Gear VR. These systems can be expensive, from several hundreds to small thousands of dollars, after you factor in peripherals and, potentially, purchasing a new computer to power them. Given these prices, the top end of consumer virtual reality is still for early adopters, not the mass market. It would be great if there was a lower-cost VR option for the average consumer, and for developers who aren't ready to make a big financial commitment. Thankfully, we have one such option in Google's *Cardboard VR*.

Google introduced Cardboard VR in 2014 to enable low-cost virtual reality using existing smart phones. The idea was that, by simply dropping a mobile phone into a cardboard box costing about US$2 in parts, anyone can experience VR. Google's original Cardboard VR unit, debuted at the I/O conference in May 2014, is pictured in Figure 6-1.

Figure 6-1. Google's Cardboard VR Viewer

By Google's accounting, as of early 2015 more than a million Cardboard headsets had already been distributed, a number that far exceeds the existing installed base of all the high end systems combined. As we will see later in this chapter, applications written for Cardboard can also be used with several other types of "drop-in" virtual reality stereo viewers, making the potential market for Cardboard applications even larger. Cardboard is also proving to be a popular choice among developers, with hundreds of applications already available for both Android and iOS.

This chapter covers how to build applications for Cardboard VR. There are actually several ways to do this:

- **Use the Cardboard SDK for Android** to create native applications in Java with OpenGL. This is a good option for experienced native developers; however it only works for Android.
- **Use the Cardboard SDK for Unity** to create native applications with the Unity3D engine. This approach works well for people already conversant with Unity and C# programming, and makes it easy to build for both Android and iOS.
- **Use HTML5, WebGL and JavaScript** with a mobile browser such as mobile Chrome or mobile Firefox to create a web application. If you are most comfortable with web development, and/or you are already creating WebVR desktop applications (see Chapter 5), this might be a good option for you.

We will go over each of these in some detail. This chapter covers broad territory--three different programming languages and as many development environments. If you can work your way through all of it, you will be approaching VR Jedi status. But

if you don't, that's OK. Feel free to focus only on the environments you know. If you are most comfortable with native Android, then the first section will be the most valuable. If you are a Unity3D developer, or think that Unity is your best approach to covering all the platforms with the least amount of work, then go for it. And if you're already a web developer, then the last section will be for you. In this respect, Cardboard VR is kind of like the lottery: many ways to play, more ways to win!

Cardboard Basics

Before we dive into the specifics of developing with each environment, let's get an overview of the world of Cardboard, including the phones and operating systems that can run it, where to buy headsets (or how to build your own), how the technology works in general, and some of the cool applications that have already been built.

Supported Devices and Operating Systems

Cardboard runs on most Android phones. The original specifications were designed to accommodate phones with screen sizes up to about 5.2 inches, but with advent of larger phones and phablets, cardboard manufacturers have been making newer headsets that can hold phones with display sizes up to about 6 inches.

Though the scheme was originally designed for use with Android phones, there is actually nothing Android-specific about programming a Cardboard application. Cardboard apps written for iOS run well on newer iPhone models, specifically the iPhone 5 and 6 series.

The Cardboard Situation for iOS

As of this writing, the majority of Cardboard VR applications are Android-based, but we can expect to see that change as overall interest in VR increases. I attend and organize quite a few VR meetups in the San Francisco Bay area, and the topic of iOS comes up often. According to many San Francisco-based Cardboard developers, iOS support is one of the most frequent feature requests coming from their users. This may be anecdotal evidence, but it is still telling.

Headset Manufacturers

Cardboard is actually a reference specification. Google doesn't offer it as a product--though they give plenty of the headsets away at trade shows and marketing events. You get the specifications from Google, and build a headset of your own. The Cardboard specifications can be found at

https://www.google.com/get/cardboard/manufacturers.html

If you don't have the time or inclination to build a cardboard from scratch, you can also purchase a ready-to-assemble kit from one of several manufacturers, including DODOcase, I Am Cardboard, Knox Labs, and Unofficial Cardboard. In addition to selling headset kits, each of these manufacturers also provides a mobile app, downloadable from the Google Play store and/or the iTunes store, that provides a handy list of Cardboard-ready VR applications.

Other Drop-In Virtual Reality Headsets

As it turns out, Cardboard isn't the only mobile phone-based virtual reality viewing system; in fact, it wasn't even the first. There are several other "drop-in" viewers on the market. These headsets are made from a variety of materials, such as hard plastic and foam rubber, and come in different form factors with support different lens shapes, distortion methods and fields of view. They tend to cost more than Cardboard, but are typically priced at around US$100. Here are a few of the notable products:

- The *Durovis Dive* (http://www.durovis.com/index.html), arguably the first smartphone VR headset, came out well before Google Cardboard.
- *MergeVR* (http://mergevr.com/) is a durable foam rubber headset that also comes with Bluetooth input devices to enhance the experience. MergeVR is depicted in Figure 6-2.
- The *Wearality Sky* (http://wearality.com/) a newer entry to the market, features a comfortable lightweight plastic frame and a wide field of view.

Figure 6-2. The MergeVR Drop-In Headset

Each of these headsets comes with its own Android and/or Unity SDKs that fully exploit the device's various features. This could be a recipe for chaos-- requiring

developers to create custom builds for each device-- but thankfully, the rise in popularity of the Cardboard specification has forced most of them to support Cardboard in a compatibility mode. Long story short, most of the techniques described in this chapter should work with the other drop-in headsets right out of the box, and you can be confident that your Cardboard app will run fairly well on them.

Cardboard Applications

Go the Google Play Store at https://play.google.com/store and type "Cardboard VR" into the search box. You will find hundreds of applications, including games, 360-degree video and photo viewers, educational simulations and at least two ports of the famous Tuscany demo. Given the low cost of manufacturing, Cardboard seems like a natural fit for live events and advertising campaigns: it has already powered VR experiences with big-name entertainers like Sir Paul McCartney and Jack White, and brands such as Volvo. Not surprisingly, games make up the lion's share of entries in the Play Store, such as Sharks VR (https://play.google.com/store/apps/details?id=com.lakento.sharksvr), the undersea adventure game depicted in Figure 6-3.

Figure 6-3. Sharks VR, an Undersea VR Adventure Game

The Play Store also features several "launcher" apps, which present a curated list of VR applications that makes it easier to find the best experiences. Most of the headset makers provide a launcher app to go with their Cardboard device, such as Google's demo application named, simply, Cardboard (https://play.google.com/store/apps/details?id=com.google.samples.apps.cardboarddemo), pictured in Figure 6-4.

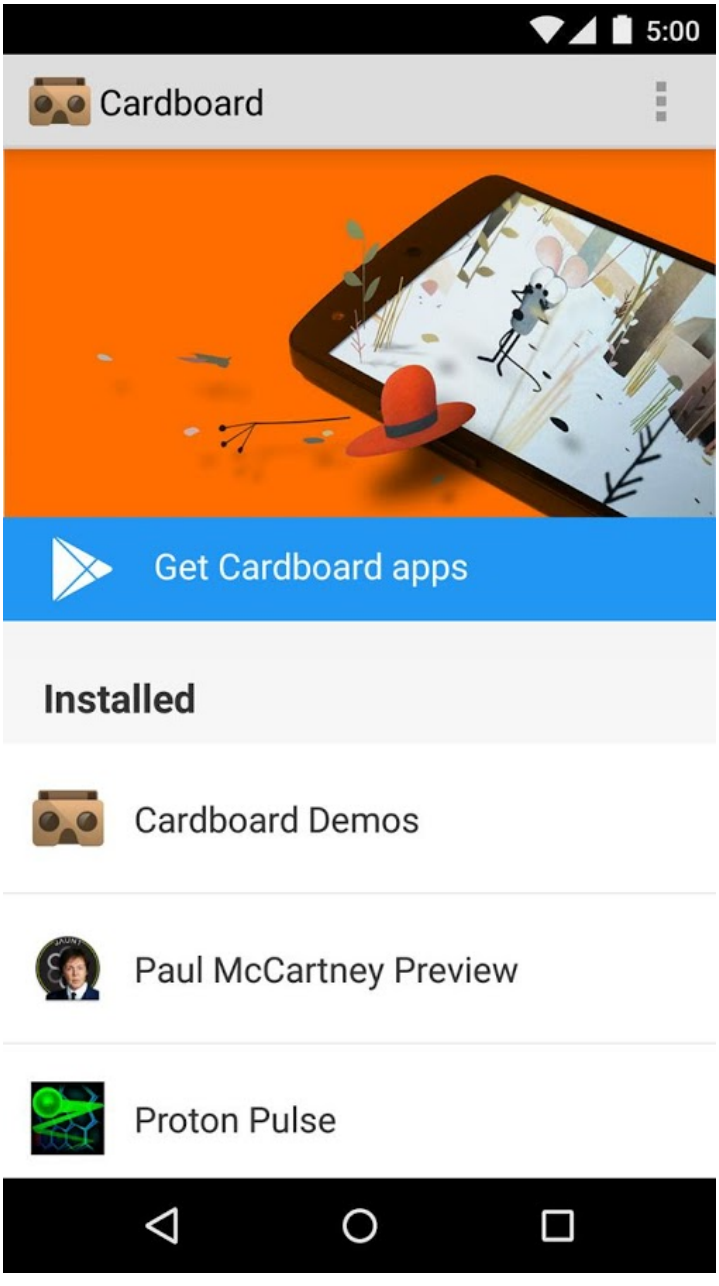


Figure 6-4. Google's Cardboard Demo Application

The Cardboard demo application can launch applications by tapping one of the icons, or you can use the novel virtual reality interface to start its featured applications from within the app itself

. The VR launcher mode is shown in Figure 6-5.

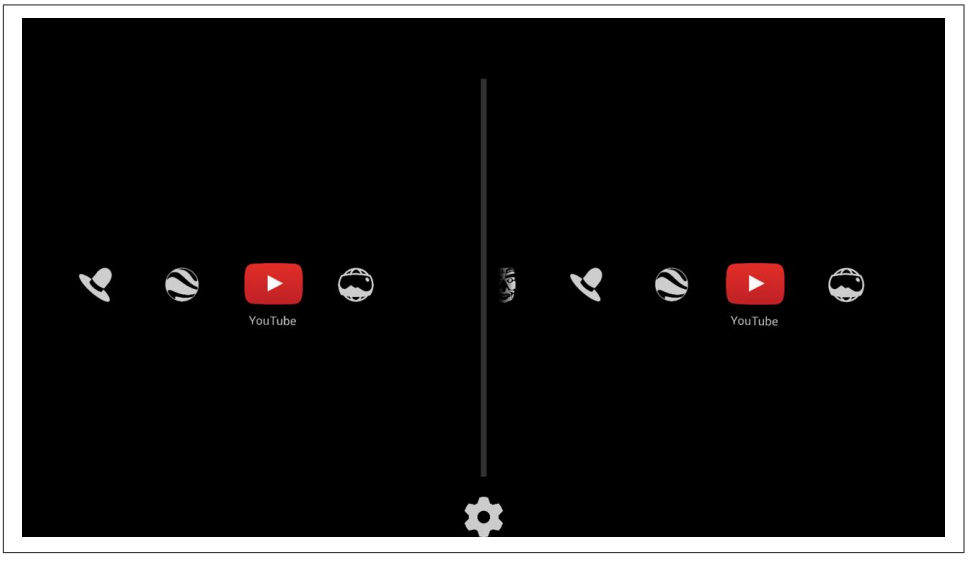


Figure 6-5. Google's Cardboard Demo Application in VR Launcher Mode

Many of the Cardboard VR headset makers have developed their own launcher apps, including with their own VR exploration modes, such as the DODOcase VR Store application (https://play.google.com/store/apps/details?id=com.dodocase.vr).

Input Devices for Cardboard

Input for Cardboard VR can be challenging. The phone is fully contained inside a box, so the user doesn't have access to the most commonly used mobile input mechanism, namely the touch screen. Different makers of Cardboard and other drop-in headsets have devised different solutions to the problem.

Google's input device is shown in the image of the Cardboard headset depicted in Figure 6-1. There is metal ring on the upper left of the headset, on which you pull downward when you want to select something in the VR scene. The ring is actually a magnet, and there is a second magnet on the inside of the box. By pulling down on

the magnet, the user is perturbing a magnetic field, and this perturbation is detected by the magnetometer on the phone. This is a brilliant solution! But while the magnet switch works very well, it doesn't work with all phones, because it is highly dependent on the position of the magnet on the phone. The magnet switch also only works with Android native code; it can't be used with iOS, nor in a mobile browsers, because HTML5 browsers do not support an API for detecting changes to the phone's magnetic sensor.

DODOcase, another popular Cardboard manufacturer, wasn't satisfied with the lack of device coverage provided by the magnet switch, so they devised a completely different mechanism. Refer to Figure 6-6 depicting a DODOcase headset. A wooden stick protrudes from the upper right side of headset. By pressing downward on the stick, a piece of capacative plastic inside the headset bends and touches the phone's touch screen, essentially mimicking a finger tap. This is also a brilliant solution, and more universal than the magnet switch, because it works with many more Android phones, as well as iPhones and mobile browsers on both platforms.

Figure 6-6. The DODOcase Cardboard Viewer with Universal Input Switch

While the DODOcase input switch is a step in the right direction, it is still experimental and made of inexpensive materials. (No, you're not imagining things; that is indeed a Popsicle stick protruding from the box!). It was devised for early developers

and experimenters to play with. If it ever becomes widely adopted, the company can make it out of something more durable.

Other Cardboard makers are experimenting with more sophisticated input devices, such as Bluetooth-connected game controllers. The Sharks VR app shown in Figure 6-3, for example, uses a game controller. Also, some headset companies are working on new types of Bluetooth controllers specifically designed for VR; note the plastic controller shown in the foreground of the MergeVR screenshot in Figure 6-2 above.

But each of these kinds of controllers requires custom programming, and as of this writing, there is no standardized input mechanism for mobile VR. This situation has led to many apps being designed to only use gaze tracking and a cursor countdown. We will see examples of both later in the chapter.

Cardboard Stereo Rendering and Head Tracking

The Cardboard VR approach to stereo rendering is simpler than Oculus: it is a 90 degree horizontal field of view without the barrel distortion. This allows applications to do simple side-by-side rendering in two viewports, one for each eye. Figure 6-7 shows an example of a 3D scene rendered side-by-side for Cardboard, from the app *Stadiums for Cardboard VR*.

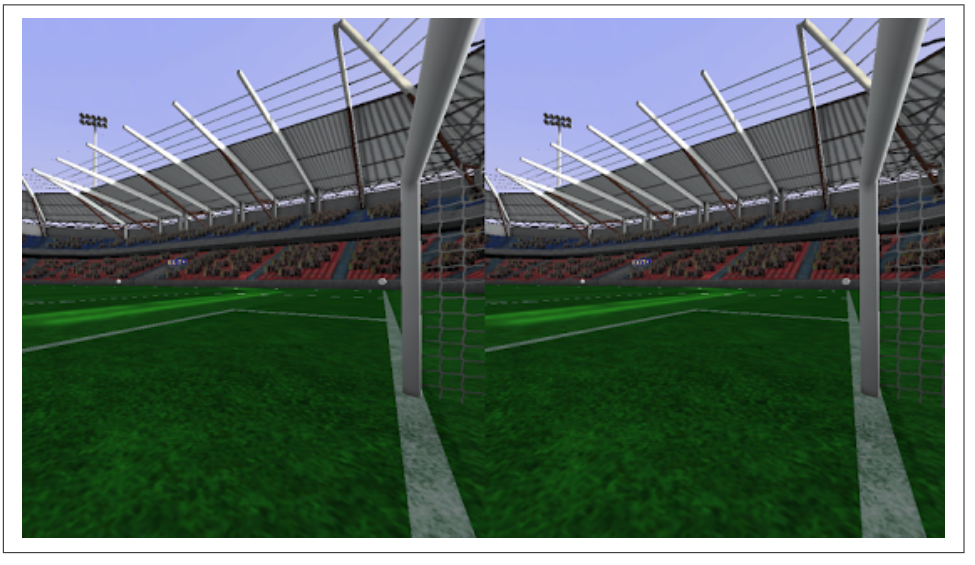


Figure 6-7.
Side-by-side Rendering from Stadiums for Cardboard VR

https://play.google.com/store/apps/details?id=indie.AlbuSorinCalin.StadiumsDemoVR

Head tracking in Cardboard is also a bit simpler than with Oculus. It uses the existing operating system orientation events generated by the phone's compass and accelerometer, collectively known as the *inertial measurement unit*, or IMU. Note that today's phone operating systems deliver IMU changes at approximately 60 frames per second. This is slower than the 75-120 FPS targets cited by Oculus and other high-end HMD makers; however because of the smaller field of view, most users find the slower tracking acceptable, at least for short experiences.

Now, the details of programming the stereo rendering and head tracking differ greatly between native Android, Unity3D-based applications, and HTML5 web code. But the principles are the same for all of them: render the scene twice-- once for each eye with a natural stereo separation-- and update the camera based on the IMU's orientation.

So Is Cardboard VR Good Enough VR?

Because of the 60 FPS head tracking, the narrower field of view, and for older phones, the lower display resolution (compared to the Samsung Galaxy Note 4's sexy 1280 pixels per eye), many VR insiders consider Cardboard an inferior experience. Clearly, that is a subjective assessment. Proponents of Cardboard argue that the generally lower-end experiences are acceptable for shorter periods of use, i.e. as "bite-size" VR that makes the medium more affordable and thus accessible to more people.

Also, it is worth keeping in mind that the Cardboard team at Google isn't sitting still. They know the current limitations better than anybody, and have alluded to making changes in future versions of Android to address both the field of view and tracking speed limitations. So it's possible that a future version of Cardboard will get closer to parity with higher-end VR platforms.

Developing with the Cardboard SDK for Android

In this section we will explore Java-based development of native Android VR applications. If you are not interested in using Java and OpenGL to create Cardboard applications for Android, or need to run an app that will also run on iOS, feel free to skip this section and move on to the sections on Unity3D and HTML5.

Google's Java-based SDK, known as the *Cardboard SDK for Android*, comes with a ready-to-build sample application and a detailed tutorial. There is no reason to duplicate all of that information here, and to do so would be beyond the scope of this

book. But we will take a look at the highlights so that we can get a feel for the tools, technical concepts and development process. The SDK example is a treasure hunt game, depicted in Figure 6-8.

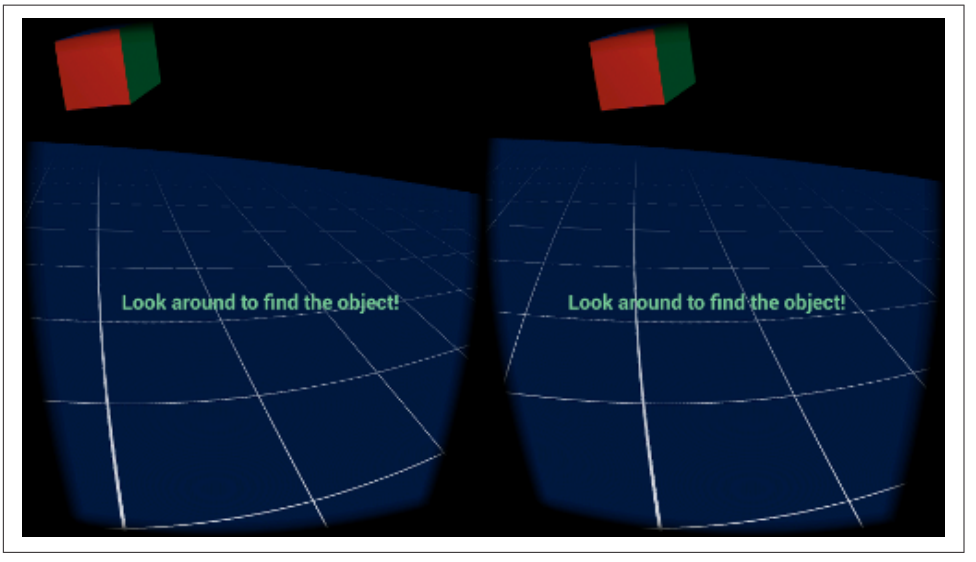


Figure 6-8. Treasure Hunt Sample Included with the Cardboard SDK for Android

The object of the game is simple: look for cubes floating in space. When you gaze over one, it highlights; pull down on the magnet to select it; the cube disappears, you score a point, and you are prompted to look around for another one; then do it again. It's not the most satisfying game play, but it does illustrate core Cardboard development concepts:

- Creating a stereo view rendered in OpenGL
- Using gaze tracking to select objects
- Using the magnet switch for input

Setting Up the Environment

Google has done an excellent job documenting the steps for creating your first native Cardboard application. To get going, visit the main Android SDK page at https://developers.google.com/cardboard/android/, where you will find links and instructions for downloading the tools and samples, as well as complete tutorials.

As of this writing, native Cardboard development is best done using *Android Studio*, a new integrated development environment (IDE), dubbed "The official Android IDE." Over time, Android Studio is supplanting Eclipse as the main visual development tool for Android. (You may recall that Eclipse is the development tool currently

being used to create Gear VR applications, as described in Chapter 4.) Figure 6-9 shows a screenshot of Android Studio.

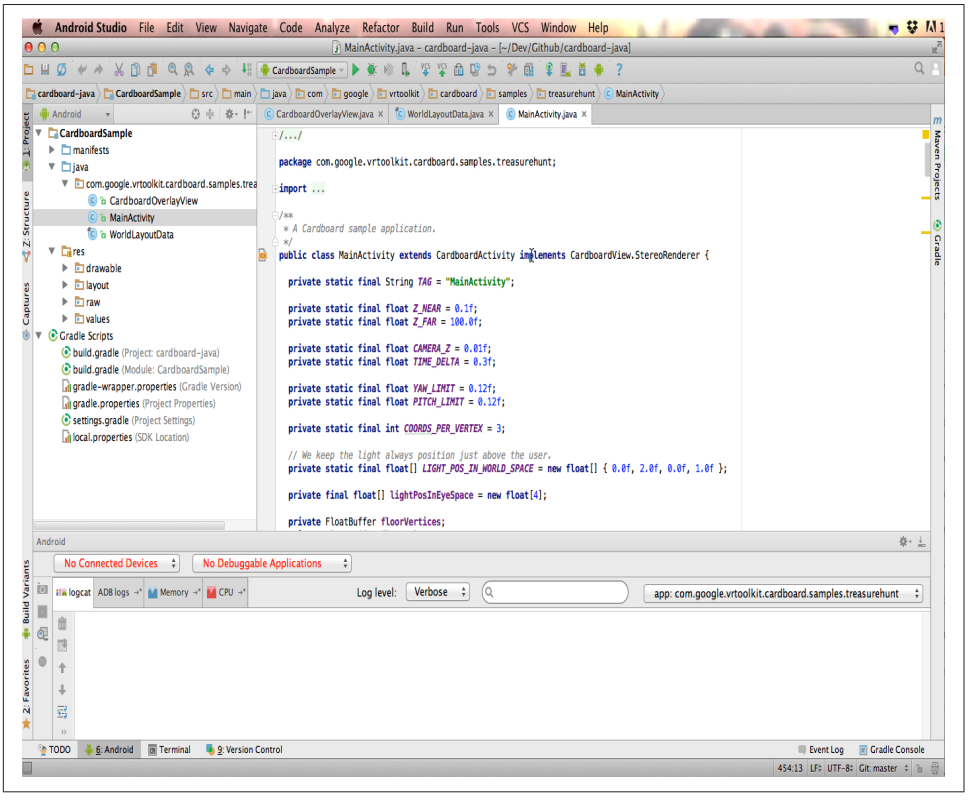


Figure 6-9. Android Studio, Version 1.2

While Android Studio is much improved over Eclipse, it has similar issues in that setup and configuration can be frustratingly hard. Make sure to carefully follow the instructions at the official site, https://developer.android.com/sdk/index.html and refer to various online documentation sources on how to set up emulators for testing and how to build for devices.

Walking Through the Code

The **Getting Started** section of the Android SDK documentation has a complete walk-through of the "Treasure Hunt" sample app:

https://developers.google.com/cardboard/android/get-started

Let's take a look at some of the more interesting parts here.

Creating and Rendering a Stereo View

The first thing we need to do is create a main application view, in this case a full-screen Android view that we will use to render the 3D scene in stereo. We do this by creating a new Android *activity* that represents the application. The source code for this activity, implemented in the Java class `MainActivity`, resides in the file MainActivity.java. Let's look at the first line of the class definition:

```
/**
* A Cardboard sample application.
*/
public class MainActivity extends CardboardActivity
    implements CardboardView.StereoRenderer {

... // class code for MainActivity here

}
```

The SDK supplies a `CardboardActivity` class that all Cardboard applications can extend. `CardboardActivity` provides easy integration with Cardboard devices, by exposing events to interact with Cardboard headsets, and handling many of the details commonly required when creating an activity for VR rendering. You don't have to implement this interface in your code, but by doing so you will save yourself a lot of work. In addition to extending `CardboardActivity`, your main application class should implement the interface `CardboardView.StereoRenderer` along with a couple of rendering methods specific to your application, as we will see below.

The first method your class should implement is `onNewFrame()`, which is called by the SDK each time the application is ready to draw the view, that is, for each *frame* (also known as a tick, time slice, or update cycle). Your application should use this method to perform per-frame updates that are not specific to either eye, such as updating objects in the scene and tracking the current head position.

```
/**
* Prepares OpenGL ES before we draw a frame.
*
* @param headTransform The head transformation in the new frame.
*/
@Override
public void onNewFrame(HeadTransform headTransform) {
// Build the Model part of the ModelView matrix.
Matrix.rotateM(modelCube, 0, TIME_DELTA, 0.5f, 0.5f, 1.0f);

// Build the camera matrix and apply it to the ModelView.
Matrix.setLookAtM(camera, 0, 0.0f, 0.0f, CAMERA_Z, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);

headTransform.getHeadView(headView, 0);

checkGLError("onReadyToDraw");
}
```

Our version of this method does two things: 1) it rotates the cube by updating values in the `modelCube` and `camera` matrices associated with it; and 2) it uses the `headTransform` helper class to track the phone's orientation, saving the values into the class member `headView`. `headView` will be used below to do gaze tracking.

Now that we have done our per-frame update, it's time to draw. Because we are doing stereo rendering for Cardboard, we actually draw the scene twice, once from each eye. This is done in the method `onDrawEye()`:

```
public void onDrawEye(Eye eye) {
  GLES20.glClear(GLES20.GL_COLOR_BUFFER_BIT | GLES20.GL_DEPTH_BUFFER_BIT);

  checkGLError("colorParam");

  // Apply the eye transformation to the camera.
  Matrix.multiplyMM(view, 0, eye.getEyeView(), 0, camera, 0);

  // Set the position of the light
  Matrix.multiplyMV(lightPosInEyeSpace, 0, view, 0, LIGHT_POS_IN_WORLD_SPACE, 0);

  // Build the ModelView and ModelViewProjection matrices
  // for calculating cube position and light.
  float[] perspective = eye.getPerspective(Z_NEAR, Z_FAR);
  Matrix.multiplyMM(modelView, 0, view, 0, modelCube, 0);
  Matrix.multiplyMM(modelViewProjection, 0, perspective, 0, modelView, 0);
  drawCube();

  // Set modelView for the floor, so we draw floor in the correct location
  Matrix.multiplyMM(modelView, 0, view, 0, modelFloor, 0);
  Matrix.multiplyMM(modelViewProjection, 0, perspective, 0,
    modelView, 0);
  drawFloor();
}
```

`onDrawEye()` is called by the SDK once for each eye. It takes as an argument an `Eye` object, which contains information on the world space position and perspective projection for each eye. Let's walk through what this method does, step by step:

1. Save the eye information into the matrix view, for use in subsequent matrix calculations.
2. Calculate the scene light's position in view space. This will be used to shade the cube and floor.
3. Calculate the cube's position in view space using the current values of the `modelView` matrix using the view matrix and the `modelCube` matrix (calculated previously in method `onNewFrame()`).
4. Draw the cube by calling the helper method `drawCube()`. That method will draw the cube either normally or highlighted in yellow, depending on whether the user is looking at it. We will cover that below when we discuss gaze tracking.

5. Calculate the floor's position in view space using the current values of the `modelView` matrix using the view matrix and the `modelFloor` matrix.
6. Draw the floor by calling the helper method `drawFloor()`.

The Smart Phone as Reality Engine

Now that we have walked through our rendering code, it's worth taking a step back and reflecting on how far computers have come in a short time. In this sample app - admittedly a simple one - `onDrawEye()` does all this work, calculating matrices, setting up vertex and color buffers, drawing to the OpenGL hardware pipeline...

every frame...

once for each eye...

sixty times per second...

on a phone.

Think about that for a minute.

Selecting Objects Using Gaze Tracking

As noted , `drawCube()` renders the cube either normally or highlighted in yellow if the user is looking at the cube. Let's see how we determine that. The details are implemented in the helper method `isLookingAtObject()`.

```
/**
* Check if user is looking at object by calculating where the object is in eye-space.
*
* @return true if the user is looking at the object.
*/
private boolean isLookingAtObject() {
float[] initVec = { 0, 0, 0, 1.0f };
float[] objPositionVec = new float[4];

// Convert object space to camera space. Use the headView from onNewFrame.
Matrix.multiplyMM(modelView, 0, headView, 0, modelCube, 0);
Matrix.multiplyMV(objPositionVec, 0, modelView, 0, initVec, 0);

float pitch = (float) Math.atan2(objPositionVec[1], -objPositionVec[2]);
float yaw = (float) Math.atan2(objPositionVec[0], -objPositionVec[2]);

return Math.abs(pitch) < PITCH_LIMIT && Math.abs(yaw) < YAW_LIMIT;
}
```

We determine whether the user is looking at the object by figuring out the angle between our eye view vector (i.e. the direction we are looking in) and the location of the center of the object. If the angle is smaller than a certain rotation about the X and

Y axes (also known as *pitch and yaw*), then we are looking at the object. The math in here might seem a bit counter-intuitive, but it is a straightforward and fast way to calculate the intersection, in contrast to intersecting a pick ray with the object. And it is accurate enough for this simple demo. Here's how it works:

1. Calculate the cube's position relative to the camera space, by multiplying the cube's model and view matrices, and then multiplying that result by an origin vector (`initVec`).
2. Calculate the rotation between the cube's origin in camera space, and the origin of the camera itself in camera space, which is, by definition, at the actual origin (`0, 0, 0`).
3. Use trigonometry (`Math.atan2()` function) to find the rotation between the cube's position and the camera's position, in both the X and Y dimensions.
4. If the X and Y rotations are within the defined limits `PITCH_LIMIT` and `YAW_LIMIT`, return true; otherwise, return false.

Detecting Input from the Magnet Switch

Now we just need to figure out when the user presses the magnet switch and "finds the treasure," i.e. is looking at the cube when the switch is pressed. To do this, we can override `CardboardActivity.onCardboardTrigger()` in our application's activity. This method is called by the SDK when the switch is pressed. When this happens in our app we update the score, change the prompt in the overlay, and hide the cube. Actually we don't hide the cube; we move it to another random position out of the user's current view, so that we are ready for the next round. As a flourish, we also vibrate the phone to provide additional user feedback.

```
/**
* Called when the Cardboard trigger is pulled.
*/
@Override
public void onCardboardTrigger() {
Log.i(TAG, "onCardboardTrigger");

if (isLookingAtObject()) {
score++;
overlayView.show3DToast("Found it! Look around for another one.\nScore = " + score);
hideObject();
} else {
overlayView.show3DToast("Look around to find the object!");
}

// Always give user feedback.
```

```
vibrator.vibrate(50);
}
```

Developing with the Cardboard SDK for Unity

Google's Cardboard tools come with support for creating applications in the popular Unity3D game engine. The main page of the *Cardboard SDK for Unity* can be found at https://developers.google.com/cardboard/unity/. It contains a Unity3D version of the Treasure Hunt example (see the previous section), and a rich developer guide and programming reference. Let's walk through building Treasure Hunt for Unity.

Setting up the SDK

To get set up, follow the instructions at https://developers.google.com/cardboard/unity/download. We can summarize the steps here:

1. First, make sure you have a recent version of Unity3D. To build the examples in the book, I used Unity5 Pro. You can always find the latest at http://unity3d.com/get-unity/download. If you're not familiar with Unity, or are feeling a bit rusty, refer to the extensive information in Chapter 3.
2. Next, get the Cardboard SDK for Unity. This is available as a direct download, or in source code form at Google's Github repository at https://github.com/google-samples/cardboard-unity.
3. If you don't already have it, install the Android SDK. This is the SDK for general Android development, not just VR. The Unity packages require it. If you do have the Android SDK, make sure to update it. The main page for the Android SDK is http://developer.android.com/sdk/index.html.

Now, you are ready to start using the Cardboard SDK for Unity. The SDK is a Unity package that you import into your project. So, we start by creating an empty Unity project. Let's do that... call it *UnityCardboardTest* so that we're all on the same page. Now that the project has been created, we will import the SDK into it and build the Treasure Hunt sample.

Figure 6-10 shows a screenshot of the package importing process. To import the Cardboard SDK package into the new project, follow these steps:

1. Find the **Assets** pane of the **Project** tab in the Unity IDE. **Choose Assets -> Import Package ->Custom Package...**

2. You should see a file dialog box. Use it to navigate to the location of the downloaded Unity SDK.
3. Select the file *CardboardSDKForUnity.unitypackage*. If you downloaded the file by cloning the Github repo, you will find this file in the root directory.
4. Once you have clicked **Open**, Unity will scan the file and present you with a list of package contents to import. For now, let's just bring them all into the project: make sure that all of the objects in the list are checked, and click the **Import** button. You will now see assets present in the **Assets** pane, where there were none before.

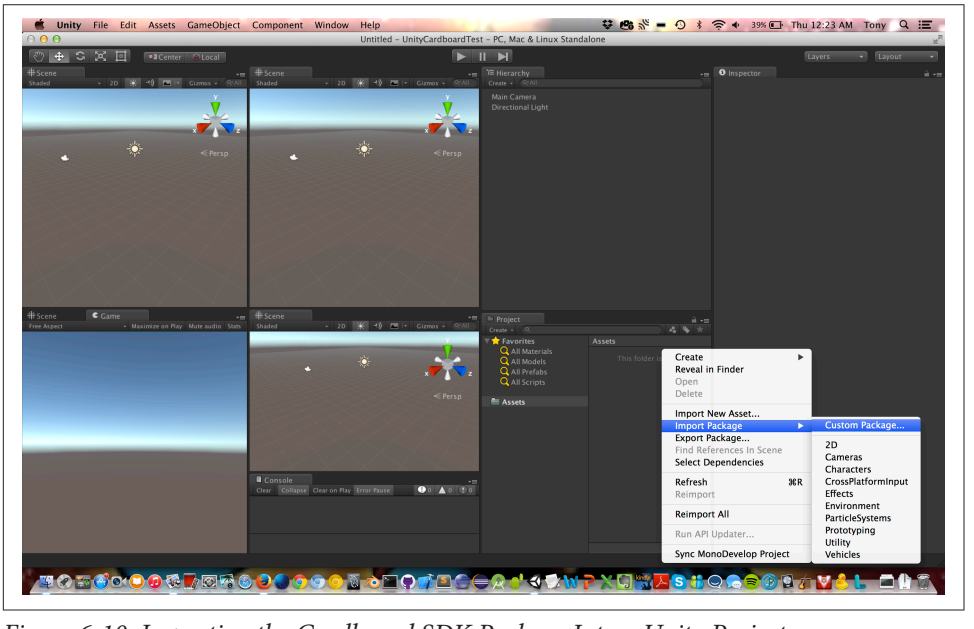


Figure 6-10. Importing the Cardboard SDK Package Into a Unity Project

You're good to go! You can now build Cardboard VR applications using Unity.

Building *Treasure Hunt* for Unity

Once you have imported the Cardboard SDK into your project, it's really easy to build the Treasure Hunt demo. The demo comes packaged with the SDK, so it is already in your project, and in just a few steps you can have it running on your phone. The demo is pictured in Figure 6-11. Note that the Unity version has a nice touch: a dynamic particle system floating around the cube.

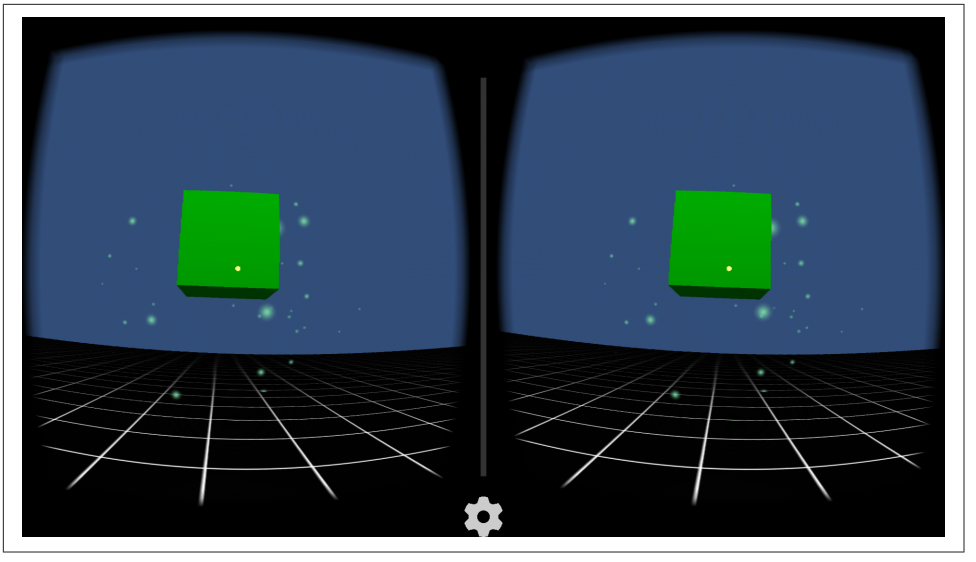


Figure 6-11. The Unity3D Version of Treasure Hunt

To get the assets for *Treasure Hunt* built into the new project, select the folder **Assets->Cardboard->DemoScene**. In the detail pane you will see an icon for a Unity scene named **DemoScene**; double-click that icon. You should now see the scene in the main editor views. You can hit the Play button at the top of the Unity window to get a preview on your computer.

To build the app for your phone, you need to change a couple of settings. First, open the Build Settings dialog by selecting **File->Build Settings...** from the main menu. You will see a dialog that resembles the screen shot in Figure 6-12.

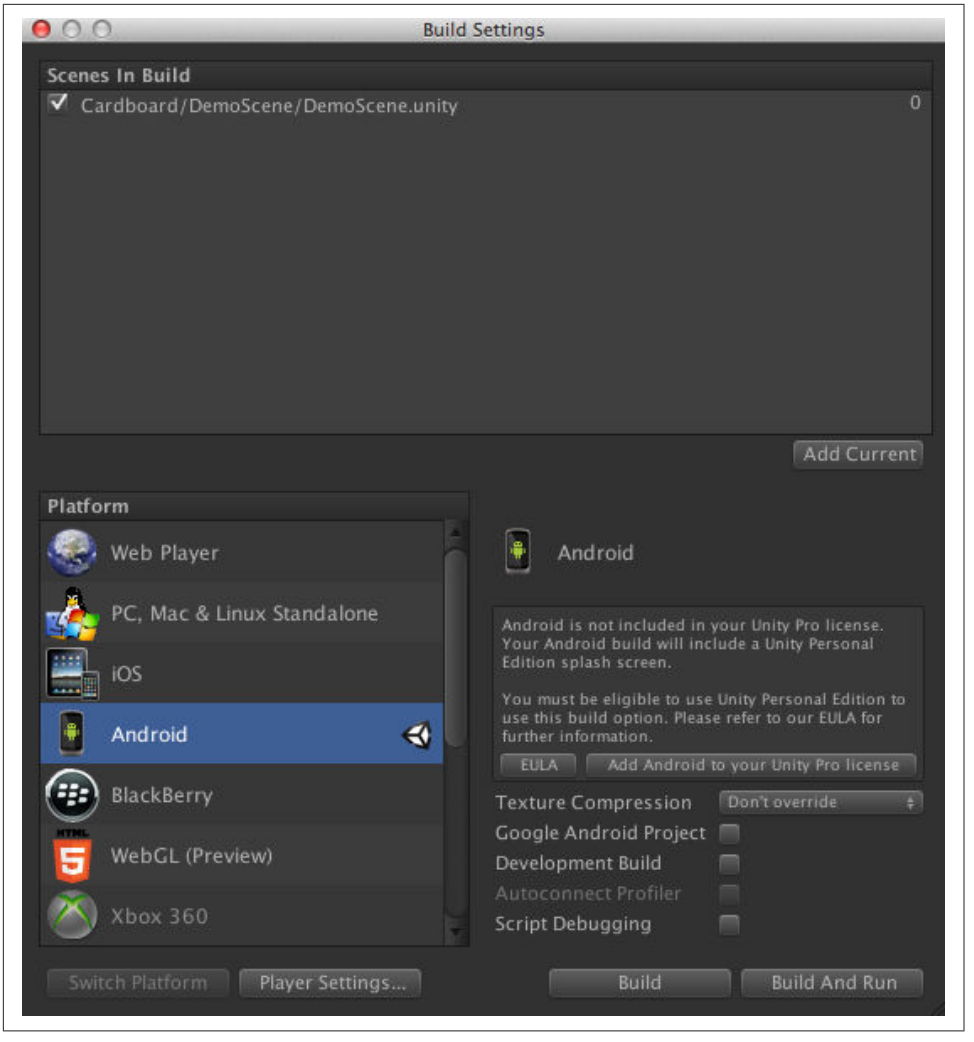


Figure 6-12. Build Settings for the Treasure Hunt Android App

Now perform the following steps:

1. Select Android as your platform. Click on the **Android** item in the list and then click the **Switch Platform** button.
2. Add the demo scene to your build. The Scenes in Build list at the top will be empty to start; you need to add the current scene. You should already be viewing it in the Editor view, so if you click the **Add Current** button, it should be added to the list, and you will see a checked item named *Cardboard/DemoScene/DemoScene.unity*.

3. Click the **Player Settings...** button on the bottom left. This will open the Inspector to the Player Settings panel. (You should see it on the far right of the Unity interface.) Find the setting named Bundle Identifier, and change its value to a reasonable Android package name. Make sure to change it from the default value, or Unity will complain during the build step. I used the value *vr.cardboard.UnityCardboardTest*.
4. Finally, back to the Build Settings dialog. Click the **Build and Run** button. You will be prompted for a file name under which to save the .apk (Android package) file. I chose the name *UnityCardboardTest.apk*. That's the name that will show up on the phone in your list of apps. Hit **Save** to save the .apk.

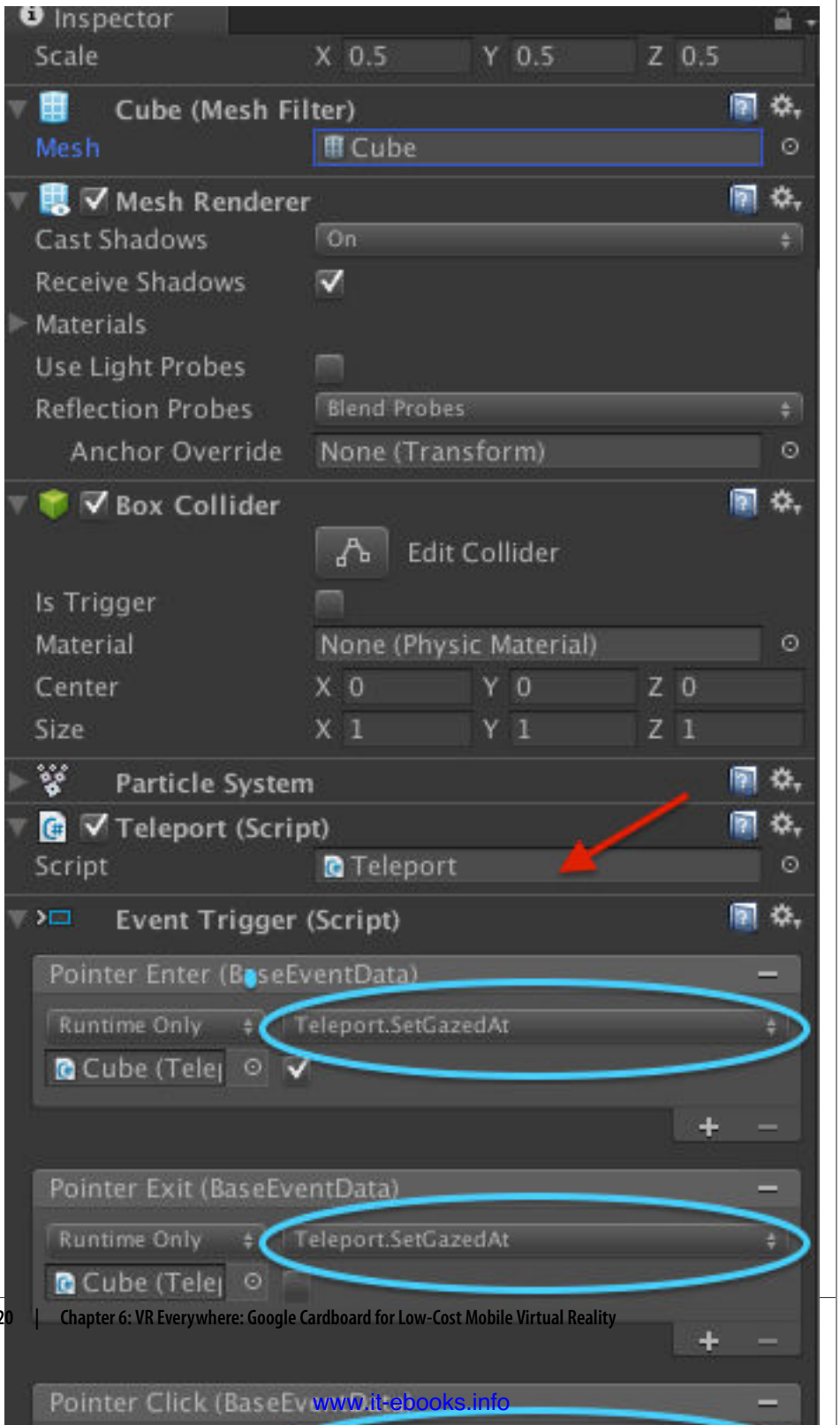
The app should now launch. Pop it into your Cardboard viewer and play!

A Walk-Through of the Unity Code

With the Cardboard SDK in place, it's surprisingly easy to build a working Cardboard application using Unity. Most of the work takes place in the Unity editor: construct the 3D scene using the visual editor, drop in the Prefabs from the Cardboard SDK, and implement a little bit of logic.

For the Treasure Hunt demo, we simply need to detect the gaze/tap combination and move the cube to a new location. Let's see how this is put together. First stop is the Unity editor:

- In the editor, select the cube. You can do this either by clicking directly on the cube in one of the Scene views, or click on the name of the object ("Cube") in the Hierarchy pane.
- Now, go to the Inspector pane. You should see several properties, grouped into sub-panes. Scroll to the bottom and you will see three Event Trigger properties: one each for Pointer Enter and Exit, and one for Pointer Click. See the blue outlined areas in Figure 6-13.



The Cardboard SDK comes with a module called `GazeInputModule`, which manages when the camera is looking at objects. When a new object is gazed at, this module converts that into a standard Unity pointer event, that is, treating it as if the mouse rolled over the object on a desktop computer. `GazeInputModule` also handles magnet switch and touch screen events such that, if those input actions take place, they are delivered to the currently gazed-at object and treated as if they were mouse clicks. By supplying the values `Teleport.SetGazedAt` and `Teleport.TeleportRandomly` in the property sheet, we are telling the Unity runtime to call those named methods in response to the gaze events. With this simple wiring in place, all that's left to do is actually handle the actions in a script.

Note the property value annotated with the red arrow in Figure 6-13 above. This is the name of the script that implements the event triggers, in our case a script class named `Teleport`. Double-click on this value and *MonoDevelop*, Unity's C# programmer's editor, will open the file Teleport.cs from the project sources. The entirety of the class implementation is in the following code listing.

```
// Copyright 2014 Google Inc. All rights reserved.
...

using UnityEngine;
using System.Collections;

[RequireComponent(typeof(Collider))]
public class Teleport : MonoBehaviour {
  private Vector3 startingPosition;

  void Start() {
    startingPosition = transform.localPosition;
    SetGazedAt(false);
  }

  public void SetGazedAt(bool gazedAt) {
    GetComponent<Renderer>().material.color = gazedAt ? Color.green : Color.red;
  }

  public void Reset() {
    transform.localPosition = startingPosition;
  }

  public void TeleportRandomly() {
    Vector3 direction = Random.onUnitSphere;
    direction.y = Mathf.Clamp(direction.y, 0.5f, 1f);
    float distance = 2 * Random.value + 1.5f;
    transform.localPosition = direction * distance;
  }
}
```

`Teleport` is defined as a subclass of `MonoBehaviour`, which is the base class for all scripts. The script only does two important things:

1. Method `SetGazedAt()` highlights the cube green if it is being gazed at, otherwise it sets the color to red. Visual objects in Unity contain a Renderer component; we simply set the color of its material to either green or red.
2. Method `TeleportRandomly()` uses the Unity built-in class Random to generate a random direction vector on a unit sphere and move the cube a random distance along that vector.

And that's actually it. Unity is such a powerful, professional 3D authoring system that putting together applications like this is almost trivial-- once you master the intricacies of the interface. Combining the Cardboard SDK with Unity, we were able to put together a basic Cardboard demo in no time at all.

Developing Cardboard Applications Using HTML5 and a Mobile Browser

And now for something completely different. If you worked through the Unity Cardboard project in the previous section, you saw how simple putting together an app can be using pro tools. Well, now we are going to survey the Wild West of developing Cardboard applications using a mobile browser. There are few tools to speak of; we'll have to cobble our code together from open source libraries. But on the plus side, we will have a no-download Cardboard VR application that you can launch just by opening a link in your web browser!

Developing Cardboard web applications is a little different from the WebVR APIs we looked at in Chapter 5; however there are also many similarities. We code in HTML5 and JavaScript and render using WebGL, presumably with a library like Three.js. Figure 6-14 shows a Cardboard version of the simple WebVR example we coded in Chapter 5.

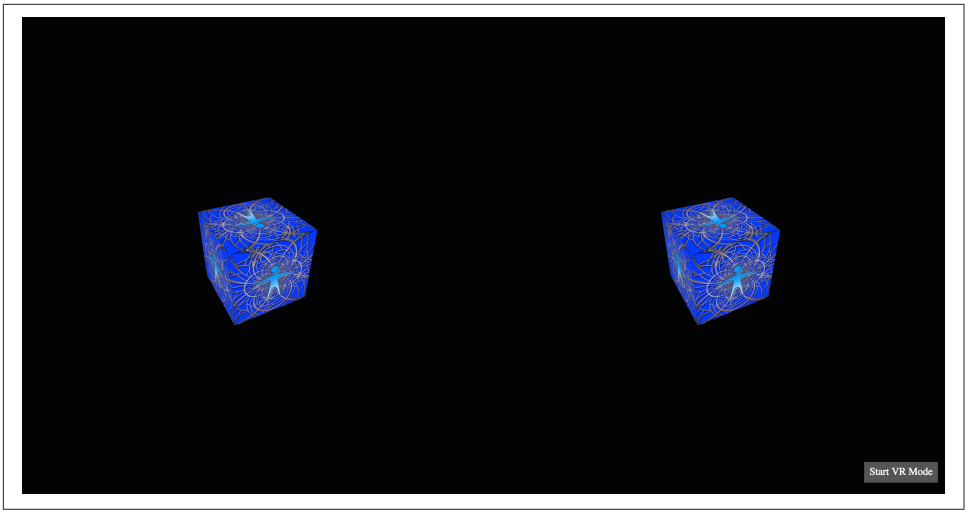


Figure 6-14. WebVR Sample Application for Cardboard

You can run this app live by visiting the URL http://tparisi.github.io/WebVR/examples/cube-cardboard.html in your mobile browser. Once the page opens, you will see a rotating cube with the WebVR logo, rendered in Cardboard stereo. Tap the **Start VR Mode** button to put the page into fullscreen mode. Then, pop the phone into your Cardboard viewer, and you can look around the scene. You should see the cube move.

Setting up the WebVR Project

To look through the code for this example, please clone the repository from Github at

https://github.com/tparisi/WebVR

If you also want to preview this file and/or make changes to it, you will need to run it from a local web server. Chapter 5 contains instructions for how to set one up simply, if you're not sure how to do that.

The JavaScript Cardboard Code

Let's walk through the JavaScript code for this example. The source code is in the file /examples/cube-cardboard.html. The application is written using the Three.js library for WebGL development. (If you're not familiar with Three.js, refer to the information on it in Chapter 5.)

First, we have the main application, defined in a jQuery `ready()` callback function that is called when the page is loaded and ready.

```
$(document).ready(
        function() {

            // Set up Three.js
            initThreeJS();

            // Set up VR rendering
            initVREffect();

            // Create the scene content
            initScene();

            // Set up VR camera controls
            initVRControls();

            // Run the run loop
            run();
        }
);
```

The `ready()` callback function calls several helper functions, also defined in this file. Let's go through each one. First, we set up stereo rendering and fullscreen mode:

```
function initVREffect() {
    // Set up Cardboard renderer
    effect = new THREE.StereoEffect(renderer);
    effect.setSize(window.innerWidth, window.innerHeight);

    // StereoEffect's default separation is in cm, we're in M
    // Actual cardboard eye separation is 2.5in
    // Finally, separation is per-eye so divide by 2
    effect.separation = 2.5 * 0.0254 / 2;

    // Set up fullscreen mode handling
    var fullScreenButton = document.querySelector( '.button' );
    fullScreenButton.onclick = function() {
        if ( container.mozRequestFullScreen ) {
            container.mozRequestFullScreen();
        } else {
            container.webkitRequestFullscreen();
        }
    };
}
```

In `initVREffect()`, we set up Cardboard VR stereo rendering using the `THREE.StereoEffect` class. This class was designed to render a scene in WebGL twice, side-by-side, using Cardboard's recommended camera values with a 90 degree field of view and typical inter-pupillary offsets for the two cameras. In addition to setting up the renderer, we also have the code to handle going into full screen mode when the button is pressed. This uses standard web browser methods for entering fullscreen mode

which, historically, use browser-specific prefixes depending on whether we are using mobile Firefox (`mozRequestFullScreen()`) or a webkit-based browser such as mobile Chrome or Safari (`webkitRequestFullScreen()`).

Next, we initialize the scene. This uses various Three.js objects to create a scene with a camera and a textured cube.

```
function initScene() {
        // Create a new Three.js scene
        scene = new THREE.Scene();

        // Add  a camera so we can view the scene
        camera = new THREE.PerspectiveCamera( 90, window.innerWidth / window.innerHeight, 1, 4
        scene.add(camera);

        // Create a texture-mapped cube and add it to the scene
        // First, create the texture map
        var mapUrl = "../images/webvr-logo-512.jpeg";
        var map = THREE.ImageUtils.loadTexture(mapUrl);

        // Now, create a Basic material; pass in the map
        var material = new THREE.MeshBasicMaterial({ map: map });

        // Create the cube geometry
        var geometry = new THREE.BoxGeometry(2, 2, 2);

        // And put the geometry and material together into a mesh
        cube = new THREE.Mesh(geometry, material);

        // Move the mesh back from the camera and tilt it toward the viewer
        cube.position.z = -6;
        cube.rotation.x = Math.PI / 5;
        cube.rotation.y = Math.PI / 5;

        // Finally, add the mesh to our scene
        scene.add( cube );

}
```

Now it's time to set up the camera controller. This object updates the camera's orientation to match the phone's orientation, based on IMU input. HTML5 mobile browsers come with a *device orientation API* that delivers IMU changes to the application in the form of `orientationchange` DOM events; `THREE.DeviceOrientationControls` handles those events and automatically updates the camera we passed in to its constructor whenever it detects a change.

```
function initVRControls() {
    // Set up VR camera controls
    controls = new THREE.DeviceOrientationControls(camera);
}
```

Our application is fully set up; now we need to run it. We do this by creating a *run loop*, a function that drives continuous animation and rendering of 3D scenes. The run loop is driven by a built-in browser function, `requestAnimationFrame()`, which calls our application every time the browser is ready to re-render the page. The listing below shows the run loop for this example.

```
var duration = 10000; // ms
var currentTime = Date.now();
function animate() {

    var now = Date.now();
    var deltat = now - currentTime;
    currentTime = now;
    var fract = deltat / duration;
    var angle = Math.PI * 2 * fract;
    cube.rotation.y += angle;
}

function run() {
    requestAnimationFrame(function() { run(); });

        // Render the scene
        effect.render( scene, camera );

        // Update the VR camera controls
        controls.update();

        // Spin the cube for next frame
        animate();

}
```

Chapter 5 contains a lengthy walk through of the run loop for the Oculus Rift version of this same example, so we won't go into all the details here. But the broad strokes are that, each time through the run loop, the application renders the scene, updates the camera based on device orientation, and animates the cube.

Chapter Summary

This chapter explored the basics of developing for Cardboard VR, an affordable way to experience virtual reality using an existing mobile phone and an inexpensive drop-in headset. Cardboard uses simple stereo rendering and the phone's built-in accelerometer to deliver modest but fun VR at a fraction of the cost of the high-end headsets on the market today.

Cardboard development can take on many forms. We can program in Java using Android Studio and Google's native Android SDK; we can build Unity3D applications in C# using the Unity Editor and an SDK developed by Google; or we can create no-download, mobile browser-based Cardboard VR using JavaScript and HTML5. Which environment is right for you depends on a combination of your needs and preferences; regardless, coding is straightforward in all of them. For developers as well as end users, Cardboard is a lightweight, low-commitment, way to create fun VR now at low cost.