



Community Experience Distilled

Mastering Leap Motion

Design robust and responsive Leap Motion applications for real-world use

Foreword by Dr. Woodie Flowers, Pappalardo Professor Emeritus MIT, Distinguished Advisor, FIRST

Brandon Sanders

[PACKT]
PUBLISHING

Mastering Leap Motion

Table of Contents

[Mastering Leap Motion](#)

[Credits](#)

[Foreword](#)

[About the Author](#)

[Acknowledgments](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers, and more](#)

[Why subscribe?](#)

[Free access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Introduction to the World of Leap Motion](#)

[Setting up the Leap Motion device](#)

[Installing the Leap Motion Developers' SDK](#)

[Installing the Java JDK](#)

[Setting up your IDE](#)

[Structure of the Leap Motion Application Programming Interface \(API\)](#)

[The Vector class](#)

[The Finger class](#)

[The Hand class](#)

[The Frame class](#)

[The Controller class](#)

[The Listener class](#)

[Creating a simple framework program within the Eclipse IDE](#)

[Setting up the project](#)

[Let's write some code!](#)

[Trying it out](#)

[Looking forward – the Skeletal Tracking API](#)

[Different fingers? Not a problem](#)

[Handedness is no longer an issue](#)

[Having confidence in tracking data](#)

[Pinching and grabbing are now much easier](#)

[A new API class – Bones](#)

[That's it!](#)

[Summary](#)

[2. What the Leap Sees – Dealing with Fingers, Hands, Tools, and Gestures](#)

[Handling hands and fingers](#)

[The Leap's field of view](#)

[The InteractionBox class](#)

[How the interaction box works](#)

[Why would you ever want to use something like the interaction box?](#)

[Detecting gestures and tools](#)

[Detecting and using tools](#)

[Gestures](#)

[Detecting gestures](#)

[Some \(albeit minor\) limitations to keep in mind](#)

[Upside-down hands can be a problem!](#)

[Needing too many hands is a bad thing](#)

[Differentiating fingers can be fun!](#)

[Lack of support for custom gestures](#)

[Summary](#)

3. What the User Sees – User Experience, Ergonomics, and Fatigue

When to use the Leap (and more importantly, when not to)

The Leap Motion user experience guidelines

Ergonomics and user fatigue

Ergonomics

User fatigue

A case study – the Artemis Quadrotor Simulator

Play testing and why you should do it

Providing as much visual feedback as possible

That's it – for now!

Summary

4. Creating a 2D Painting Application

Laying out the framework for Leapaint

LeapButton.java

LeapaintListener.java

Leapaint.java

Creating the graphical frontend

Making a responsive button – the LeapButton class

Getting our bounds

Visually responding to the user

Making a graphical user interface

Constructing a constructor

Saving images

Interpreting Leap data to render on the graphical frontend

Testing it out

Improving the application

Summary

5. Creating a 3D Application – a Crash Course in Unity 3D

A brief introduction to Unity

Installing and setting up Unity 3D

Common jargon found in Unity

[Scenes](#)

[GameObjects](#)

[Scripts](#)

[Creating a project](#)

[Setting the scene](#)

[Summary](#)

[6. Creating a 3D Application – Integrating the Leap Motion Device with a 3D Toolkit](#)

[Setting up the scene to receive Leap Motion input](#)

[A quick summary – the fundamentals of Unity scripts](#)

[Attaching a script to a GameObject](#)

[Laying out a framework of scripts](#)

[Rendering hands](#)

[LeapListener.cs](#)

[HandRenderer.cs](#)

[Preparing the scene for hand rendering](#)

[Testing out the Hand Renderer](#)

[Rendering buttons and detecting button presses](#)

[BaseSingleton – a custom singleton pattern](#)

[Colorscheme – a utility class to keep track of colors](#)

[Core – the main class, if Unity had main classes](#)

[TouchPointer – let’s draw some cursors on the screen](#)

[TouchableButton – surely, the name is self-explanatory](#)

[TitleMenu – a simple main menu](#)

[Putting it all together](#)

[Summary](#)

[7. Creating a 3D Application – Controlling a Flying Entity](#)

[Creating the flying entity](#)

[Adding the PlayerArrow and Rigidbody components](#)

[Retrieving user input with the HandController class](#)

[Interpreting user input with the Player class](#)

[Putting everything together and testing it](#)

[Improving the application](#)

[Summary](#)

[8. Troubleshooting, Debugging, and Optimization](#)

[Making sure your Leap is connected](#)

[The Diagnostic Visualizer](#)

[Keeping the Leap Motion SDK updated](#)

[Cutting back on Leap Motion API calls](#)

[Handling the NoSuchMethod and NoClassDefFound errors in Java](#)

[Custom calibration of the Leap Motion Controller](#)

[Summary](#)

[9. Going beyond the Leap Motion Controller](#)

[What you've learned so far](#)

[The Leap Motion Controller standing next to other emerging technologies](#)

[Microsoft's Kinect](#)

[Oculus VR's Oculus Rift](#)

[Reliability and safety concerns with the Leap in industrial settings](#)

[Going beyond – ideas to control hardware and robots with the Leap Motion Controller](#)

[Arduino](#)

[A few things you'll need](#)

[Setting up the environment](#)

[Setting up the project](#)

[Writing the Java side of things](#)

[Writing the Arduino side of things](#)

[Deploying and testing the application](#)

[Ideas for Leap-driven applications – simulators and robots](#)

[FIRST Robotics Competition Robots](#)

[The FIRST Robotics Competition](#)

[Controlling an FRC robot with the Leap Motion Controller](#)

[Making a robot of your own!](#)

[Summary](#)

[Index](#)

Mastering Leap Motion

Mastering Leap Motion

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2014

Production reference: 1151114

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78355-139-2

www.packtpub.com

Credits

Author

Brandon Sanders

Reviewers

Rudi Chen

Lamtharn Hantrakul

Justin Kuzma

Maria Montenegro

Commissioning Editor

Usha Iyer

Acquisition Editor

Richard Harvey

Content Development Editor

Shaon Basu

Melita Lobo

Technical Editor

Edwin Moses

Copy Editors

Dipti Kapadia

Deepa Nambiar

Project Coordinator

Sanchita Mandal

Proofreaders

Paul Hindle

Sandra Hopper

Jonathan Todd

Indexers

Monica Ajmera Mehta

Rekha Nair

Graphics

Abhinash Sahu

Production Coordinator

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

Foreword

This book is part of Packt's Mastering series. The author assumes that you have some programming background.

A decade ago, *Mastering Leap Motion* would not likely have been published. It is part of a wave of crowdsourced, informal, and targeted publications. eBooks are changing things. Innovation and creative projects are sprouting everywhere in part because digital technology has lowered the barriers to the dissemination of knowledge and information. Grade-school kids are creating smartphone apps. 3D printing has arrived in schools and home workshops.

Gone is the time when one would be considered a professional and/or contributor because they knew certain things. Now, you must do something with knowledge to be in the winner's circle. Real celebrity will go to those who understand both the technology and the creative process. With this book, you can make technology dance in the celebration of creativity.

If you listen to the author's backstory, you will be inspired to use your newfound understanding in creative ways. Brandon Sanders has spent several years working with others in the FIRST (For Inspiration and Recognition of Science and Technology) programs. FIRST gives young people complex and difficult problems related to robotics to deal with. Good FIRST teams create delightful robots that bristle with innovation. Brandon discusses some of those machines at the end of [Chapter 9](#), *Going beyond the Leap Motion Controller*.

I will encourage you to follow the author through a conversation about how to use Leap Motion Controllers. Then, have your own conversation with an innovative use of Leap Motion. Do something delightful! Do something that will make others smile.

Dr. Woodie Flowers

Pappalardo Professor Emeritus MIT

Distinguished Advisor, FIRST

About the Author

Brandon Sanders is an 18-year-old roboticist who spends much of his time designing, building, and programming new and innovative systems, including simulators, autonomous coffee makers, and robots for competition. At present, he attends Gilbert Finn Polytechnic (which is a homeschool) as he prepares for college. He is the founder and owner of Mechakana Systems, a website and company devoted to robotic systems and solutions.

As a home-educated student, he's had the unique opportunity to focus his efforts on the fields that interest him. This has made him successful as the team captain for the FIRST Robotics teams: #4982 Café Bot and #1444 the Lightning Lancers. He has also served as a scientific research assistant to the Chairman of the Washington University Physics Department, where he wrote software to aid in the calculation of equations of state for dense matter in neutron stars.

He has received numerous awards and accolades as a result of his involvement in various programs. Two of his most notable achievements are FIRST Robotics Competition Dean's List Award and FIRST Tech Challenge World Championship Inspire Award.

Acknowledgments

First and foremost, I would like to thank Dr. Woodie Flowers for not only being an inspiration to me and my peers but also for graciously taking out the time to write the foreword for this title.

In addition, I would also like to thank all my friends and family who helped double-check my work during the lengthy process of writing this title, including Dr. Anne Jensen-Urstad, Ethan Michaelicek, Jonas Kersulis, and my parents, Kim and Robert Sanders.

Finally, I wish to thank my editors and reviewers, Richard Harvey, Shaon Basu, Melita Lobo, Edwin Moses, Rudi Chen, Justin Kuzma, Lamtharn Hantrakul, and Maria Montenegro for their continued commitment throughout the duration of writing this title.

About the Reviewers

Rudi Chen is a software developer from the University of Waterloo and has worked for companies such as Side Effects Software and Dropbox.

Lamtharn Hantrakul is an international student from Thailand who is double majoring in Applied Physics and Music at Yale University. His research interests include instrument acoustics, signal processing, and musical HCI. He has published and presented his work, which combines Leap Motion and musical HCI, at Institut de Recherche et Coordination Acoustique/Musique (IRCAM) and at the International Computer Music Conference (ICMC). He enjoys composing music, playing jazz piano, building music controllers, and learning about traditional Thai instruments such as *Saw-U* and *Saloh*. He speaks Thai, English, French, and Chinese, and outside of Music and Physics, likes to read and write modern nonfiction essays. His projects, compositions, writing, and research can be found on his website at <http://lh-hantrakul.com/>.

Justin Kuzma is a freelance engineer and software developer based in Burlington, VT. He has experience in creating mechanical designs and digital art installations in addition to iOS app development. Using Leap Motion, he has created intuitive interfaces that spark the imagination as they blur the lines between the digital and the physical worlds.

Maria Montenegro is a computer scientist and an electronic media artist. Currently, she is pursuing a Master's degree in Entertainment Technology from the Entertainment Technology Center (ETC) at Carnegie Mellon University. She is passionate about developing new ways of entertainment with the use of new technology to promote and enhance learning. She believes that interactive storytelling and interactive installations can have a huge impact on people worldwide, showing them different perspectives of things.

As a computer scientist, she focuses more on computer graphics, artificial intelligence, and computer vision to exploit the most of the technology in use. She likes pushing the limits of a new technology to bring completely new experiences to users by knowing and understanding its limitations.

For more information, visit <http://www.fusion-sky.com/>.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

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

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Preface

The Leap Motion Controller is a revolutionary system that blends the boundary between man and machine, or at least, between our hands and monitors. What if you wanted to literally grab a 3D model with your hands and manipulate it however you want? This is where the Leap Motion device and its prospective developers (you!) come in. After using the powerful Leap API and with some innovation and a lot of patience, you will be aware how this new device has the potential to revolutionize the way we work with our computers.

As you are no doubt already familiar with the Leap device itself, we'll keep the introduction brief. As you've probably gathered from the title, this book is devoted to mastering the process of designing, writing, and testing programs for the Leap Motion Controller. It will teach you how to develop with the Leap Motion device effectively while creating a polished user experience. Throughout the book, we'll cover a broad range of topics ranging from the API basics to user experience all the way to robotics integration. Yes, robotics integration. As my primary field of study (and subsequently, one of my forms of recreation) is robotics, I've integrated a bunch of different robots with the Leap device, ranging from simplistic quadrotor simulators all the way to competition robots that cost thousands of dollars. I'll be sharing a few of these different projects with you toward the end of the book!

Once you start getting a hang of things, we'll create both 2D and 3D applications to put all of this knowledge to work. Of course, we'll also cover the troubleshooting and debugging of programs. After all, nothing is worse than an app that's broken because of no obvious reason, right?

This book emphasizes user experience, which is the most important thing when using the Leap Motion Controller. You might not think about it at first, but something as simple as user fatigue is a big problem when programming for the Leap. You will need to make sure that your user can use the application, but at the same time, you need to make sure that their hands don't get tired from a series of repetitive gestures or motions. We will cover problems such as this one as we progress through this book.

In keeping with the natural user interface that the Controller offers, a good Leap application should be simple, intuitive, and easy to use. It's okay if the underlying software is complex, but the user interface and its controls should be obvious. For example, instead of using a fancy U-shaped gesture to perform a simple task like an undo operation, why not just have the user make a quick swipe to the left with one of their hands? A good developer should always be on the lookout for opportunities to simplify their interface without making it unusable!

What this book covers

[Chapter 1](#), *Introduction to the World of Leap Motion*, shows you how to set up and test the Leap Motion device and a programming environment to use with it. Once everything is set up, we'll review the API briefly and finish the chapter off with a simple example program to make sure everything's working.

[Chapter 2](#), *What the Leap Sees – Dealing with Fingers, Hands, Tools, and Gestures*, covers the software and hardware side of any Leap Motion application. This includes basic tracking data such as hands and fingers as well as more advanced features such as tools and gestures. We'll finish off with an overview of some of the limitations that you might run into when working with the API and the device.

[Chapter 3](#), *What the User Sees – User Experience, Ergonomics, and Fatigue*, covers the user side of any Leap Motion application. This includes when and when not to make use of the Leap in an application, the importance of ergonomics, and the prevention of user fatigue.

[Chapter 4](#), *Creating a 2D Painting Application*, walks you through the creation of a two-dimensional (or 2D) painting application for the Leap. We'll start out simple with the basic framework and graphical frontend and then move straight into rendering user input onto the screen.

[Chapter 5](#), *Creating a 3D Application – a Crash Course in Unity 3D*, introduces you to a three-dimensional (or 3D) toolkit (Unity 3D) to prepare you for the next few chapters. We'll cover the basic installation and setup of the environment, which is followed by the creation of a blank template project for use in the next few chapters.

[Chapter 6](#), *Creating a 3D Application – Integrating the Leap Motion Device with a 3D Toolkit*, walks you through the basic steps of integrating the Leap Motion device with an external 3D toolkit. You'll learn how to render hands, fingers, and buttons. We'll finish off by covering the detection of user input via the Leap. We will be using C# in this chapter instead of Java.

[Chapter 7](#), *Creating a 3D Application – Controlling a Flying Entity*, guides you through the completion of our 3D application. We'll create a 3D entity, retrieve user input from the Leap, and then use that data to control the entity.

[Chapter 8](#), *Troubleshooting, Debugging, and Optimization*, is devoted to the inevitable things that will arise during application development: bugs and problems and optimization. This chapter will go over a few different things you can use to fix common problems with your device or application, in addition to a few general best practices.

[Chapter 9](#), *Going beyond the Leap Motion Controller*, covers a variety of subjects that go beyond the Leap Motion device itself. I'll talk about what you've learned so far, where the Leap Motion stands next to other emerging technologies, some concerns regarding the reliability and safety of the device in the industry, and even some ideas to control robots!

What you need for this book

Before you begin with this book, there are a few things you'll need. These include:

- A Leap Motion Controller
- A computer
- An Internet connection (to download various things such as the Leap SDK)

In addition to these, you should have an understanding of one or more object-oriented programming (OOP) languages. Prior experience with the Unity 3D toolkit is also a good thing to have for the later chapters, but it's not required as we will review it in [Chapter 5, *Creating a 3D Application – a Crash Course in Unity 3D*](#).

In this book, we'll use the Java programming language most of the time. However, we will switch over to C# briefly for [Chapter 5, *Creating a 3D Application – a Crash Course in Unity 3D*](#); [Chapter 6, *Creating a 3D Application – Integrating the Leap Motion Device with a 3D Toolkit*](#); and [Chapter 7, *Creating a 3D Application – Controlling a Flying Entity*](#), when we start working with the Unity 3D toolkit.

Who this book is for

This book is for developers who have some experience with the Leap Motion device and want to turn that experience into mastery of the device.

As this book is intended for more experienced developers, I highly suggest that you have experience working with at least one object-oriented programming (OOP) language before you begin reading; if you don't, it will make the tutorials in this book rather frustrating. On the flip side, even if you don't have a whole lot of experience with the Leap, this book will still provide you with a plethora of information to utilize within your projects as you gain more experience!

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: “The `addlistener` method, on the other hand, is used to register a custom `Listener` class with the device.”

A block of code is set as follows:

```
Finger frontMost = frame.fingers().frontmost();
Vector position = new Vector();
position.setX(frontMost.tipPosition().getX());
position.setY(frontMost.tipPosition().getY());
position.setZ(frontMost.tipPosition().getZ());
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public class MyListener extends Listener
```

Any command-line input or output is written as follows:

```
> java -jar C:\Users\YourPath\SimpleLeap.jar
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: “Following this, navigate to the **Downloads** tab at the top of the Leap Motion website and click on it.”

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

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

Errata

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

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

Piracy

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

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

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

Questions

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

Chapter 1. Introduction to the World of Leap Motion

In this chapter, we will walk through the setup and installation of the various tools that you need to begin working with the Leap Motion Controller. You will learn how to install the Leap Motion Controller, the Developers' SDK and, optionally, an Integrated Development Environment called Eclipse. Afterwards, we'll go through the Leap Motion API in detail, including the API structure, basic terminology, and anatomy of a program. At the end of this chapter, we'll create a simple program that you can use as a stepping stone to the more complex Leap-driven applications that appear later in this book.

Note

Keep in mind that a majority of the code-related jargon used throughout this book is relative to the Java programming language.

Should you come across any issues during this chapter, refer to [Chapter 8, Troubleshooting, Debugging, and Optimization](#), for troubleshooting tips and tricks.

This chapter is sprinkled with periodic *Fun facts* that offer high-level and entry-level factoids about scripting and programming for your reading pleasure.

Setting up the Leap Motion device

If you've already set up your Leap Motion device, you can safely skip this section. Otherwise, read on!

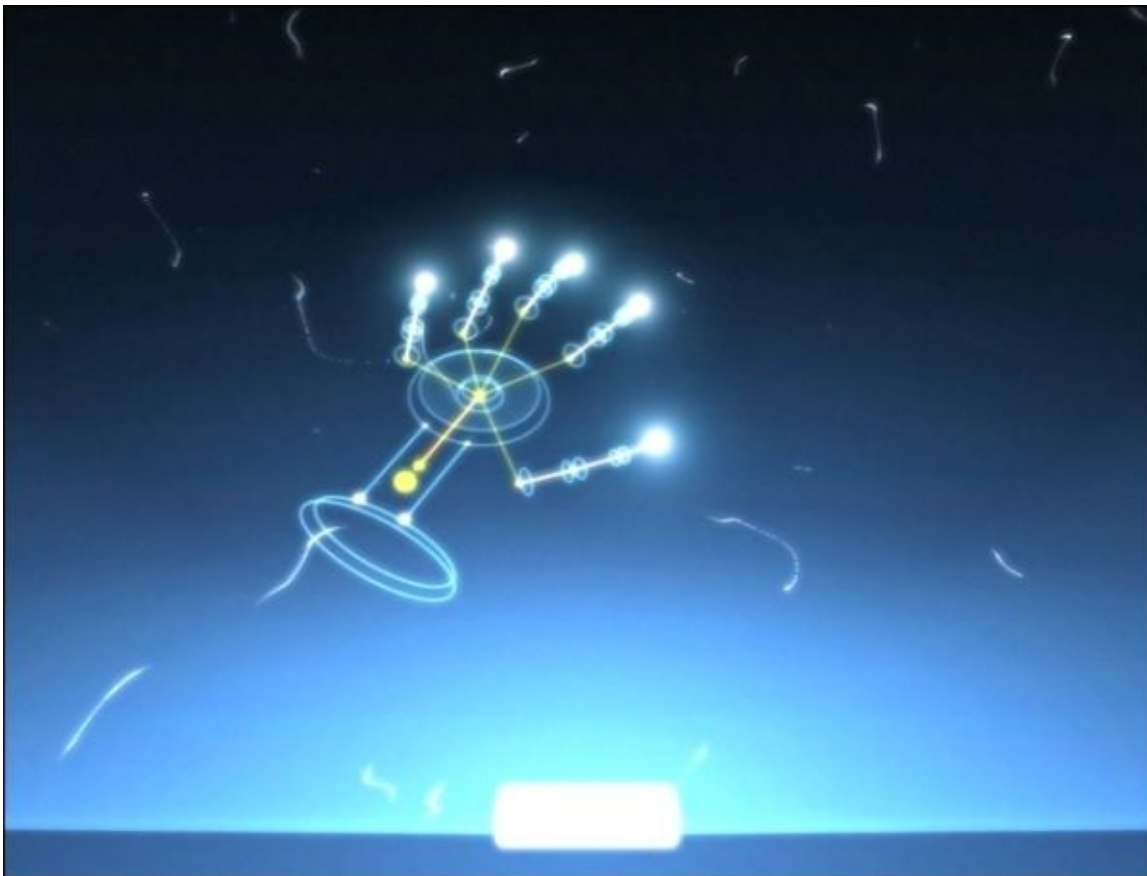
Note

This section (and the rest of the book) will assume that you are on the Windows operating system.

Compared to the early days of Leap Motion development (when `regedit.exe` was far too common), setting up the device is quite easy now. All you need to do is follow these simple steps to get your device up and running:

1. Plug in your device. It may or may not begin installing the firmware; no worries if it doesn't.
2. Following this, head on over to <https://www.leapmotion.com/setup> and download the Leap Motion installer for your platform (Windows in our case).
3. Once the installer has been downloaded, go ahead and run it; this will get your device fully set up.

Once the installer has completed, go to your Start menu in Windows, type in "Leap Motion Visualizer" in the search bar (you can show the search bar in Windows 8/8.1 by pressing Windows Key + Q) and hit *Enter*:



You're all set! You can try out your device by waving your hands in front of the Leap Motion device; wireframe representations of your fingers and hands will be rendered on the screen by the Leap Motion Visualizer, similar to the preceding screenshot.

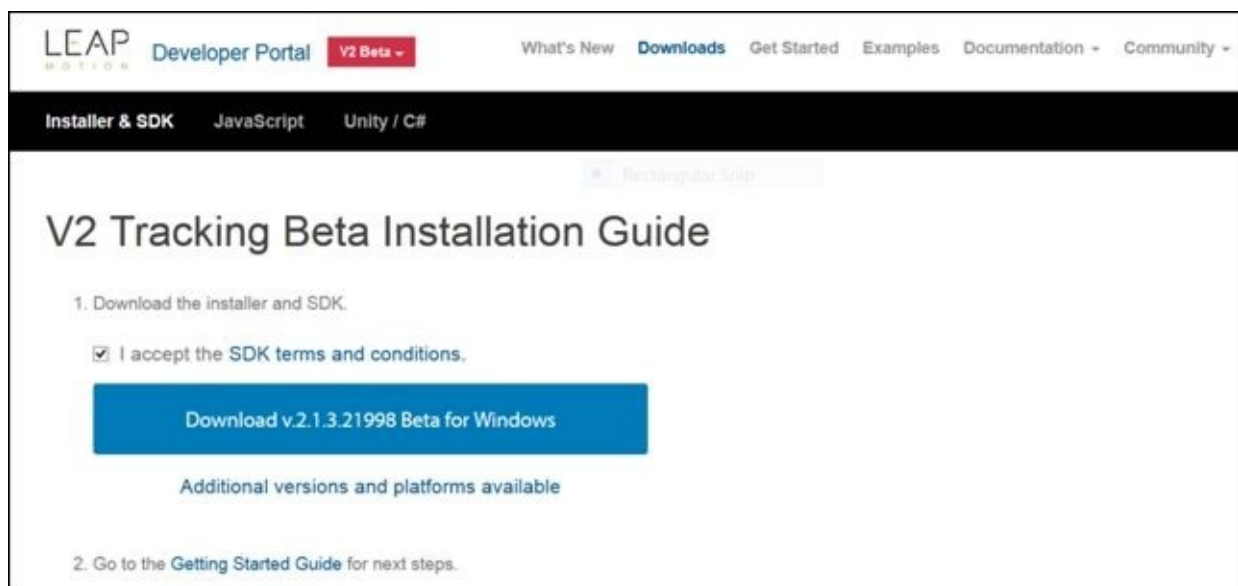
If things did not go as anticipated, skip to [Chapter 8](#), *Troubleshooting, Debugging, and Optimization*, to learn about troubleshooting and debugging the Leap Motion Controller to see what went wrong.

Installing the Leap Motion Developers' SDK

Now that we've got your device installed and ready to go, we need to download the Leap Motion Developers' Software Development Kit, or SDK.

The SDK contains a series of language-specific libraries, DLLs, and examples for any developer to freely use. Needless to say, it's very important that you have it installed if you're going to develop anything for Leap! So, without further ado, let's get the SDK installed with the following steps:

1. Head to <https://developer.leapmotion.com/> and sign in. If you do not have a developer's account, create one when prompted.
2. Following this, navigate to the **Downloads** tab at the top of the Leap Motion website and click on it.
3. You should then see a page that looks something like the one shown in this step. Leap Motion will attempt to autodetect your platform, presenting a screen that looks similar to the following one. If the information is correct, accept the terms and conditions (without reading them, naturally) and begin the download!



Once the download completes, extract the contents of the downloaded .zip file (assuming you're using Windows) to a safe place. We'll be referring back to this folder quite a bit later on. With that, you're done!

Installing the Java JDK

Before we install Eclipse, the IDE you'll be using for the remainder of this book, we need to install an appropriate Java Development Kit, or JDK. Oracle (the company that manages Java) is constantly changing its site, so here is a generalized step-by-step process to get a JDK installed:

1. Go to <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
2. Look for a box that says **JDK Download** in big letters. Click on the **Download** button.
3. You will be taken to a page with a list of JDKs and download options. Scroll down until you see your platform listed; in the case of Windows, the platform will be named either **Windows x86** (32-bit systems) or **Windows x64** (64-bit systems).
4. Accept the license agreement, download the file, and run it to install the JDK.

You're done! Now we can move onto installing Eclipse.

Setting up your IDE

With the Leap Motion software and SDK installed and out of our way, we can move onto getting your Integrated Development Environment, or IDE, installed and set up!

When I'm developing applications for the Leap Motion device (usually to control robots), I prefer to use the Eclipse Integrated Development Environment.

The Eclipse IDE was originally created for Java, but it has since expanded into many other languages, including C/C++ and Python. Throughout this book, all of my examples and instructions will assume you're using the Eclipse IDE. While you're welcome to use any other IDE, or even a text editor, I highly suggest that you install Eclipse, as it will make this book easier to use!

So first off, you have to find the IDE. Head to <https://www.eclipse.org/downloads/> and download Eclipse Standard Edition, as shown in the following screenshot:



Note

Make absolutely sure that you choose a version that matches your Java JDK (32-bit Eclipse for a 32-bit JDK, 64-bit Eclipse for a 64-bit JDK, and so on), or else there will be issues!

Once the download has completed (which can take a while, since the IDE is usually more than 150 megabytes in size), extract the ZIP file to wherever you'd like Eclipse to be located. Since Eclipse doesn't use an installer, everything you need is contained in the folder you have just extracted.

At this point, there's nothing left to do with the setup; we can now move onto the Leap Motion device itself!

Structure of the Leap Motion Application Programming Interface (API)

The Leap Motion API, while containing many complex and advanced features, is relatively simple. At the time of writing this book, there are about 23 different classes within the API, each one serving a different task.

Many of the classes are utilities that you won't directly instantiate; `FingerList`, `HandList`, `ToolList`, `Pointable`, and so forth are some examples of these kinds of classes. On the other hand (no pun intended), we have classes that contain data about a specific hand, finger, or other object, such as the `Finger`, `Hand`, and `Tool` classes. In almost all cases, the API classes are intuitively named and relatively easy to remember. Now, why don't we go over some of the more common classes and what they do?

Note

An up-to-date API documentation can always be found at <https://developer.leapmotion.com/documentation>.

The Vector class

The Vector class is a kind of utility class. It contains and manages a single set of x , y , and z coordinates. It also has a slew of built-in functions to make more common operations easier to perform. You'll find yourself using this class all the time, whether you know it or not. Almost every Leap API class makes use of it! The following is a brief example of how we can use the Vector class to check a hand's yaw (or simply put, to detect the rotation of your hand):

```
Vector position = new Controller().frame().hands().get(0).direction();
System.out.println("Hand 0 Yaw: " + position.yaw());
```

Note

Fun fact

When using 3D coordinates, we use three special words to refer to the rotation of an object. These words are Pitch, Yaw, and Roll, and they correspond to the forward and backward tilt (pitch), left and right rotation (yaw) and left and right tilt (roll) of an object.

This will first create a new Vector object, fill it with directional data for the first hand in Leap's field of view and then output its approximate yaw (in degrees) to the console window. This illustrates just one of the many functions you can perform with the Vector class.

The Finger class

Next on the list is the `Finger` class. This class contains tracking data for a single finger that is, or was, within Leap's field of view. Similar to the `Vector` class we just covered, this class contains a multitude of coordinates (which happen to be vectors themselves). However, it also includes a series of other things, including the hand that it belongs to, the current position of its tip, the direction its tip is facing, and more. Here is an example of how we can retrieve the position of a finger's tip:

```
Finger finger = new Controller().frame().fingers().frontmost();
System.out.println("Frontmost Finger data:" + "\nTip Position (X|Y|Z): " +
finger.tipPosition().getX() + "|" + finger.tipPosition().getY() + "|" +
finger.tipPosition().getZ());
```

The Hand class

Next in line after the `Finger` class is the `Hand` class! This class is just like the `Finger` class in respect to what it contains, with the addition of a few more items like the `Sphere Radius` and `Palm Position` of the hand. The `Sphere Radius` of the hand is the approximate radius of the biggest sphere that the hand can hold. Note that while it isn't effective for gauging the overall size of a hand, it can be useful to detect how spread apart a hand's fingers are. The `Palm Position` of the hand is relatively straightforward; this is the position of the hand's palm in the x , y , and z coordinates. Here is an example of how we can retrieve the position of a hand's palm:

```
Hand hand = new Controller().frame().hands(0);
System.out.println("First Hand data:" + "\nPalm Position (X|Y|Z): " +
hand.palmPosition().getX() + "|" + hand.palmPosition().getY() + "|" +
hand.palmPosition().getZ());
```

The preceding example is almost identical to our previous one with the `Finger` class; it simply outputs the three vector coordinates of the oldest hand in view to the console window.

The Frame class

Now for the `Frame` class. Before we delve into the specifics of this class too much, it's important to understand that the Leap Motion Controller works using frames of information; that is, it's much like a video game or movie. Frames are processed as they are received from Leap, and as such, have no set refresh rate (or *frame rate*).

Each frame contains a complete set of tracking data for all of the hands, fingers, tools, gestures, and other things that were within Leap's field of view when the frame was taken. The following code counts the objects within Leap's field of view for a single frame:

```
Frame frame = new Controller.frame();
System.out.println("Frame data:" + "\nHand count: " + frame.hands().count()
+ "\nFinger count: " + frame.fingers().count() + "\nTool count: " +
frame.tools().count() + "\nTimestamp: " + frame.timestamp());
```

The preceding example will retrieve the most recent frame from the Leap device and output the number of fingers, hands, and tools along with a timestamp to the console window.

The Leap Motion API is kind enough to cache the most recent sixty frames, allowing us to look at data from previous frames for whatever reason. To retrieve a specific frame, use the following syntax:

```
Frame frame = new Controller.frame(frameNumber);
```

Here, `frameNumber` is the number, 1-60, of the frame that you want to fetch. The oldest frame in memory is number 60, whereas the newest one is 1. To get the current frame, do not specify any frame number at all.

The Controller class

The `Controller` class at last! This class is your portal to the world of Leap motion, so to speak; you'll notice that we've used it in all of the previous mini-examples up until this point. The class itself is rather simple on the outside. The two methods that we'll be using most often are `frame` and `addListener`. The `frame` method, as the name would suggest, returns the most recent frame received from Leap. The `addListener` method, on the other hand, is used to register a custom `Listener` class with the device; this brings us to the next topic.

The Listener class

The `Listener` class, well, listens. To be specific, it's an event-driven class (or a callback, if you will) that is registered with the `Controller` class and responds to various things that are going on with Leap. It's a little hard to explain it with just text, so let's look at a practical example here, which is the `MyListener.java` file:

```
public class MyListener extends Listener
{
    public void onFrame(Controller controller)
    {
        Frame frame = controller.frame();

        if (!frame.hands().empty())
            System.out.println("First Hand data: " +
                "\nPalm Position (X|Y|Z): " + frame.hand(0).palmPosition().getX() + "|" +
                frame.hand(0).palmPosition().getY() + "|" +
                frame.hand(0).palmPosition().getZ());
    }
}
```

Then, in a separate file, `Main.java`, use this:

```
public class Main
{
    public static void main(String args[])
    {
        Controller controller = new Controller();

        MyListener listener = new MyListener();

        controller.addListener(listener);

        while (true) {}
    }
}
```

This was a bit longer than our previous examples, wasn't it? Let's break it down into chunks. First, let's look at the lines in the first class, `MyListener`:

public class MyListener extends Listener

The first part of this class defines our new class, which will be extending the `Listener` class; this is what allows it to be compatible with the Leap API. Following this, let's take a look at the next line:

```
public void onFrame(Controller controller)
```

The second part of this class defines an override for the `Listener` class' internal `onFrame` method so that we can respond whenever a new frame is received from the controller. Thusly, this function (once this class is registered, of course) will be called every time a new frame is generated by Leap. The next few lines from that point should be fairly familiar at this point; if they aren't, head back and review the other classes that we've covered so far!

Now, let's look at the lines of the `Main` class. The first few lines should be familiar (they simply initialize the controller), so I'm only going to focus on one particular line:

```
controller.addListener(listener);
```

This line registers our instance of the `MyListener` class with the Leap API, allowing it to receive events from Leap. In this case, once registered, the `onFrame` method from our `MyListener` instance will be called every time a new frame is received from the controller.

Then, there's the last line:

```
while (true) {}
```

This line prevents our main class from exiting, allowing our `Listener` implementation to continue running and receiving events until we're ready to quit.

Note

Fun fact

The Leap Motion Listener system runs inside its own thread at all times; this means that any listeners that you register will continue to receive updates, even if your application is single threaded and receives a blocking call or something similar.

Creating a simple framework program within the Eclipse IDE

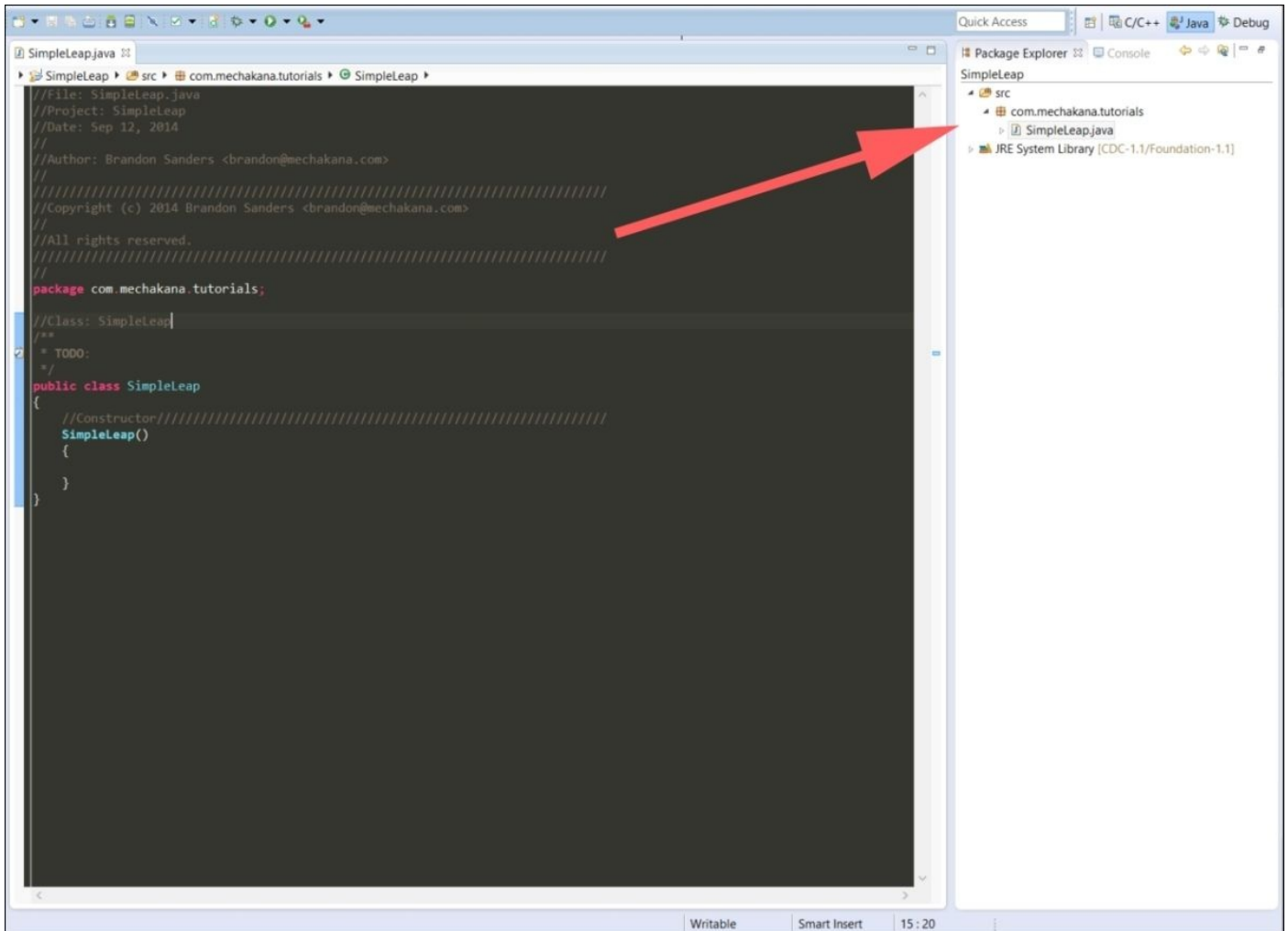
Based on everything we've done this far, we can go ahead and create a basic framework program to build off. The end result will be an all but empty program that continuously outputs tracking data from a single hand within Leap's field of view. You can then use this *template* with the examples featured later on within this book. In addition, this section will help familiarize you with the Eclipse IDE if you haven't used it before.

So, without further ado, let's get started!

Setting up the project

First off, we need to create a new Eclipse Java project. This can be achieved by heading over to **File | New | Java Project** from within the IDE; you will then be greeted by a project creation wizard. Choose a name for your project (in this case, I called mine SimpleLeap) and then click on the **Finish** button.

Once your project is created, navigate to it in the Package Explorer window. The following screenshot illustrates what the package explorer should look like:



While my IDE layout will most likely drastically differ from yours, the preceding screenshot should give you a general idea of what the Package Explorer looks like (the giant orange arrow is pointing to it). It most conveniently displays all of your Java packages as a folder hierarchy, making navigating large and more complex projects a breeze.

For now, we should go ahead and create a new package to put our classes in. This can be achieved by right-clicking on the src folder for your project in Package Explorer and then going to **New | Package** tooltip. You can name it whatever you like; standard Java naming convention dictates that package names consist of your web address in the reverse order followed by the project name, so I named mine `com.mechakana.tutorials`.

Note

Fun fact

Java developers use web addresses in the reverse order (like `com.mechakana`) as package names so that different developers can distribute packages with unique names. If I just named my package “tutorials”, it could potentially overwrite somebody else’s package named `tutorials`—if I use my own website and name it `com.mechakana.tutorials`, I can guarantee I’m the only person with that name.

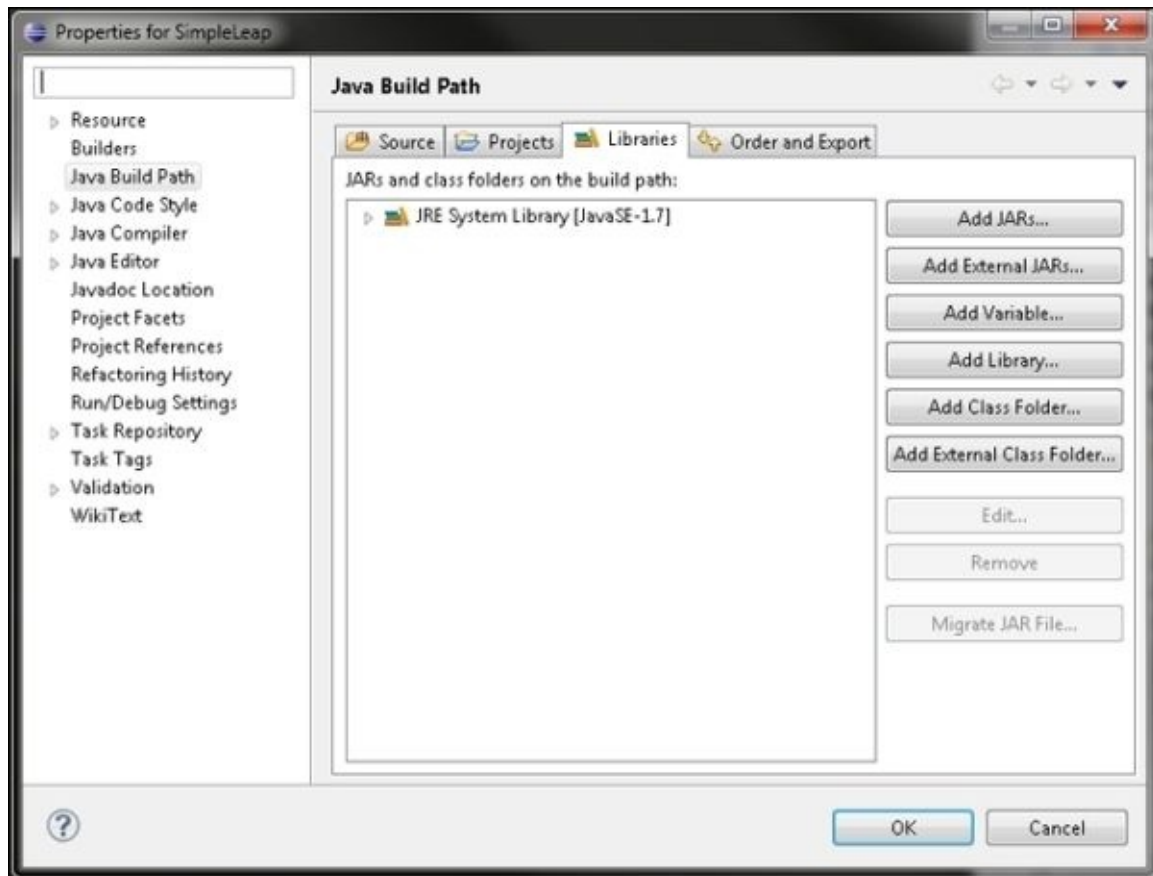
Once our package is created, we can fill it with two empty files: `SimpleLeap.java` and `LeapListener.java`. To create a new class in Eclipse, right-click on our package (`com.mechakana.tutorials` in my case) and then go to **New | Class**. You should do this twice; once for the `SimpleLeap` class and again for the `LeapListener` class. Only specify a name for the class—keep all the other settings as their defaults.

Now, we need to configure the Leap libraries. While it would be super convenient to just start coding, it wouldn’t do us a whole lot of good if we didn’t have access to Leap, right? Fortunately, this is a relatively easy task that can be done in a few simple steps:

1. Navigate to your Leap Motion SDK directory that we created earlier and go to `LeapSDK/lib/`. Copy the `LeapJava.jar` file contained within to the root of our project in Eclipse. On Windows, you can simply drag the file from the Leap SDK folder into Eclipse and it will automatically be copied.
2. While we’re still within `LeapSDK/lib/`, navigate one level deeper into the `x86` folder. Copy the `Leap.dll` and `LeapJava.dll` files contained within to the root of our project in Eclipse.
3. Our project should now look similar to the following screenshot when viewed from within the Eclipse IDE:



4. Now, with all of the items we need contained inside our project folder, hover over the `SimpleLeap` project in Package Explorer (inside Eclipse) and right-click on it. Navigate to **Build Path | Configure Build Path** in the tooltip that pops up.
5. Once inside, you will be presented with a window that contains a series of tabs, similar to the one pictured here:



6. Navigate to the **Libraries** tab and hit the **Add JARs...** button. In the window that pops up, open our SimpleLeap project, navigate to the LeapJava.jar file and then double-click on **OK**.

With that, you should be done setting up the project for development! Aside from the creation of our two files, these steps are common to pretty much any Eclipse project that uses the Leap Motion device; be sure to remember them!

For more information about Eclipse, you can go to their official website at <http://www.eclipse.org>.

Let's write some code!

With our project set up and ready to go, let's start writing! First off, we need to create an implementation of the `listener` class. Go ahead and open up the `LeapListener.java` file, which we created earlier, and enter the following:

```
package com.mechakana.tutorials;

import com.leapmotion.leap.*;

public class LeapListener extends Listener
{
    public void onFrame(Controller controller)
    {
        Frame frame = controller.frame();

        if (!frame.hands().isEmpty())
            System.out.println("First Hand data:" +
                "\nPalm Position (X|Y|Z): " + frame.hand(0).palmPosition().getX() + "|" +
                frame.hand(0).palmPosition().getY() + "|" +
                frame.hand(0).palmPosition().getZ());

        if (!frame.fingers().isEmpty())
            System.out.println("Frontmost Finger data:" +
                "\nTip Position (X|Y|Z): " +
                frame.fingers().frontmost().tipPosition().getX() + "|" +
                frame.fingers().frontmost().tipPosition().getY() + "|" +
                frame.fingers().frontmost().tipPosition().getZ());
    }
}
```

Now, let's discuss what each major chunk of code does:

```
import com.leapmotion.leap.*;
```

This line imports the entire Leap API into the context of our file. Needless to say, without this statement, we cannot make use of Leap:

```
public class LeapListener extends Listener
```

The preceding line defines our `LeapListener` class and states that it extends the Leap API's `Listener` class. This is what allows us to later register the class with a `Controller` object.

```
public void onFrame(Controller controller)
```

This line overrides the `Listener` class' built-in `onFrame` method. Once our `LeapListener` class is registered with a `Controller` object, this method will be called every time a new frame is received from Leap.

The rest of the lines contained in this example code have been covered previously in this chapter, but basically, they collect tracking data from hands and fingers and output it to the console window.

Moving on, let's fill out the `SimpleLeap` class. Go ahead and open up `SimpleLeap.java`

and fill it with the following code:

```
package com.mechakana.tutorials;
import com.leapmotion.leap.*;

public class SimpleLeap()
{
    public static void main (String args[])
    {
        Controller controller = new Controller();

        LeapListener listener = new LeapListener();

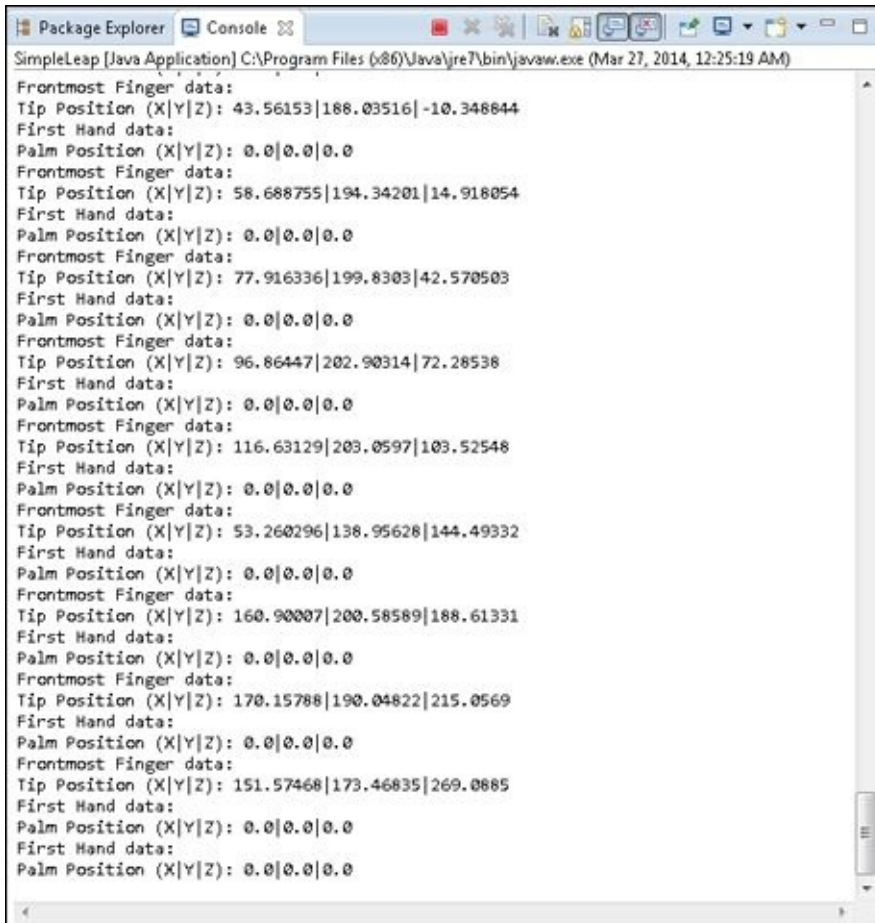
        controller.addListener(listener);

        while (true) {}
    }
}
```

This code should look pretty familiar to you; in fact, we've used it before in this chapter during the previous `Listener` example! Since there's nothing new to this code, I'll give you the rundown; it registers an instance of our `LeapListener` class with `Leap` so that it can receive callbacks. Once registered, it loops forever until the user closes the application.

Trying it out

With all of the hard stuff out of the way, we can finally test out our code! With Eclipse, it's quite easy for us to launch a console application; simply click on the green arrow at the top of your toolbar or press *Ctrl + F11*. If everything worked correctly, you'll see a console tab pop up somewhere within your workspace. Try waving your hands around in front of the Leap device, and you'll see an output similar to the following screenshot:



```
SimpleLeap [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Mar 27, 2014, 12:25:19 AM)
Frontmost Finger data:
Tip Position (X|Y|Z): 43.56153|188.03516|-10.348644
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
Frontmost Finger data:
Tip Position (X|Y|Z): 58.688755|194.34201|14.918054
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
Frontmost Finger data:
Tip Position (X|Y|Z): 77.916336|199.8303|42.570503
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
Frontmost Finger data:
Tip Position (X|Y|Z): 96.86447|202.90314|72.28538
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
Frontmost Finger data:
Tip Position (X|Y|Z): 116.63129|203.0597|103.52548
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
Frontmost Finger data:
Tip Position (X|Y|Z): 53.260296|138.95628|144.49332
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
Frontmost Finger data:
Tip Position (X|Y|Z): 160.90007|200.58589|188.61331
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
Frontmost Finger data:
Tip Position (X|Y|Z): 170.15788|190.04822|215.0569
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
Frontmost Finger data:
Tip Position (X|Y|Z): 151.57468|173.46835|269.0885
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
First Hand data:
Palm Position (X|Y|Z): 0.0|0.0|0.0
```

With that, you're ready to dive into the world of Leap Motion.

Let's get started!

Looking forward – the Skeletal Tracking API

Recently, the Leap Motion development team has released a new version of the Leap SDK (v2.0), which contains an all new API for skeletal tracking (as in, you know, tracking the bones in your hand).

This new API brings not only a slew of new possibilities, but also an increase in the precision of current tracking and vision recognition functions. As the new API was still in the beta phase during this book with no stable release in sight (beta software in a book isn't fun for anyone!), we will not be making heavy usage of it in the projects that we work on.

With that said, as it is unlikely that the skeletal tracking API will change too much in the coming months and the ideas it brings to the table are relatively straightforward, I thought we might as well go over what it has to offer compared to the current API. Depending on your level of experience with the Leap Motion Controller up to this point, some of the things I talk about in the next few pages might or might not mean much to you—but I assure you, it's all relevant!

Different fingers? Not a problem

We'll be covering the primary disadvantages of Leap and how to correct them later in this book, but I'm going to go ahead and say it now: the new Skeletal Tracking API eliminates a lot of the issues present in the current API. It is, in simple terms, *freaking awesome*.

With the new API, individual fingers can now be differentiated from each other. This probably seems like something rather simple, but it can make a world of difference in many cases. Let's take a first-person shooter for example.

During normal gameplay, the player holds his or her primary hand out in front of the Leap device in the form of a gun (making a fist with only the index finger and thumb pointing out). Once the player has aimed at a target they want to shoot, they'll close their thumb on their hand as if firing the gun. This will cause the in-game weapon to fire in the direction aimed by their finger, ideally.

Now, if you can't tell the difference between the thumb and the pinky finger, or an index finger and a ring finger, this seemingly trivial programming task—calling a function when the user's thumb disappears from view or enters a specific threshold—suddenly becomes a beast of a programming task; this is where the new Skeletal Tracking API comes to the rescue!

Earlier, we differentiated the fingers on a hand by trying various tricks like sorting the *x* indices of each finger from left to right or right to left and then guessing and based on this data, which finger belonged to which “named” finger (such as a thumb, index, or pinky). We'll be covering this in the next few chapters. However, with the new API, all we have to do is check the `type` field on any valid finger object and the API will work its magic, providing you with an integer that represents the proper name for the finger that the object denotes.

The bare-bones snippet here illustrates, in two lines, how you'd query the finger type data using the new API:

```
if (frame.hand(0).finger(0).type == 0)
    System.out.println("The first finger on the first hand is a thumb.");
```

The preceding example snippet will print out a notice if, and *only if*, the first finger on the first hand in a given frame is a thumb (finger 0 on hand 0, since the Leap Motion API is zero indexed).

Handedness is no longer an issue

Gone are the days of wondering if a given hand belonged to somebody's right arm or left arm. Using the magic of mathematics and other unknown formulae, relating to the structure of the skeleton of the average human hand, the shiny new API can now differentiate between the left and right hands.

Again, this is not as simple as it sounds. There are times when you want to figure out whether a user has their left or right hand out; why, I won't ask, but there are such times. Using the preskeletal tracking API, one of your best bets for figuring out which hand is which is detecting the side of a hand the farthest finger (the thumb) was on; if the thumb was on the left, it's a right hand, and vice versa. Of course, this is little better than a guess—what if the user's hand is upside down, for instance? You'd then have to take that into account, and so on, and so on...not fun.

Yet again, the new Skeletal Tracking API comes to the rescue with the `isLeft()` and `isRight()` functions, which call all of our `Hand` objects home in the new API! I certainly hope that the names of these functions are self-explanatory, but just in case, for the sake of clarification, I shall clarify. The `isLeft()` function for any `Hand` object will return a result of `true` if the object is a left hand and `false` otherwise. Likewise, the `isRight()` function for any `Hand` object will return a result of `true` if the object is a right hand and `false` otherwise. This is far superior to the previous method of guessing.

Having confidence in tracking data

This is slightly less of a concern to the average developer, but it is a concern nonetheless. The *confidence* that your software has that a tracked hand is, indeed, a hand! The amount of confidence the Skeletal Tracking API has about the accuracy of a hand is based on how well the tracking data for a hand matches a predetermined model of an ideal hand, including posture and finger positions.

In the 2.0 Beta API, this *confidence* value is rated on a scale of 0.0 to 1.0, with 0.0 meaning it is extremely unlikely that a tracked hand is valid, while a value of 1.0 means it is extremely likely that a tracked hand is valid.

Since every `Hand` object has a confidence rating, all you have to do to get the confidence rating for a hand is call the `confidence()` member function of a `Hand` object and you will get a return value of 0.0 to 1.0, like so:

```
System.out.println("First Hand Confidence Rating: " +  
String.valueOf(frame.hand(0).confidence()));
```

The preceding example snippet will print out the confidence rating of the first hand in a given frame to your console window.

Pinching and grabbing are now much easier

Gone are the days of trying to detect gestures or interpolating finger coordinates in an attempt to pick up pinches and grabs from the user. The new API introduces two new member functions for the `Hand` class that are specifically for detecting grabbing (closing of the fist) and pinching—`grabStrength()` and `pinchStrength()`, respectively.

The first function, `grabStrength()`, will return a value between 0.0 and 1.0 that represents how tightly a given hand is making a fist. A value at or near 0.0 means a hand is almost perfectly flat and not making a fist, while a value at or near 1.0 means a hand is making a tight fist. In other words, the more a hand curls its fingers inward into the form of a fist, the higher the returned value from `grabStrength()` will be. Similar to the other new methods we've covered already, usage of this one is relatively simple:

```
System.out.println("First Hand Grab Strength: " +  
String.valueOf(frame.hand(0).grabStrength()));
```

The preceding example snippet will print out the approximate grab strength of the first hand in a given frame to your console window.

Moving on to the second function, `pinchStrength()` will return a (you guessed it) value between 0.0 and 1.0 that represents how close a hand's thumb is to any other finger on the hand. This one is just a tad more confusing than the other functions; basically, a hand's pinching strength is defined by a hand's thumb being very close to any other finger on the hand—it doesn't matter which finger, as long as it's a finger.

A value at or near 0.0 means that a hand's thumb isn't very close to any other finger, and therefore the hand isn't pinching very hard (or at all). On the other hand, a value at or near 1.0 means that a hand's thumb is very close to other fingers and is pinching rather hard. Basically, the closer a hand's thumb is to another finger, the higher the returned value from `pinchStrength()` will be. Just like `grabStrength()`, `pinchStrength()` is relatively simple to use:

```
System.out.println("First Hand Pinch Strength: " +  
String.valueOf(frame.hand(0).pinchStrength()));
```

The preceding example snippet will print out the approximate pinch strength of the first hand in a given frame to your console window.

A new API class – Bones

Of course, the Skeletal Tracking API update wouldn't make much sense if it didn't introduce a facility for dealing with, well, skeletons.

This is where the new Bone class comes in. Using it, you can glean all kinds of information from any valid Finger object, such as the length of the metacarpals, proximal phalanges, intermediate phalanges, and distal phalanges on a given finger. Don't worry; I didn't know what these names meant either until I read the shiny new API docs for developers. We won't go too in depth about this new class since the API was (or still is) in the beta stage at the time of writing this, but let's go ahead and touch on the basics.

In the new Skeleton Tracking API, every single Finger object contains an array of Bone objects—four to be precise. The exception is for thumbs; while they technically have four Bone objects, they only have three literal bones in real life. Therefore, the length of the metacarpal bone (the one closest to the hand, which thumbs don't have) on a thumb will always be zero.

Every Bone object has a variety of useful and/or fun member functions, including:

- The standard `isValid()` and `invalid()` functions to check data integrity
- The `length()` function to check the length of the given bone
- The `prevJoint()` function that returns the `Vector` object for the point the given bone is anchored to
- `nextJoint()` that returns the `Vector` object for the tip of the given bone
- `type()` for getting the proper name, or type, of the given bone

For what it's worth, the names of the individual bones on a finger, in the order from the closest to the hand to the furthest from the hand, are: the metacarpal bone, proximal phalange bone, intermediate phalange bone, and the distal phalange bone.

That's it!

As you can see, the new Skeletal Tracking API is poised to revolutionize the way the Leap Motion software will be written by developers like you and me. Everything is easier, from simple things such as picking out named fingers to the more complex things such as iterating over the bones of different fingers to create more accurate 3D representations of our hands. Only time will tell what awesome and crazy applications will be written using this new API. Now...back to *Mastering the Leap Motion Controller*!

Summary

In this chapter, we covered all of the basics of the Leap Motion device. You installed and set up the Leap Motion device, its libraries, and an IDE to help you program for it. We then went over all of the different members of the Leap Motion API, including vectors, fingers, hands, frames, controllers, and listeners. Afterwards, you created a simple framework program with Eclipse to make sure everything was working correctly. We finished off with a lengthy look at the new Skeletal Tracking API and what it brings to the development table. Armed with this knowledge, you are now ready to tackle the next few chapters of this book.

In the next chapter, we'll begin diving into the world of Leap Motion. You'll be learning about how Leap interprets your hands, various ways you can detect user input, and some unavoidable limitations of the tracking software.

Chapter 2. What the Leap Sees – Dealing with Fingers, Hands, Tools, and Gestures

In this chapter, you will learn about the more complex aspects adopted within any Leap Motion application. This includes basic tracking and coordinate data, such as hands and fingers, as well as more advanced features, such as tools and gestures. Throughout this chapter, we'll go through various sets of example code to give you an idea of everything that can be done to grab input from a user.

Note

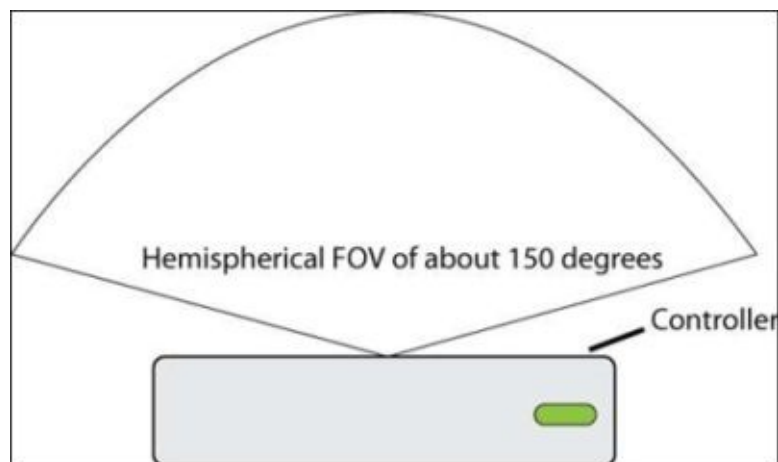
This chapter is sprinkled with periodic *Fun facts* that offer high-level and entry-level factoids about scripting and programming for your reading pleasure.

Handling hands and fingers

In the previous chapter, you learned about the top-level functionality of the `Hand` and `Finger` classes. How about we go over the specifics?

The Leap's field of view

When the Leap is plugged in and turned on, it is constantly looking for and tracking any hand, finger, or tool-like objects within a certain area, commonly referred to as its **field of view (FOV)**. The FOV is just 30 degrees short of being a perfect hemisphere, measuring in as a 150-degree area protruding directly from the device. You can see this perfectly in the following diagram:



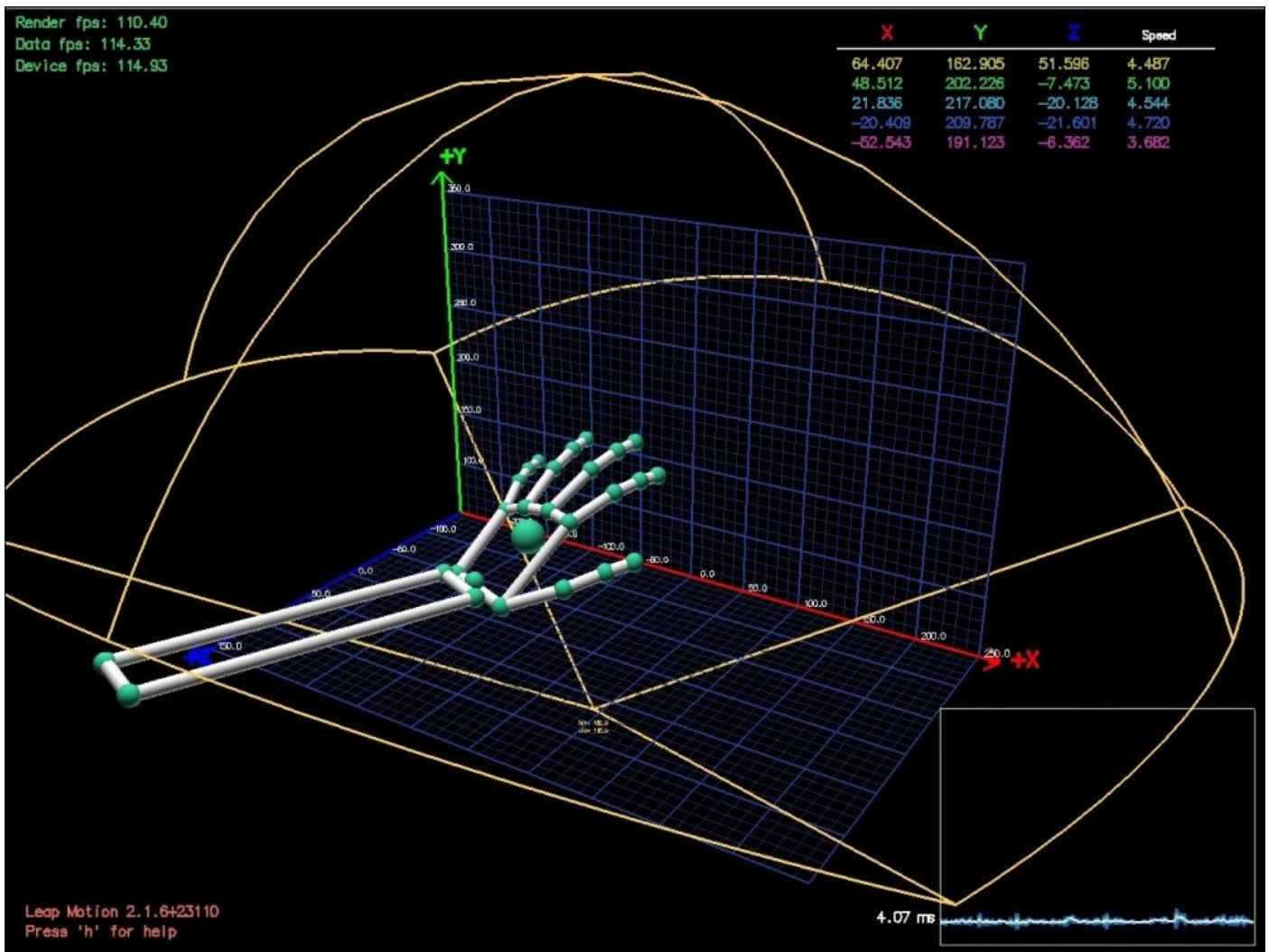
The actual range of the FOV is about 20" (20 inches or 50 centimeters, for those who are not familiar with the convention) from the device in the upward direction and 10" from the device in any lateral direction. Anything within the field of view will automatically be detected by the Leap and then forwarded to the API classes, which we will use when programming applications.

To give you a better idea of the three-dimensional appearance of the Leap's FOV, I've included the following screenshot. The boundaries of the FOV are the yellow lines, with one hand visible towards the center of the image.

Note

Fun fact

This screenshot was taken from the Leap Motion Diagnostic Visualizer—please refer to [Chapter 8, Troubleshooting, Debugging, and Optimization](#), for information on what it is and how to use it.

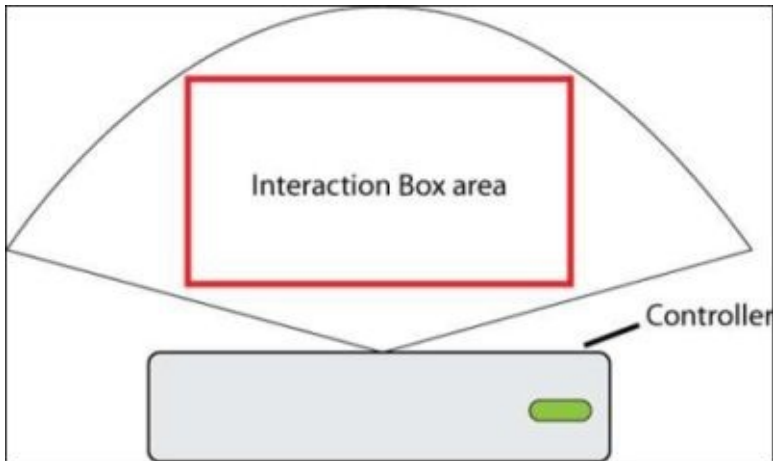


When writing an application for the Leap, you should avoid placing user interaction areas (menus, buttons, sliders, and so on) near the edges of the screen (and thus, near the edges of the Leap’s field of view); this will help mitigate the potential for erratic and strange feedback from the Leap.

Unfortunately, however, there are many cases where the best place for a button or a menu is at the screen’s edge. The question is how do we place items near the edge of the screen, and thus in the field of view, without the risk of violating the edge of the Leap’s detection zone? Easy; we can use the `InteractionBox` class from the Leap Motion APIs.

The InteractionBox class

With the InteractionBox class, we can normalize the Leap's coordinate system and map it to the resolution of a user's screen. How does the interaction box work, you might ask? It's quite simple in principle; the interaction box represents a (mostly) perfect cuboid (box-shaped object) that exists entirely within the limits of the Leap's FOV, as pictured here:



Now, the easiest way to demonstrate how this class works is with a practical example; let's write some code. If you're still using the simple Leap app template from the previous chapter, replace your listener's onFrame method with the following code:

```
public void onFrame(Controller controller)
{
    Frame frame = controller.frame();

    //Retrieve an InteractionBox reference.
    InteractionBox box = frame.interactionBox();

    if (!frame.fingers().isEmpty())
    {
        //Retrieve the vector of the frontmost finger's tip.
        Vector frontmost = frame.fingers().frontmost().tipPosition();

        //Normalize the frontmost vector to a 0...1 scale.
        frontmost = box.normalizePoint(frontmost);

        //Print out the vector. Left, front and bottom are represented by 0.
        System.out.println("Frontmost Finger normalized coordinates (X|Y|Z):
" + frontmost.getX() + "|" + frontmost.getY() + "|" + frontmost.getZ());
    }
}
```

OK, let's break down this code. Check this line:

```
InteractionBox box = frame.interactionBox();
```

Here, we're fetching a reference to the current frame's InteractionBox object and assigning it to an empty InteractionBox reference.

Note

You should never ever try to initialize your own `InteractionBox` object, as the frame-supplied one takes into account the user's current configuration settings and the state of the motion controller device.

You should be pretty familiar with the following line:

```
Vector frontmost = frame.fingers().frontmost().tipPosition();
```

Here, we're simply getting the vector coordinates for the frontmost finger's tip position.

The next line is *the* important one, and the main reason why we use `InteractionBox`:

```
frontmost = box.normalizePoint(frontmost);
```

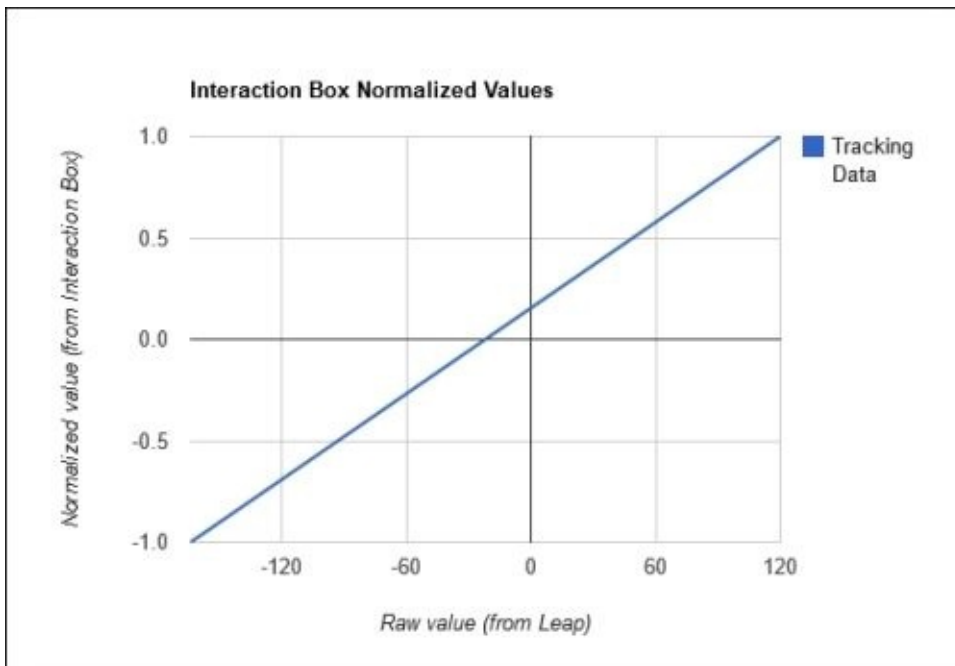
This line normalizes the value of the passed vector to a 0 – 1 scale, based on the internal vector coordinate system of the frame that this `InteractionBox` reference belongs to. This allows us to map the Leap coordinates to pretty much any percentage-based coordinate system in existence.

Note

What does normalize mean?

Well, when you normalize a vector using the `InteractionBox` object's `normalizePoint` function, you're converting the distinct vector coordinates within this vector to floating point values between 0 and 1. This effectively turns a z value of, say, -164 (all the way to the front on a standard Leap device) to a z value of 0 or 0 percent. In turn, a z value of 0 before being normalized will become a z value of about 50 percent or will be centered within the Leap's FOV. This normalization allows us to map the Leap input, in a meaningful way, to other coordinate systems such as those found on graphical user interfaces or games.

The `InteractionBox` class' normalization process is described by the following graph, assuming a maximum range of -164 (all the way forward) to 120 (all the way back):



Finally, the last line simply prints out the three normalized coordinates of the frontmost finger. Try the code out and you'll see values with quite a few decimal places that range from 0 to 1—although the values can be better read as “0 to 100 percent in a given direction.”

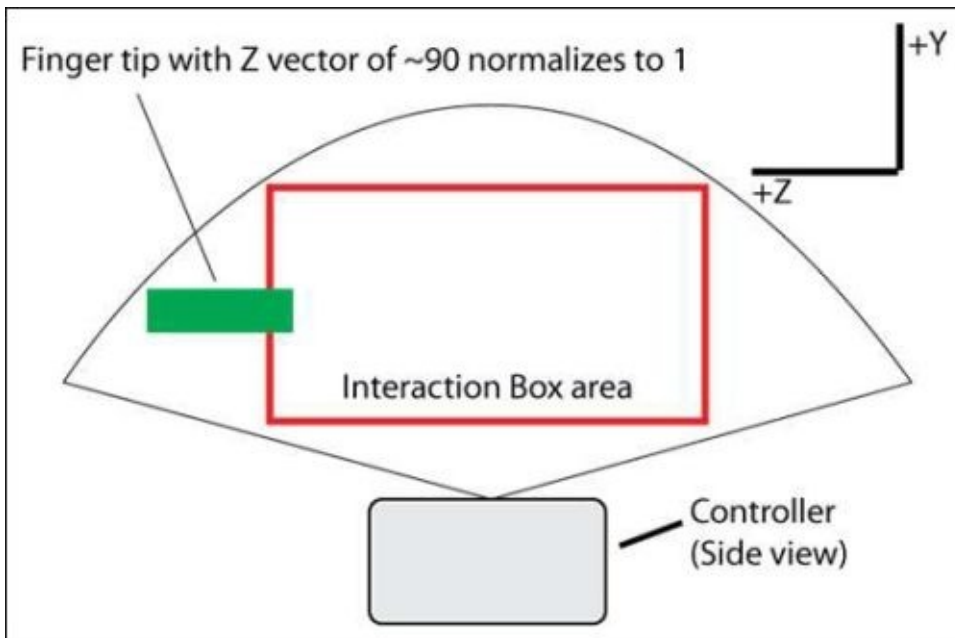
How the interaction box works

Behind the scenes, the Leap Motion device is always tracking and gathering data on anything that comes into its field of view. These items are then tagged as fingers, hands, tools, or other objects and packed into a frame that is sent to the Leap API.

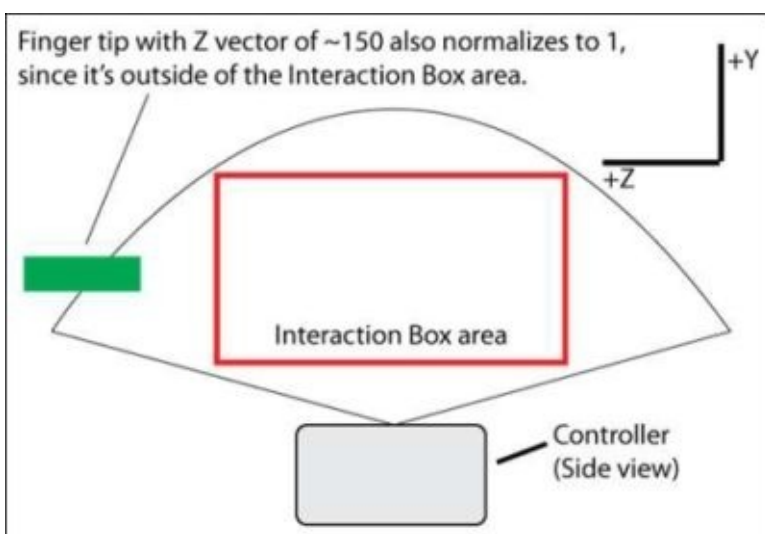
In addition to these items, the Leap also appends an `InteractionBox` object to every single frame. The box represented by this object is equal to the size of the biggest possible box that can fit within the Leap's current field of view. Now, since the Leap's FOV is anything but a box (it's basically a hemisphere), it's impossible for the `InteractionBox` area to fill the entire effective field of view for any Leap device.

In practice, this means that, for example, two different fingertips, one with a z value of 90 (towards the back of the Leap's FOV) and the other with a z value of 150 (even further back), will be normalized to the same value of 1.0 (all the way *back*). This is because of the fact that once the fingertips exit the effective range of the interaction box area, the `InteractionBox` class will automatically clamp the passed vectors down to the maximum possible value of 1.

To give you a better illustration of how this looks, take a look at the following two diagrams. In the previously shown screenshot and diagram, showing Leap's FOV, you can see the Leap Motion Controller from its right side when in normal operation, with the left side of the image being the one *facing* the user.



In the first situation, shown in the preceding diagram, the finger is *breaking the wall* of the interaction box area and is, therefore, within it. Since it is at the edge of the box, its z vector will be normalized to a value of 1.



In the second situation, pictured here, the finger is completely outside the interaction box's area but still within the field of view of the Leap device. Since its z vector cannot be normalized to anything outside the 0 - 1 range, the InteractionBox class will automatically *clamp* the z value passed to the `normalizePoint` function to 1 because this is the closest applicable value.

Why would you ever want to use something like the interaction box?

Well, the interaction box is one of the most useful (and for me, one of the coolest) features incorporated in the Leap API. You can use it to map the Leap coordinates to screen coordinates or even other coordinate systems within games and simulators.

A good example of the interaction box in action is in a quadrotor simulator that I wrote a while back, designed to simulate—you guessed it—quadrotors. This simulator, called **Artemis**, was originally written to help me train with the Leap Motion Controller before I deployed it in my real-life quadrotors...wouldn't want to crash a flying robot into a wall now, would we? In Artemis, I used the `InteractionBox` class to achieve two different things:

- Using the interaction box, I was able to map tracking data from the Leap to three-dimensional world-space coordinates within the simulator, allowing me to render 3D hands on the screen.
- Using the same method, I was able to map data from the Leap directly onto the user interface to give visual feedback as to where the user was pointing.

Once you start working with 3D toolkits in [Chapter 5, *Creating a 3D Application – a Crash Course in Unity 3D*](#), you'll not only get to work on a simulator similar to Artemis, but we'll also cover the actual application of the interaction box and how it is essential for user interface design.

Detecting gestures and tools

Next up on the list of things to tackle are gestures and tools. Let's start with tools, since they're pretty simple...

Detecting and using tools

The Leap Motion API documents do the best job of defining what exactly a tool is to the Leap:

“Tools are pointable objects that the Leap Motion software has classified as a tool. Tools are longer, thinner, and straighter than a typical finger. Get valid LeapTool objects from a LeapFrame object.

Tools may reference a hand, but unlike fingers they are not permanently associated. Instead, a tool can be transferred between hands while keeping the same ID.”

In other words, any instance of the `Tool` class represents a three-dimensional object that is longer and thinner than a regular finger. In addition, since `Tool` objects have their own distinct ID and are tracked independently of hands, the hand that *owns* them can change over the duration of the existence of the `Tool` object.

In practice, manipulation of tools is almost identical to manipulation of fingers; the only difference is that Leap will only register objects that meet the aforementioned criteria as tools, and they don't necessarily always belong to the same hand throughout their lifetime. The following code gives a brief demonstration of how to detect and read data from tools present within Leap's FOV:

```
Frame frame = controller.frame();

if(!frame.tools().isEmpty())
{
    System.out.println("Frontmost Tool data:" + "\nTip Position (X|Y|Z): " +
        frame.tools().frontmost().tipPosition().getX() + "|" +
        frame.tools().frontmost().tipPosition().getY() + "|" +
        frame.tools().frontmost().tipPosition().getZ());
}
```

Easy, right? To test this code, start up the code and then grab something such as a pencil or pen and wave it around within the Leap's field of view. You should be greeted by an output; you'll notice that if you try to place your fingers in the field of view of the Leap, it will ignore them because they aren't tools.

Gestures

OK, so these are just a tad more complex than the previous topics we've covered so far. However, I digress; they're still relatively simple. As gestures are one of the more complicated aspects of the Leap Motion API, we'll walk through the different gestures (and what they do) one at a time.

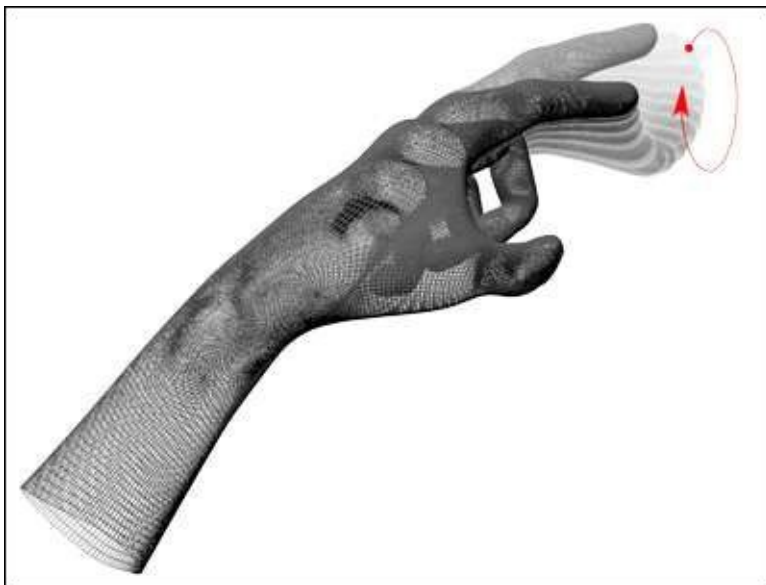
Detecting gestures

Whenever the Leap Motion Controller is turned on and is active, it is watching for activity within its FOV that resembles certain kinds of movement patterns that are typical of a user gesture or command. As an example, a circular movement of the user's finger might be detected as `CircleGesture`, while a hand moving from side to side might indicate `SwipeGesture`.

As per the Leap API documents, whenever the Leap's tracking software detects what might be a gesture, it assigns the gesture an identifier and adds a corresponding `Gesture` object to the gesture list for that frame. In the case of continuous gestures that take place over multiple frames, the Leap software will update the gesture by adding a `Gesture` object with an identical identifier to each subsequent frame.

The gestures that are currently supported by the Leap are pictured here:

- **Circle gestures:** These denote the circular motion of a single finger:



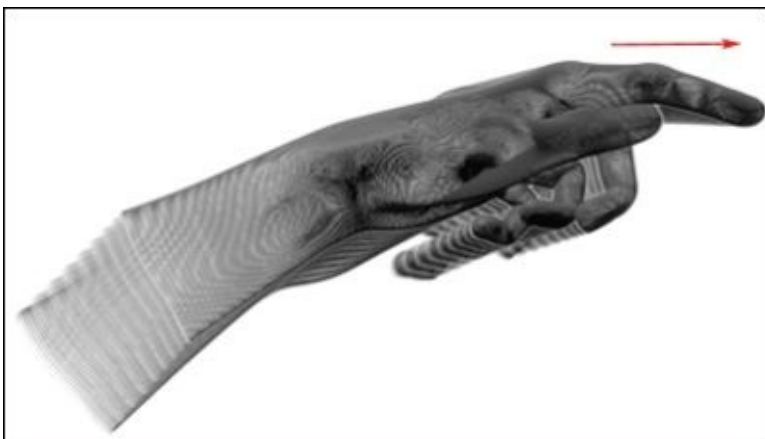
- **Swipe gestures:** These denote the swiping motion of a hand, finger, or tool:



- **Screen tap gestures:** These denote a finger tapping the screen by poking forward and then returning to its original position:



- **Key tap gestures:** These denote a finger rotating down towards the palm and then returning to its original position:



Now, how about an example of a gesture in action? Let's try out how to detect a circular gesture by adding the following code to your `onFrame` function:

```
//Enable detection of circular gestures.
controller.enableGesture(Gesture.Type.TYPE_CIRCLE);

if (!frame.gestures().isEmpty())
{
    //Loop over all of the gestures detected by the Leap.
    for (Gesture gesture : frame.gestures())
    {
        //If it's a circle gesture, print data for it.
        if(gesture.type() == Gesture.Type.TYPE_CIRCLE)
        {
            CircleGesture circleGesture = new CircleGesture(gesture);

            System.out.println("Detected Circle Gesture:" +
                "\nRadius: " + circleGesture.radius() +
                "\nRotations: " + circleGesture.progress());
        }
    }
}
```

OK, let's discuss each line. Here is the first one:

```
controller.enableGesture(Gesture.Type.TYPE_CIRCLE);
```

This line tells the Leap to start looking for circular gestures. By default, the Leap will not detect any gestures, so it's very important to call this function. Any `Controller` object will support this. Currently, there are four gesture types that you can pass to the `enableGesture` function:

- `TYPE_CIRCLE`
- `TYPE_SWIPE`
- `TYPE_SCREEN_TAP`
- `TYPE_KEY_TAP`

These functions cover the preceding four gestures, respectively, and can be accessed via the `Gesture.Type` enum, as seen in the preceding line of code.

The next three lines check whether the current frame has gestures, and if it does, it iterates over all of them looking for circular gestures.

Then, if a circular gesture is found, this line is executed:

```
CircleGesture circleGesture = new CircleGesture(gesture);
```

This line creates a new `CircleGesture` instance from the detected circular gesture. You cannot cast a `Gesture` object to a `CircleGesture` object, no matter how convenient this will be; you must explicitly call the `CircleGesture` constructor and pass it to the `Gesture` object that you want to use, as seen in the preceding line of code.

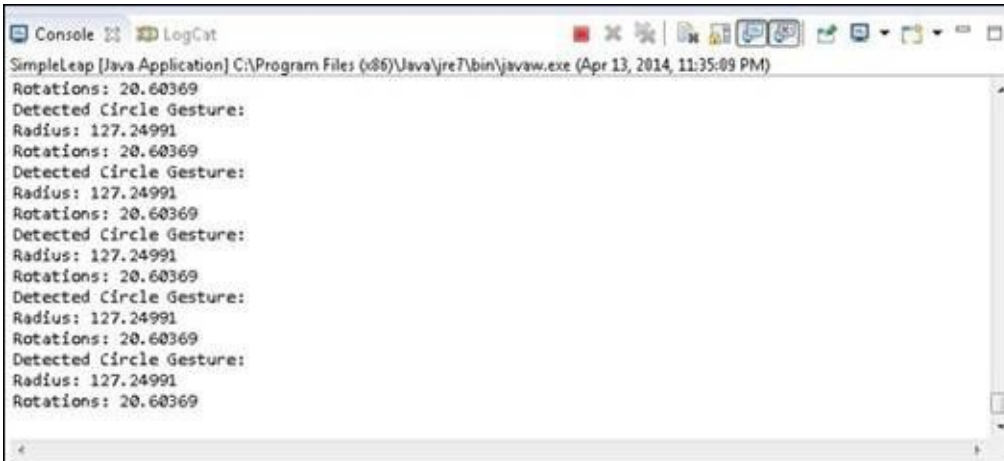
Finally, refer to the last line:

```
System.out.println("Detected Circle Gesture:" + "\nRadius: " +
```

```
circleGesture.radius() + "\nRotations: " + circleGesture.progress());
```

This line prints out the radius (in millimetres, via the `CircleGesture.radius` member) of the circular gesture as well as the amount of times a complete circle has been created by the finger drawing the gesture (via the `CircleGesture.progress` member).

If you drop all of this into your simple Leap template project in Eclipse, run it, and then draw some circles with one of your fingers, you should be greeted by a console output that is similar to the output shown in the following screenshot:



```
SimpleLeap [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Apr 13, 2014, 11:35:09 PM)
Rotations: 20.60369
Detected Circle Gesture:
Radius: 127.24991
Rotations: 20.60369
Detected Circle Gesture:
Radius: 127.24991
Rotations: 20.60369
Detected Circle Gesture:
Radius: 127.24991
Rotations: 20.60369
Detected Circle Gesture:
Radius: 127.24991
Rotations: 20.60369
Detected Circle Gesture:
Radius: 127.24991
Rotations: 20.60369
```

Pretty simple, right? You'll notice that the more circles you make at once with your hand, the higher will be the number of rotations detected by the Leap. I'm sure there are plenty of creative uses for this in applications...

Some (albeit minor) limitations to keep in mind

While the Leap Motion Controller is able to track all of your hands and fingers with remarkable accuracy (as long as you stay within the field of view, of course!), there are some limitations that need to be kept in mind when developing. The list is, fortunately, by no means exhaustive (and there are probably a few other minor caveats I missed while writing this, too, as this is all based on experience).

Upside-down hands can be a problem!

The Leap sometimes has trouble detecting hands that are upside down. If a hand enters its field of view upside down, the Leap will take the best guess and assume that the hand is right-side up. If a hand starts right-side up, however, and then flips upside down, then the Leap will usually correctly detect it as upside down.

However, fear not! The new Skeletal Tracking API from Leap Motion helps mitigate these issues, thanks to its, well, skeletal tracking. Of course, it's still possible to confuse the Leap Motion Controller when you place a hand upside down into the field of view—but if this happens, the Leap Motion API will automatically detect this mistake after a few seconds and correct it.

Needing too many hands is a bad thing

More obvious than other limitations, it's not practical to require more than two hands in the field of view of the device at any given time. Doing this can cause the device to (albeit rarely) confuse individual hands and fingers or even not see them at all (for example, if one hand was above another). Fortunately, needing more than two hands within the field of view should be a very rare thing...right?

Differentiating fingers can be fun!

This used to be a problem in the days that predated the Skeletal Tracking API from Leap Motion, but no longer! Thanks to (rapid) advances in tracking and image analysis technologies, developers are now able to get fingers on a given hand by name—heck, we can even get the individual *bones* in the fingers!

Of course, for the purposes of completeness, I believe it will still be nice to talk a little bit about what developing for the Leap Motion Controller was like in the...old days, as it were.

In the old days, there was no reliable way to distinguish between the different fingers on a hand (such as the thumb, pinky, or index finger); they were all just *fingers*. It was, however, possible to take a guess as to which finger was which by doing some basic math and ordering the fingers by, say, their *x* axis locations.

This process is best illustrated by the following code, taken from one of my Leap projects from long ago, which you can copy-and-paste into a file and use as a Leap listener:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import com.leapmotion.leap.*;

public class LeapListener extends Listener
{
    //Finger enumeration to make the code more readable.
    public enum kFingerName
    {
        THUMB(0), INDEX(1), MIDDLE(2), RING(3), PINKY(4);

        private int value;

        kFingerName(int newValue) { value = newValue; }

        public int getValue() { return value; }
    }

    //Comparator to sort the finger vectors from left to right.
    class FingerTipVectorXComparator implements Comparator<Finger>
    {
        public int compare (Finger finger1, Finger finger2)
        {
            return (int) (finger1.tipPosition().getX() -
            finger2.tipPosition().getX());
        }
    }

    //Function: getNamedFinger
    /**
     * Detects the specified finger, by name, on a hand and then returns it.
     */
}
```

```

* @param hand: The hand which contains the finger we're looking for.
* @param fingerName: The name of the finger that we're looking for.
* @param left: Set to true if this is <strong>assumed</strong> to be a
*             left hand instead of a right one.
*
* @return the Finger on the passed hand specified by fingerName.
*/
public Finger getNamedFinger(Hand hand, kFingerName fingerName, boolean
left)
{
    //Finger vectors.
    List<Finger> fingers = new ArrayList<Finger>();

    //Retrieve the vector coordinates for all five fingers on the passed
hand.
    for (Finger finger : hand.fingers())
        fingers.add(finger);

    //Using our custom comparator, sort the list of vectors from "left" to
"right" based on their X-axis.
    if (left)
        Collections.sort(fingers, Collections.reverseOrder(new
FingerTipVectorXComparator()));
    else
        Collections.sort(fingers, new FingerTipVectorXComparator());

    //Return the specified finger if it is contained in our array.
    if (fingerName.getValue() + 1 <= fingers.size()) return
fingers.get(fingerName.getValue());

    //Return an empty finger if the specified one was not contained in our
array.
    else return new Finger();
}

//OnFrame method.
public void onFrame(Controller controller)
{
    Frame frame = controller.frame();

    if (!frame.hands().isEmpty())
        System.out.println("Right-most hand Ring Finger data:" + "\nTip
Position (X|Y|Z): " +
            getNamedFinger(frame.hands().rightmost(), kFingerName.RING,
true).tipPosition().getX());
}
}

```

OK! Allow me to explain how this code works...

The first segment of the code (`public enum kFingerType...[]`) defines a finger type enumeration that allows us to specify which finger we're looking for later on in the code. I won't go over the specifics, as they aren't entirely relevant to this example.

The next segment of code (`class FingerTipVectorXComparator...[]`) is similar to our

kFingerType enumeration; it serves as a utility to help us sort the fingers later on in the code. Again, I won't go over the specifics, as they are not relevant to this example.

Now, for the meat of the code, the `getNamedFinger` function—time to go over some specifics! It takes three main values when called; `Hand`, `kFingerType`, and `Boolean` telling it whether it's parsing a left or right hand. Now, let's break down the contents of the function:

```
List<Finger> fingers = new ArrayList<Finger>();  
  
for (Finger finger : hand.fingers())  
    fingers.add(finger);
```

These first few lines create a basic list of the (presumably five) fingers on the passed `Hand` object. Let's take a look further:

```
if (left)  
    Collections.sort(fingers, Collections.reverseOrder(new  
FingerTipVectorXComparator()));  
  
else  
    Collections.sort(fingers, new FingerTipVectorXComparator());
```

These next four lines then order the `fingers` list based on the individual `x` axis of each finger, ordering them from left to right if it's a right hand and from right to left if it's a left hand, as specified by the `Boolean` type passed to the function. You'll notice that we use the `FingerTipVectorXComparator` class to allow comparisons between them and the sorting of the complex `Finger` data type. Let's move further:

```
if (fingerName.getValue() + 1 <= fingers.size()) return  
fingers.get(fingerName.getValue());  
  
else return new Finger();
```

Finally, these two lines parse the `fingers` list for the finger we want, as specified by the `kFingerType` enum passed to the function.

With that out of the way, this brings us to the last block of code, the `onFrame` method. The code is quite simple and only prints out tracking data for the rightmost hand's (with respect to the Leap's FOV) ring finger. Try it out! When it is run, the output looks similar to the output shown in the following screenshot:

```
<terminated> SimpleLeap [Java Application] C:\Program Files (x86)\Java\jre8\bin\javaw.exe
Hand 1 Ring Finger data:
Tip Position (X|Y|Z): 25.619003
Hand 1 Ring Finger data:
Tip Position (X|Y|Z): 25.414427
Hand 1 Ring Finger data:
Tip Position (X|Y|Z): 25.18283
Hand 1 Ring Finger data:
Tip Position (X|Y|Z): 24.871561
Hand 1 Ring Finger data:
Tip Position (X|Y|Z): 24.521173
Hand 1 Ring Finger data:
Tip Position (X|Y|Z): 24.123919
Hand 1 Ring Finger data:
Tip Position (X|Y|Z): 23.666086
Hand 1 Ring Finger data:
Tip Position (X|Y|Z): 23.19691
```

Now, in this code, we're assuming that the passed hand had five (or more, if that's possible) fingers, allowing the `kFingerType` enumeration to map perfectly to the size of the `fingers` list when parsing for fingers. In practice, this method almost always works... the only case where there might be trouble is when all your five fingers are out but one of your fingers isn't detected for some reason; in this event, it might mistake your thumb for your index finger, your pinky for your ring finger, and so on—this is very rare, though.

So, as you can see, there are some ways of getting around this problem. Nowadays, though, we can just type:

```
Finger indexFinger =
hand.fingers().fingerType(Finger.Type.TYPE_INDEX).get(0);
```

Assuming `hand` is an instance of the `Hand` class, the preceding code will simply assign the first (and hopefully the only) index finger attached to a given hand to the `indexFinger` object.

Lack of support for custom gestures

To be honest, the lack of custom gesture support is (in my mind) not really a problem for or a drawback of the Leap Motion Controller—it's actually a good thing. Why, you might ask? For a moment, imagine if every application out there was responsible for defining its own set of gestures—each time you download a new application, you'd need to learn a (potentially) entirely new set of gestures. That's no fun.

Instead, by standardizing the usage of gestures and restricting developers to predefined ones, Leap Motion guarantees that different apps will offer similar user experiences. This allows users to transition from application to application with minimal difficulty, needing to memorize only a few predefined gestures.

The main point here is this: avoid using custom gestures!

That's it! Fortunately, there are very few common limitations and the ones that exist can almost always be overcome with a little bit of extra code or some minor redesigns to your user experience.

Summary

In this chapter, we went through the more complex features of the Leap as well as some of the things that are going on in the background.

We talked about the Leap's field of view and how it interacts with the `InteractionBox` class, before making a simple application demonstrate how vector coordinates can be normalized using this class. We then went over gestures and tools and how they work; tools are basically really long, straight fingers, and gestures are unique patterns made by a user's hands and fingers. We then finished off this chapter with a brief look at some more common limitations of the Leap motion device and its API and how they can be overcome or avoided in the first place.

In the next chapter, we will dive into what is arguably the most important part of making an application with the Leap: the user experience!

Chapter 3. What the User Sees – User Experience, Ergonomics, and Fatigue

Armed with the information to grab user input and make use of it, it's time to look at the other half of your application: the user. In this chapter, we'll take a break from the Leap-Motion-specific jargons and discuss a slew of the common things that developers can often forget, including user fatigue, ergonomics, and the overall experience of the application—all the while keeping in mind how these apply to the Leap.

In this chapter, we'll be covering the following topics:

- When to use the Leap (and more importantly, when not to)
- The Leap Motion user experience guidelines
- A note on ergonomics and user fatigue
- A case study: the Artemis Quadrotor Simulator

Note

This chapter is sprinkled with periodic *Fun facts* that offer high-level and entry-level factoids about scripting and programming for your reading pleasure.

When to use the Leap (and more importantly, when not to)

You just finished unpacking your shiny new Leap device and installing the developers' software development toolkit. You fire up the Diagnostic Visualizer (don't be afraid to admit it; we all know it's a fun way to pretend to be doing real work when the boss comes by). And then it dawns on you, "I could make some really awesome and amazing applications with this"—applications such as gesture-recognizing typewriters or maybe a controller for your robots...yes, robots.

Hold that thought.

This is all fine and good but what exactly should you use the Leap for, aside from those times when you want to look at three-dimensional representations of your hands?

Well, as an example (I by no means claim to have understood all the possible applications for this device), you can look at the applications for the Leap from two different angles; the first is as a replacement for preexisting controls such as joysticks, keyboards, and mice. The second, more interesting (in my opinion), is as an entirely new interface that allows the creation of applications that we've never even thought of before.

In the first case, where you're replacing controls such as joysticks, keyboards, and mice, you have to ask yourself, "Do I really need to replace this if it already works? Will it improve the experience my end users and operators have?" If the answer to either of these questions is no, then you might wish to rethink your strategy—the Leap is there to make a more intuitive experience for the user, not a more complicated one!

However, when the time comes to create an entirely new interface and experience, you're in good company—that's exactly what the Leap is for.

The Leap Motion user experience guidelines

If you've developed applications for end users before, you've probably heard of **user experience (UX)**. Oh, the user experience! Arguably, it's the single most important part of any Leap-driven application—heck, of any software application really!

Without a good UX, you'll have users tearing their hair out and/or punching the screen in frustration when even the seemingly simplest things they try to do don't go quite as planned. Or in a milder, less drastic case, your user will develop carpal tunnel syndrome. Either way, a good user experience is better than a bad one.

Note

Fun fact

Carpal tunnel syndrome develops when excess strain is placed on your hands, specifically the median nerve (the nerve in the wrist that allows feeling and movement to parts of the hand). Carpal tunnel syndrome can lead to numbness, tingling, weakness, or muscle damage in the hand and fingers. Not good!

Over the course of the next few pages, I've listed and expanded on the guidelines for user experience design given by the official Leap Motion developers. The original texts from the development team are encased in quotes, for your reading pleasure.

Now, without further ado, here are the texts from the official guidelines with some added commentary:

“Keep in mind that symbology can be difficult to learn and memorize.

Avoid forcing users to learn complex hand gestures to interact with your application.”

In other words, avoid utilizing complex Harry Potter-esque hand motions within your application. While they're really cool from a developer's standpoint, your users are going to hate you. Of course, there are exceptions; if you're writing something such as a sign language interpreter or perhaps a wizard/spell-casting emulator, go right ahead—just don't make the user draw an ampersand (&) or something to perform a simple task such as navigating to the next page or confirming a dialogue.

“Instead, draw inspiration from physical interaction and real-world behaviors.

The more physically inspired interactions are, the less training a person needs and the more intuitive and natural your application feels.”

For example, if the user needs to grab some kind of object, a ball for example, don't make them *tap* or *swipe* the ball to pick it up—have them literally grasp it. This exact task can be achieved with a few simple steps:

1. Check whether the user's palm coordinates are within the grabbing range of the ball.
2. If it is, using the new Skeletal API, check whether the hand's *grab strength* is within a certain threshold, such as 0.7 or higher. If you're not using the Skeletal API, an alternative will be to check whether the hand's sphere radius is within a given threshold.
3. Finally, if all these conditions are true, begin moving the ball in relation to the user's hand.

Little things such as these can make a world of difference to your users, even if they do add a bit more complexity to the programming side of the application.

“Don't feel constrained by the limitations or inconveniences of the real-world—this is your world.”

Interaction doesn't have to be the way it has always been. It can be any way we imagine it to be. Why force the user to reach all the way out and grab an object? Why not have the object reach back?—Give them “the force”!”

There's not a whole lot to say about this one—it's a fairly obvious guideline, right?

Having said that, just make sure that if you do something, you make it consistent! If your world has different rules for interaction, make sure that you implement them in a predictable fashion. If you decide to give your users “the force,” make sure that they can always use it and don't find themselves stuck wondering why they can't grab the object.

“The user should feel as if their intent is amplified rather than subdued or masked.

For example, users often like their movements to be amplified when using a mouse (i.e. they don't need 10 inches of mouse movement to move 10 inches on screen). For gestural interactions, amplifying or exaggerating responses can have an even more positive result. Keep in mind that some people are more sensitive than others, so link this exaggeration to a sensitivity setting for users to modify this effect to their preference.”

This one is a pretty basic concept but an important one nonetheless. As a general rule, the Leap's coordinate system maps pretty well to the virtual realm...but even a normalized box can only get you so far.

With a little bit of multiplication (or division, if you're one for the decimal system), you can very easily modify the Leap Motion Controller's inputs and exaggerate or subdue them. This is particularly useful when you want to incorporate a user-adjustable sensitivity feature into your application, allowing users to configure how much (or little) they want to move their hands and fingers to achieve a given task.

“Concentrate on giving the user dynamic feedback to their actions. The more feedback they have, the more precisely they can interact with your software.

For example, the user will need to know when they are “pushing” a button, but can

be more effective if they can see when they are hovering over a button, or how much they are pressing it.”

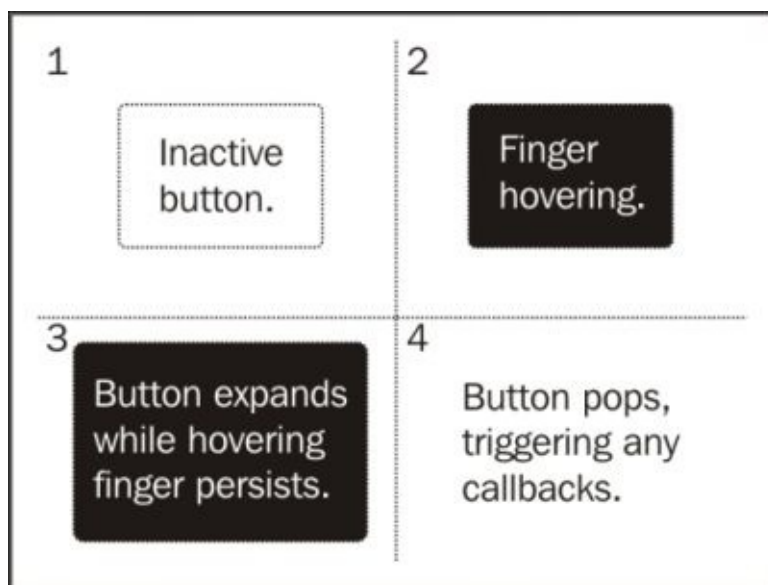
This one is a biggy! In fact, I think I spent more time working with this guideline than anything else on some of the applications I’ve made.

Think about it; how does your user know if they’re pushing a button? They can’t exactly feel something because their finger is being waved around in mid-air and then being projected into a virtual realm...so, what do you do?

For those of you who are inclined toward modern-day touch devices such as tablets and smartphones, you’d probably jump straight to basic haptic feedback (vibration) and visual feedback (transparent circles where the user has touched and so forth). Unfortunately, at the moment, we don’t have any technology at hand (pun intended) to enable physical feedback for something you’re not physically touching. So, what do we do?

We give the user as much visual feedback as possible.

The most common form of visual feedback is in the case of a button, of course. The method I use to inform my users that they’re pushing a button is to slowly expand the size of the button until it *pops* (triggering the button-click event), kind of like blowing up a bubble. Take a look at the following diagram for a better explanation:



Let’s take a look at the state of the button throughout this process, starting at the top-left corner of the diagram:

1. The button is inactive with no fingers pressing it.
2. Finger enters the button area, causing it to change color.
3. Finger remains in the button area, causing the button to expand.
4. The button pops and triggers any callbacks or events.

Of course, there are many different ways to supply visual feedback to your user—anything

from gradually expanding buttons or onscreen cursors to virtual representations of the user's hands.

Just remember: visual feedback is one of the most powerful tools in your arsenal in order to improve the user experience!

“Onscreen visuals (such as representations of hands, tools, or digital feedback) should be simple, functional, and non-intrusive.

The user should not be distracted from the task by their tools or environment. Decoration should not distract from your purpose.”

This guideline is ever-so-slightly less important than the other ones, but it is still important nonetheless. It's good to have a pretty (and somewhat detailed) user interface, but don't make the representation of a hand, finger, or other element so complicated and detailed that it draws away from the main purpose of your application.

“Require more deliberate actions for destructive or non-reversible acts than for harmless ones.

Subtle gestures should be reserved for subtle actions. Conversely, an act such as closing an application or deleting a file can be a non-reversible event requiring a more deliberate action. Double check with the user when unsure, such as a prompt for confirmation.”

This is a critical note—don't ever make it easy to do destructive, irreversible things such as data deletion or save file overwrites. Then again, don't make the gestures to perform operations such as these so impossible that your user can't remember them; just exercise caution.

This deserves even more attention due to the very nature of the Leap Motion Controller—on a touch screen (tablet, smartphone, and so on), you have the option of simply not touching the screen, thereby avoiding the possibility of triggering any undesirable actions. This isn't the case with the Leap Motion Controller because it's always watching and waiting for input, meaning developers have to exercise a higher level of caution when designing and implementing *destructive* actions.

“Provide a clear delineation and specific sense of modality between acts of navigation and interaction, unless both are simple or one is handled automatically (or with assistance). Mixing the two in a complex situation can lead to confusion or disorientation.

For example, moving an object while having the user simultaneously position their viewing angle inside a 3D environment is inherently difficult. However, if the viewing angle moves automatically in response to the user's movement, then working with the object is easier. Likewise, when navigating a large data set the user will want the view to move easily, but when highlighting a portion of the data the view should remain still.”

Essentially, if your application is three-dimensional (in many cases, it will be), make sure that a lot of the camera and viewing work is handled automatically in a meaningful way that responds to current user input.

From personal experience, I can tell you that trying to fly a virtual quadrotor with one hand while moving the camera with another is a very difficult task. Also, it's not much fun or practical either.

To illustrate this better, let's take a look at modern first-person shooter movements and camera controls. Typically, you will use two joysticks at any given time—one for moving around, and one for looking around. It can take some people a little getting used to at first, but these controls are very simple and easy to use because you're only thinking about managing two fingers.

Now, let's try doing the same thing, except with your hands—one for moving around, and one for controlling the camera. If you try, you will notice that you are suddenly using a lot more brain power (and energy) because you're trying to coordinate and manage two hands instead of two thumbs. While some people can definitely pull it off, none of the people I've had test my applications were able to pick it up and play without a bit of training and practice beforehand, defeating the purpose of an intuitive interface.

“Overall, imagine that your user is faced with no instructions or tutorials on how to use your application.

Strive at all costs to make their first intuitive guesses the right ones. Where appropriate, create more than one proper way to do something.”

This is, perhaps, the most important guideline listed by the Leap Motion crew.

If your application can be used without a tutorial, based purely on intuitive guesses, you've succeeded. The examples of this include Apple and Dropbox; both are famous for their simplicity and the ease-of-use that their platforms possess. Always strive to simplify the interface so that obvious things can be done in obvious ways: tapping to trigger buttons, grasping and grabbing to manipulate objects, swiping to navigate pages, and so on.

A lot of the things we just covered were relatively basic concepts but following and remembering them while developing can mean the difference between a happy set of users and...potentially...no users at all. Preferably, you want a happy set of users!

Ergonomics and user fatigue

So, we just finished covering the underlying, basic concepts of what makes a good, solid user experience. However, we didn't cover two things that also heavily impact the user experience: ergonomics and user fatigue. I won't spend too much time on these concepts, as they can be highly situational, but I thought it might be good if we discussed what they are briefly.

Ergonomics

Have you ever played a game for a while (perhaps 10 hours straight) on a keyboard or gamepad? Perhaps, afterwards you noticed that you had severe wrist or hand cramping and pain? This is the result of bad ergonomics.

The longer you have a user hold an interesting position or perform complex actions, the more strain it puts on their hands, causing cramping over time. This can be caused by simple things such as having the user twist their palms about the z axis (in other words, rotate their entire hand to the left or right) or more complex things such as weaving complicated gestures to perform relatively mundane actions.

I'm not an expert at human physiology, but I will say this: avoid having the user perform actions such as those previously listed whenever possible, or else you run the risk of making the user experience quite uncomfortable!

User fatigue

You could say that the concept of user fatigue is less obvious than that of ergonomics and is a bit more exclusive to human interfaces such as the Leap device. Sometimes, though, it can directly affect the ergonomics (and vice versa)!

In its simplest form, user fatigue is caused by having the user perform a given action for an extended period of time, effectively tiring them out. This can be demonstrated by some apps where you have to hold your hand out to control the throttle on a ship or, in the case of my quadrotor simulator (which we'll discuss next), control the direction of movement of some kind of object. Sometimes, this is unavoidable!

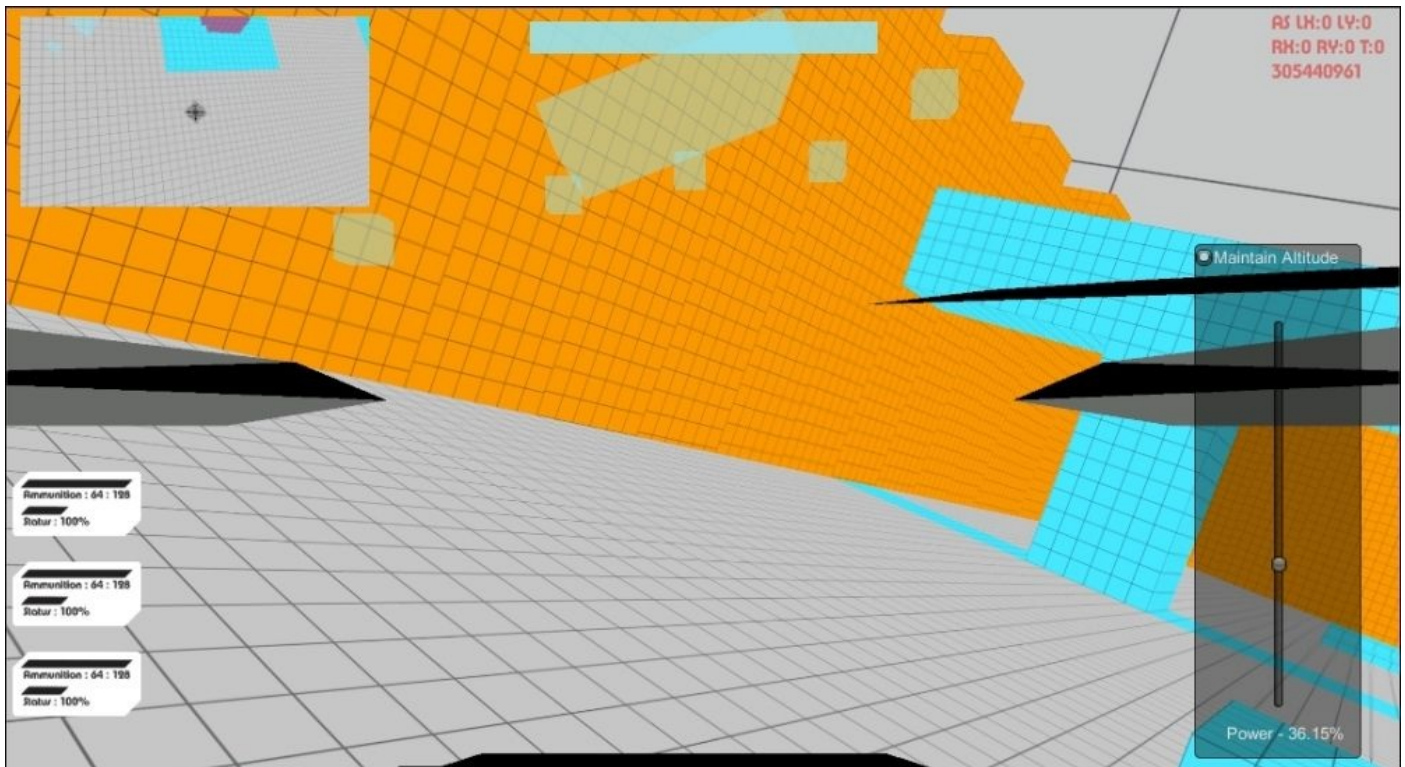
However, when possible, try to avoid actions that induce user fatigue; favor short, quick interactions with the screen as opposed to long, continuous actions. Giving the user the ability to remove their hands from the field of view between interactions without affecting the application can also be a very important tool when attempting to mitigate user fatigue.

If your application requires uninterrupted precision control for long periods of time, it's possible that the Leap is not the correct choice for your application's user interface.

Always keep an eye out for how you can make your application more comfortable and less tiring for the user. However, don't try to accommodate the user so much that it detracts from the functionality of your application—that's no good either!

A case study – the Artemis Quadrotor Simulator

Throughout this book (and chapter), you've probably noticed that I've made a few comments here and there about one of my first Leap projects, a quadrotor simulator called Artemis. To complete this chapter, I thought I'd spend some time talking about how I developed the user experience for Artemis using an assortment of methods. Visual feedback took the form of many things in Artemis, including hands, height meters, fuel gauges and copies of the Leap's tracking data (for debugging), as shown here:



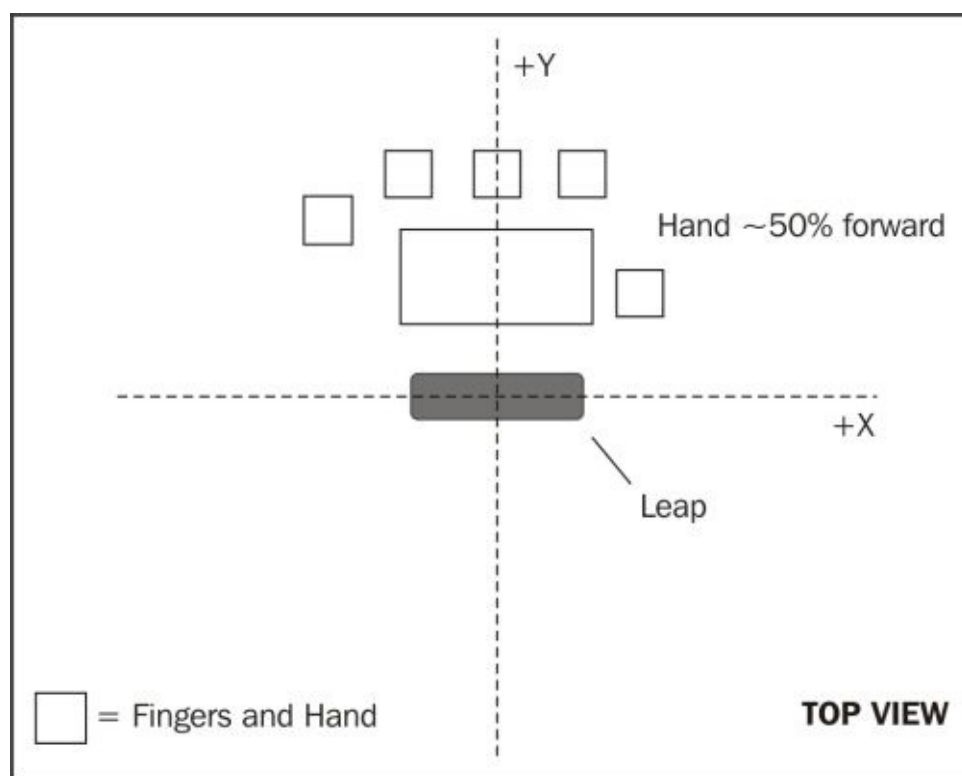
Play testing and why you should do it

First, play testing or hallway testing is the art of grabbing an unsuspecting friend or co-worker from the Internet or proverbial hallway and making them sit down in front of your application to, well, use it. If you've developed any frontend applications before, you're probably already familiar with this concept. Trust me, it helps a lot.

By having your friends and coworkers (who have potentially never even seen your application) interact with your program, you can see how people try to use it versus how you intended them to use it. I'll repeat myself yet again, saying that this is a fairly basic concept of user experience, but it's still a concept. Also, it applies to the Leap.

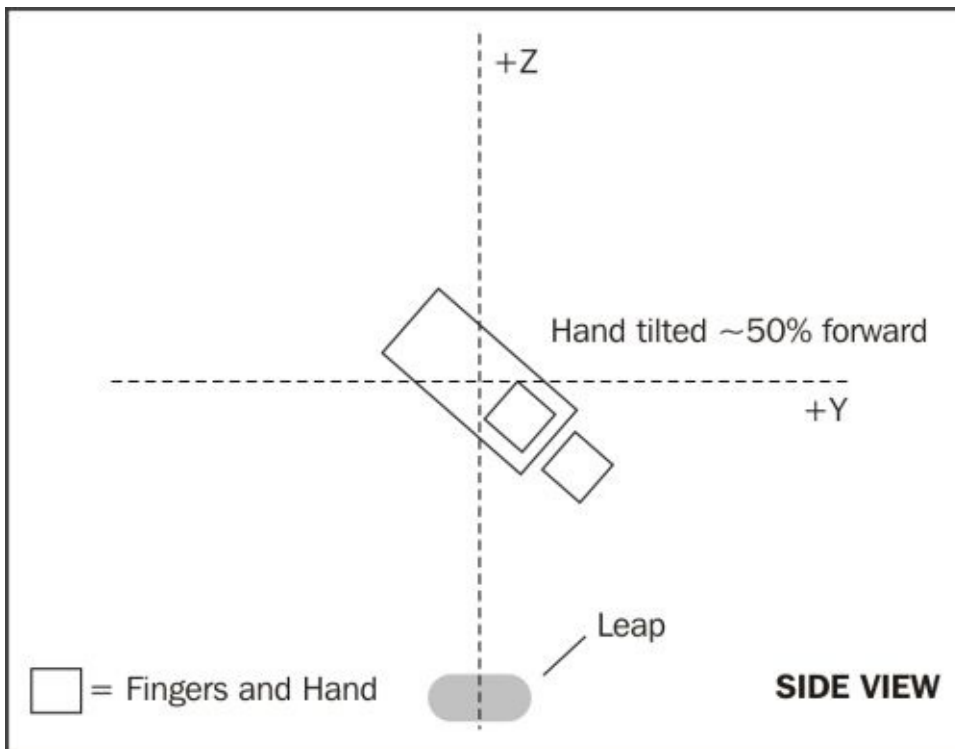
When performing play testing, it's important that you don't tell your testers how the application works. This will allow you to observe what they try to do in comparison to what you want them to do. Do they get frustrated? Do they try to do things you didn't anticipate or code? These are all important things to know when creating a user experience.

With Artemis, I set out to make an application that would allow anyone to control a virtual quadrotor with just their hand. Originally, I used the x , y , and z coordinate system on the Leap to control the motion of the quadrotor, as seen in the following diagram:



However, this didn't work quite as well as I'd hoped. When my friend tried to take control over the quadrotor, it went just everywhere except where he wanted it to go. What I soon noticed was that every person I put in front of the simulator would try to tilt their hand to control the motion of the quadrotor, as if the virtual quadrotor became a literal extension of their hand.

After seeing the same behavior over and over again, I modified the controls to use the pitch and roll of the hand instead so that if a user's hand were to tilt forward, the virtual quadrotor would tilt forward, and so on. You can see a diagram of how this works here:

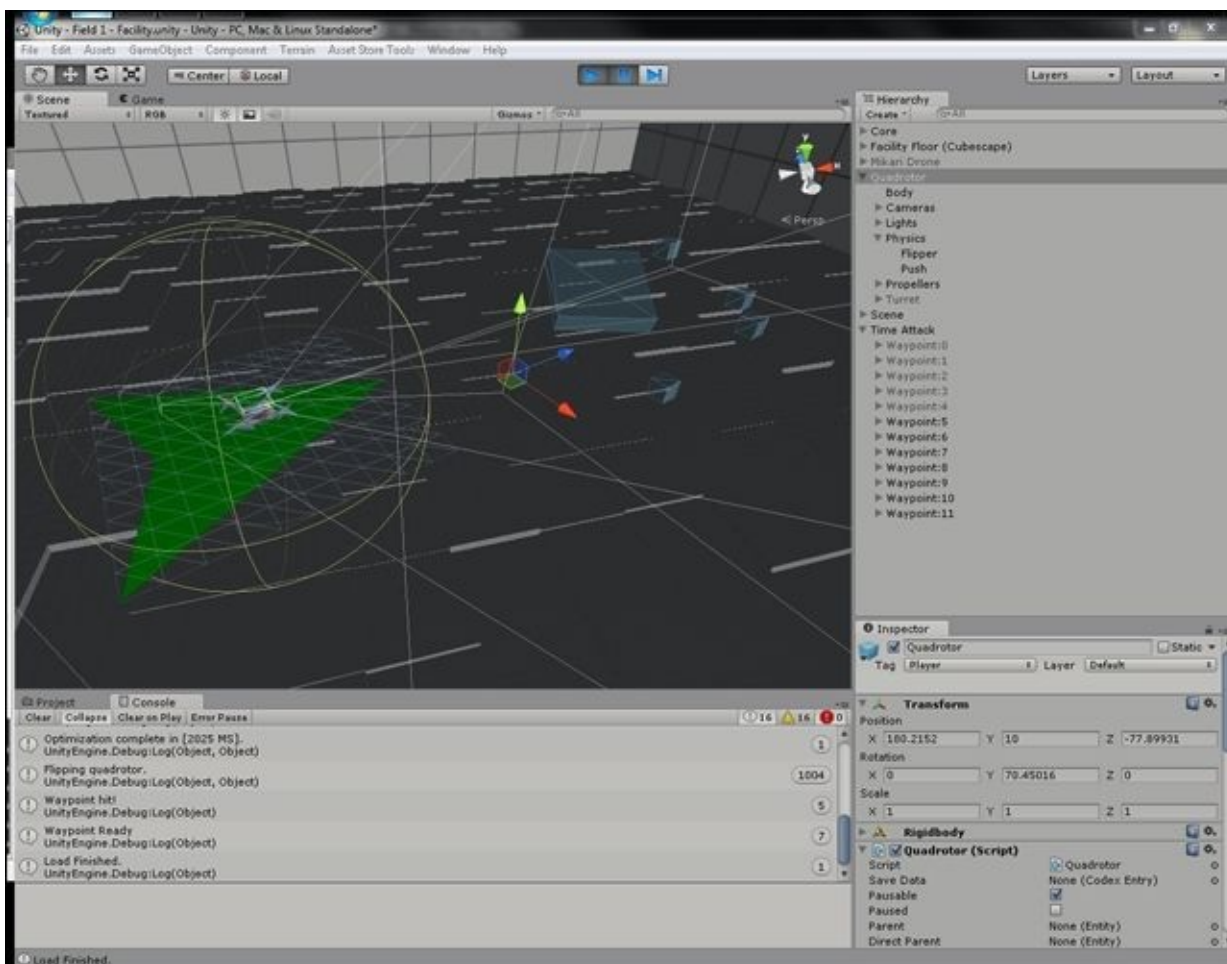


After making this change, I asked my friends to try using the simulator one more time. This time they were able to navigate an obstacle course just fine and found it natural on the first try!

As you can see, play testing can help you to discover tiny issues that impact the experience of the user tremendously, allowing you to fix these issues before making a release of your application.

Providing as much visual feedback as possible

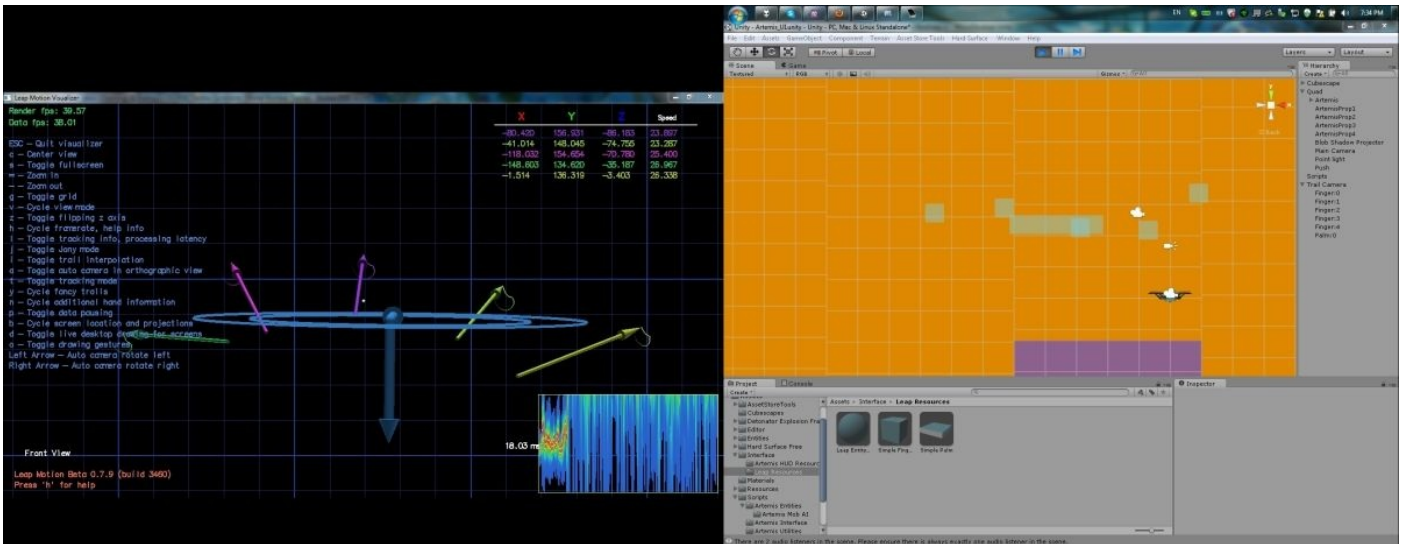
When I first began creating Artemis, I was trying to make a controllable three-dimensional version of a quadrotor to test my Leap with, as I didn't want to crash one of my real ones (after all, watching a thousand dollars of hardware plummet viciously to the ground isn't a good time for anyone, especially the owner). Thus, the visual feedback of what was going on in my simulator's brain was a key concept during development! You can see an assortment of the various things used in a 3D application like Artemis to provide feedback; a virtual hand, a giant green radar arrow, and more, in the following screenshot:



At this point in time, I had seen a few Leap applications here and there that used different methods to show user input; the more 2D-oriented apps used little dots on the screen to represent fingers, whereas the more 3D-oriented ones used simplistic models of the user's entire hand on the screen.

I opted for the 3D-oriented approach.

After a few hours of testing and debugging, I finally had a working solution to render hands on screen, as seen in the following screenshot:

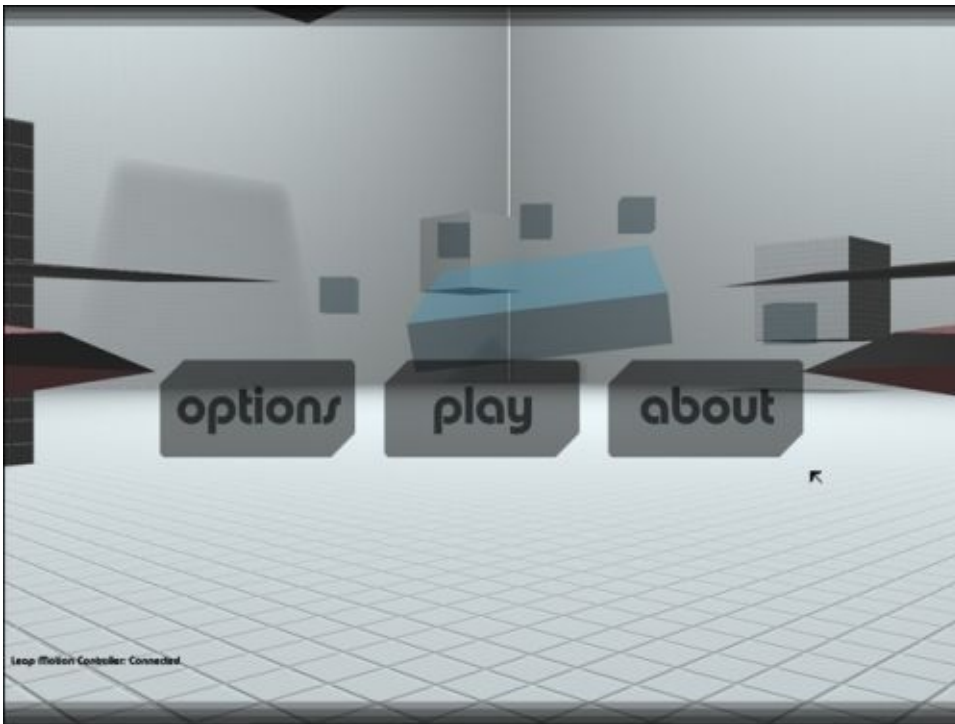


Allow me to explain the preceding screenshot. On the left-hand side of the screen, you can see frozen tracking data for a person’s left hand within the Leap’s diagnostic visualizer. On the right, you can see five blue cubes and a larger blue rectangle, which represent the same left hand’s fingers and palm, respectively—except in the virtual game space.

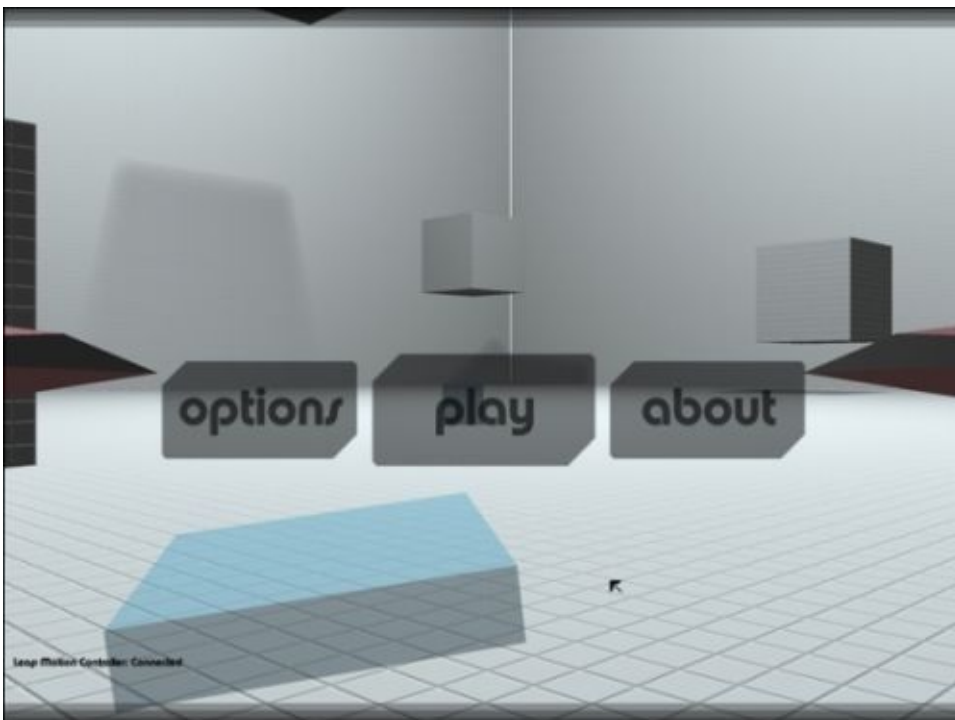
After adding this feature, I sat one of my friends down in front of the computer yet again and asked them to fly the virtual quadrotor around. They were now able to control the quadrotor even better, thanks to the visual feedback of being able to see what the Leap is seeing.

Before I added this feature, we would tilt our hands around randomly, hoping the quadrotor was seeing our real hands and moving in response. After this addition though, the user’s confidence shot through the roof, as they were able to see what the Leap was seeing and know that the simulator was reflecting the desired actions.

Another feature that I added towards the end of the development was visual feedback on buttons. We’ve discussed how you can apply visual feedback to buttons in this chapter already, but I’ll go ahead and reiterate. By notifying users when they’re interacting with a button, you can give them increased confidence and prevent them from doing things they don’t necessarily want to do. Now, refer to the following two screenshots:



In this first screenshot, you can see the Start menu of Artemis with my left hand in the field of view. At this moment, none of the buttons are engaged and everything is idle.



In this second screenshot, you can see the same Start menu. However, now I've moved one of the fingers of my left hand over the **play** button and, if you look closely, you can see it starting to expand—it will continue to expand until it pops, so to speak, taking the user to the actual game.

As you can see, this simple visual addition can tremendously improve the feedback that

the user has.

Of course, some more advanced users might not want to wait for a full second to trigger a button. In this case, you might consider implementing a configuration setting that shortens the delay to half a second or less. Ultimately, the decision lies with the developer...and the play testers!

Note

Another thing

It's always a good thing to notify the user if their Leap Motion Controller is plugged in and detected by your application. If they have no way of knowing this information, they might think that your application is just plain unresponsive, even if there are other forces at work.

In the preceding screenshot, you can see a little indicator in the bottom left side (if you look closely) that says **Leap Motion Controller: Connected**; in the event that the device gets disconnected, this text changes to **Leap Motion Controller: Disconnected** to notify the user that something is amiss.

That's it – for now!

This pretty much summarizes the user experience aspects of Artemis—between play testing, rendering users' hands, and providing visual feedback for button presses, I was able to satisfy the requests of everyone who had ever picked up and played it.

So, remember: always give the users some kind of relevant visual feedback and always make sure that your interface is as *natural* as possible. When in doubt, grab some friends and do some play testing or hallway testing, whatever you prefer to call it!

Summary

In this chapter, we covered a series of more abstract concepts related to the Leap, including the user experience, ergonomics, fatigue, and even a brief case study.

You learned about the different guidelines that Leap developers have in place for designing a user experience and the author's interpretation of these guidelines based on past experiences. We then briefly covered why it's important to pay attention to ergonomics and user fatigue without compromising the functionality of an app. We finished this chapter with a brief case study of the Artemis Quadrotor Simulator and how its user experience was developed.

In the next chapter, we're going to begin writing a two-dimensional drawing application using all the things we've learned up until now!

Chapter 4. Creating a 2D Painting Application

As you are familiar with all the essentials of the Leap Motion device, you should now have mastered the basic concepts of developing with it. How about we apply that knowledge to make a two-dimensional painting application? In this chapter, we'll create a painting application with the Java programming language and the Leap Motion API called **Leapaint**. Here we go!

In this chapter, we'll be covering the following topics:

- Laying out the framework for Leapaint
- Creating the graphical frontend
- Interpreting Leap data for rendering on the graphical frontend
- Testing the application
- Improving the application

Note

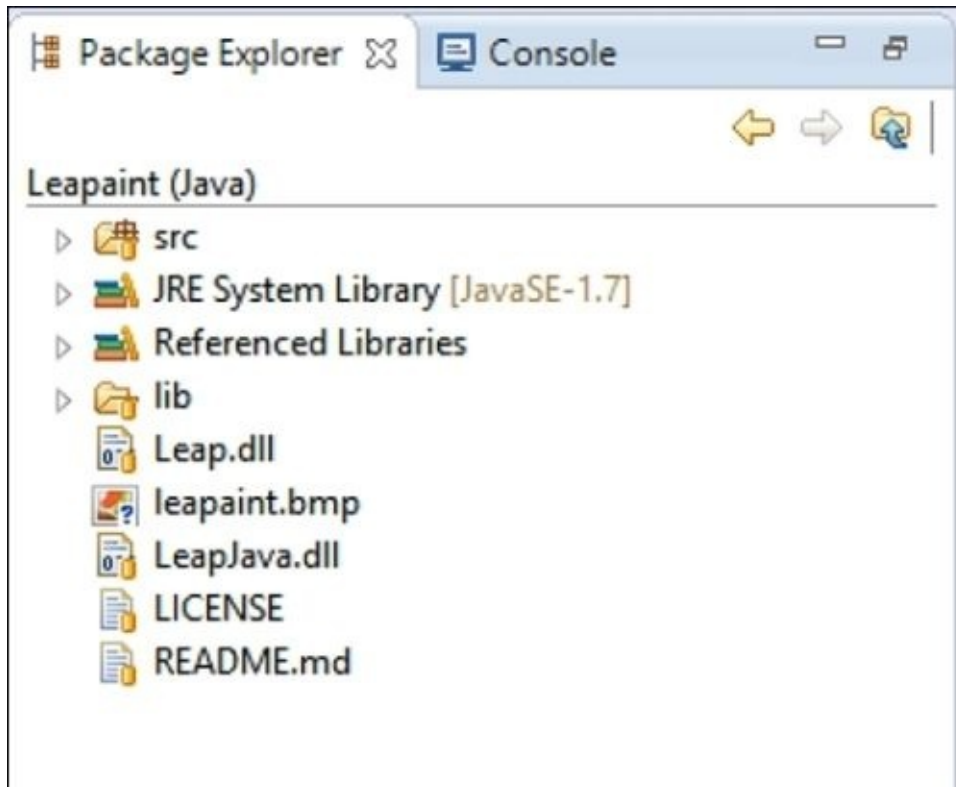
This chapter involves writing an application entirely in the Java programming language. Many of the user interface items will not be directly portable to other languages, as Swing is heavily utilized. However, the core logic and design is easily transferrable to languages such as C/C++, Python, and so forth.

You can find more information about the Java Swing API on Oracle's official website at <http://docs.oracle.com/javase/7/docs/technotes/guides/swing/>.

This chapter is sprinkled with periodic *Fun facts* that offer high-level and entry-level factoids about scripting and programming for your reading pleasure.

Laying out the framework for Leapaint

We're going to jump straight into working in this chapter! Have a look at the following screenshot, which shows the Leapaint package:



The first step in creating Leapaint will be laying out the baseline framework of files (or classes, if you will) that we'll be using. This first section will involve writing skeleton versions of the three primary classes in Leapaint; these classes will contain all the variables and functions that the final ones contain, without going overboard in defining what the functions actually do.

Before we begin writing any code, create the following three files in Eclipse and place them in a package under the `src` directory in a new Java project, as seen in the preceding screenshot:

- `Leapaint.java` (the main file)
- `LeapaintListener.java` (the Leap interface)
- `LeapButton.java` (a special class for Leap-enabled buttons)

Note

If you are unsure as to how to do the things listed here, refer back to the first chapter on setting up Java projects in Eclipse. Keep in mind that we're making an entirely new project, so don't reuse the one from [Chapter 1](#), *Introduction to the World of Leap Motion*.

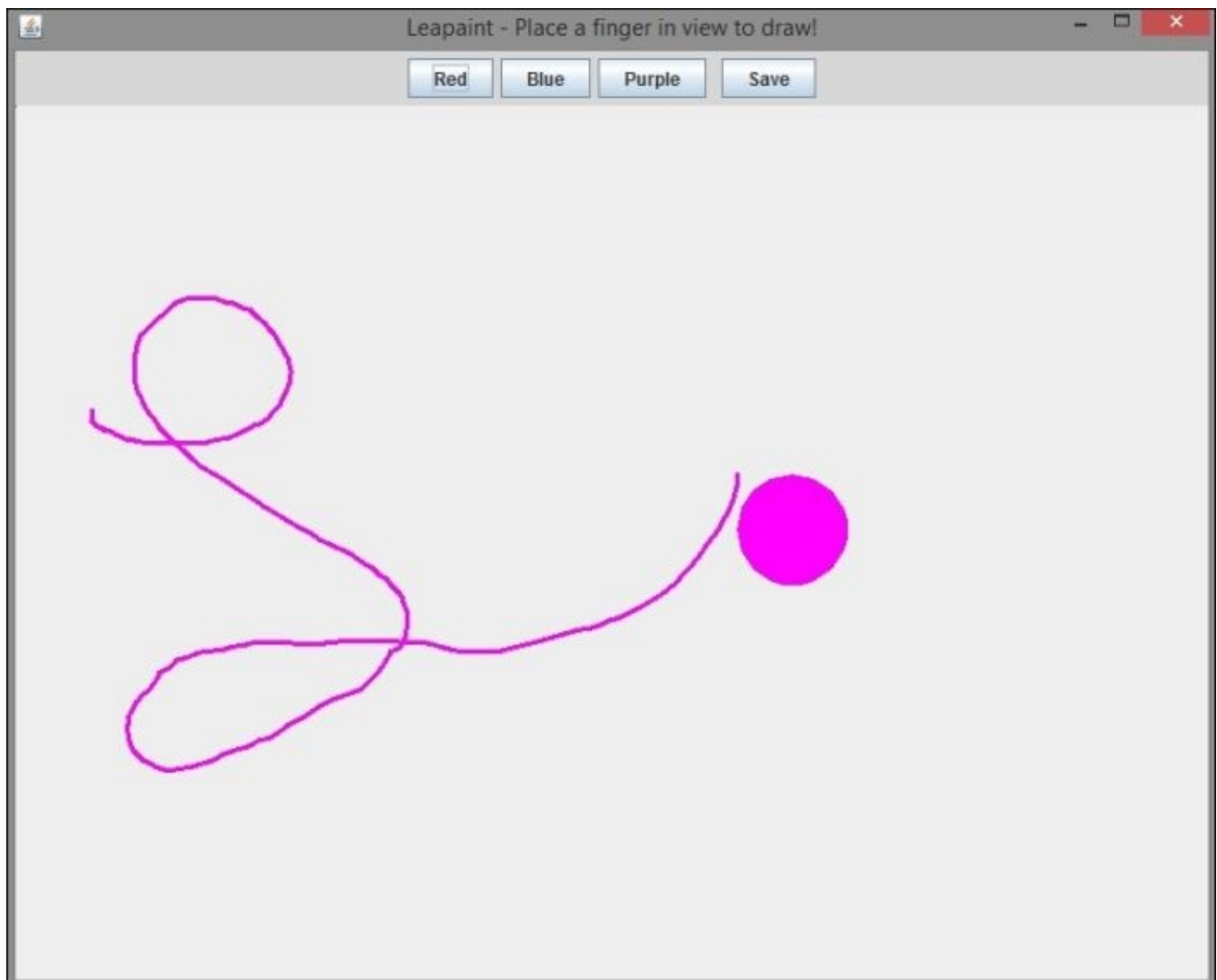
With those files created, let's go ahead and talk about what each of them will do when we're done writing code.

The `LeapaintListener` class, as the name suggests, is our Leap Listener implementation for this project. It will be designed to read values from the Leap Motion Controller, normalize them based on the `InteractionBox` class, and then forward their coordinates to the main `Leapaint` class. It is also responsible for checking whether fingers are hovering over buttons and then triggering the said buttons if they are.

The `LeapButton` class represents, as you might expect, a Leap-enabled button. This class renders buttons on screen that are capable of visually *expanding* and then *popping* when they are triggered. Technically, this class does not contain any Leap Motion code; instead, it relies on an external class or function (the `LeapaintListener` class in this case) to trigger and/or stop the expansion of the button.

Finally, we have the `Leapaint` class, which is our main class in this project. This class controls the **graphical user interface (GUI)** of our application, allowing the various buttons and lines that we draw to be rendered onto the user's screen. In addition, it initializes and coordinates the `LeapaintListener` and `LeapButton` classes. This class is the *glue* that holds our project together.

When you put all of these classes together, you will have a working application that allows a user to use their fingers to draw on screen, just like in the following screenshot:



Now then, let's go ahead and start off by writing the smallest class in the Leapaint project, LeapButton.

LeapButton.java

First up is the LeapButton class. This class extends the basic functionality of the JButton class in Java's Swing API by adding the ability to trigger visual *inflation* and *deflation* of the button to allow for visual feedback when using the Leap to push buttons.

Let's start by writing the bare-bones file with all the data members and function definitions. When you finish writing, the file should look something like this (don't forget to add an appropriate package definition to the top such as package com.mechakana.tutorials):

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Rectangle;

import javax.swing.JButton;

public class LeapButton extends JButton
{
    //Expanding state of the button.
    private boolean expanding = false;

    //Original button size.
    private int originalSizeX, originalSizeY;

    //Button expansion multiplier; defaults to 1.5 times as big.
    private double expansionMultiplier;

    //Allow expansion?
    public boolean canExpand = false;

    //Constructor
    LeapButton(String label, double expansionMultiplier)
    {
        //Always call the superclass methods with Swing.
        super(label);

        //Assign values.
        this.expansionMultiplier = expansionMultiplier;
    }

    //Member Function: getBigBounds - We'll write this later!
    public Rectangle getBigBounds() { return new Rectangle(); }

    //Member Function: expand - We'll write this later!
    public void expand() { }
}
```

As you can see, this class is pretty simple—there are a few data members to keep track of basic state information and a constructor to define the name of the button.

Note

At this point, you're most likely going to see several warnings (and possibly errors) in

Eclipse that will tell you certain variables aren't being used—or possibly don't even exist! Don't worry about this, as we will be writing code later on that will make full use of these variables.

If you were to add an instance of this class to a `JFrame` class (more on this later), you'd already see a full-fledged button appear on screen; the Java Swing API has a way with simplifying a lot of the work for us developers.

However, before we get ahead of ourselves, let's proceed to writing the other two *skeleton* files. We're not going to write the actual functions for `getBigBounds` and `expand` in the `LeapButton` class just yet, as we're still laying out a basic framework.

LeapaintListener.java

Next up is the LeapaintListener class. This class will allow our main class, Leapaint, to communicate with the Leap Motion device—this is where all the fun, Leap-related stuff will happen.

Your skeleton file should look something like this when you're done writing, again omitting the content of some member functions:

```
import com.leapmotion.leap.*;

public class LeapaintListener extends Listener
{
    //Leap interaction box.
    private InteractionBox normalizedBox;

    //Leapaint instance.
    public Leapaint paint;

    //Controller data frame.
    public Frame frame;

    //Constructor.
    public LeapaintListener(Leapaint newPaint)
    {
        //Assign the Leapaint instance.
        paint = newPaint;
    }

    //Member Function: onInit
    public void onInit(Controller controller)
    {
        System.out.println("Initialized");
    }

    //Member Function: onConnect
    public void onConnect(Controller controller)
    {
        System.out.println("Connected");
    }

    //Member Function: onDisconnect
    public void onDisconnect(Controller controller)
    {
        System.out.println("Disconnected");
    }

    //Member Function: onExit
    public void onExit(Controller controller)
    {
        System.out.println("Exited");
    }

    //Member Function: onFrame - We'll write this later!
    public void onFrame(Controller controller) { }
```

```
}
```

Unlike our previous `LeapButton` class, this one requires a bit of explanation before we charge ahead to the next class.

As we discussed earlier, the `Listener` class within the Leap Motion API is our primary entry point into accessing Leap tracking data; needless to say, there are numerous functions for us to define (technically, we are overriding the functions) in any `Listener` implementation, if we, as developers, so desire. The `LeapButton` class is no exception to this, and the following are several examples of these functions:

- The `onInit` function is called when the Leap Motion software itself initializes; this is not to be confused with the Leap Motion Controller being connected or initialized.
- The `onConnect` function is called when the Leap Motion software connects to a physical Leap Motion Controller.
- The `onDisconnect` function is called when the Leap Motion software is disconnected from a physical Leap Motion Controller.
- The `onExit` function is called when the `Listener` class disconnects from the Leap Motion software.
- Finally, the `onFrame` function is called whenever the `Listener` class receives a new frame from the physical Leap Motion Controller; this function can be thought of as a sort of `while` or `for` loop. This is where we'll be doing all of the work with the Leap later on in this chapter.
- As a general rule, the functions previously listed will be called in this order (assuming a controller is already connected to the computer and nothing goes terribly wrong):
 - `onInit`
 - `onConnect`
 - `onFrame` (called many times per second)
 - `onDisconnect`
 - `onExit`

You can find in-depth documentation on these functions at the official Leap Motion website, <http://developer.leapmotion.com/>.

Now, let's move onto the final bare-bones class that we'll be writing before we begin fleshing out the details.

Leapaint.java

Lastly, we have the Leapaint class. This is the main class for our project, so needless to say, it's very important! Not only does it initialize the LeapListener class and configure all the LeapButton instances, but it also contains almost all of the graphical user interface and code.

The skeleton file should look something like this by the time you finish writing in code. For instructional and reference purposes, the relevant Leap-related lines have been highlighted:

```
import java.awt.BasicStroke;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Point;
import java.awt.Rectangle;
import java.awt.Robot;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BufferedImage;
import java.io.File;
import java.util.ArrayList;
import java.util.List;

import javax.imageio.ImageIO;
import javax.swing.Box;
import javax.swing.JFrame;
import javax.swing.JPanel;

import com.leapmotion.leap.Controller;

public class Leapaint extends JFrame
{
    //Static reference to this class.
    private static Leapaint paint;

    //X, Y and Z coordinates of the user's finger. These are set via the
    LeapaintListener class.
    public int prevX = -1, prevY = -1;
    public int x = -1, y = -1;
    public double z = -1;

    //Current drawing color.
    public Color inkColor = Color.MAGENTA;

    //Line data structure used to keep track of the lines we'll be drawing.
    public class Line
    {
        public int startX, startY, endX, endY;
        public Color color;
    }
}
```

```

Line(int startX, int startY, int endX, int endY, Color color)
{
    this.startX = startX;
    this.startY = startY;
    this.endX = endX;
    this.endY = endY;
    this.color = color;
}
}

//Lines drawn. We need to keep track of these, or they'll be lost every
time the screen refreshes.
public List<Line> lines = new ArrayList<Line>();

//Leap-enabled buttons.
public LeapButton button1, button2, button3, button4;

//Panels that we'll be drawing on.
public JPanel buttonPanel;
public JPanel paintPanel;

//Constructor—we'll write this later!
Leapaint() { }

//Member Function: saveImage—we'll write this later!
public void saveImage(String imageName) { }

//Member Function: main
public static void main(String args[])
{
    //Create a new instance of the Leapaint class.
    paint = new Leapaint();

    //Create a new listener and controller for the Leap Motion device.
    LeapaintListener listener = new LeapaintListener(paint);
    Controller controller = new Controller();

    //Start the listener.
    controller.addListener(listener);
}
}

```

This file has just a little bit more going on compared to the previous ones...and we haven't even started filling in the empty constructors and functions yet.

The more mundane items in this class, such as the Line class and its respective array, are relatively straightforward and easy to understand, so I will let the code and comments do the explaining for those.

There are a few lines of code that I'd like to provide a short explanation for, nonetheless, as they are intrinsic to the Leap side of this class's functionality:

- The field, `public int x, y`, is set by the LeapaintListener class from the outside using normalized screen coordinates and are used for drawing the lines and cursors

- The field, `public double z`, is also set by the `LeapaintListener` class and is used to both provide visual feedback to the user and determine when to draw lines

I'd also like to bring your attention to the last three lines:

- `LeapaintListener listener = new LeapaintListener(paint)`: This line creates a new instance of our `LeapaintListener` class and passes in the `Leapaint` instance, `paint`, that we initialized on the previous line.
- `Controller controller = new Controller()`: This line creates a new instance of the Leap API's `Controller` class, in turn creating our entry point to connect to the Leap Motion control software.
- `controller.addListener(listener)`: This line registers our `LeapaintListener` (or `Listener` if you will) instance with the Leap Motion software, allowing it to receive *callbacks* for the various functions that we talked about at the end of the `LeapListener.java` section of this chapter.

Many (if not all) applications that involve the Leap Motion Controller will end up using these three lines somewhere along the line (pun not intended), no matter how simple or complex the application in question might be.

At this point, we now have three basic classes loaded up with a lot of variables and empty member functions. If you were to run the application now, there probably wouldn't be any errors...but nothing would really happen.... So, how about we start filling in those functions?

Creating the graphical frontend

With our skeleton framework written and laid out, we can now work on flushing out the graphical component of Leapaint. As we're using the Java programming language for this application, we will write the graphical side of things using the Java Swing API. First up is the `LeapButton` class.

Note

As you read this section and the remainder of this chapter, keep in mind that as this book is titled *Mastering Leap Motion* and not *Mastering the Java Swing API*, I will not be discussing the fine details of what the Swing API calls are doing or how they work.

Making a responsive button – the LeapButton class

It's time to make your first Leap-driven, *visually responsive* button. There are many, many different ways to go about coding the logic behind a button for the Leap Motion Controller, and less so when you're integrating with a preexisting API such as Java's Swing.

Our LeapButton class does not listen to or otherwise check the values coming out of the Leap Motion Controller. Instead, it relies on an external class or function (LeapMouseListener in this case) to tell it what to do.

In order for us to achieve this behavior, I've incorporated two functions into the LeapButton class, which more or less make up the entirety of the class.

Note

A brief disclaimer: the LeapButton class is not the be-all and end-all way of designing an interactive and/or responsive button for Leap Motion. Rather, it is the most simple and direct method I could think of for this chapter. However, it works, and it works well.

Getting our bounds

The first function, `getBigBounds`, will be used by our `LeapListener` class to detect where the button is on the screen and whether or not a finger is within it. In essence, this function returns an exaggerated version of the location and size of the button on screen, making it easier for users to trigger.

Shown next is the aforementioned function. You should replace the contents of the placeholder `getBigBounds` function that we wrote earlier in `LeapButton.java` with the following lines of code:

```
//Retrieve original bounds.
Rectangle rect = getBounds();

//Increase height and width of the button.
rect.width = rect.width + 30;
rect.height = rect.height + 30;

//Reposition the button so that its central coordinates more or less remain
the same.
rect.x = rect.x - 15;
rect.y = rect.y - 15;

//Return the new button size.
return rect;
```

That's it! At the end of the day, it's a very simple function; now for the slightly more complex function.

Visually responding to the user

The second function, `expand`, does exactly what it says: it makes the `LeapButton` visually enlarge until it pops, triggering any registered callbacks. This function fulfills one of the

requirements for good design that we discussed in the previous chapter, giving the user visual feedback.

Unlike what you might expect, this function is not integrated with the Leap Motion device in any shape or form; it is purely math and logic driven, being triggered and stopped from the outside by our `LeapButtonListener` class.

In order for the button to cleanly expand and contract without halting our main execution loop, this function starts an anonymous inner thread that handles the logic behind expanding the button, popping the button, triggering any callbacks, and then resetting the button. At any given point, this thread can be effectively terminated by setting the `canExpand` member variable to `false` from outside the `LeapButton` instance, which we'll demonstrate later on.

The `expand` function is used in the following code. Again, you should replace the contents of the placeholder `expand` function that we wrote earlier in `LeapButton.java` with the content of this one:

```
//Don't start anything if this button is already expanding.
if (!expanding)
{
    //Begin expanding.
    canExpand = true;
    expanding = true;

    //Create an anonymous inner thread, so as not to freeze the main loop.
    (new Thread()
    {
        public void run()
        {
            //Change the button's color to green for even better visual feedback.
            Color originalColor = getBackground();
            setBackground(Color.green);

            //Store the original button size.
            originalSizeX = getPreferredSize().width;
            originalSizeY = getPreferredSize().height;

            //Calculate the target size based on this LeapButton's expansion
multiplier.
            int targetSizeX = (int) (originalSizeX * expansionMultiplier);
            int targetSizeY = (int) (originalSizeY * expansionMultiplier);

            //Calculate the amount to increase button size by in terms of steps.
            int stepX = (targetSizeX - originalSizeX) / 10;
            int stepY = (targetSizeY - originalSizeY) / 10;

            //Loop while expanding is ok and we haven't reached the target size.
            while (canExpand && getPreferredSize().width < targetSizeX)
            {
                //Increase button size.
                setPreferredSize(new Dimension(getPreferredSize().width + stepX,
getPreferredSize().height + stepY));
            }
        }
    });
}
```

```

        //Repaint (update) the button.
        revalidate();

        //Wait a moment before increasing size again.
        try { Thread.sleep(75); }
        catch (Exception e) { }
    }

    //Trigger all callbacks if the button size on loop exit meets or
    exceeds our target expansion size.
    if (getPreferredSize().width >= targetSizeX) doClick();

    //Otherwise, revalidate (update) the button to make sure renders,
    since doClick() would normally handle this.
    else
        revalidate();

    //Reset the size of the button to its original dimensions.
    setPreferredSize(new Dimension(originalSizeX, originalSizeY));

    //Revalidate (update) the button.
    revalidate();

    //This button is no longer expanding.
    expanding = false;

    //Restore the original button color.
    setBackground(originalColor);
}
}).start();
}

```

So, setting aside the fact that this was a good chunk of code you just wrote (or read), the overall content of the function is relatively straightforward:

1. First, we check whether the button is already expanding and exit if it is.
2. We make a note of the button's current color and then switch it to another one.
3. We obtain the current size of the button and then calculate how big it should grow to be.
4. We then use the data from the previous step to calculate how many pixels the button should increase size by so that we perform more or less exactly 10 steps over the course of a second.
5. Now we enter a loop, continuously incrementing the size of the button and waiting for a fraction of a second until the button is told to no longer expand or it reaches the target size.
6. We exit the loop, triggering any registered callbacks if the button managed to fully expand.
7. Finally, we completely reset the state of the button and exit the function.
8. Even better, this process can be applied to *any* programming language with any API. It's not exclusive to Java in any way; in fact, there are very few Swing-specific method calls in this function!

You can see this expansion method in action in the following screenshot on the finished version of the Leapaint application:



Note

If you run the application right this moment, you will not see anything like the preceding screenshot, as we haven't written the Leap tracking code or the graphical frontend.

Now it's time to move on to the graphics-heavy class and the brunt of the Leapaint application's coding, Leapaint!

Making a graphical user interface

The `Leapaint` class, as you've probably gathered from the name, is central to the application we're creating; it houses the main GUI components, it starts up all of the services, and it generally just ties everything together. Now, we just have to fill in the blanks that were left in the skeleton version we wrote earlier, starting with the constructor.

Constructing a constructor

Swing can, at times, involve some pretty big constructors. This is one of those times. As this is mostly Swing-related code, I will let comments do a majority of the talking for this particular snippet.

Here are the contents of the `Leapaint` constructor. You should replace the contents of the constructor in the skeleton file with these:

```
//Always call the superclass constructor when overriding Java Swing classes.
```

```
super("Leapaint - Place a finger in view to draw!");
```

```
//Configure the button panel.
```

```
buttonPanel = new JPanel(new FlowLayout());
```

```
buttonPanel.setBackground(new Color(215, 215, 215));
```

```
//Configure the buttons.
```

```
button1 = new LeapButton("Red", 1.5);
```

```
button1.addActionListener(new ActionListener()
```

```
{  
    public void actionPerformed(ActionEvent e)
```

```
    {  
        inkColor = Color.RED;
```

```
    }
```

```
});
```

```
button2 = new LeapButton("Blue", 1.5);
```

```
button2.addActionListener(new ActionListener()
```

```
{  
    public void actionPerformed(ActionEvent e)
```

```
    {  
        inkColor = Color.BLUE;
```

```
    }
```

```
});
```

```
button3 = new LeapButton("Purple", 1.5);
```

```
button3.addActionListener(new ActionListener()
```

```
{  
    public void actionPerformed(ActionEvent e)
```

```
    {  
        inkColor = Color.MAGENTA;
```

```
    }
```

```
});
```

```
button4 = new LeapButton("Save", 1.5);
```

```
button4.addActionListener(new ActionListener()
```

```

{
    public void actionPerformed(ActionEvent e)
    {
        saveImage("leapaint");
    }
});

//Add the buttons to the button panel.
buttonPanel.add(button1);
buttonPanel.add(button2);
buttonPanel.add(button3);
//Put a space between the color and save buttons.
buttonPanel.add(Box.createVerticalStrut(1));
buttonPanel.add(button4);

//Configure the paint panel.
paintPanel = new JPanel()
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);

        //Setup the graphics.
        Graphics2D g2 = (Graphics2D) g;
        g2.setStroke(new BasicStroke(3));

        //Only start drawing if the user's finger is in view and not on the
        button panel.
        if (z <= 0.5)
            lines.add(new Line(prevX, prevY, x, y, inkColor));

        //Draw all registered lines.
        for (Line line : lines)
        {
            g2.setColor(line.color);
            g2.drawLine(line.startX, line.startY, line.endX, line.endY);
        }

        //Repaint all the buttons.
        buttonPanel.repaint();

        //Draw the cursor if a finger is within in view.
        if (z <= 0.95 && z != -1.0)
        {
            //Set the cursor color to the inkColor if painting, and green
            otherwise.
            g2.setColor((z <= 0.5) ? inkColor : new Color(0, 255, 153));

            //Calculate cursor size based on depth for better feedback.
            int cursorSize = (int) Math.max(20, 100 - z * 100);

            //Create the cursor.
            g2.fillOval(x, y, cursorSize, cursorSize);
        }
    }
};

```

```
//Make sure the paint panel doesn't obscure any other elements.
paintPanel.setOpaque(false);

//Add the panels to the primary frame.
getContentPane().add(buttonPanel, BorderLayout.NORTH);
getContentPane().add(paintPanel);

//Make sure the application exits on close.
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

//Set initial frame size and become visible.
setSize(800, 800);
setVisible(true);
```

Almost all of the preceding code was Swing related, but there are a few things worth noting before we move on to the next function.

The lines starting with `button1 = new LeapButton("buttonname", 1.5)` or `button1.addActionListener(new ActionListener())` create new instances of the `LeapButton` class and add anonymous action listeners to them, allowing them to, you guessed it, respond to input when pushed via a finger or clicked by a mouse.

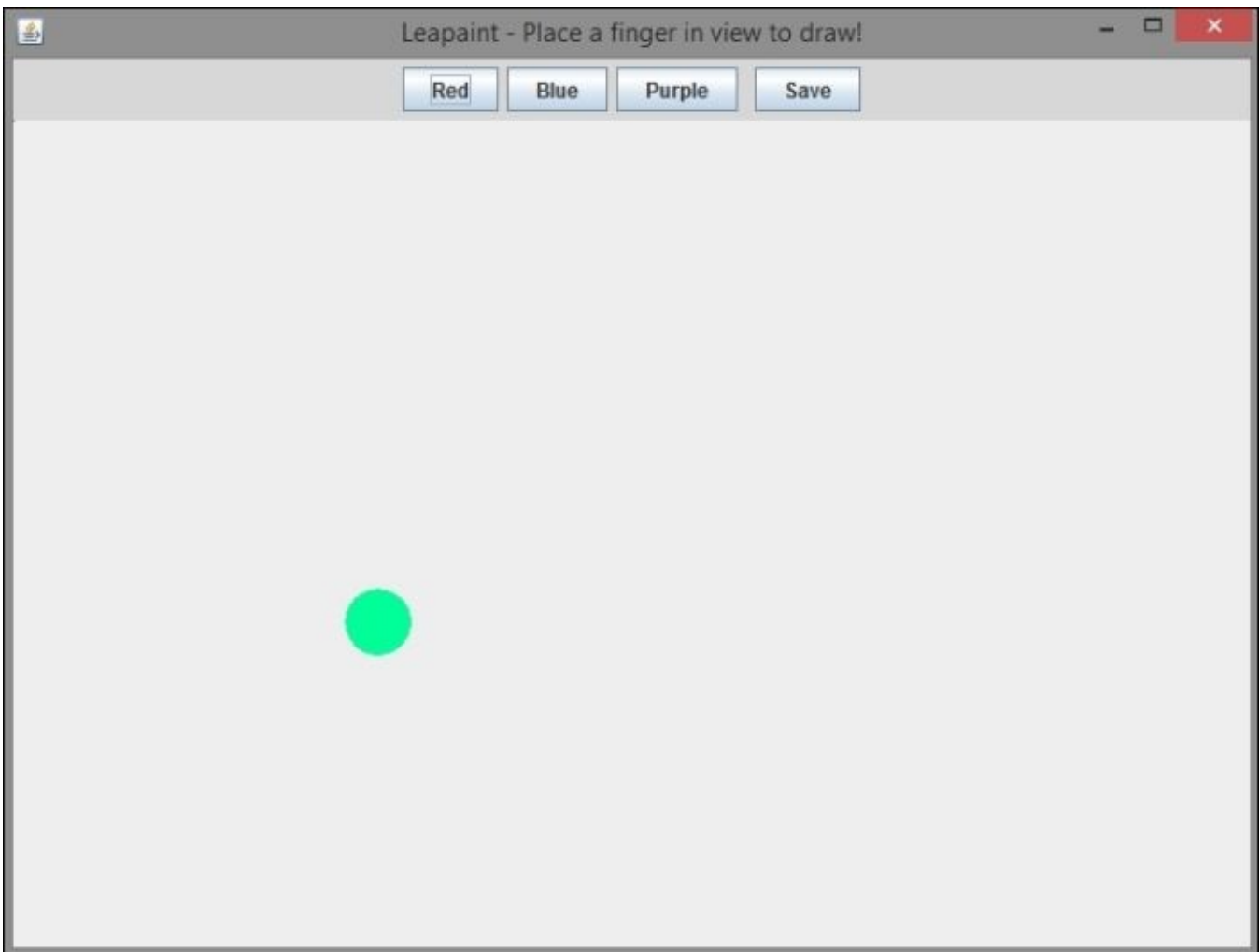
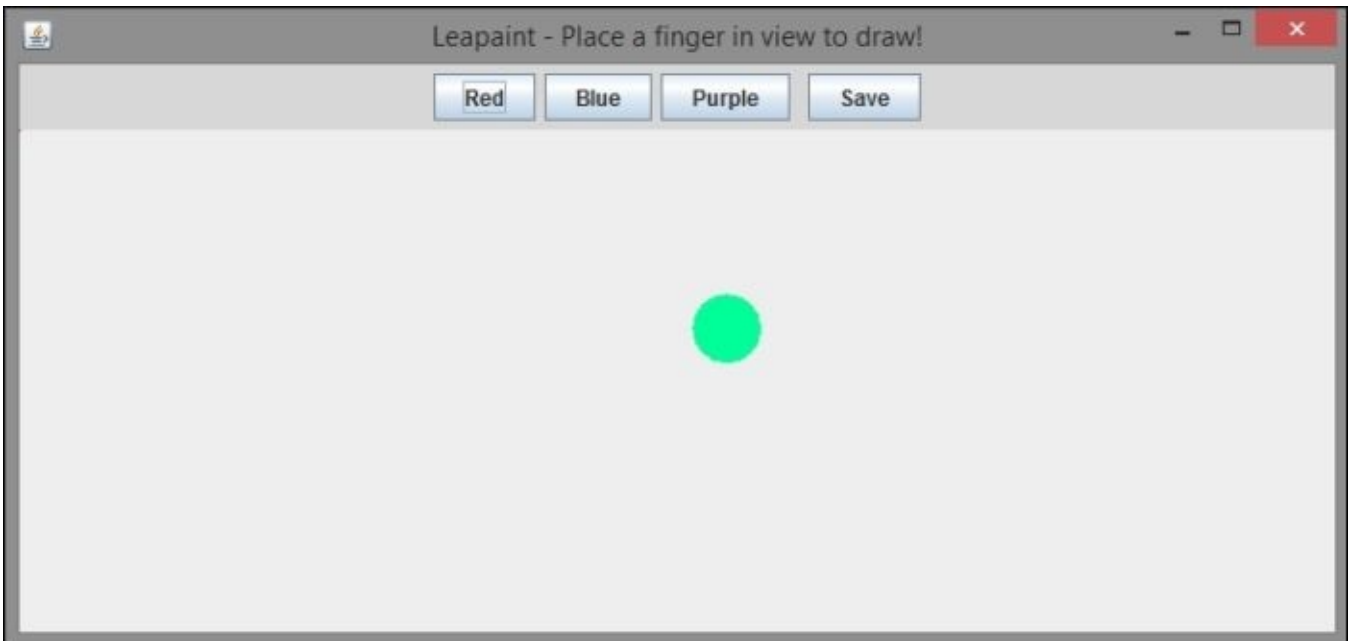
The block of code that starts with the line, `public paintComponent(Graphics g)`, highlighted in the preceding code for your convenience, contains most of the Leap-related logic. The first few lines are pretty straightforward; we set up some graphics references, store values for lines, and then draw some paint on the screen. One of the lines in there, `if (z <= 0.5) lines.add(new Line(prevX, prevY, x, y, inkColor))`, uses the z axis of the Leap device to tell whether or not it's okay to paint; Leapaint will only start drawing lines if the user's finger passes a certain threshold, which we will be setting later via an `InteractionBox` instance.

However, wait; there is more. We also draw an onscreen cursor representing where the user's finger is and what it is doing. There are a total of three lines involved with configuring and drawing the user's cursor; we'll go over each one:

- `g2.setColor((z <= 0.5) ? inkColor : new Color(0, 255, 153))`: This determines which color the cursor should be. If the user's finger is close enough to the screen to be painting (the value for the z axis less than 0.5), then it will match the color of the ink that we are drawing with. Otherwise, it will be a soothing mint green. While these checks could have been done using a basic if-else statement, the use of a ternary operator here allows everything to fit on one line, which is nice.
- `int cursorSize = (int) Math.max(20, 100 - z * 100)`: This calculates the exact size the cursor should be, again using the depth (the z axis) of the user's finger as the primary metric. The closer the finger gets to the screen, the bigger the circle will become. While it's not required to provide visual feedback like this, it's a great way to provide visual feedback for people so that they aren't poking around blindly (or guessing how far in their finger is).
- `g2.fillOval(x, y, cursorSize, cursorSize)`: This finally renders the cursor on screen as a circle using the current coordinates of the pointer and the size we

calculated on the previous line.

This results in a shape-shifting cursor that gets bigger and smaller as a finger gets closer to and farther from the screen, as seen in the following screenshots:



Note

If you run the application right this moment, you will not see the cursor, as the Leap tracking code has not been written yet.

This completes the Leapoint constructor! All that's left before this class is finished is a single function, saveImage.

Saving images

The `saveImage` function in `Leapaint` is in no way related to developing for the Leap Motion Controller, but it's a great toy—I mean, tool—to have in hand when you're writing a graphical drawing application. We will cover it very briefly before moving on to the long-awaited Leap-side of things.

Find the code for `saveImage`. Be sure to replace the content of `saveImage` in your skeleton file with the following code:

```
//Get the location and bounds of this JFrame.
Point pos = getContentPane().getLocationOnScreen();
Rectangle screenRect = getContentPane().getBounds();
screenRect.x = pos.x;
screenRect.y = pos.y;

//Attempt to take a screen capture and pipe it to the image file.
try
{
    BufferedImage capture = new Robot().createScreenCapture(screenRect);
    ImageIO.write(capture, "bmp", new File(imageName + ".bmp"));
}

catch (Exception e) {}
```

I won't go into detail on how this function works, or why, but I will briefly cover the fun part of how we go about getting the screen capture:

1. We first retrieve a set of onscreen coordinates for the location and size of the content area of our application.
2. We then attempt to use Java's `Robot` class (not literally related to robots, sadly) to take a screen capture of the onscreen area depicted by the coordinates that we just retrieved.
3. If all goes well when called, `saveImage` will pipe out an image to the file depicted by the `imageName` variable.

If you were to fire up `Leapaint` right now using Eclipse, you will be greeted by a window that looks something like the following (assuming there are no errors; if there are, check your code!):



As you can see, the screen is blank and ripe to be painted on. All we need to do now is catch our user input from the Leap Motion Controller!

Interpreting Leap data to render on the graphical frontend

At long last, we can begin writing the Leap code! The only code we'll be editing from this point forward is the `LeappaintListener` class.

Go ahead and open up `LeappaintListener.java` now and scroll down to the `onFrame` method. Replace its content with the lines of code I've written here:

```
//Get the most recent frame.
frame = controller.frame();

//Detect if fingers are present.
if (!frame.fingers().isEmpty())
{
    //Retrieve the front-most finger.
    Finger frontMost = frame.fingers().frontmost();

    //Set up its position.
    Vector position = new Vector(-1, -1, -1);

    //Retrieve an interaction box so we can normalize the Leap's coordinates
    to match screen size.
    normalizedBox = frame.interactionBox();

    //Retrieve normalized finger coordinates.

position.setX(normalizedBox.normalizePoint(frontMost.tipPosition()).getX())
;

position.setY(normalizedBox.normalizePoint(frontMost.tipPosition()).getY())
;

position.setZ(normalizedBox.normalizePoint(frontMost.tipPosition()).getZ())
;

    //Scale coordinates to the resolution of the painter window.
    position.setX(position.getX() * paint.getBounds().width);
    position.setY(position.getY() * paint.getBounds().height);

    //Flip Y axis so that up is actually up, and not down.
    position.setY(position.getY() * -1);
    position.setY(position.getY() + paint.getBounds().height);

    //Pass the X/Y coordinates to the painter.
    paint.prevX = paint.x;
    paint.prevY = paint.y;
    paint.x = (int) position.getX();
    paint.y = (int) position.getY();
    paint.z = position.getZ();

    //Tell the painter to update.
    paint.paintPanel.repaint();
}
```

```

//Check if the user is hovering over any buttons.
if (paint.button1.getBigBounds().contains((int) position.getX(), (int)
position.getY()))
    paint.button1.expand();

else paint.button1.canExpand = false;

if (paint.button2.getBigBounds().contains((int) position.getX(), (int)
position.getY()))
    paint.button2.expand();

else paint.button2.canExpand = false;

if (paint.button3.getBigBounds().contains((int) position.getX(), (int)
position.getY()))
    paint.button3.expand();

else paint.button3.canExpand = false;

if (paint.button4.getBigBounds().contains((int) position.getX(), (int)
position.getY()))
    paint.button4.expand();

else paint.button4.canExpand = false;
}

```

Ah, it's good to have some almost pure Leap code after our almost 20-page respite from having any at all. I'll go ahead and break everything down line by line:

- The first line, `frame = controller.frame()`, retrieves the most recent frame from the Leap Motion Controller. This is, of course, critical if we're to do any finger tracking.
- The `if (!frame.fingers().isEmpty())` statement makes sure that we only analyze the tracking data from the Leap if the frame we got has fingers inside. Otherwise, there's not much point, is there?
- The `Finger frontMost = frame.fingers().frontMost()` statement obtains a copy of the frontmost (furthest toward the screen) finger in the frame the controller gave us. There were many different routes that we could've taken here in order to get a reference to a single finger. We could have used the Skeletal Tracking API to get an index finger, or we could have averaged the location of all the fingers in the frame, and so on. I chose this method because it's simple and straightforward, making it great for a tutorial.
- The `Vector position = new Vector(-1, -1, -1)` statement initializes the set of coordinates that we will be passing to the `Leapaint` class later on.
- The `normalizedBox = frame.interactionBox()` statement retrieves an `InteractionBox` instance from the frame; as we discussed earlier, always get an `InteractionBox` instance from the Leap Motion API—never create a new one from outside.
- The line,

`position.setX/Y/Z(normalizedBox.getNormalizedPoint(frontMost.tipPosition))` retrieves the normalized 0.0 to 1.0 coordinates of the frontmost finger from our `InteractionBox` instance.

- The two lines, `position.setX(position.getX() * paint.getBounds().width)` and `position.setY(position.getY() * paint.getBounds().height)`, adjust the normalized values to fit Leapaint's `JFrame`, no matter what size it is set to. The cool thing about floating point (decimal) numbers that are fixed to a scale of 0.0 to 1.0 is that, when multiplied by the maximum dimension (for example, size) of another coordinate system, they scale and integrate into this coordinate system perfectly. Ergo, this method works for all kinds of things, ranging from 2D applications like this one to 3D applications, which we'll be getting into in the next chapter.
- The next two lines, `position.setY(position.getY() * -1)` and `position.setY(position.getY() + paint.getBounds().height)`, invert the normalized y axis coordinates of our frontmost finger's tracking data. If we didn't do this, a finger that is pointing at the bottom of the screen would end up painting on the top half of the screen and vice versa. This is due to a minor discrepancy in how the Leap tracks the y axis as compared to how many of the common GUI frameworks handle their coordinate systems.

Note

Fun fact

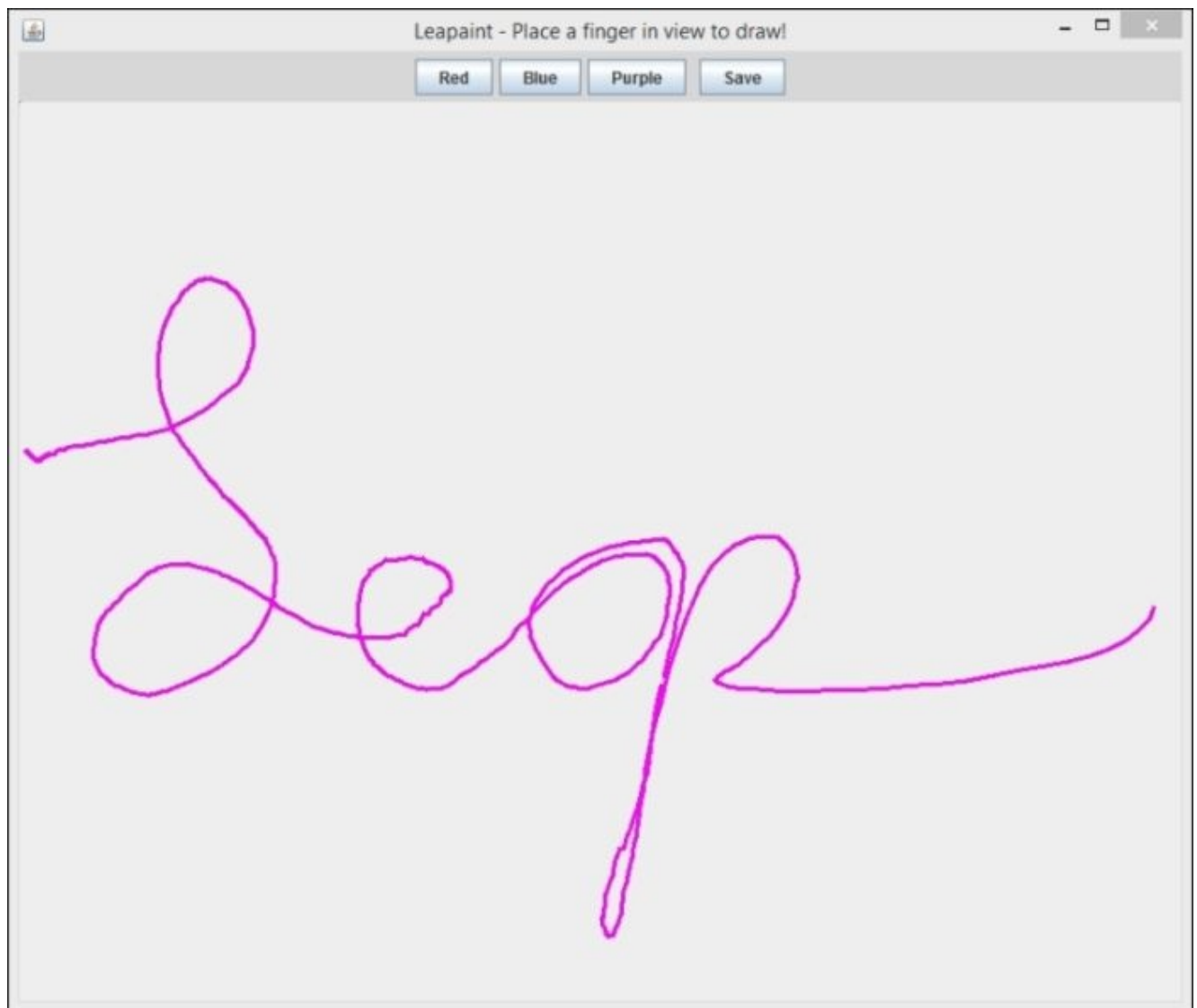
The Leap Motion Controller sees a higher (positive) value in the y axis to be further away from the Leap, or higher. Conversely, many GUI systems, Java Swing included, see higher values in the y axis as being further away from the origin or top-left corner of the screen. In other words, a normal GUI framework will see higher values of y as being lower on the screen, whereas the Leap Motion Controller sees them as being higher.

- The next few lines, such as `paint.prevX = paint.x` and `paint.x = position.getX()`, pass our fully normalized and converted finger coordinates to the Leapaint class instance so that it can start drawing lines.
- The `paint.paintPanel.repaint()` statement tells the Leapaint instance to *repaint* the drawing area of our application; in Swing terms, this basically tells the window to update itself visually to respond to the new finger coordinates.
- Finally, the last few lines like `if (paint.buttonX.getBigBounds().contains((int) position.getX(), (int) position.getY()))`, `paint.buttonX.expand()`, and `else paint.buttonX.canExpand = false` detect if the frontmost finger is *touching* any of our four LeapButtons instances (the three color switchers and the save image button), and triggers their associated expand function if it is. Otherwise, if a button isn't being touched by a finger, these lines make sure it stops expanding.

Testing it out

Now, for the long-awaited moment where we fire up the Leapaint application:

1. After verifying that all of the code in this chapter thus far has been written and put in the correct places, plug in your Leap and hit the run button (if you're using Eclipse)!
2. If all goes well, you should be able to start drawing—like the image shown in the following screenshot (do forgive this author's inability to draw freehand; I much prefer vector graphics):



With this, you've completed a simplistic, but functional, 2D drawing application for the Leap Motion Controller. Next on the list is a 3D application, the Leap's native domain so to speak.

Improving the application

With the bare-bones Leapaint application complete, you now have a great platform to build off and improve. Why not use this as an opportunity to sharpen your skills before moving on to the next chapter?

There are quite a few ways you can enhance Leapaint, including:

- A prettier user interface and better-looking buttons.
- Smoother, less random lines (occasionally, using this code, drawn lines will stutter; this is because we interpolate the location of the lines by drawing them between points).
- More colors to pick from; maybe by using a color wheel or slider?
- Anything else that you can possibly think of!

In addition, if you wish to further your skills with Java GUIs, you can find more information about the Java Swing API (which Leapaint uses quite extensively) on Oracle's official website at <http://docs.oracle.com/javase/7/docs/technotes/guides/swing/>.

Summary

In this chapter, you wrote a two-dimensional (2D) drawing application for the Leap Motion Controller using only the Leap and Java Swing APIs.

You covered how to make simple expanding buttons, responsive cursors, and a basic but effective user experience. Heading forward, this knowledge will make the coming chapters much easier as we begin working with a 3D application, which isn't a whole lot more complex than a 2D application once you get used to the third dimension, that is. You also wrote a lot of code in this chapter.

In the next chapter, you will begin learning about the Unity 3D development toolkit as it pertains to the Leap Motion Controller so that you can begin writing three-dimensional applications for the Leap; we're just getting started!

Chapter 5. Creating a 3D Application – a Crash Course in Unity 3D

Now that we've tackled the design and development of a simple 2D painting application, how about we move on to more advanced 3D applications? In this chapter, you'll learn about the Unity 3D toolkit, the go-to software suite for many Leap developers who are working on applications that utilize three dimensions.

Topics that we'll be covering in this chapter include:

- A brief introduction to Unity
- Installing and setting up Unity
- Common jargon in Unity
- Creating a project
- Setting the scene

Note

Disclaimer: this chapter contains absolutely no Leap-Motion-related content or information. It exists solely to teach you the basics of using Unity to prepare you for the next two chapters, which will contain a lot of Leap-Motion-related content.

This chapter is sprinkled with periodic Fun facts that offer high-level, entry-level factoids about scripting and programming for your reading pleasure.

A brief introduction to Unity

So far in this book, you've covered many of the fundamental concepts of the Leap, ranging from simple things such as the API and configuring the IDE to more complex things such as how the Leap sees tracking data and how user ergonomics can be enhanced. Along the way, you also covered the creation of a simple two-dimensional application to help tie all of these ideas together.

The following is a screenshot of the Unity 3D application, Artemis Quadrotor Simulator:



A Unity 3D application, the Artemis Quadrotor Simulator

Now, it's time for us to delve into the native realm of the Leap Motion Controller: three-dimensional applications, including their design, creation, and integration with the Leap. Over the course of the next three chapters, we will cover the setup, design, coding, and testing of a simple 3D application for the Leap Motion Controller where the user will control a simple *floating* object by moving their hand around.

Writing a 3D application can be a daunting task; you'll need to pull together assets like 3D models, 2D textures, audio clips, scripts, and so forth. How do you stitch all of these together into a single, Leap-driven application? That is where 3D toolkits come into play; and as I'm sure you've guessed from the chapter title already, we will be using the Unity 3D toolkit in the next few chapters as we create a three-dimensional application.

Unity allows game designers, artists, and developers to create games and simulators with little difficulty. Since the early days of Leap Motion, when things were still in beta and just being kicked off, Unity was the go-to toolkit for creating applications that make full use of all three dimensions that the Leap offers—this means there’s a lot of preexisting tools, assets, and references to develop for the Leap with Unity. Thanks to the extremely large user base that Unity has, there’s an extremely wide selection of resources and documentation (often available freely) on the Internet!

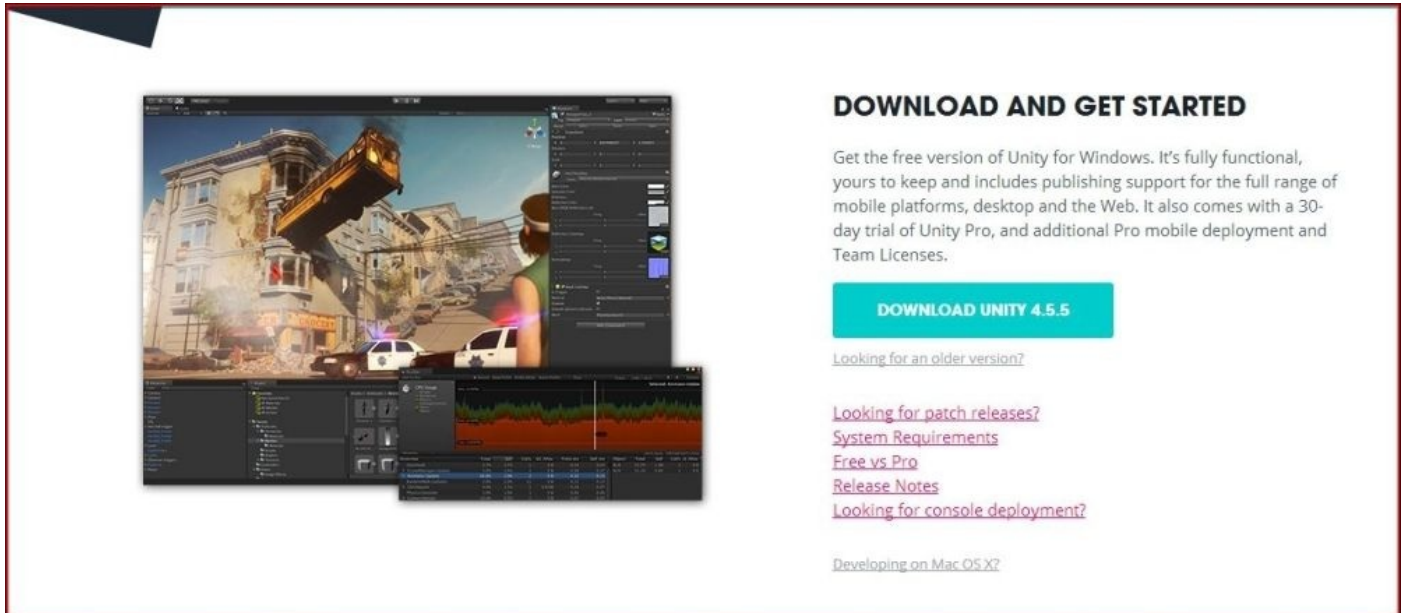
Of course, Unity is a very complex and useful program with a large number of features and capabilities—an outright guide to using it is worthy of a book in its own right (and I believe there are quite a few books on it too). Therefore, so as to avoid leaving the scope of this book (mastery of Leap Motion development), this chapter (and the following two) will touch only on the high-level aspects of using Unity—you will learn exactly what you need to know to get the job done.

If you want to learn even more about Unity, or you get stuck at some point during the next few chapters, you can find excellent documentation at <http://unity3d.com/learn>.

Let’s get straight to it then! In this chapter, you will install Unity and then create a basic scene to place our 3D app in.

Installing and setting up Unity 3D

The first step in developing with Unity is, well, installing Unity. Head on over to <http://www.unity3d.com/unity/download> and click on the giant blue button, as seen in the following screenshot; this will begin downloading the installer for Unity. It's a fairly large file, so expect to wait a little while for it to finish.



Once it finishes downloading, locate the installer file and run it. Follow all the prompts that you see, with the end result (hopefully) involving the installation of Unity.



After a while, you will see a screen like the preceding one. Go ahead and tick the **Activate a free 30-day trial of Unity Pro** box and click on **OK**. The installation of Unity should now be complete.

Common jargon found in Unity

Before we continue any farther and begin creating a Unity project, let's go over three of the most common things that a project contains.

Scenes

Every single *level* created in Unity 3D is called a **scene**; scenes are to a Unity project as chapters are to a book. Scenes can contain an arbitrary amount of GameObjects (limited only by the user's system resources), sky boxes, terrain, and so forth. You can think of them as the canvas on which we create a given area in a 3D application.

GameObjects

Every single item contained within a scene, be it a mountain, a wall, a robot, a light, or even some kind of user interface element, is a GameObject. There's even a GameObject class when scripting with Unity! Think of these as the building blocks of a 3D application.

Scripts

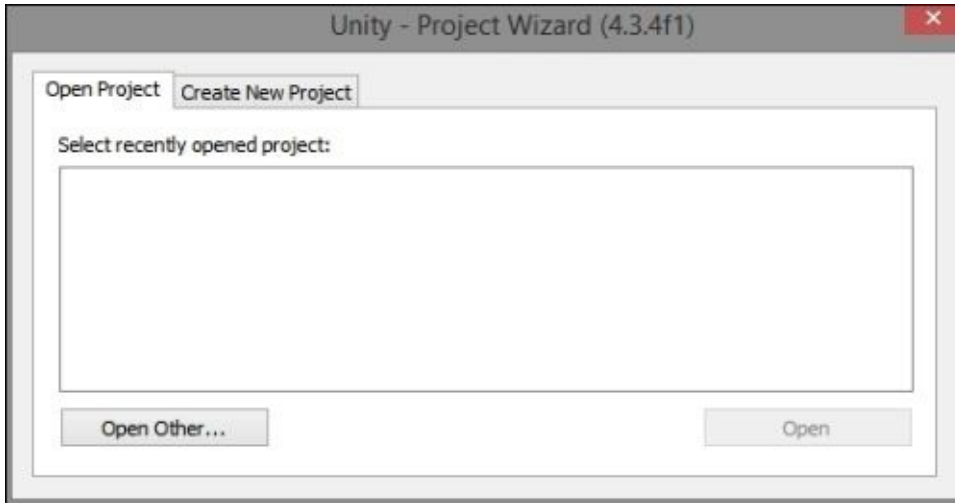
Any `GameObject` in a given scene that performs some kind of function or series of functions has one or more **scripts** attached to it. Scripts define the behavior of a `GameObject`, allowing it to move around, interact with other `GameObjects`, respond to user input, and even change shape and size! These are both the backbone of any control scheme and the entry point for Leap Motion development when building a 3D application.

The three scripting languages supported by Unity are `UnityScript` (JavaScript for Unity), `Boo`, and `C#`. We will be using `C#` for the duration of this book, as it has great integration with the Leap Motion Controller and it's quite similar to Java in terms of syntax.

As a short summary, every Unity project contains at least one scene. In turn, every scene contains one or more `GameObjects` (usually quite a few). Finally, every `GameObject` will usually have at least one or more scripts attached to it.

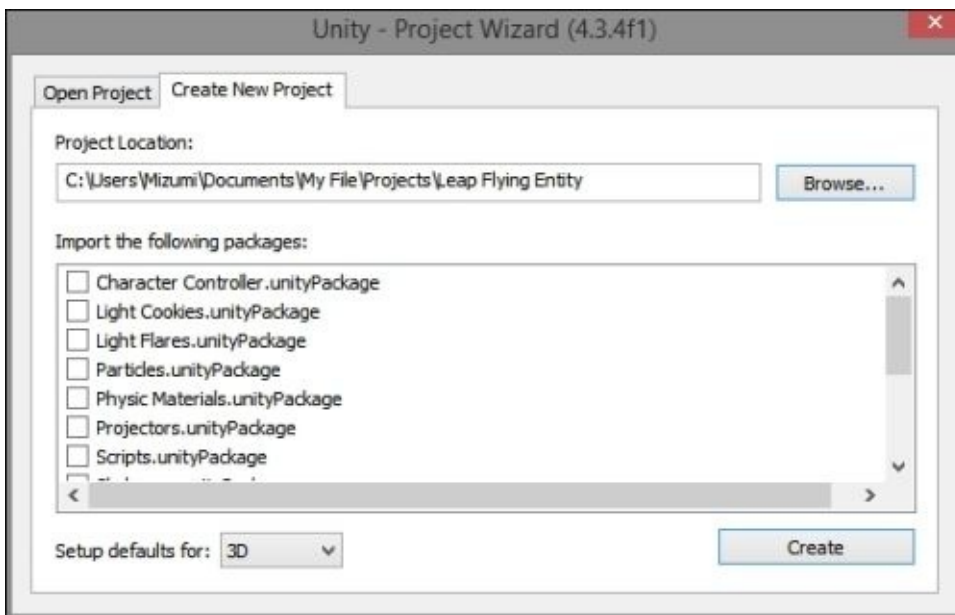
Creating a project

With the installation of Unity complete, you can now proceed to create a new project. Go ahead and launch Unity.



You will now be prompted to either open a preexisting project or create a new one. We are going to create a new one, so click on the **Create New Project** tab.

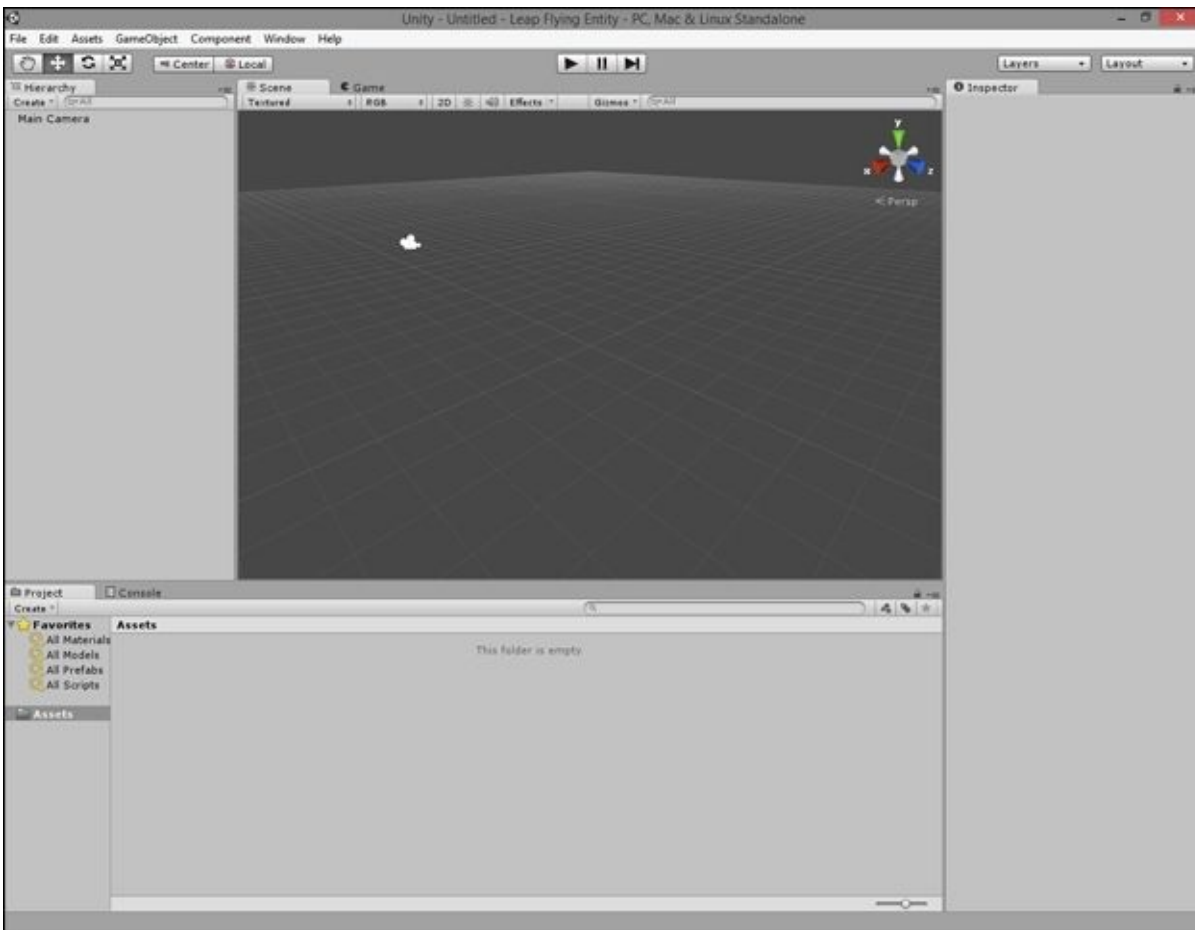
This will take you to the screen shown here:



Go ahead and browse to the folder you'd like the project to be located in via the **Browse** button. Do not worry about importing any packages; we're not worrying about that just yet.

When you're done, click on **Create**.

Upon creating the new project, you should now see a screen similar to the following one:

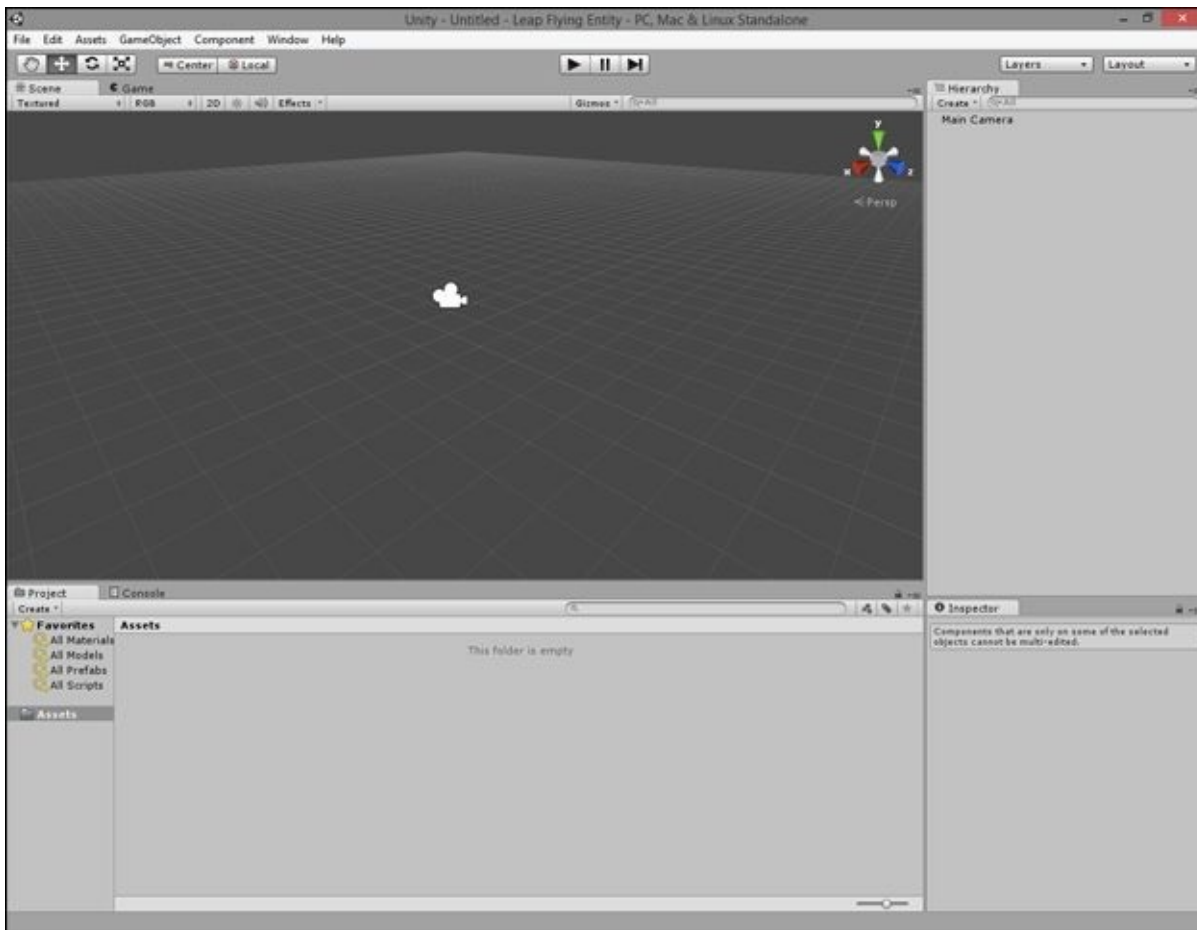


This screen is the main editor interface, which comprises multiple tabbed windows; while there's a lot going on, each individual window is rather simple (plus, we won't be making use of every single feature with this application).

There are four windows (or tabs, if you will) to pay attention to:

- **Hierarchy:** This window contains all of the GameObjects that are present in the currently active scene.
- **Scene:** This window acts as a viewport of the currently active scene.
- **Project:** This window contains a sort of tree-style file browser for the currently active project. In the next two chapters, we will be using this to organize and access the various scripts and assets that are created. You can think of it as Unity's version of the Project Explorer in Eclipse.
- **Inspector:** This window contains all the data for the currently selected GameObject in the **Scene** or **Hierarchy** window.

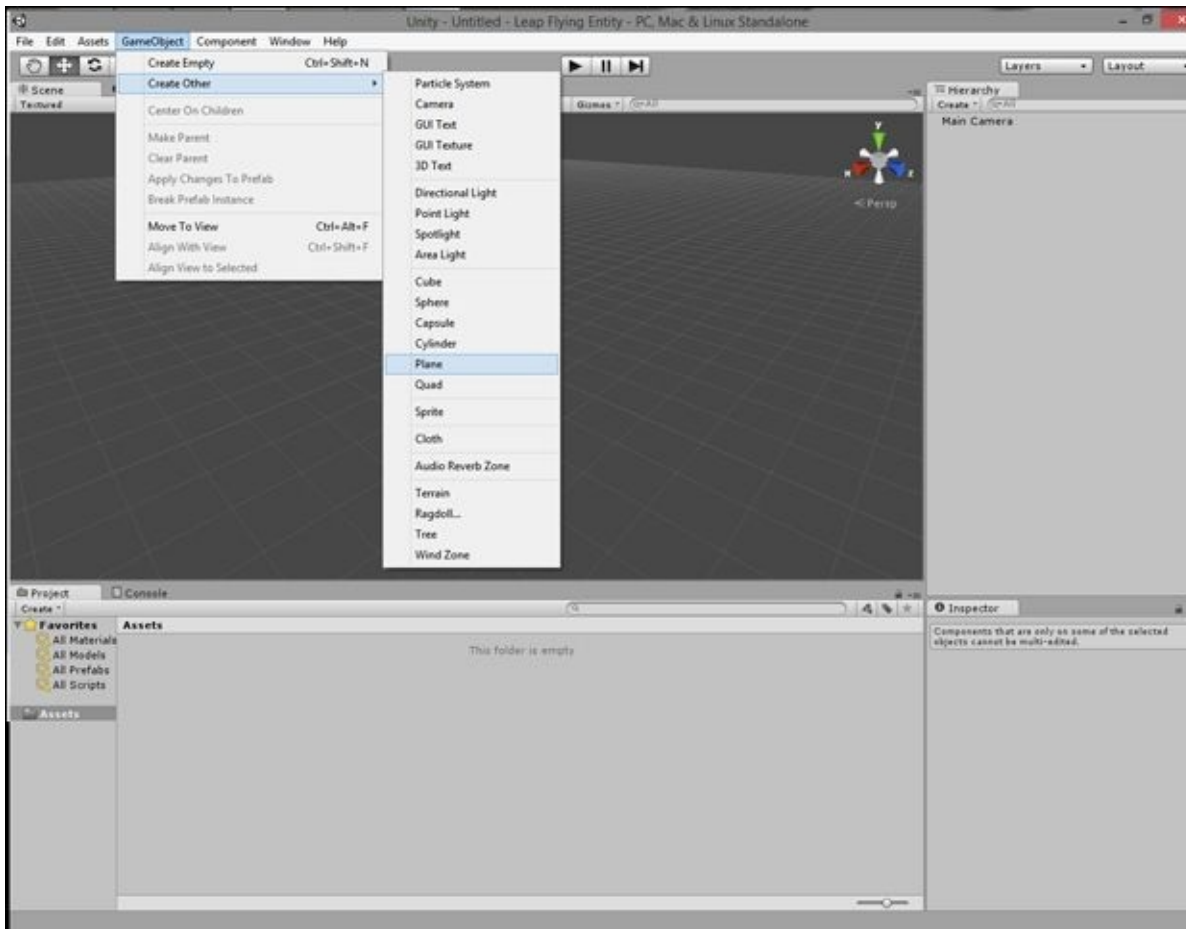
We will utilize the rest of the windows on an as-needed basis.



The preceding screenshot shows the editor layout that the author uses during development, but you can lay out the editor however you want by simply clicking on the **Window** tabs (**Project**, **Console**, **Inspector**, **Scene**, and so on) and dragging them around the screen.

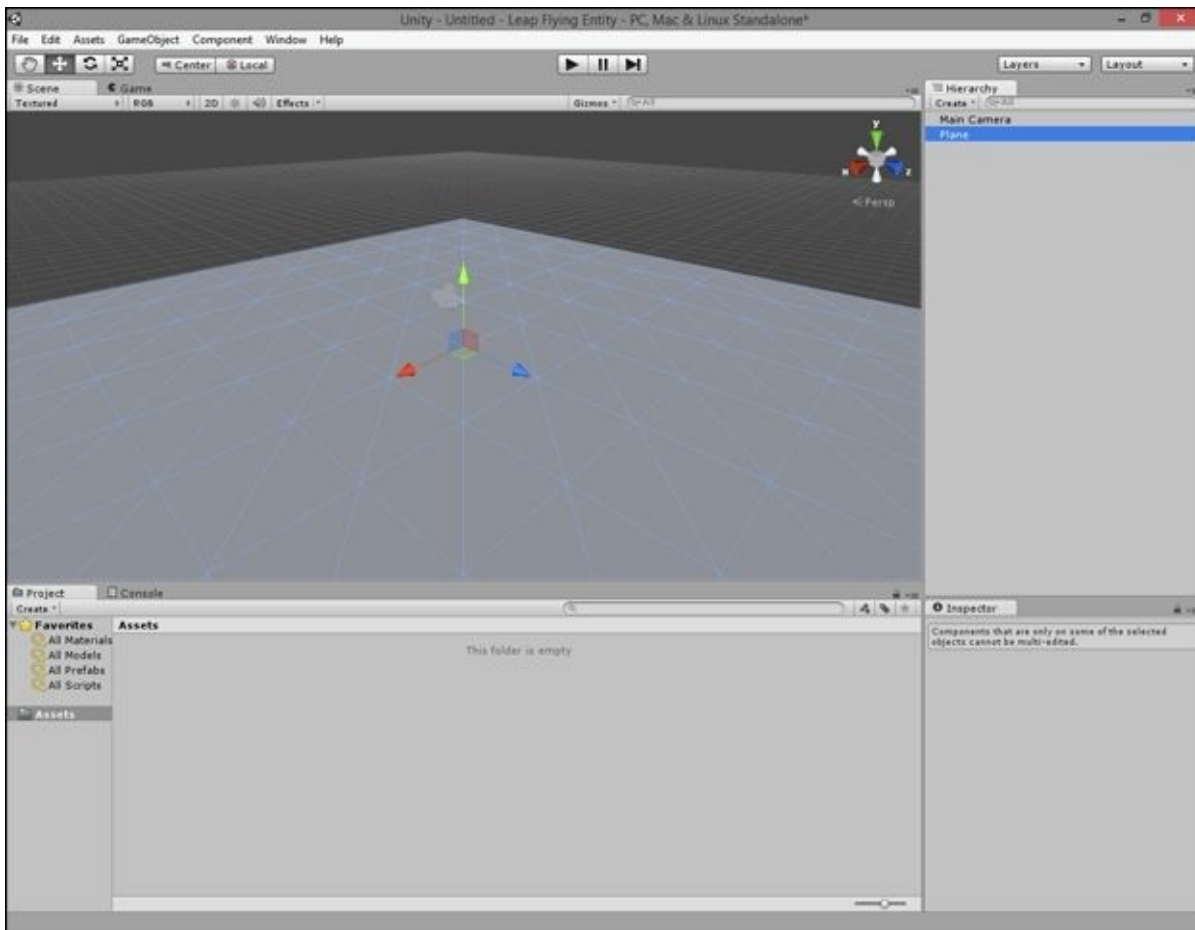
Setting the scene

With our project created and the editor configured, we can now set the stage (or scene, as it were) for this application.



To start off, you'll want to create a basic plane GameObject to act as the *ground* for this application. This can be achieved by navigating to **GameObject | Create Other | Plane** in the toolbar, as shown in the preceding screenshot.

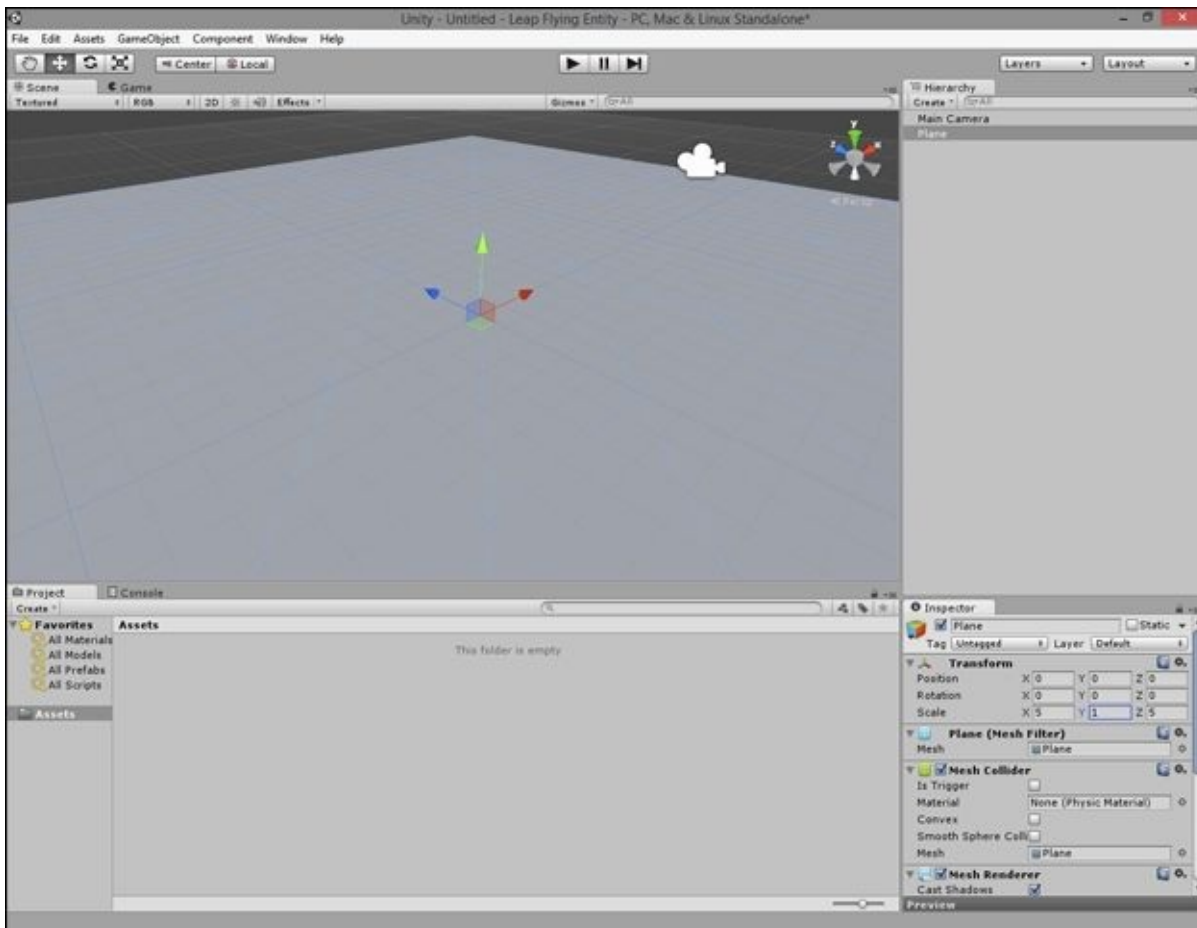
After clicking on the **Plane** button under **Create** in the previous step, you will see a perfectly flat plane appear in the **Scene** window, as seen in the following screenshot:



Note

Fun fact

All of those blue triangles that you see on any GameObject you select in the **Scene** window are a visual representation of that object's three-dimensional mesh.

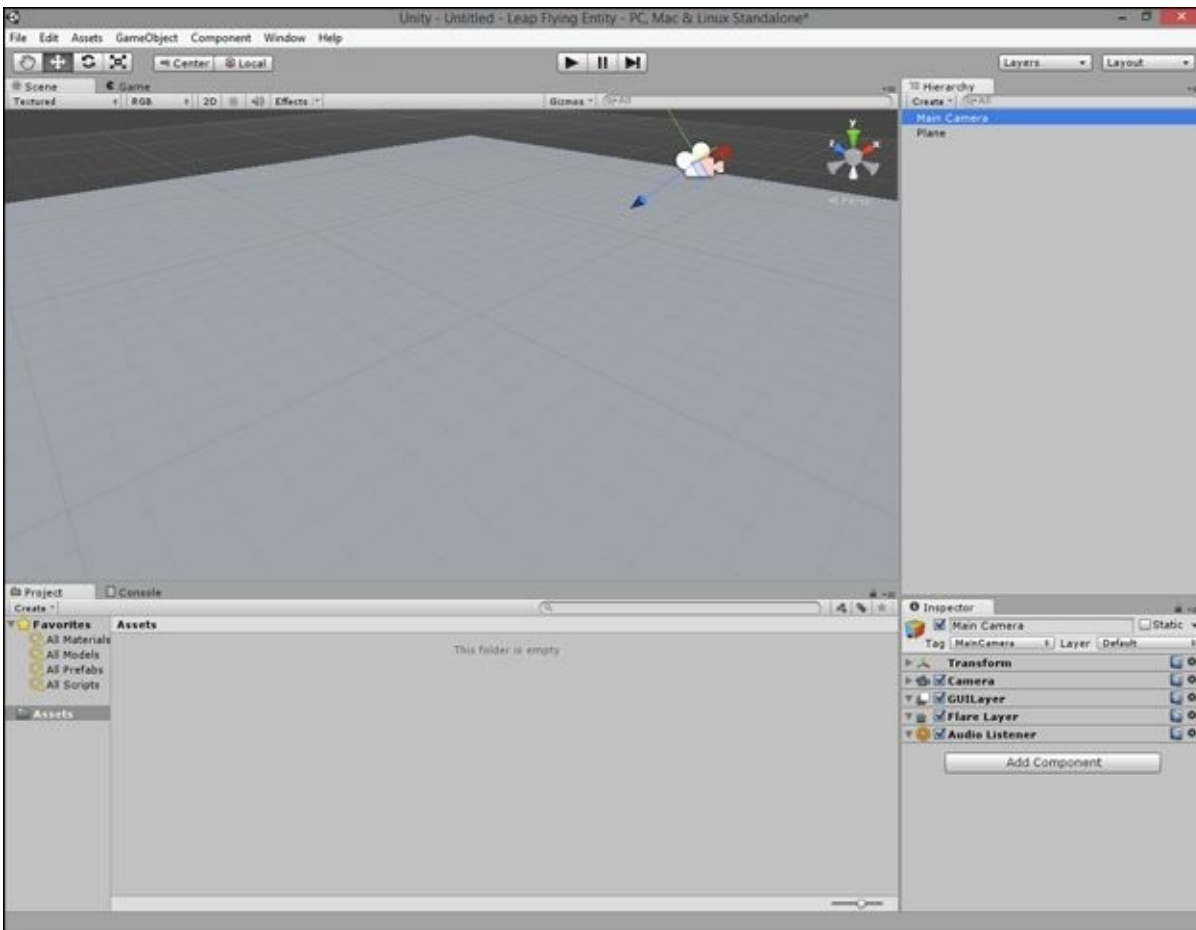


Now, as our 3D application (or game, as it was) is going to require a good deal of space for navigating, you'll want to make the plane a whole lot bigger.

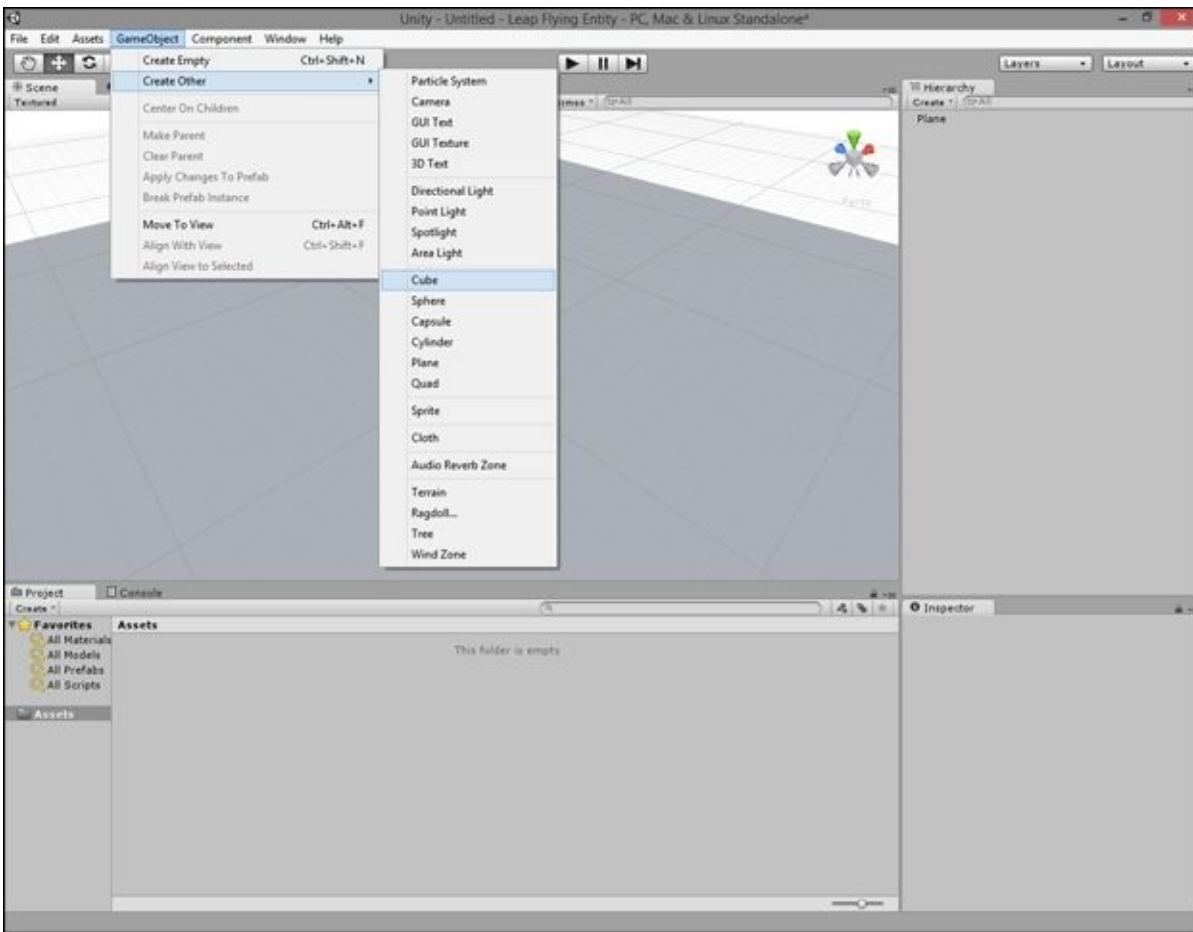
Let's go ahead and make the plane a bit bigger via the following steps (these will apply to any GameObject):

1. Select (click on) your newly created plane in the **Hierarchy** window. You'll notice that a bunch of information appears in the **Inspector** window.
2. Click on the **Transform** tab within the **Inspector** window. You will now see a list of fields labeled **Position**, **Rotation**, and **Scale**.
3. Select the **X** and **Z** fields that are next to the **Scale** label (the **Y** field is denoted by the blue arrow in the preceding screenshot) set them both to 5.
4. With this, your plane should be a whole lot bigger—five times bigger, to be exact. You can make it even bigger, but for now it's plenty.

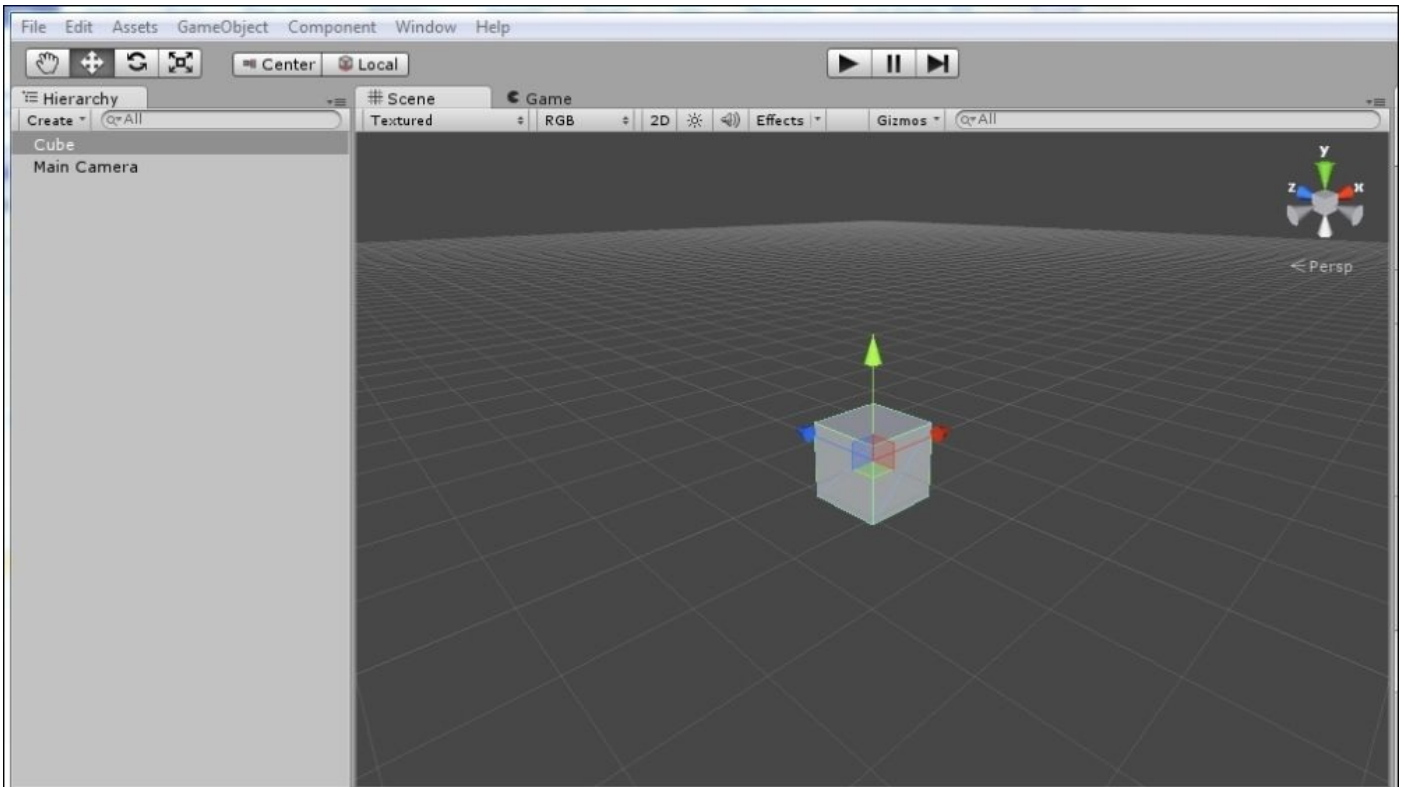
You've probably noticed the **Main Camera** GameObject hanging around in the **Hierarchy** window for a while now. This is shown in the following screenshot:



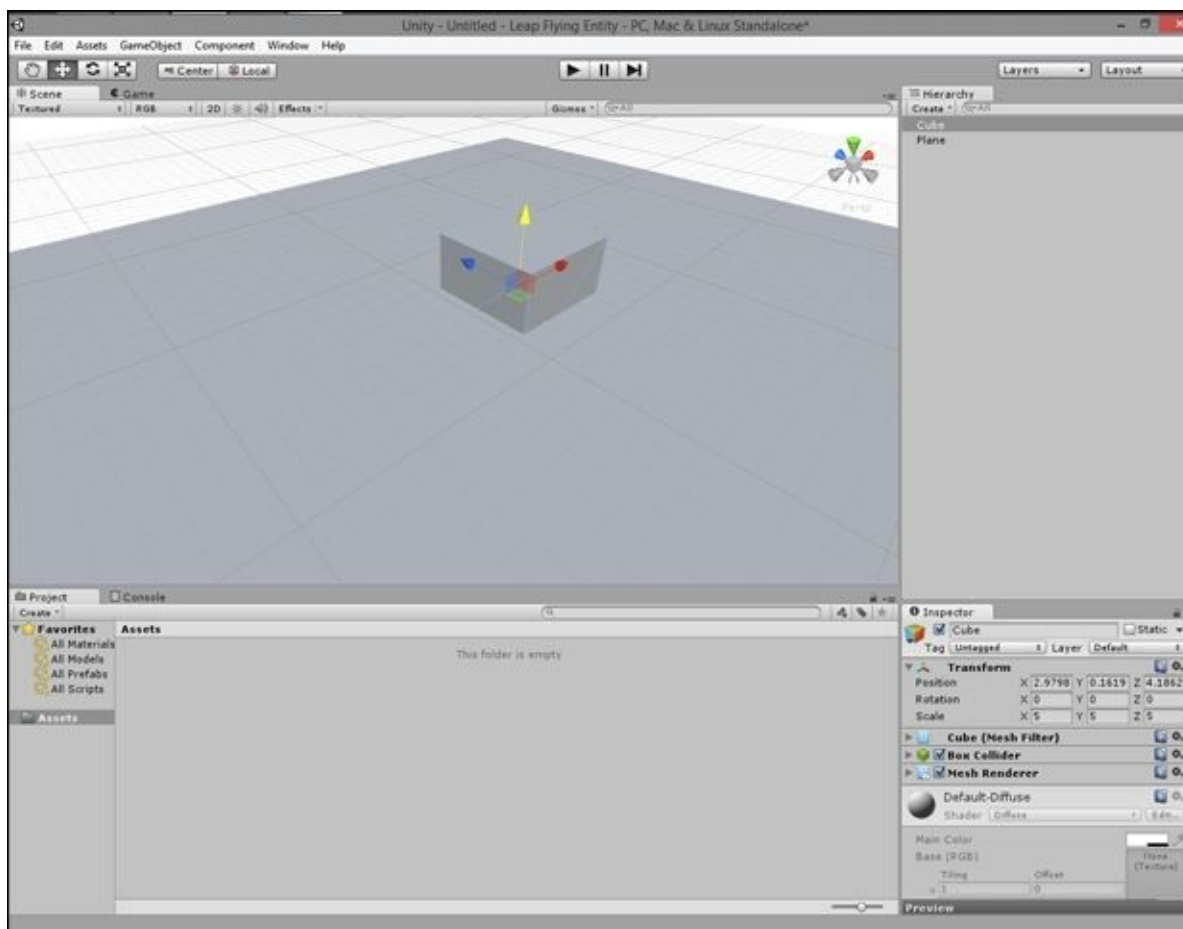
We will be making a new one later on, so go ahead and click on it and delete it (right-click on **Main Camera** and click on **Delete**). At this point, you should now have what is essentially an empty scene with the exception of a flat plane. How about we add some simplistic terrain to that plane? Head on over to the toolbar again and go to **GameObject | Create Other | Cube**, as shown here:



Behold, a lone cube is now present in the middle of the vast emptiness that is the **Scene** window, as shown here:



Moving on to the next step, let's make the cube a wee bit larger. Just as we did for the plane, go into the cube's **Transform** settings (via the **Inspector** window after clicking on the cube in the **Hierarchy** window). Then, go into the **Scale** field and set the **X**, **Y**, and **Z** values to 5. This will make your cube a whole lot bigger, as shown in the following screenshot:

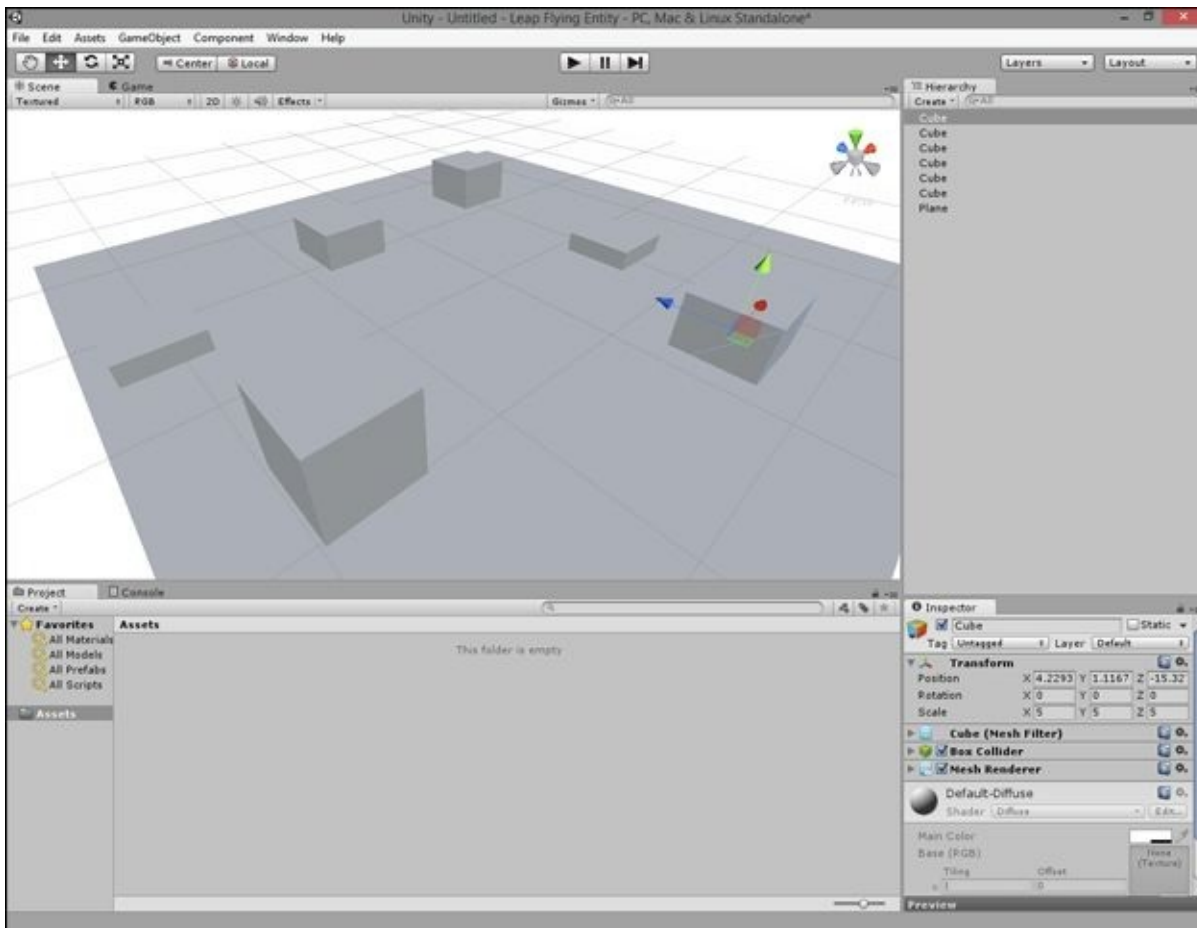


Now for the *artistic* part of laying out the scene—place the cube wherever you like in a logical fashion.

By now, you've probably noticed the three red, yellow, and blue arrows that protrude from the currently selected GameObject in the **Scene** window. These *handles* are used to move objects around within the **Scene** window without having to type in coordinates by hand from within that object's inspector.

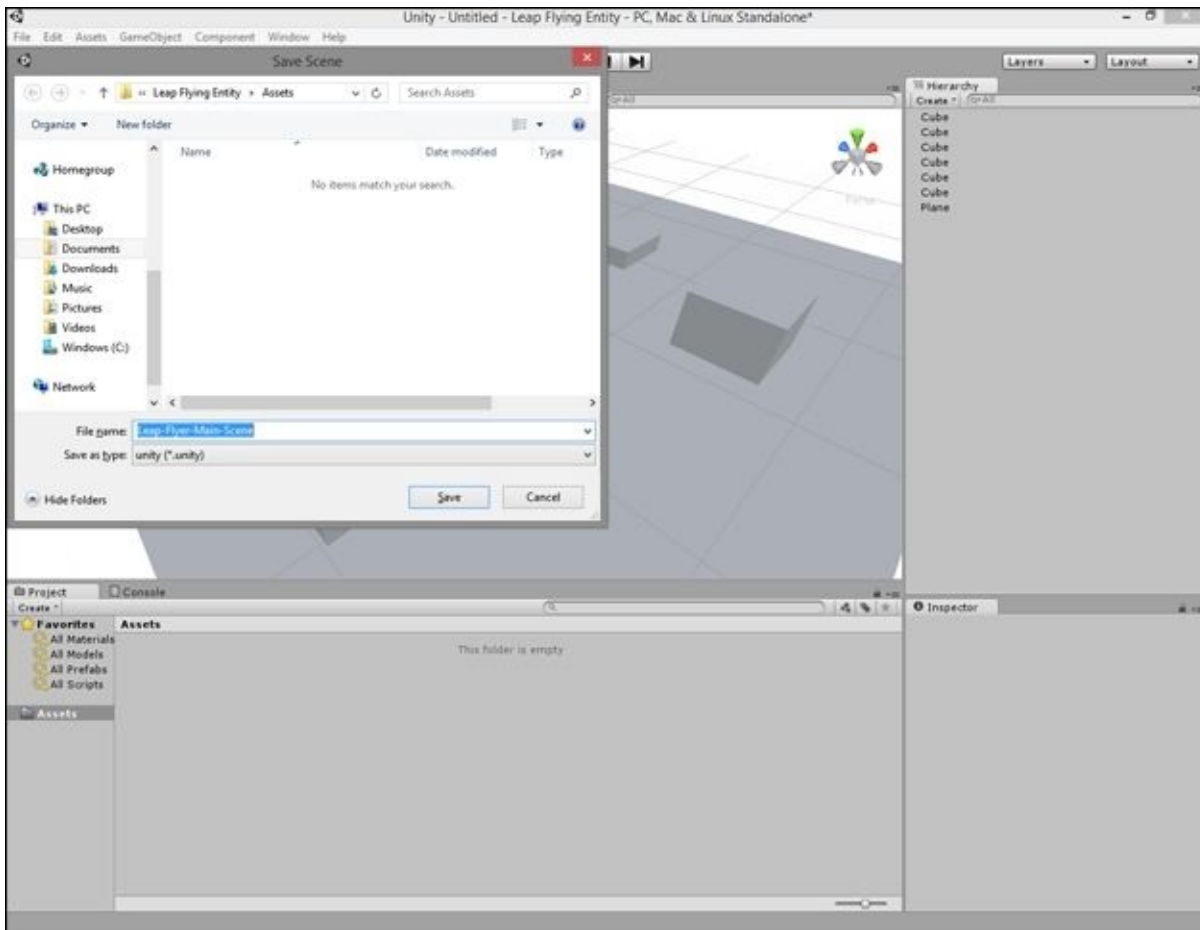
Go ahead and try it out for yourself: simply click and hold on any one of the arrows and then drag your cursor across the screen. When you're happy with the position of the object, stop clicking. Use this method to move the cube to a spot that's intersecting (or at least touching) the plane.

You can very quickly copy and paste new cubes into the scene by selecting one in the **Hierarchy** window, right-clicking its name, and then clicking on **Duplicate**. Go ahead and do this now: copy, paste and randomly position about five more cubes until your plane looks kind of like the one in the following screenshot:



Voilà! You now have a basic scene to work with. This chapter was probably either informative or slightly less interesting than the ones you've read thus far. Possibly both! However, I hope that it got you ready for 3D Leap Motion development for the next two chapters.

Before you forget, be sure to save the Unity project that you just finished working on! Simply go to the toolbar yet again and navigate to **File | Save Project** to save your project. If this is the first time you are saving your project, you will be greeted by a dialog like the one in the following screenshot that will ask you to give a name for the default scene—I named it Leap-Flyer -Main-Scene but you can use whatever name you like.



Summary

In this chapter, you learned about the Unity 3D editor, which will help you follow the upcoming chapters. We started by covering the installation of Unity, followed by a review of some common terms present in Unity, such as scenes, GameObjects, and scripts. You spent the remainder of the chapter learning about the creation of a project and the design and layout of a simple scene that you'll be using in the chapters to come.

You finished off the chapter by saving the project for the first time, although I hope you were actually saving all along.

In the next chapter, we'll begin integrating the Leap Motion Controller with the Unity 3D toolkit, including touchable buttons and real-time representations of a user's hands.

Chapter 6. Creating a 3D Application – Integrating the Leap Motion Device with a 3D Toolkit

Now that you're familiar with the Unity 3D toolkit, we can begin integrating the Leap into a 3D application. In this chapter, we'll cover the integration of the Leap device as well as the rendering of hands, fingers, and buttons using the C# programming language.

We'll be covering the following topics in this chapter:

- Setting up the scene to receive Leap Motion input
- A quick summary – the fundamentals of Unity scripts
- Laying out a framework of scripts
- Rendering hands
- Rendering buttons and detecting button presses

Note

Both this chapter and the next one make extremely heavy use of the C# programming language. If you're already familiar with it, great! If not, rest assured that it's, at first glance, quite similar to Java and/or C++. In addition, this chapter is sprinkled with periodic *Fun facts* that offer high-level and entry-level factoids about scripting and programming for your reading pleasure.

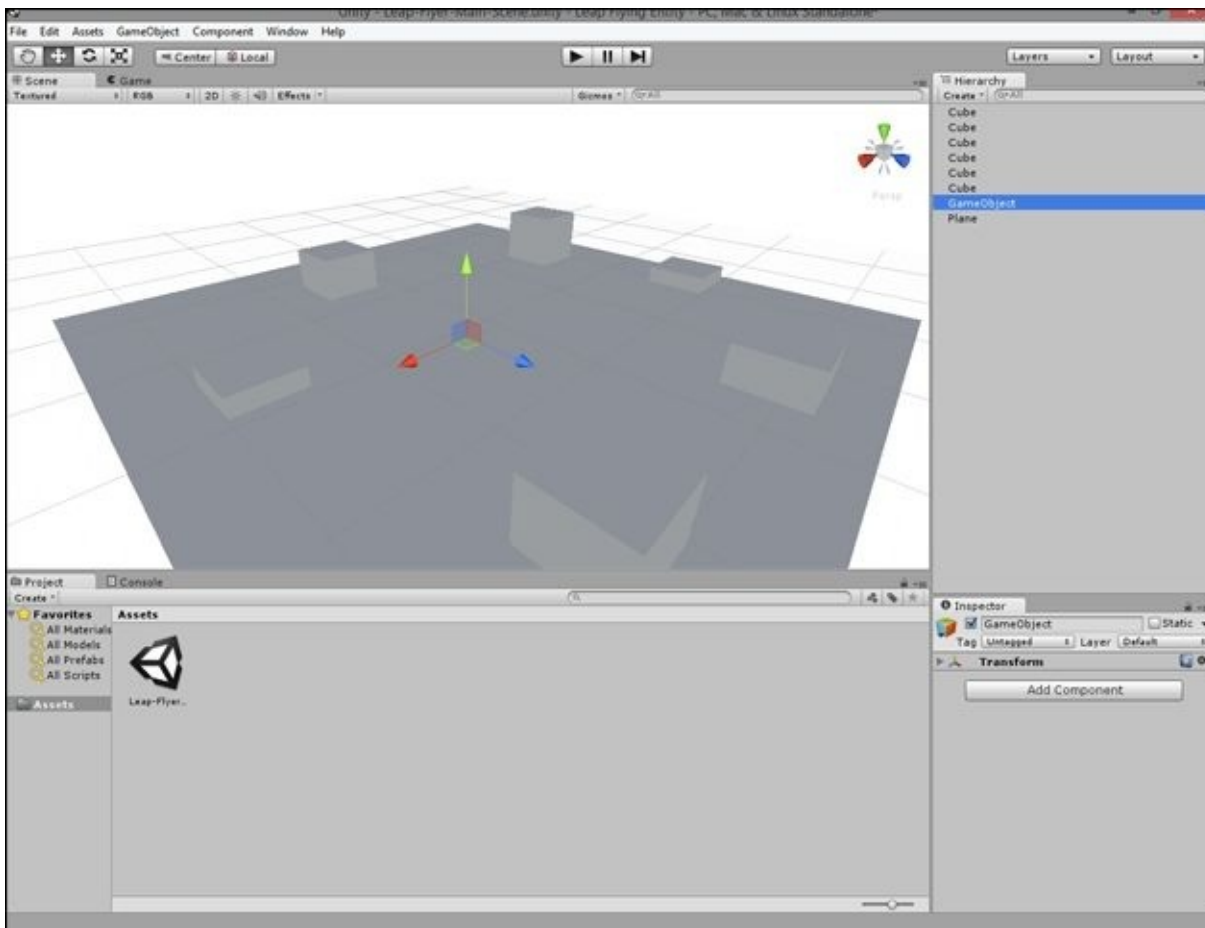
Setting up the scene to receive Leap Motion input

Welcome to *Chapter 6*. Let's get started!

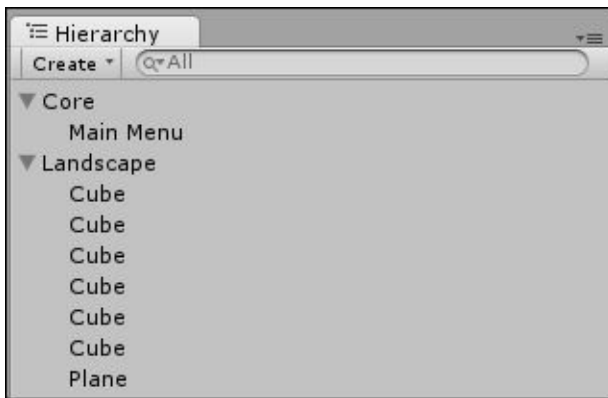
Before you can retrieve input and tracking data from the Leap Motion device, the scene must be modified to accommodate the required scripts to capture tracking data, manage menus, and so on.

Unity requires you to attach every single script you write to a `GameObject`; this means that you cannot just create a script and have it automatically work too. Fortunately, you can create *empty* `GameObjects` to attach our scripts with minimal hassle, and this is exactly what you're going to do now. So, without further ado...

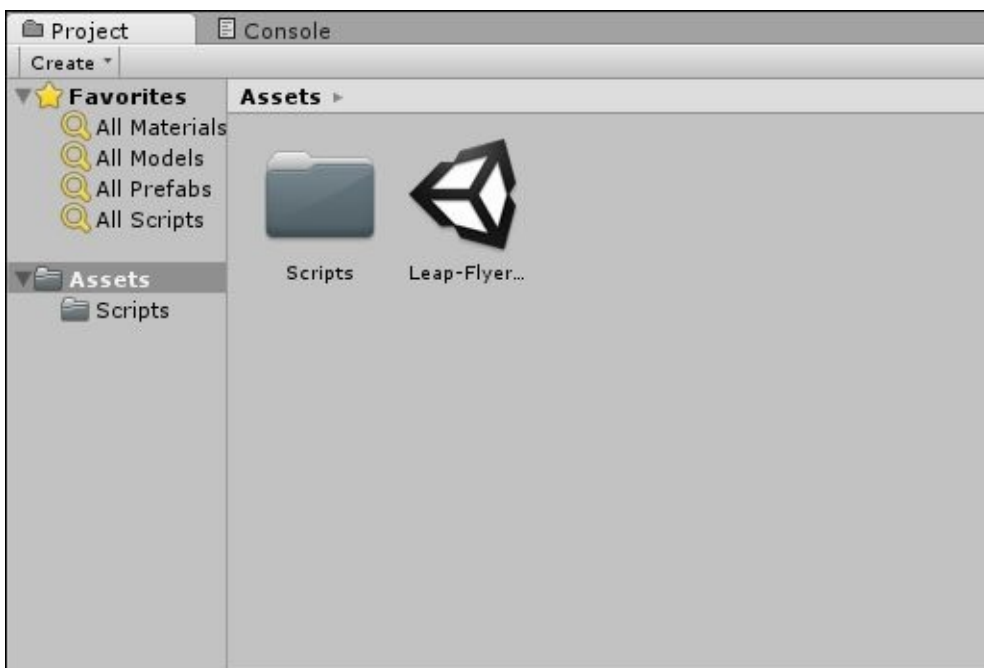
Within Unity, navigate to **GameObject | Create Empty** in the menu bar and select it. This will create a new empty `GameObject` in your scene and hierarchy windows, as shown in the following screenshot:



Right-click on the newly created `GameObject` and rename it to `Core`. Now, create one more `GameObject` called `Main Menu` and nest it underneath `Core`. Your **Hierarchy** window should now look like the following screenshot (for organizational purposes, I also grouped all the cube objects underneath a single empty `GameObject`):



Next, go ahead and create a Scripts folder in the **Assets** area, as shown in the following screenshot:



Almost done. Now, you need to drop the Leap DLL files into your project. If you don't, your code will not work, as it will try to utilize libraries that aren't there! The three DLLs we need are `Leap.dll`, `LeapCSharp.dll`, and `LeapCSharp.NET3.5.dll`, all of which can be found within the Leap SDK folder (refer back to *Chapter 1, Introduction to the World of Leap Motion*, if you need help finding it). Make sure that you choose the appropriate ones for your platform (x86 versus x64), otherwise nothing will work correctly!

Each one of these DLLs needs to go in a specific place, as listed here:

- `Leap.dll` and `LeapCSharp.dll` need to go in the root folder of your Unity project. As an example, if your project was named `My Leap 3D App`, the root folder would be `My Leap 3D App/`.
- `LeapCSharp.NET3.5.dll` needs to go in the `Assets` folder of your project. This is located under `[Project Name]/Assets`. Alternatively, you can simply locate the DLL and drag it directly into the assets window in Unity.

With this, you're ready to move on to the next step: writing code.

A quick summary – the fundamentals of Unity scripts

All software written for a Unity application is written inside files that Unity refers to as scripts. Scripts can be written in JavaScript, C#, and Boo, but we're going to focus only on C#, as that's what you'll be using to develop with Leap Motion.

Each C# script in Unity contains at least one class that extends Unity's built-in `MonoBehaviour` class. This class will almost always override one or more of the following functions from the parent `MonoBehaviour` class:

- `Awake`: This is called when the script is being loaded
- `OnEnable`: This is called when the script is enabled
- `Start`: This is called on the frame when the script is enabled just before any of the `Update` methods are called for the first time
- `Update`: This is called once during every frame
- `OnGUI`: This is called to render and handle GUI events
- `OnDisable`: This is called when the script is disabled
- `OnDestroy`: This is called when the script is destroyed

You can find a more exhaustive breakdown of the `MonoBehaviour` class and how it works at <http://docs.unity3d.com/ScriptReference/MonoBehaviour.html>.

Also, in addition to extending `MonoBehaviour`, every single script in Unity must be attached to a `GameObject` in order to work; just writing a script won't make anything happen.

Note

Fun fact

Now is a good time to take note of the spelling of `MonoBehaviour`. You will notice that it uses the British spelling of *behaviour* instead of the American spelling of *behavior*. Many Unity developers have fallen victim to this subtle difference in syntax, ending up with programs that “can't find the `MonoBehavior`” class!

Attaching a script to a GameObject

Attaching a script to a GameObject is as simple as performing a few steps. The following are the steps to attach a script to a GameObject:

1. Click on the name of the GameObject to which you want to add a script in the **Hierarchy** window.
2. Click on the **Add Component** button in the **Inspector** window for that GameObject.
3. Search for the name of the script you want in the resulting dialog and select it.

Laying out a framework of scripts

At long last, it's time to write some scripts and enjoy the oh-so-sweet instant gratification of seeing your hands in the three-dimensional format.

Note

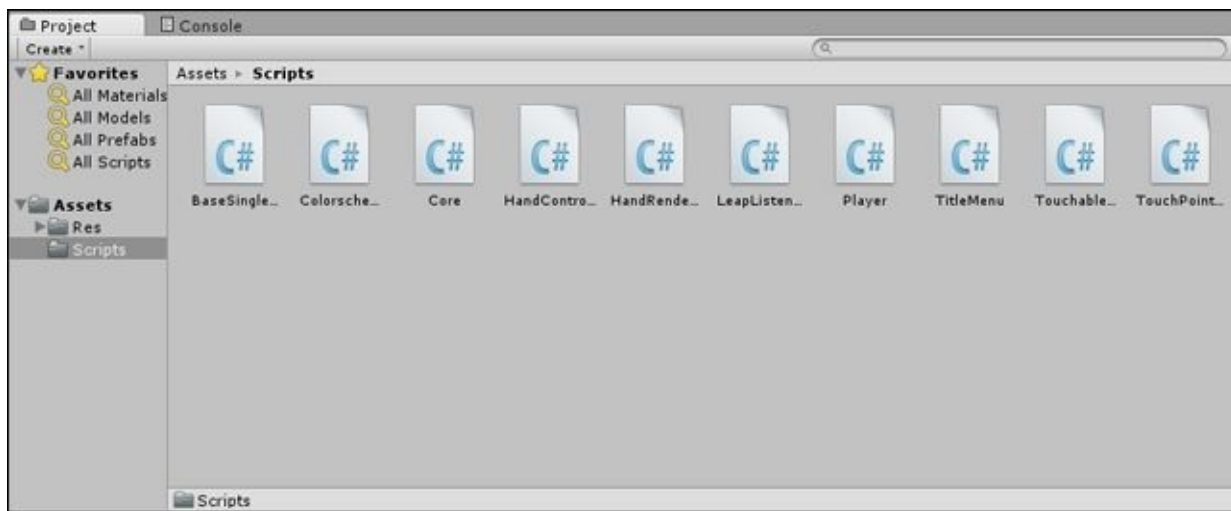
Fun fact

Both this chapter and the next one make use of adapted, simplified versions of scripts from the engine used in Artemis Quadrotor Simulator.

Before you begin writing code, create the following script files (we won't be filling in all of them right away) inside the Scripts folder by right-clicking on Scripts and going to **Create | C# Script**:

- BaseSingleton
- Colorscheme
- Core
- HandRenderer
- LeapListener
- TitleMenu
- TouchableButton
- TouchPointer

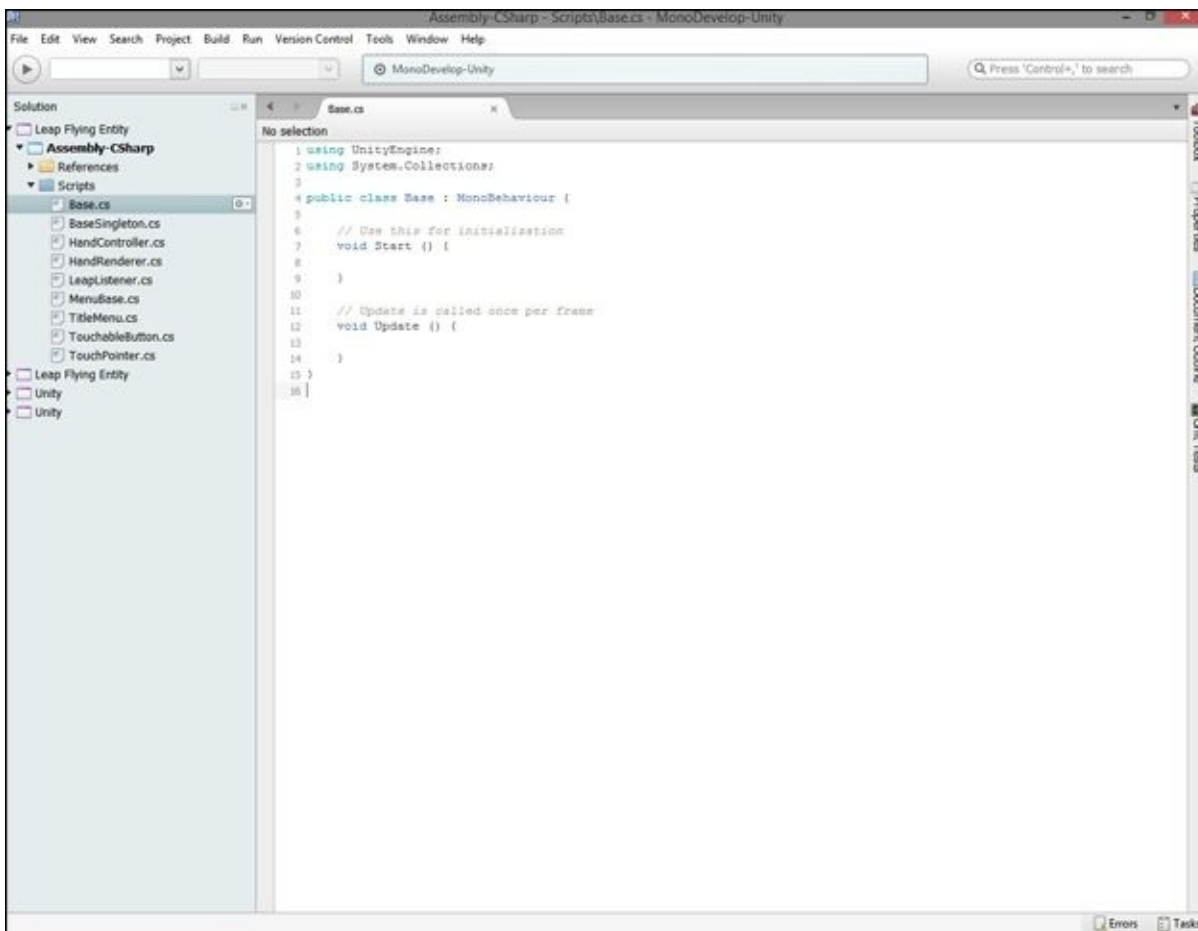
Your Scripts folder should look similar to the following screenshot when you're done (though not with quite as many scripts in it):



Rendering hands

For your first adventure out into the realm of Unity scripting, you'll be writing only two scripts, which will render the user's hands into the three-dimensional game world.

Go ahead and double-click on the `LeapListener` file that you created earlier inside the `Scripts` folder. This will make Unity automatically open MonoDevelop, the Integrated Development Environment used when writing scripts for Unity. Your screen should look a little bit like this:



The basic layout of MonoDevelop is pretty similar to Eclipse and other IDEs; you have the **Solution** browser on the left (which is basically just your project explorer), the workspace in the middle, and the file tabs at the top.

Go ahead and expand your `Scripts` folder in the **Solution** browser by clicking on [Name of your Unity Project Here] and going to **Assembly-CSharp | Scripts**, and then selecting `LeapListener` (if it wasn't already opened when you launched MonoDevelop).

LeapListener.cs

Your first bit of Leap Motion code lies directly ahead. This class, as I'm sure you've guessed, is the standard Leap Motion listener.

As there isn't a whole lot to say about this class, go ahead and open up this LeapListener.cs file and enter the following lines into it:

```
using Leap;

public class LeapListener
{
    //Leap controller.
    private Leap.Controller controller = null;

    //Minimum distance from hand for thumb to be recognized.
    public static float thumbDistance = 40;

    //Current frame.
    public Frame frame = null;

    //Fingers contained in the current frame.
    public int fingers = 0;

    //Hands contained in the current frame.
    public int hands = 0;

    //Various easy-access hand values.
    public float handPitch = 0.0F;
    public float handRoll = 0.0F;
    public float handYaw = 0.0F;

    //Hand and finger positions.
    public Vector handPosition = Vector.Zero;
    public Vector fingerPosition = Vector.Zero;
    public Vector handDirection = Vector.Zero;
    public Vector fingerDirection = Vector.Zero;

    //Quick-find for the right thumb.
    public Leap.Finger thumb = null;

    //Timestamp of the current frame.
    public long timestamp = 0;

    //Is the Leap connected?
    public static bool connected = false;

    //Member Function: refresh
    public bool refresh()
    {
        //Try.
        try
        {
            //If there's no controller, make a new one.
            if (controller == null) controller = new Leap.Controller();
        }
    }
}
```

```

//Check if the controller is connected.
connected = controller.IsConnected && controller.Devices.Count > 0 &&
controller.Devices[0].IsValid;

//If we're connected, update.
if (connected)
{
    //Get the most recent frame.
    frame = controller.Frame();

    //Assign some basic information from the frame to our variables.
    fingers = frame.Fingers.Count;
    hands = frame.Hands.Count;
    timestamp = frame.Timestamp;

    //If we see some hands, get their positions and their fingers.
    if (!frame.Hands.IsEmpty)
    {
        //Get the hand's position, size, and first finger.
        handPosition = frame.Hands[0].PalmPosition;
        handDirection = frame.Hands[0].Direction;
        fingerPosition = frame.Hands[0].Fingers[0].TipPosition;
        fingerDirection = frame.Hands[0].Fingers[0].Direction;

        //Get the hand's normal vector and direction.
        Vector normal = frame.Hands[0].PalmNormal;
        Vector direction = frame.Hands[0].Direction;

        //Get the hand's angles.
        handPitch = (float) direction.Pitch * 180.0f / (float)
System.Math.PI;
        handRoll = (float) normal.Roll * 180.0f / (float) System.Math.PI;
        handYaw = (float) direction.Yaw * 180.0f / (float)
System.Math.PI;

        thumb = null;

        //Find the thumb for the primary hand.
        foreach (Leap.Finger finger in frame.Hands[0].Fingers)
        {
            if (thumb != null && finger.TipPosition.x < thumb.TipPosition.x
&& finger.TipPosition.x < handPosition.x)
                thumb = finger;

            else if (thumb == null && finger.TipPosition.x < handPosition.x
- thumbDistance)
                thumb = finger;
        }
    }
}

//Otherwise, reset all outgoing data to 0.
else
{
    //Fingers contained in the current frame.

```

```

    fingers = 0;

    //Hands contained in the current frame.
    hands = 0;

    //Various easy-access hand values.
    handPitch = 0.0F;
    handRoll = 0.0F;
    handYaw = 0.0F;

    //Hand and finger positions.
    handPosition = Vector.Zero;
    fingerPosition = Vector.Zero;
    handDirection = Vector.Zero;
    fingerDirection = Vector.Zero;

    //Quick-find for the right thumb.
    thumb = null;
}

return true;
}

//In the event that anything goes wrong while reading and converting
tracking data, log the exception.
catch (System.Exception e) { UnityEngine.Debug.LogException(e); return
false; }
}

//Member Function: rotation
public UnityEngine.Vector3 rotation (Leap.Hand hand)
{
    //Create a new vector for our angles.
    UnityEngine.Vector3 rotationAngles = new UnityEngine.Vector3(0, 0, 0);

    //Get the hand's normal vector and direction.
    Vector normal = hand.PalmNormal;
    Vector direction = hand.Direction;

    //Set the values.
    rotationAngles.x = (float) direction.Pitch * 180.0f / (float)
System.Math.PI;
    rotationAngles.z = (float) normal.Roll * 180.0f / (float)
System.Math.PI;
    rotationAngles.y = (float) direction.Yaw * 180.0f / (float)
System.Math.PI;

    //Return the angles.
    return rotationAngles;
}
}

```

By now, this should all look very familiar; thankfully, writing a listener for use with the Unity scripting engine isn't a whole lot different from writing one in Java or any other language.

The only item I'd like to bring your attention to is the following function:

```
public UnityEngine.Vector3 rotation (Leap.Hand hand){  
    ...  
}
```

This function calculates the rotational values of the passed Leap hand and returns them as a set of Unity-recognizable vectors. Quite handy in the files to come!

Now, let's move on to the last of our files before we can do some playing/testing.

HandRenderer.cs

This class is the meat of our Leap Motion code so far; it renders all of the hands currently in view of the Leap on to the user's screen, which is a good way to get started with integrating the Leap into a 3D application.

So, without further ado, let's get started. Open up `HandRenderer.cs` and copy the following content into it (you can also find the entire file online at <https://github.com/Mizumi/Mastering-Leap-Motion-Unity-Project> in the `HandRenderer.cs` file under **Assets | Scripts**):

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

class HandRenderer : MonoBehaviour
{
    //Leap listener.
    private LeapListener listener;

    //Leap box.
    private Leap.InteractionBox normalizedBox;

    //Currently active fingers.
    private GameObject[] fingers;

    //Currently active palms.
    private GameObject[] hands;

    //Camera to render the hands on.
    public Camera camera = null;

    //Finger object.
    public GameObject fingerTip = null;

    //Palm object.
    public GameObject palm = null;

    //Distance modifiers.
    public float depth = 20.0F;
    public float verticalOffset = -20.0F;

    //OnEnable.
    public void OnEnable() {listener = new LeapListener();}

    //OnDisable.
    public void OnDisable()
    {
        //Reset the hands array.
        if (hands != null)
            for (int i = 0; i < hands.Length; i++)
                Destroy(hands[i]);

        //Reset the fingers array.
    }
}
```

```

    if (fingers != null)
        for (int i = 0; i < fingers.Length; i++)
            Destroy(fingers[i]);
}

//Update.
public void Update()
{
    if (listener == null) listener = new LeapListener();

    //Update the listener.
    listener.refresh();

    //Get a normalized box.
    normalizedBox = listener.frame.InteractionBox;

    //First, get any hands that are present.
    if (listener.hands > 0)
    {
        //Reset the hands array.
        if (hands != null)
            for (int i = 0; i < hands.Length; i++)
                Destroy(hands[i]);

        //Initialize our hands.
        hands = new GameObject[listener.hands];

        //Loop over all hands.
        for (int i = 0; i < listener.hands; i++)
        {
            try
            {
                //Create a new hand.
                hands[i] = (GameObject) Instantiate(palm);

                //Set its properties.
                hands[i].transform.parent = camera.transform;
                hands[i].name = "Palm:" + i;

                //Set up its position.
                Vector3 palmPosition = new Vector3(0, 0, 0);

                palmPosition.x += listener.frame.Hands[i].PalmPosition.x / 10;
                palmPosition.y += verticalOffset; palmPosition.y +=
listener.frame.Hands[i].PalmPosition.y / 10;
                palmPosition.z += depth; palmPosition.z +=
(listener.frame.Hands[i].PalmPosition.z * -1) / 10;

                //Move the hand.
                hands[i].transform.localPosition = palmPosition;

                //Set the hands rotation to neutral.
                Quaternion lr = hands[i].transform.rotation;

                Vector3 leap = listener.rotation(listener.frame.Hands[i]);

```

```

        lr.eulerAngles = new Vector3(leap.x * -1, leap.y, leap.z);

        hands[i].transform.localRotation = lr;
    }

    //Watch out for those pesky "index out of bounds" errors.
    catch (System.IndexOutOfRangeException e) { Debug.LogException(e);
}
    }
}

//If there aren't any, delete any active palms.
else if (hands != null)
    for (int i = 0; i < hands.Length; i++)
        Destroy(hands[i]);

//Get any fingers that are present.
if (listener.fingers > 0 && listener.hands > 0)
{
    //Reset the fingers array.
    if (fingers != null && listener.fingers != fingers.Length)
        for (int i = 0; i < fingers.Length; i++)
            Destroy(fingers[i]);

    //Initialize our fingers.
    if (fingers == null || listener.fingers != fingers.Length)
        fingers = new GameObject[listener.fingers];

    //Loop over all fingers.
    for (int i = 0; i < listener.fingers; i++)
    {
        try
        {
            //Create a new finger.
            if (fingers[i] == null)
                fingers[i] = (GameObject) Instantiate(fingerTip);

            //Set its properties.
            fingers[i].name = "Finger:" + i;
            fingers[i].transform.parent = camera.transform;

            //Set up its position.
            Vector3 tipPosition = new Vector3(0, 0, 0);

            tipPosition.x += listener.frame.Fingers[i].TipPosition.x / 10;
            tipPosition.y += verticalOffset; tipPosition.y +=
listener.frame.Fingers[i].TipPosition.y / 10;
            tipPosition.z += depth; tipPosition.z +=
(listener.frame.Fingers[i].TipPosition.z * -1) / 10;

            //Move the finger to where it belongs.
            fingers[i].transform.localPosition = tipPosition;

            //Set the fingers rotation to neutral.
            Quaternion lr = fingers[i].transform.rotation;

```



```

        lr.eulerAngles = Vector3.zero;

        fingers[i].transform.localRotation = lr;
    }

    //Watch out for those pesky "index out of bounds" errors.
    catch (System.IndexOutOfRangeException e) { Debug.LogException(e);
}
    }
}

//If not, delete any active fingers.
else if (fingers != null)
    for (int i = 0; i < fingers.Length; i++)
        Destroy(fingers[i]);
}
}

```

I'm fairly certain that `HandRenderer` is one of the single longest scripts that you'll be writing in this chapter. Let's go ahead and break it down.

`OnEnable` and `OnDisable` are pretty straightforward, but as far as `Update` goes, this is where all the action takes place. The first thing we do is refresh the tracking data from the Leap listener, followed by retrieving an `InteractionBox` instance to *normalize* our values with.

Next, we check whether any hands are present. If there aren't any, all the hands that are currently loaded into the game world are deleted along with their fingers (but I promise it isn't nearly as gruesome as it sounds).

If there *are* hands in the current frame in the Leap, we proceed to calculate where to place the user's hands in the game world. First, we loop over all the hands that are currently loaded into the game world (if any) and delete them. We then loop over all the hands that are contained in the most recent frame from the Leap, applying some fun math and transformation to their coordinates to normalize them into the game world.

We repeat this process for any fingers in the current frame in the Leap (but only if hands were present in that same frame).

Note

Fun fact

Unity automatically renders any `GameObjects` created via scripting on screen. You were probably wondering when and/or how all those hands and fingers that we were putting into arrays were going to get displayed; the truth is, they already are!

You might have noticed the definitions for two arrays of `GameObjects` at the start of the file: `GameObject[] fingers` and `GameObject[] hands`. Any of the `GameObjects` we place in here (in fact, any `GameObjects`' period) are automatically rendered and handled by Unity. These two arrays are used so that we can keep track of all the active hands and fingers to make deleting or refreshing them all at once a piece of the proverbial cake.

Preparing the scene for hand rendering

After all that coding, let's go ahead and return to the *graphical* realm for a little bit.

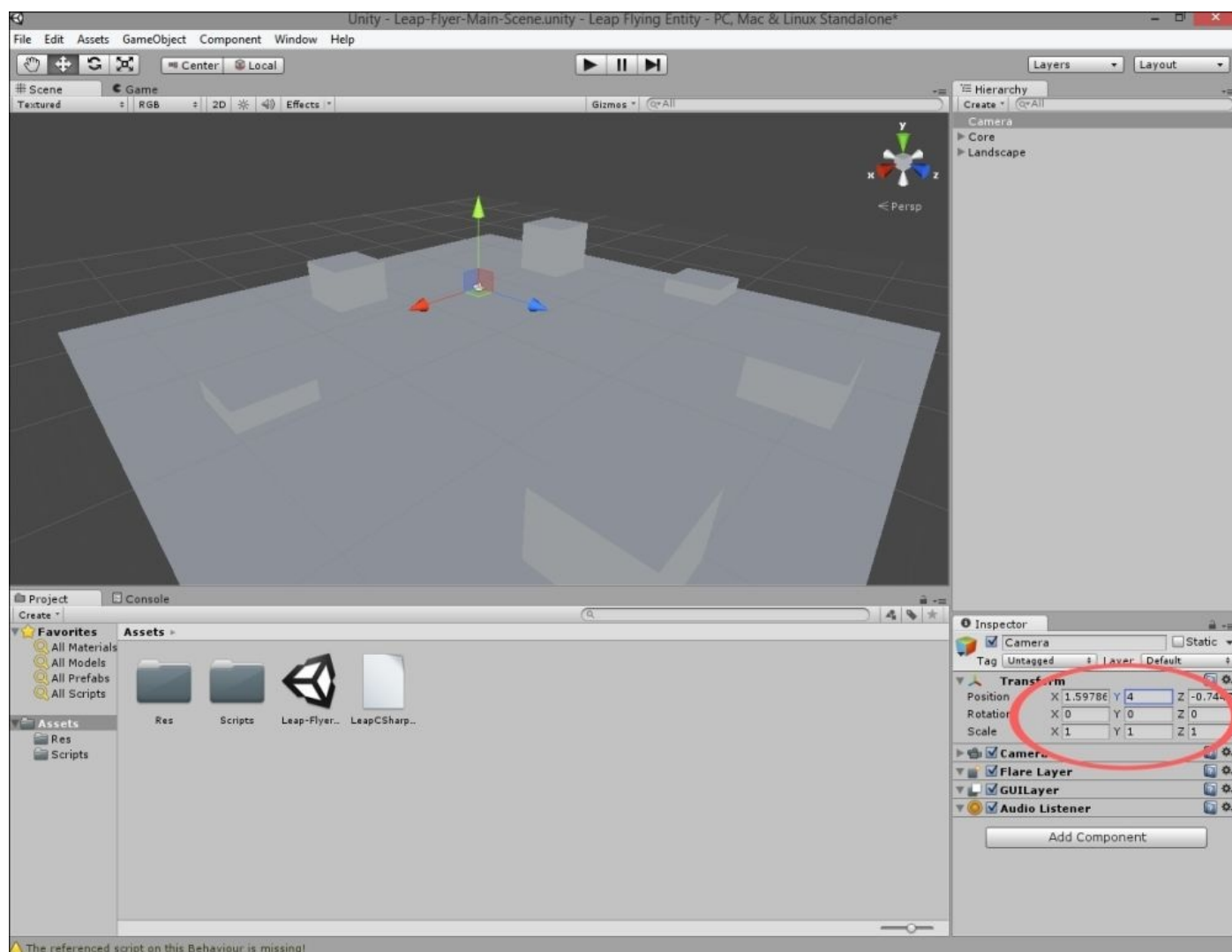
First things first: we need to add in a camera and some **prefabs** so that we can see the user's fingers rendered on screen. Go ahead and return to the Unity Editor (which will probably start compiling your scripts).

Note

Fun fact

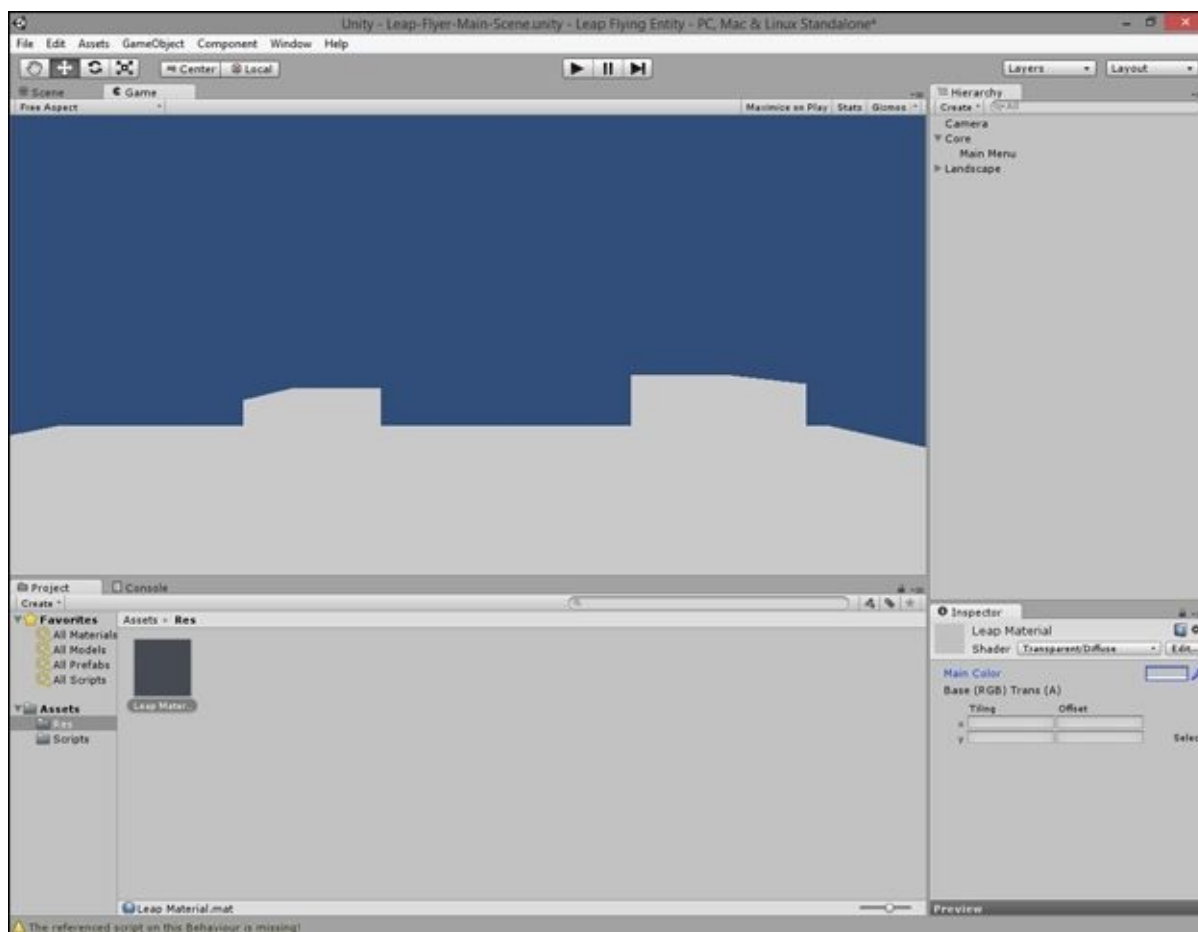
A prefab, as Unity calls it, is a predefined collection of GameObjects and their configurations. These wonderful little things allow developers to save complex combinations of GameObjects (including scripts and customizations made via the Inspector) into a single file for easy portability and creation via scripts later.

Now, add a new camera to the scene by going to **GameObject | Create Other | Camera** and setting its y axis position (circled in the following screenshot) in the Inspector to something around 4 so that it isn't underneath the floor:

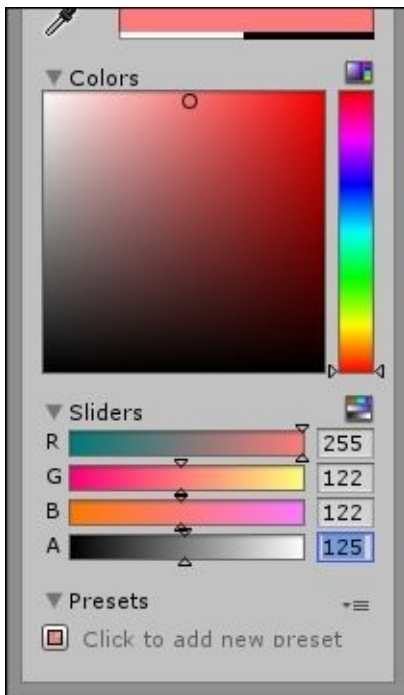


With that done, create a new folder in your Assets folder called Res (or resources if you

will). Now, right-click on the Res folder and go to **Create | Material**. Your Res folder should now look similar to the following screenshot:



Go ahead and select your newly created material and rename it Leap Material for convenience sake. Once you do this, within the **Inspector** window for your new material, click on the **Shader** drop-down box and go to **Transparent | Diffuse** from the list. After you do this, click on the little box to the right of **Main Color** in the **Inspector** window for Leap Material. You'll now see a color picker dialog similar to the one shown here:



You can choose literally any color that takes your fancy (I went with red in the preceding screenshot), but make sure to set the **A** (alpha) value to 125; this will make the material half transparent.

Note

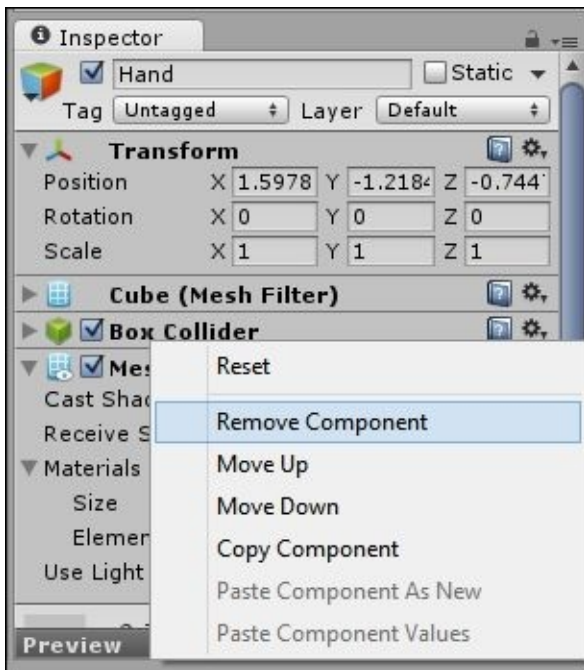
Fun fact

Every single color within Unity (in scripting, anyway) is an instance of the `Color` class. This class contains four main values that you'll use quite often: R, G, B, and A, which stand for red, green, blue, and alpha, respectively.

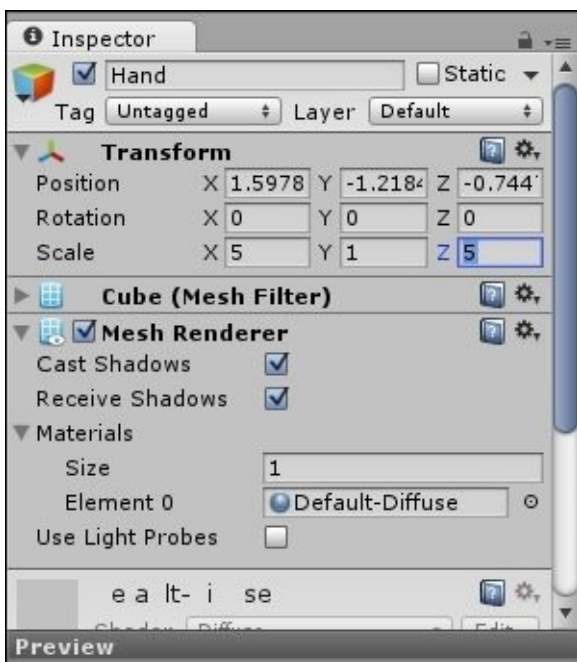
The first three values should be pretty familiar to you already, but it's possible that you might not be familiar with the alpha property of a color; the alpha of a color determines its perceived transparency within Unity, on a scale of 0 to 255, with 0 being completely transparent and 255 being completely solid.

Now, it's time to make prefabs for the fingers and hands that will be shown in the game world by the `HandRenderer` class. Go ahead and create two new cubes by going to **GameObject | Create Other | Cube** and then rename one to `Hand` and the other to `Finger` via the **Hierarchy** window.

With the `Hand` and `Finger` cubes created, select both of them from within the **Hierarchy** window and then look toward the **Inspector** window. You should see a component there called **Box Collider**. Right-click on this and remove it, as shown in the following screenshot:



With Box Collider removed (see the preceding screenshot), you should now select just the Hand cube and set its **X** and **Z** Transform Scales to 5 via the Inspector, as shown in the following screenshot:



Finally, select both the Hand and Finger cubes again, this time expanding the **Mesh Render** component and clicking on the little circle next to the box that says **Default-Diffuse** next to **Element 0**, highlighted in the following screenshot:

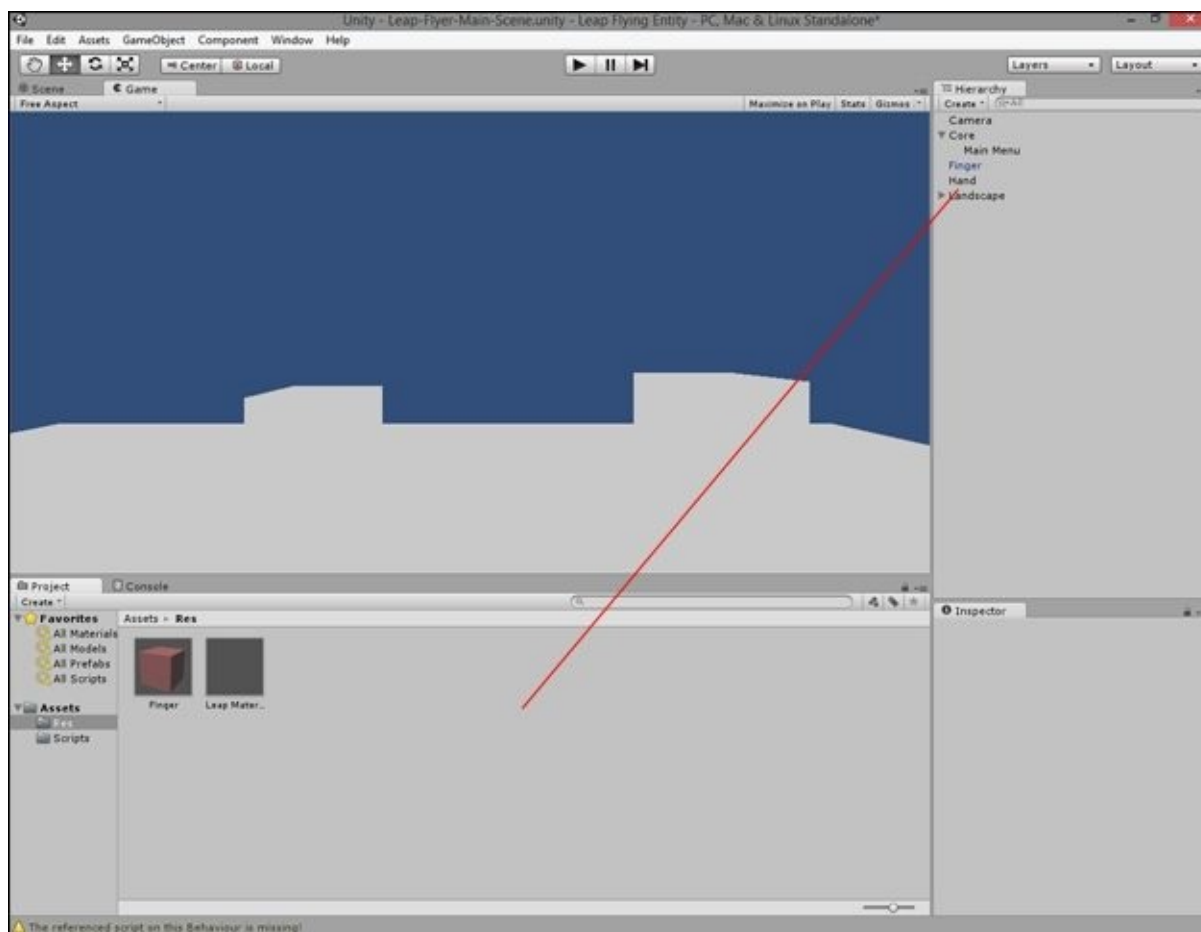


You will now see a dialog window appear on the screen, similar to the following screenshot:



Go ahead and select **Leap Material** from the dialog window that appears; this will make both cubes appear to be of the color that you chose earlier when setting up the **Leap Material** object.

For the last cube-related step, drag both cubes (one at a time) from the **Hierarchy** window into the Res folder under Assets, shown in the following screenshot:

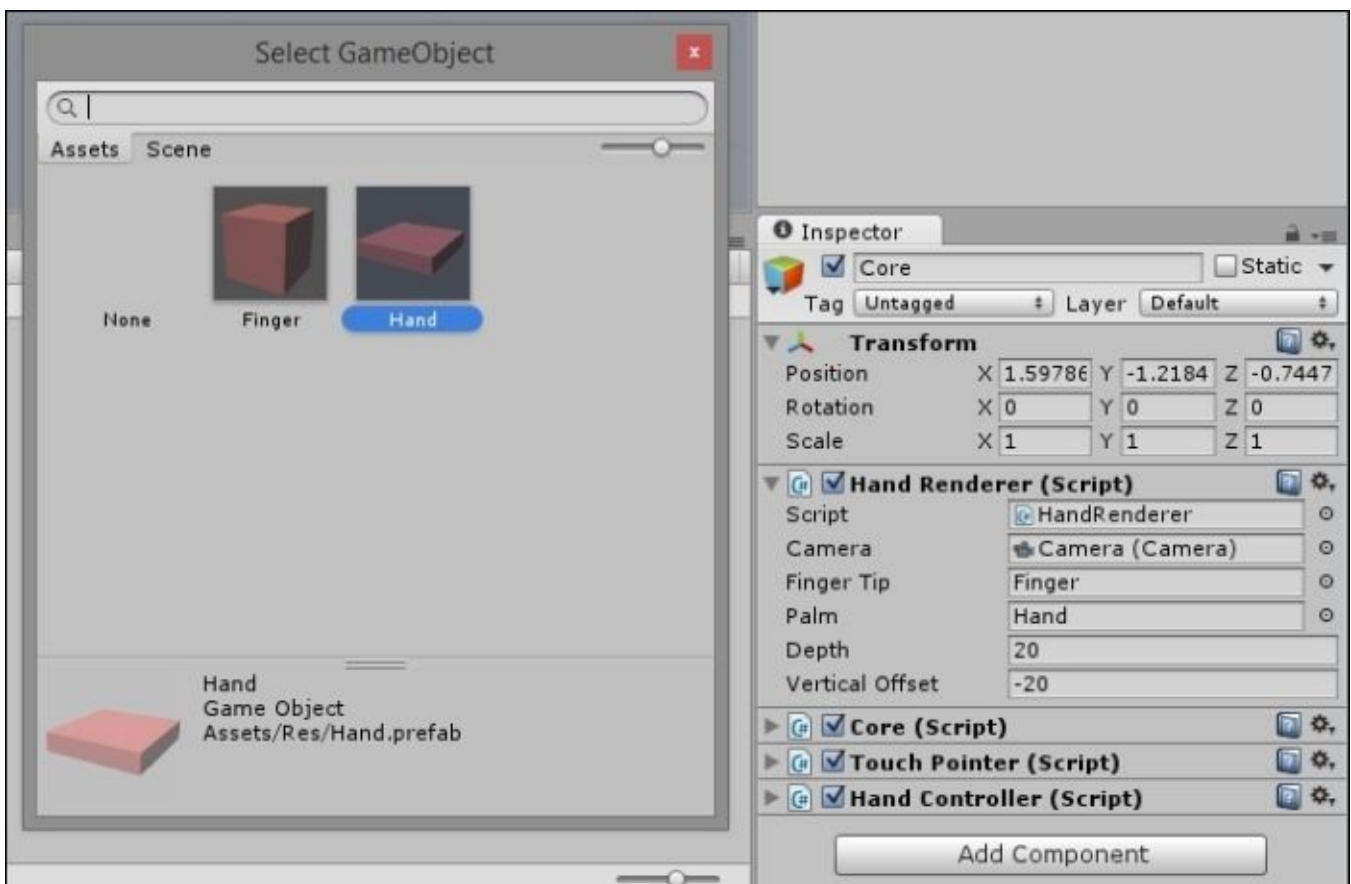


Once you drag both of these cubes in, delete them from the **Hierarchy** window, as they're no longer needed there; in the process of dragging the cubes into the Assets folder, you created two new prefabs that contain all the configuration data for their respective cube, allowing you to make an infinite number of them later.

Finally, we need to activate the `HandRenderer` class. To do this, click on the Core GameObject in the **Hierarchy** window and navigate to **Add Component | Scripts | HandRenderer.cs**, as shown in the bottom right-hand corner of the following screenshot:



This will add the Hand Renderer script to the Core GameObject, making it visible in the Inspector, as shown in the following screenshot:



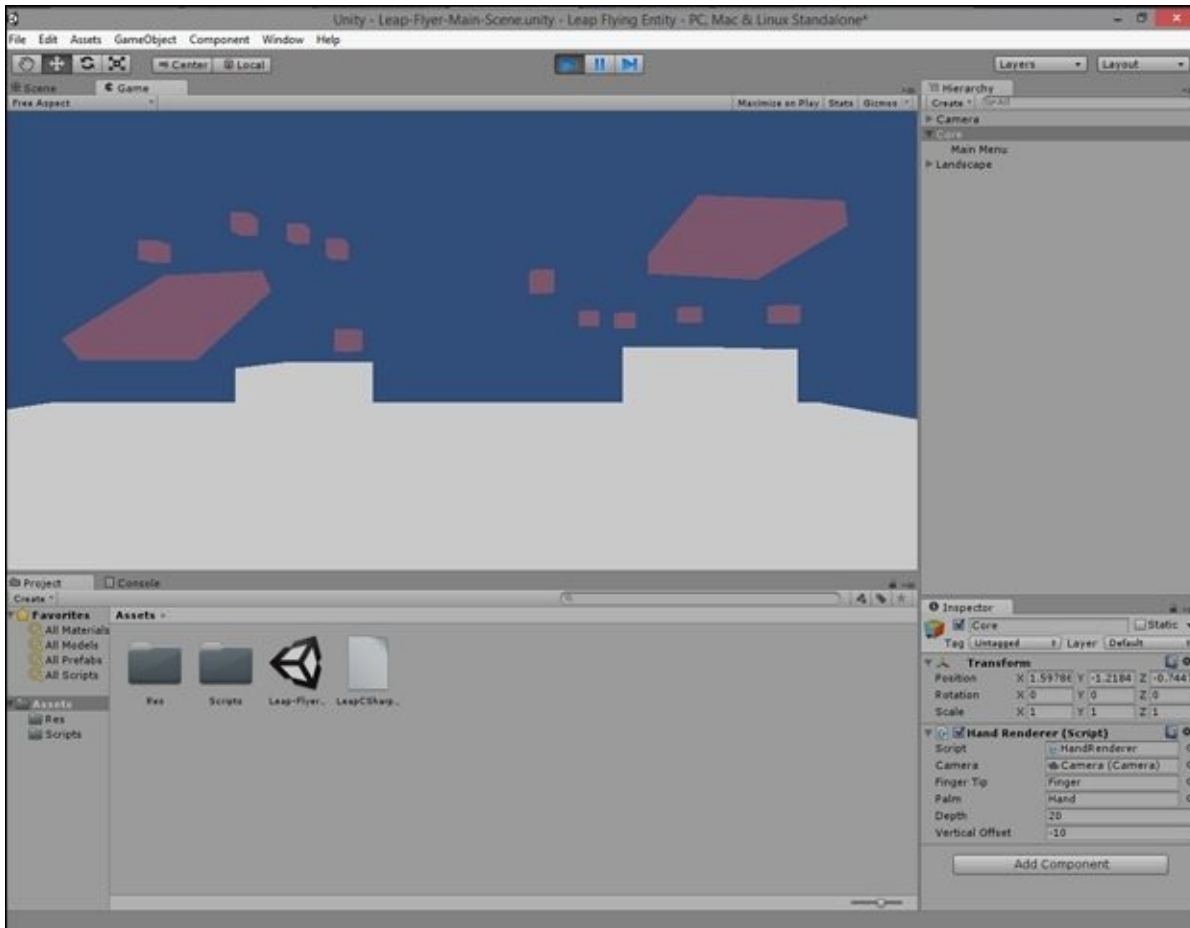
You'll notice that the **Camera**, **Finger Tip**, and **Palm** fields listed underneath the Hand Renderer (script) component are empty. Click on the little circle next to each one, selecting the appropriate item as listed here:

- Select the **Finger** prefab for the **Finger Tip** field
- Select the **Hand** prefab for the **Palm** field
- Select **Camera** from **Hierarchy** for the **Camera** field

Now you're done.

Testing out the Hand Renderer

All that's left is to run it and see whether everything works. To do this, simply click on the arrow icon above the **Scene** window. If all goes well, you should see virtual representations of your hands when you place them into view, as shown in the following screenshot:



Rendering buttons and detecting button presses

Now that you've got some hands on the screen, let's give those hands something to manipulate; to me, buttons would seem to be the next logical step. Don't you agree?

Get ready to write lots of code (six files, to be exact), because touchable buttons are no laughing matter in a 3D toolkit (well, they aren't so bad, but still). Head on over to the `Scripts` folder and double-click on the `BaseSingleton.cs` file; this should open up MonoDevelop if it wasn't already open to begin with.

BaseSingleton – a custom singleton pattern

Our first class, as the name suggests, is a custom implementation of the singleton pattern. What's a singleton, you ask? Read on!

In every program—and consequently, every game—you need to have some form of global logic that can be accessed from anyone, anywhere, and at any time, be it the current high score in a game or a list of available sensors on a robot. Of course, in almost all object-oriented languages, all of your logic will be contained in instances of classes. This results in, you guessed it, multiple instances of their variables; that simply won't do. So, what do you do when you want to share global values within classes while guaranteeing that there's only ever one set of these values?

Meet the singleton pattern. In its simplest form, a singleton is a very special kind of class that will only ever have one instance in existence at any given moment. This allows us to guarantee that we'll only ever have one instance of a given set of variables, functions, and so on. We achieve this by utilizing a combination of a private constructor and private instance of the class, as seen in the simplistic code here:

```
public class MySingleton()  
{  
    private static final MySingleton instance = new MySingleton();  
  
    private MySingleton() { };  
  
    public String variable = "something";  
  
    public static MySingleton getInstance() { return instance; }  
}
```

You can then access the variables in `MySingleton` from anywhere in your code using the following syntax:

```
String myLocalVariable = MySingleton.getInstance().variable;
```

Now, you'll notice a few things about the preceding code: there's a static instance of the `MySingleton` class that is immediately initialized when the code is loaded, a private constructor that can never be called from outside of the `MySingleton` class, and a static `getInstance` function that returns a reference to the static instance of `MySingleton`. All of these things work together, creating the common singleton pattern.

But wait, there's more! As we're using Unity, there are a few extra problems to solve:

- Unity doesn't use constructors; instead, it uses a public `Awake` method. This means there's no way to prevent our singleton's initialization function from being called from outside of the class.
- As any script, and therefore any class, can be attached to an arbitrary number of `GameObjects`, there is absolutely nothing that stops Unity from trying to load multiple instances of the same singleton.

So then, given the insurmountable odds working against us, what do we do? No worries,

the solutions are quite simple:

- We can solve the lack of constructors by keeping track of whether or not the `Awake` function is called. When it's called, we'll set a static Boolean to true.
- When our singleton executes the `Awake` function, it will first check whether there is a pre-existing instance in play (and whether it's already *awoken*). If there is, our singleton will immediately call `Destroy` on itself, preventing it from coming into existence.

Note

Fun fact

Unity's built-in `Destroy` function completely and utterly decimates the passed `GameObject` or script, removing it from this realm of existence.

It's a great way to clean up memory, but it's also an excellent way to crash your game if you're not careful!

Although the behavior I've described here is rather unusual for a normal singleton, it's absolutely required if we're going to create a Unity-compatible singleton. Of course, talking about code is all well and good, but how about we turn all of this into a class called `BaseSingleton`? Go ahead and write the following lines of code into the `BaseSingleton.cs` file:

```
using UnityEngine;

//Class: BaseSingleton
public class BaseSingleton<T> : MonoBehaviour where T : MonoBehaviour
{
    //Instance of this type.
    private static T instance;

    //Has this type already been enabled?
    private static bool awoken = false;

    //Member Function: Awake
    public void Awake()
    {
        //If an instance already exists, delete this one.
        if (instance != null && instance != this)
            Destroy(this);

        //Otherwise, proceed to initialization.
        else
        {
            //Check if we should wake up.
            if (!awoken)
            {
                //Awake will now no longer be called.
                awoken = true;

                //Wake up.
                onAwake();
            }
        }
    }
}
```

```

    }
}

//Member Function: onAwake.
public virtual void onAwake() {}

//Member Function: getInstance.
public static T getInstance()
{
    //If there is no instance to return, generate a new one.
    if (instance == null)
    {
        //First attempt to see if there's already an instance
        //attached to an object and use that.
        try
        {
            instance = (T) Object.FindObjectOfType(typeof(T));
        }

        //Otherwise, create a new object with an instance.
        catch
        {
            GameObject instanceObject = new GameObject(typeof(T).ToString());
            instanceObject.AddComponent<T>();
            instance = instanceObject.GetComponent<T>();
        }

        //Call the instance's awake.
        if (!awoken)
        {
            awoken = true;

            instance.Invoke("onAwake", 0.0F);
        }
    }

    //Return the current instance.
    return instance;
}
}

```

Let's break down the BaseSingleton class.

In one of the first few lines, we have this:

```
public class BaseSingleton<T> : MonoBehaviour where T : MonoBehaviour
```

We define the BaseSingleton class as a template class and extend the MonoBehaviour class so that any of the classes that extend this one can be attached to GameObjects. This allows any of our classes to extend it and automatically become a singleton. Isn't that nifty?

Note

Fun fact

Template classes are special classes that can take an arbitrary data type (that is, class) during instantiation and then use this type to process data later. Although template classes are out of the scope of this book, this one in particular (`BaseSingleton`) is useful, as it saves us from having to rewrite the same code three or four times later on.

In the next few lines:

```
public void Awake()  
{  
    //If there's already an instance, delete this one.  
    ...  
  
    //Otherwise, proceed to initialize.  
    ...  
}
```

We override the `Awake` function of `MonoBehaviour` to check whether an instance of this class has already been created. If an instance has been created, the new one will be immediately destroyed so as to avoid having multiple instances. Otherwise, our `onAwake` function will be called.

As we're overriding the `Awake` function of `MonoBehaviour`, we can't let classes that inherit from `BaseSingleton` override `Awake` again—that'd be disastrous! This is where our next function, `onAwake`, comes into play:

```
public virtual void onAwake() {}
```

Classes that inherit from `BaseSingleton` will now have to override `onAwake` instead of `Awake`. When the class gets loaded into the scene (this usually happens when a `GameObject` the script is attached to gets loaded), the `Awake` function from `BaseSingleton` will be called. Then, if the singleton hasn't been initialized yet, it will call `onAwake`. This guarantees that the initialization routines for any given class that decides to extend `BaseSingleton` will only be called once.

Finally, in `GetInstance`, we make brief use of the two rather obscure functions that I'd like to point out: `FindObjectOfType` and `Invoke`.

The first function, `FindObjectOfType`, tries to find any currently loaded `GameObjects` that have the specified type (or script) attached to them. As this function is slow, it's usually better to figure out an alternative method to locate classes. In our case, though, it's perfectly fine—we'll only call it once under normal conditions.

The second function, `Invoke`, is a bit more controversial. In essence, `Invoke` tries to trigger a function (denoted by a string) that may or may not be contained by the object it is called on. Normally, the usage of this function is a terrible idea, because calling a function that may or may not even exist can cause all kinds of errors. However, as we want to call `onAwake` on a generic template object, which has no class (and therefore no defined `onAwake` function), we're forced to use the `Invoke` method. However, in our case, we can always guarantee the object that we're calling `Invoke` on will have an `onAwake` method, as it will have inherited from our `BaseSingleton` class.

So, now that we've spent a lot of time talking about singletons, logic, and other less than interesting things, how about we move on to something a bit more colorful?

Colorscheme – a utility class to keep track of colors

Colorscheme is a relatively simple class used to color all the menus and miscellaneous user interface (UI/GUI) elements in the Unity application that you're creating; it includes a simple palette of colors and a built-in method to convert them to grayscale equivalents.

Go ahead and open up the Colorscheme.cs file from within the solution browser in MonoDevelop and enter the following lines of code:

```
using UnityEngine;
using System;

//This class is marked as serializable to enable direct editing of the
//public colorscheme values from the Unity Inspector.
[Serializable]
public class Colorscheme
{
    //Original colorscheme that we can revert to when switching
    //between greyscale and normal colors.
    private Colorscheme original;

    //Is this colorscheme currently set to greyscale?
    private bool greyscale = false;

    //Primary color.
    public Color primary = new Color(0, 0, 0, 0);

    //Secondary color.
    public Color secondary = new Color(0, 0, 0, 0);

    //Accent color #1.
    public Color primaryAccent = new Color(0, 0, 0, 0);

    //Accent color #2.
    public Color secondaryAccent = new Color(0, 0, 0, 0);

    //Special color.
    public Color special = new Color(0, 0, 0, 0);

    //Member Function: greyscale.
    public void setGreyscale(bool grey)
    {
        //Set to greyscale if this colorscheme isn't already greyscale.
        if (grey && greyscale == false)
        {
            //Backup current colors.
            original = (Colorscheme) this.MemberwiseClone();

            //Get the greyscale versions of all colors.
            primary = new Color(primary.grayscale, primary.grayscale,
primary.grayscale, primary.a);
            secondary = new Color(secondary.grayscale, secondary.grayscale,
secondary.grayscale, secondary.a);
            primaryAccent = new Color(primaryAccent.grayscale,
primaryAccent.grayscale, primaryAccent.a);
        }
    }
}
```

```

        secondaryAccent = new Color(secondaryAccent.grayscale,
secondaryAccent.grayscale, secondaryAccent.grayscale, secondaryAccent.a);
        special = new Color(special.grayscale, special.grayscale,
special.grayscale, special.a);

        //Now greyscale.
        greyscale = true;
    }

    //Remove greyscale.
    else if (grey == false && greyscale)
    {
        //Restore original colors.
        primary = original.primary;
        secondary = original.secondary;
        primaryAccent = original.primaryAccent;
        secondaryAccent = original.secondaryAccent;
        special = original.special;

        //No longer greyscale.
        greyscale = false;
    }
}
}
}

```

There isn't a whole lot to explain here, as Colorscheme is just a simple class that acts as a container for a collection of different colors.

One thing to note before we move on to the next class is the [Serializable] flag that we've placed before the definition of the Colorscheme class. By marking the class as serializable, we are able to directly modify all the public variables (such as colors, vectors, strings, integers, and so on) directly from the Inspector in Unity. This is a very nifty feature if you want to make a class slightly generic and allow it to be customized by developers in Unity later on.

Core – the main class, if Unity had main classes

When you're working with a collection of arbitrary scripts that are barely tied together at all, you need at least one class that acts as a sort of *glue* between them: a common area for certain pieces of information to be exchanged.

In this case, the Core class will be responsible for handling the pausing and menu features contained within our example application. During each update frame, the Core class will check whether a certain number of hands are within the Leap's field of view—if they aren't, it will automatically pause the game and bring up the menu (if the menu isn't open already). Likewise, if there are hands within the view, it will make sure that the game is not paused.

This is where the Core.cs script comes into play (as well as the Core GameObject you made earlier if you recall). Despite the important functions this class serves, it's quite short compared to the other ones we've written—fewer than a hundred lines, even with comments!

So, go on ahead and open up the Core.cs file and write the following lines into it:

```
using UnityEngine;

public class Core : BaseSingleton<Core>
{
    //Leap Listener.
    private LeapListener listener;

    //Interface colorscheme.
    public Colorscheme interfaceColors = new Colorscheme();

    //Title menu.
    public TitleMenu titleMenu = null;

    //Does the application have focus?
    public bool applicationFocused = true;

    //Paused?
    public bool paused = true;

    //Member Function: onAwake.
    public override void onAwake()
    {
        //This script will not be destroyed, even when a new level loads.
        DontDestroyOnLoad(gameObject);

        //Create a new Leap Listener.
        listener = new LeapListener();
    }

    //Member Function: OnApplicationFocus.
    public void OnApplicationFocus(bool pauseStatus) { applicationFocused =
    pauseStatus; }
```

```

//Member Function: Update.
public void Update()
{
    //Update the Leap listener.
    listener.refresh();

    //If the user closes their hand while the game is not paused, pause it.
    if (listener.hands < 1 || titleMenu.open || !applicationFocused)
    {
        //Pause all entities.
        paused = true;

        //Open the title menu.
        if (titleMenu.open == false) titleMenu.enabled = true;

        //Hide the hands.
        this.GetComponent<HandRenderer>().enabled = false;
    }

    //Otherwise, keep the game running.
    else if (listener.hands >= 1 && titleMenu.open == false)
    {
        //Unpause all entities.
        paused = false;

        //Show the hands.
        this.GetComponent<HandRenderer>().enabled = true;
    }
}
}
}

```

One line I'd like to draw your attention to in this class is:

```

Public void OnApplicationFocus(bool pauseStatus) { applicationFocused =
pauseStatus; }

```

This function is called whenever the Unity application window is in focus (that is, the user is currently using it). This allows the Core class to detect whether it should pause the game world when the user loses interest in your application or switches focus to another window.

The Update function keeps track of the current state of the game, using a combination of metrics that include a value that shows whether the window has focus (see the preceding code) if hands are in view and a menu is open.

If Update thinks the game should be paused, the user's 3D hands will cease to be rendered and instead will be replaced by a plethora of 2D cursors, which represent the user's fingers (for the purpose of navigating menus); conversely, if Update thinks the game should be running, the user's 3D hands will resume rendering and the 2D cursors will be hidden.

The rest of this class should be mostly self-explanatory, so let's go ahead and move on to the next one.

TouchPointer – let's draw some cursors on the screen

The TouchPointer class pulls in coordinates from the Leap and then draws them on to the screen in the form of mouse cursors. These cursors will then be used to trigger our touchable buttons, enabling the user to intuitively (and visually) interact with your application.

So, without further ado, open up TouchPointer.cs and paste these glorious lines of code within:

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

class TouchPointer : BaseSingleton<TouchPointer>
{
    //Leap listener.
    private LeapListener listener;

    //Leap box.
    private Leap.InteractionBox normalizedBox;

    //Pointer texture.
    public Texture2D pointerNormal;

    //Vertical (Y-axis) offset of pointer coordinates.
    public float verticalOffset = -10.0F;

    //Currently active fingers.
    public List<Rect> fingers = new List<Rect>();

    //Member Function: onAwake.
    public override void onAwake() { listener = new LeapListener(); }

    //Member Function: OnDisable.
    public void OnDisable()
    {
        //Reset the fingers array.
        if (fingers != null)
            fingers.Clear();
    }

    //Member Function: Update.
    public void Update()
    {
        //Update the listener.
        listener.refresh();

        //Reset the fingers array.
        if (fingers != null)
            fingers.Clear();
    }
}
```

```

//Retrieve coordinates for any fingers that are present, but only if
the menus are visible.
if (listener.fingers > 0 && Core.getInstance().paused)
{
    //Loop over all fingers.
    for (int i = 0; i < listener.fingers; i++)
    {
        //Set up its position.
        Vector3 tipPosition = new Vector3(0, 0, 0);

        //Get a normalized box.
        normalizedBox = listener.frame.InteractionBox;

        //Finger coordinates.
        tipPosition.x =
normalizedBox.NormalizePoint(listener.frame.Fingers[i].TipPosition).x;
        tipPosition.y =
normalizedBox.NormalizePoint(listener.frame.Fingers[i].TipPosition).y;

        //Modify coordinates to equal screen resolution.
        tipPosition.x = tipPosition.x * Screen.width;
        tipPosition.y = tipPosition.y * Screen.height;

        //Flip Y axis.
        tipPosition.y = tipPosition.y * -1;
        tipPosition.y += Screen.height;

        fingers.Add(new Rect(tipPosition.x, tipPosition.y, 16, 16));
    }
}

//Member Function: OnGUI
public void OnGUI()
{
    //Make a note of the current GUI color so that we don't overwrite it.
    Color temp = GUI.color;

    //Retrieve the "special" interface color and use it for the cursors.
    GUI.color = Core.getInstance().interfaceColors.special;

    //Place a texture where the cursor currently is.
    foreach (Rect point in fingers)
        GUI.DrawTexture (point, pointerNormal);

    //Restore the original GUI color.
    GUI.color = temp;
}
}

```

All the action in this class takes place in Update and OnGUI, so we'll focus on them.

Starting with Update, we refresh the Leap tracking data and then reset the fingers array to make sure no pointers get *orphaned*, for lack of a better word. We then begin storing finger pointer coordinates if there are actually fingers on screen.

This process consists of iterating over all the fingers present in the most recent frame from the Leap, performing the following steps in order:

1. We first set up a new fingertip position vector and retrieve an interaction box from the Leap.
2. We then calculate the fingertip coordinates on screen by normalizing their values against the interaction box and then rescaling them to match the screen resolution.
3. Then, we add this set of fingertip coordinates to our array of fingers to be rendered.
4. Next up, in `onGUI` (which is called when the graphical user interface, or GUI, is refreshed), we render cursors onto the screen at the coordinates specified by the fingers array that was populated earlier by `update`. This process is relatively straightforward, so I'll let the comments do the talking for me this time around.

Now for the next class...you're almost done.

TouchableButton – surely, the name is self-explanatory

Dear reader, you're almost done. Stay with me here. This next class, as the name implies, creates a touchable button. When used in conjunction with the TouchPointer class, the TouchableButton class will allow users to interact directly with onscreen buttons using just their hands (as well as a built-in 750-millisecond delay, so as to prevent accidental button presses).

Now, go ahead and open up the TouchableButton.cs file and enter the following lines:

```
using UnityEngine;
using System.Collections;
using System.Diagnostics;

public class TouchableButton
{
    //Pointer reference.
    private TouchPointer pointer;

    //Stopwatch used to measure hover time.
    private System.Diagnostics.Stopwatch hoverTime;

    //Number of possible missed reads from a hovering finger.
    private int mistakes = 0;

    //Current size multiplier being applied to this button.
    private float size = 1.0F;

    //Member Function: valueInRange.
    private bool valueInRange(float item, float min, float max) { return
(item >= min) && (item <= max); }

    //Member Function: over.
    private bool over(Rect a, Rect b)
    {
        bool xOverlap = valueInRange(a.x, b.x, b.x + b.width) ||
            valueInRange(b.x, a.x, a.x + a.width);

        bool yOverlap = valueInRange(a.y, b.y, b.y + b.height) ||
            valueInRange(b.y, a.y, a.y + a.height);

        return xOverlap && yOverlap;
    }

    //Time in milliseconds that a finger must hover over this button in
    //order to trigger it.
    public int triggerTime = 750;

    //Constructor.
    public TouchableButton()
    {
        //Grab a pointer reference.
```

```

    if (pointer == null)
        pointer = TouchPointer.getInstance();

    //Set up the stopwatch.
    hoverTime = new System.Diagnostics.Stopwatch();
}

//Member Function: render.
public bool render(Rect location, string text)
{
    //Has the button been pressed by the mouse?
    if (GUI.Button(new Rect((location.x - ((location.width * size) / 2)) +
(location.width / 2),
                        (location.y - ((location.height * size) / 2)) +
(location.width / 2),
                        location.width * size, location.height * size), text))
        return true;

    //Has a bad value been read during iteration?
    bool bad = true;

    //Is the button being pressed by a Leap pointer?
    foreach (Rect rect in pointer.fingers)
    {
        //If a finger is over this button, begin counting the amount of time
it spends.
        if (over(location, rect))
        {
            //Begin logging hoverTime.
            if (hoverTime.IsRunning == false)
                hoverTime.Start();

            //Check to see if the hoverTime is greater than the trigger time.
            else if (hoverTime.ElapsedMilliseconds > triggerTime)
            {
                //Reset the hoverTime.
                hoverTime.Stop();
                hoverTime.Reset();

                //Reset the size.
                size = 1.0F;

                //Return true.
                return true;
            }
        }

        //Increment size.
        size += 0.005F;

        //We received at least one good value.
        bad = false;
    }
}

//If no good values were received, and the hoverTime clock is running,
increment mistakes.

```

```

if (bad && hoverTime.IsRunning) mistakes += 1;

//Otherwise, reset the mistakes.
else if (!bad && hoverTime.IsRunning) mistakes = 0;

//If our mistakes exceed 5, stop the hovertime counter.
if (mistakes >= 5)
{
    hoverTime.Stop();
    hoverTime.Reset();

    size = 1.0F;
}

//If nothing is pressing the button, return false.
return false;
}
}

```

And now, for the breakdown—hopefully not a mental one (I promise there’s only one more file left). The comments do a good job of covering the details, but let’s go over each function for safety’s sake.

The first function, `valueInRange`, is super simple—it simply verifies that the passed value is somewhere between the two other values. The function immediately following it, aptly titled `over`, is also simple (relatively speaking). It takes two rectangles and checks to see if they overlap. If they do, it returns `true` (and `false` otherwise).

The last function we’ll cover is `render`. This function, as the name implies, renders the `TouchableButton` on the screen, but it serves another important function too; it checks to see whether any of the fingers in the `TouchPointerfingers` array are overlapping the button.

In the event that a finger is overlapping the button, a series of actions and checks take place:

1. Once a finger is detected hovering over the button, a system timer is started.
2. If the finger remains over the button for an extended period of time (specifically, once the elapsed milliseconds on the `hoverTime` timer exceed the `triggerTime` value), the button returns `true` to indicate that it was clicked.
3. As the finger remains over the button, the size of the button is gradually increased to give visual feedback to the user that they’re pushing the button.
4. In the event that a finger was hovering over the button but stopped before the button could be fully pushed, the `mistakes` counter gets incremented. If this happens five times, the button is reset to its current state without being clicked. While developing this button, I was able to confirm that an average of two to three mistakes are recorded every second or so, even though the user’s finger remained steady on a button. If you didn’t use a `mistakes` counter and just reset every time a finger disappeared from the view for a second, the buttons would constantly reset and your application wouldn’t work!
5. The nice thing about this process is that it is entirely synchronous; that is, it can be

run within a single-threaded program such as Unity without stalling or slowing anything down. This is thanks to the liberal use of timers and fault checking; you wouldn't know it, but a similar process is used quite frequently to increase performance in robotic control systems that have limited processing power.

Note

Fun fact

A process is **synchronous** when there is only a single thread. This is the opposite of **asynchronous**, which involves multiple threads. Most operating systems are asynchronous, whereas a lot of lightweight applications (like those found on robots) are synchronous.

Now for the last class...at last.

TitleMenu – a simple main menu

At last, before you lies the final class in this chapter: `TitleMenu`. This class, when attached to a `GameObject`, creates a simplistic menu featuring a single play button flanked by two color setting buttons—though perhaps it's better just to show you. Go ahead and open up the `TitleMenu.cs` file and enter the following lines of beautiful code:

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

public class TitleMenu : MonoBehaviour
{
    //Smoothed button height for middle, right, and left.
    public float buttonHeight = 0.0f;

    //Font size.
    public static int fontsize = 10000;

    //Is the title menu open?
    public bool open = false;

    //Touchable buttons.
    public TouchableButton colorButton;
    public TouchableButton playButton;
    public TouchableButton greyButton;

    //Member Function: OnEnable.
    public void OnEnable()
    {
        //The menu is now open.
        open = true;

        //Initialize buttons.
        colorButton = new TouchableButton();
        playButton = new TouchableButton();
        greyButton = new TouchableButton();
    }

    //Member Function: OnDisable.
    public void OnDisable()
    {
        //The menu is now closed.
        open = false;
    }

    //Member Function: getButtonRect.
    public Rect getButtonRect(float x, float y)
    {
        return new Rect(((Screen.width / 2) - (x * ((Screen.width +
Screen.height) / 2) / 1500)),
            (Screen.height / 2 - y - GUI.skin.button.fontSize),
            360 * ((Screen.width + Screen.height) / 2) / 1500, 180 *
((Screen.width + Screen.height) / 2) / 1500);
    }
}
```

```

}

//Member Function: OnGUI.
public void OnGUI()
{
    //Set up GUI fonts.
    GUI.skin.button.fontSize = ((Screen.width + Screen.height) / 2) / 15;
    fontsize = GUI.skin.button.fontSize;

    //Set up GUI colors again.
    GUI.color = new Color(Core.GetInstance().interfaceColors.primary.r,
Core.GetInstance().interfaceColors.primary.g,
Core.GetInstance().interfaceColors.primary.b);

    //Clicking on the Play button will unpause the game and begin play.
    if(playButton.render(getButtonRect(180, buttonHeight), "play"))
    {
        //Close and disable the menu.
        open = false; enabled = false;
    }

    //Clicking the Colour button will restore the interface colorscheme to
its defaults.
    if(colorButton.render(getButtonRect(580, buttonHeight), "colour"))
    {
        Core.GetInstance().interfaceColors.setGreyscale(false);
    }

    //Clicking the Grey button will set the interface colorscheme to
greyscale.
    if(greyButton.render(getButtonRect(-220, buttonHeight), "grey"))
    {
        Core.GetInstance().interfaceColors.setGreyscale(true);
    }
}
}

```

The comments do most of the talking yet again.

However, let's go ahead and talk about `OnGUI` and what it's doing briefly. The first few lines set up the button colors and font size. We then proceed to render the touchable buttons onto the screen using a series of mathematical expressions to make sure that they're positioned just so.

When a button is pressed, the contents of its respective `if` statement will be executed. The first button, `play`, will close and disable the menu when pressed, allowing gameplay to begin. The second button, **colour**, will disable any grayscaling of the interface color scheme when pressed. Finally, the third button, **grey**, will enable grayscaling of the interface color scheme when pressed.

Enough of me talking; seeing something happen is more exciting than reading about it. Time to return to the Unity Editor!

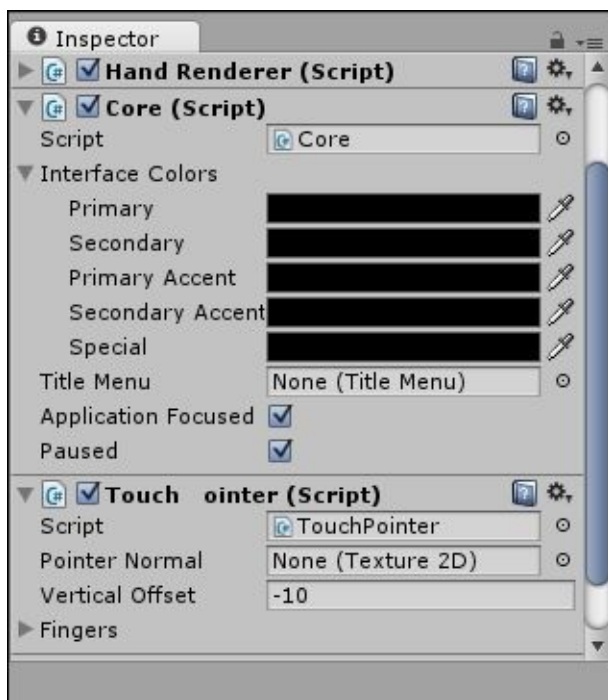
Putting it all together

At last, you're almost done; just a few more steps and you'll have one half of a working 3D application (the interface).

Return to your Unity project and attach the following scripts to their respective GameObjects, which you created earlier:

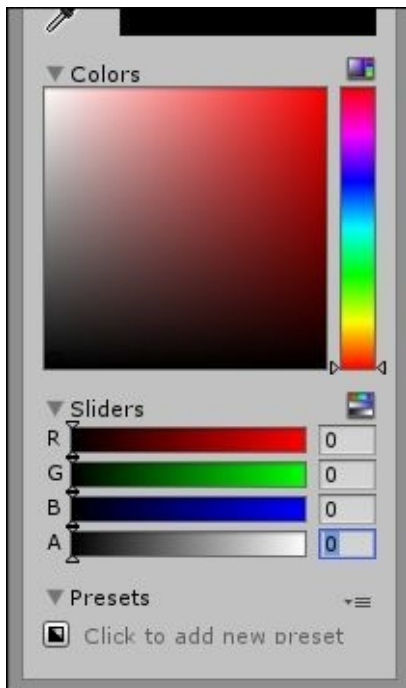
- Attach `Core.cs` and `TouchPointer.cs` to the Core GameObject.
- Attach `TitleMenu.cs` to the Main Menu GameObject.
- Now, before we continue, download a file called `cursor.png` from <https://github.com/Mizumi/Mastering-Leap-Motion-Unity-Project/blob/master/Assets/Res/Cursor.png> and place it within your Res folder. This image will be used for the cursors that the `TouchPointer` class draws on the screen.

Go ahead and select the Core GameObject from your **Hierarchy** window. Its Inspector should look something like the following screenshot:

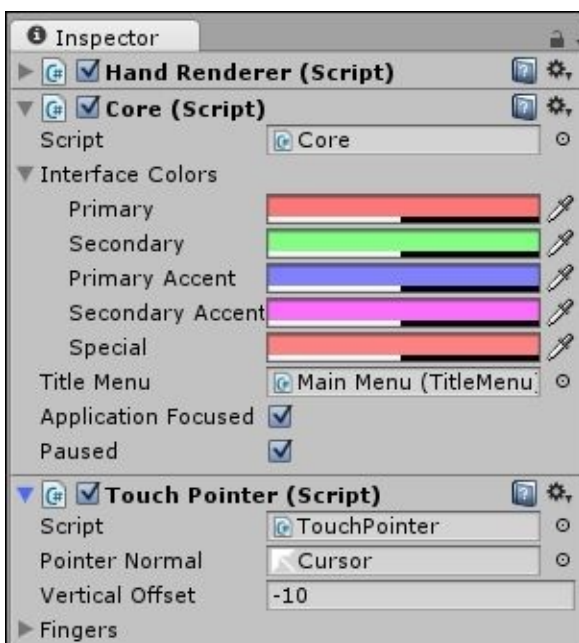


Perform the following modifications to the specified fields before we proceed:

- Click on the little circle next to the **Pointer Normal** field underneath **Touch Pointer (Script)** and select the `cursor.png` file you downloaded earlier from the dialog that appears.
- Click on the little circle next to the **Title Menu** field underneath the Core script and select the Main Menu GameObject from the dialog that appears.
- Now, we need to set up the interface colors underneath the Core script. You'll notice that right now, they appear as a series of black boxes; let's fix that. Clicking on one of the boxes will result in a dialog similar to the one here:

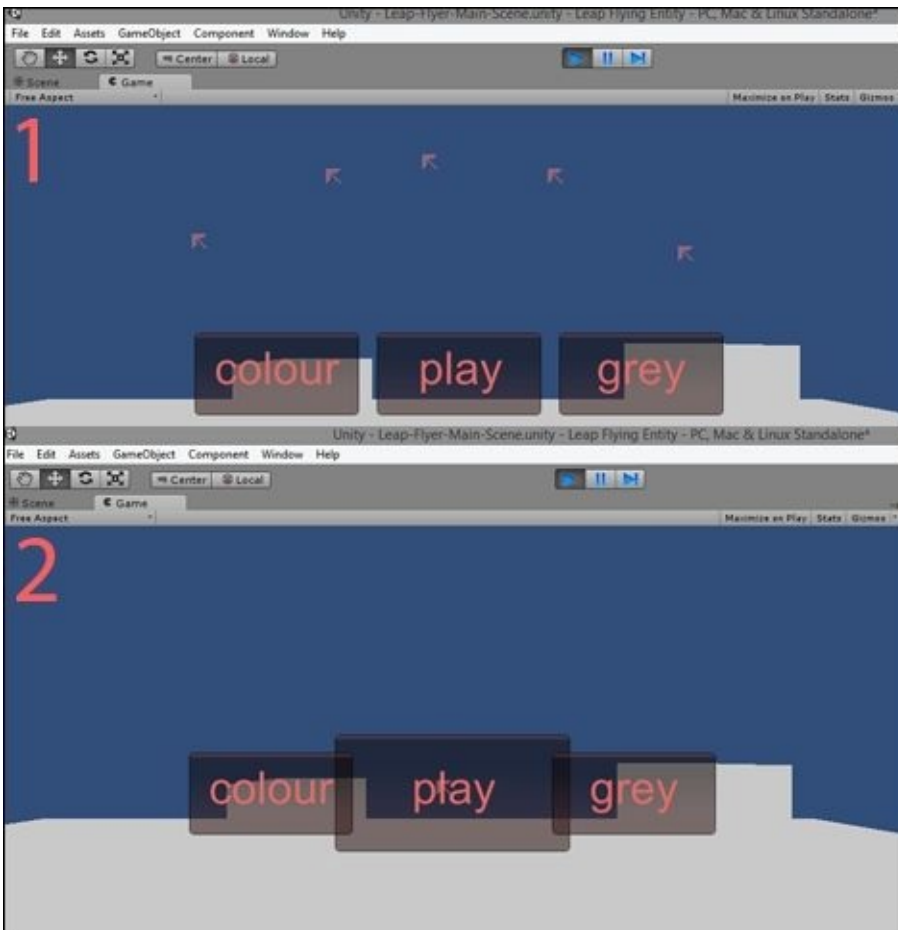


You can choose any color you like, but be sure to set the **A** value to 125 or higher; if you keep it at 0, you won't be able to see any of the colors as they'll be invisible! Repeat this process for the remaining four colors, and you should finish with an **Inspector** window that looks something like the one here:



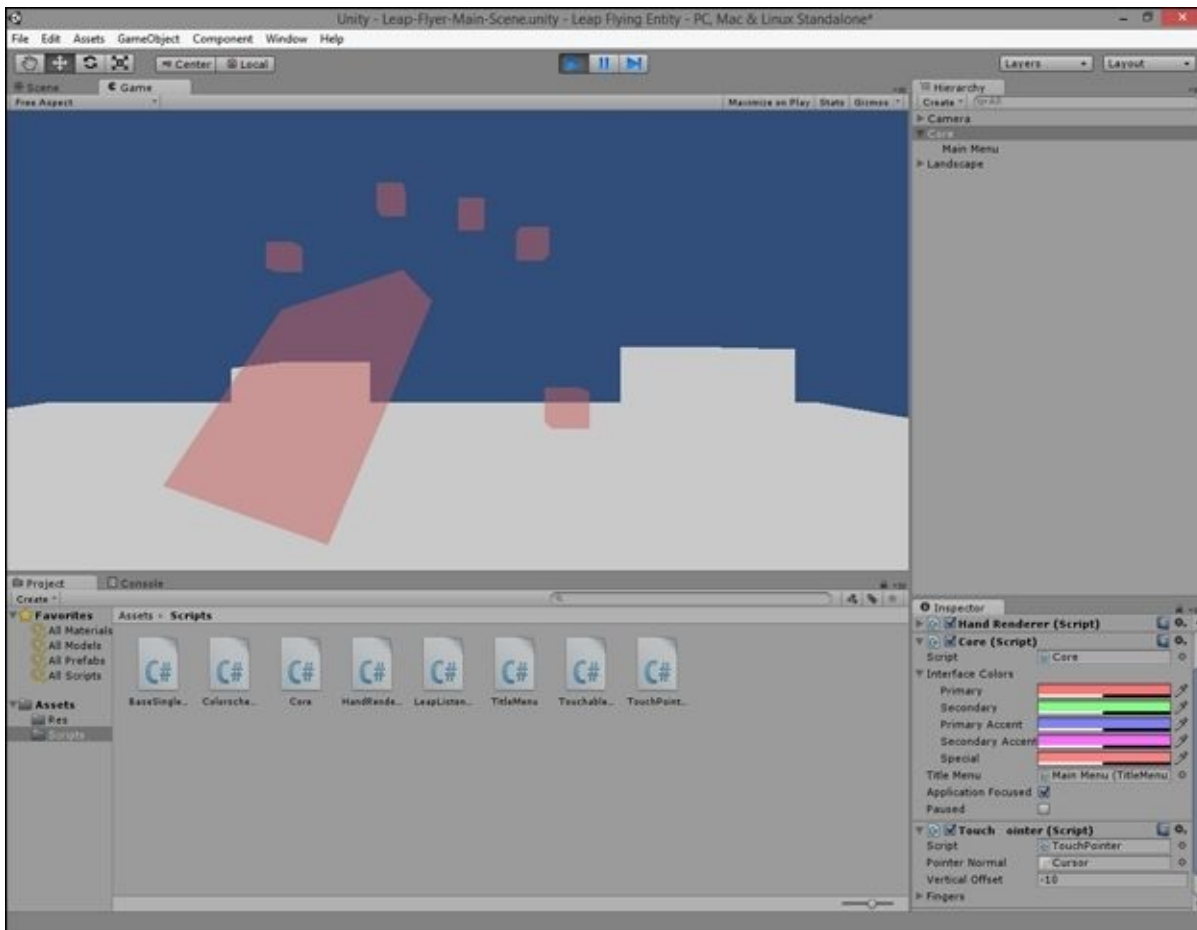
Say what you like, I like my colors pastel!

Rants about colors aside, you're now ready to test the application. Without further ado, go ahead and hit the run button. You'll initially be greeted by three buttons in all, as shown in the following screenshot:



If you place your hand within the view of the Leap, you will see some cursors appear within the field of view (as seen in the top portion of the preceding screenshot). Then, if you proceed to click on the **play** button (by hovering one of your fingers over it), you'll see the button slowly expand (as seen in the bottom portion of the preceding screenshot).

If you're successful in clicking on the **play** button, you'll see your hands come into view in the 3D realm and the menu will disappear, as shown in the following screenshot:



If you remove your hands from view, the menu will immediately reappear, giving you the ability to repeat this process over and over again—or just keep the game paused.

With this, you're done! You now have a working 3D renderer for hands, in addition to a simple interface for triggering buttons via the Leap. I admit, there were probably some close calls here and there, but all's well that ends well, right?

Summary

In this chapter, you learned the value of not only listening to directions but also listening to them well (that is, if the resultant application worked as this author expected it to). We started off by setting up the scene to support Unity scripts, followed by a quick summary of how Unity scripts work.

You then proceeded to write a series of files to render hands and fingers from the Leap Motion device on to a screen in the 3D format. After learning how to configure the scene to support these hands, you tested it out and proceeded to the next step: rendering buttons and detecting fingers on them. After writing a slew of utility classes, you learned how to map Leap input and convert it to a series of two-dimensional cursors. You then wrote a few more menu classes. Finally, you put it all together into a working user interface with a relatively solid flow.

In the next chapter, we'll combine all the work you've done so far with a flying entity that can be controlled by simply (perhaps even gracefully) moving your hand.

Chapter 7. Creating a 3D Application – Controlling a Flying Entity

In this chapter, we will take everything we covered in [Chapter 5](#), *Creating a 3D Application – a Crash Course in Unity 3D*, and [Chapter 6](#), *Creating a 3D Application – Integrating the Leap Motion Device with a 3D Toolkit*, to create a playable 3D application where you will control a virtual flying entity (similar to a quadrotor) in an empty room. We'll go over the basics of creating an entity, interpreting user input, and then moving the entity around. At the conclusion of this chapter, you should have a complete 3D application that you can control with just your hands!

Note

This chapter is sprinkled with periodic *Fun facts* that offer high-level and entry-level factoids about scripting and programming for your reading pleasure.

We will be covering the following topics in this chapter:

- Creating the flying entity
- Retrieving user input
- Interpreting user input with the `Player` class
- Putting everything together and testing it

So, without further ado, let's begin!

Creating the flying entity

At long last, it's time for us to create the **player character** for our Unity application; after all, what good is a game with no avatar to play as?

As making a complete player entity is a bit time consuming in Unity, we're going to gloss over this process entirely and use something that Unity refers to as **packages**. These nifty files allow developers to download collections of preconfigured GameObjects and their associated assets, allowing for drop-in usage. Needless to say, this can save a lot of time in bigger projects!

Note

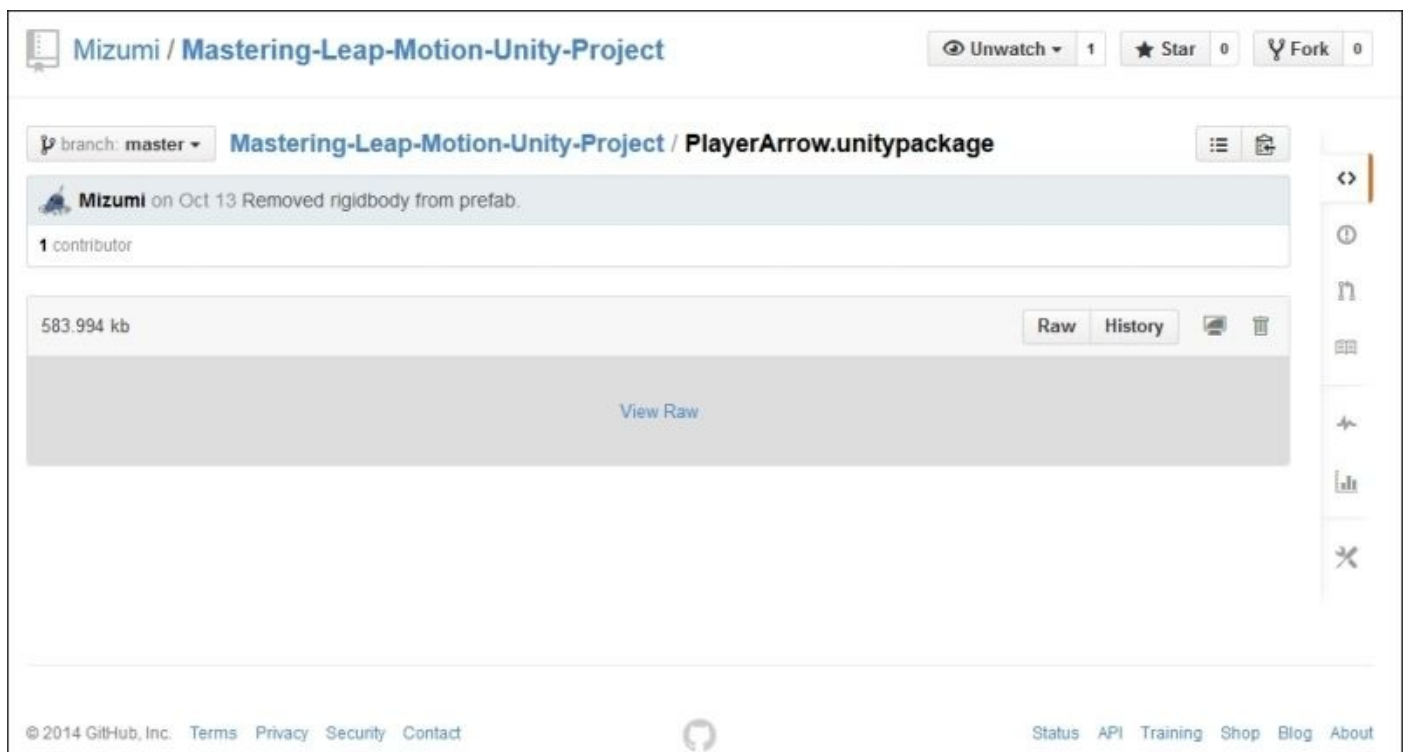
Fun fact

Unity packages can be thought of as reusable pieces of code (GameObject in this case) that can be included by simply dropping them into the scene.

So, without further ado, fire up your web browser and navigate to the following link:

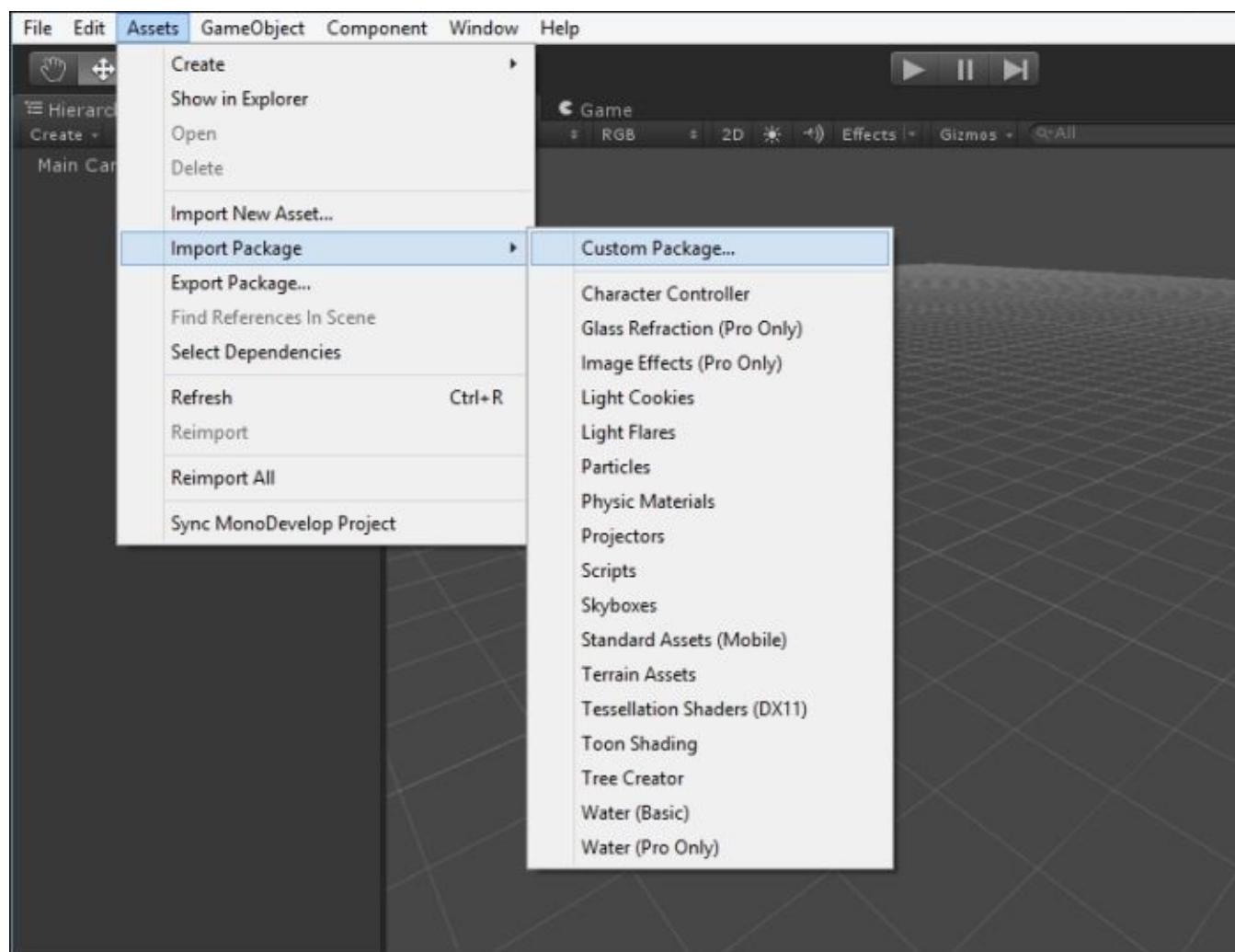
<https://github.com/Mizumi/Mastering-Leap-Motion-Unity-Project/blob/master/PlayerArrow.unitypackage>

You will be presented with a fabulous GitHub repository and a file called PlayerArrow.unitypackage, similar to what is shown in the following screenshot:



Go ahead and click on the tiny **View Raw** link. This will open a dialog prompting you to download the Player Arrow package (depending on the browser). You can also right-click on the link and click on **Save link as...** in the pop-up menu to begin the download.

When the download completes, fire up the Unity Editor and open your project. Once everything finishes loading, navigate to **Assets | Import Package | Custom Package**, as shown in the following screenshot:



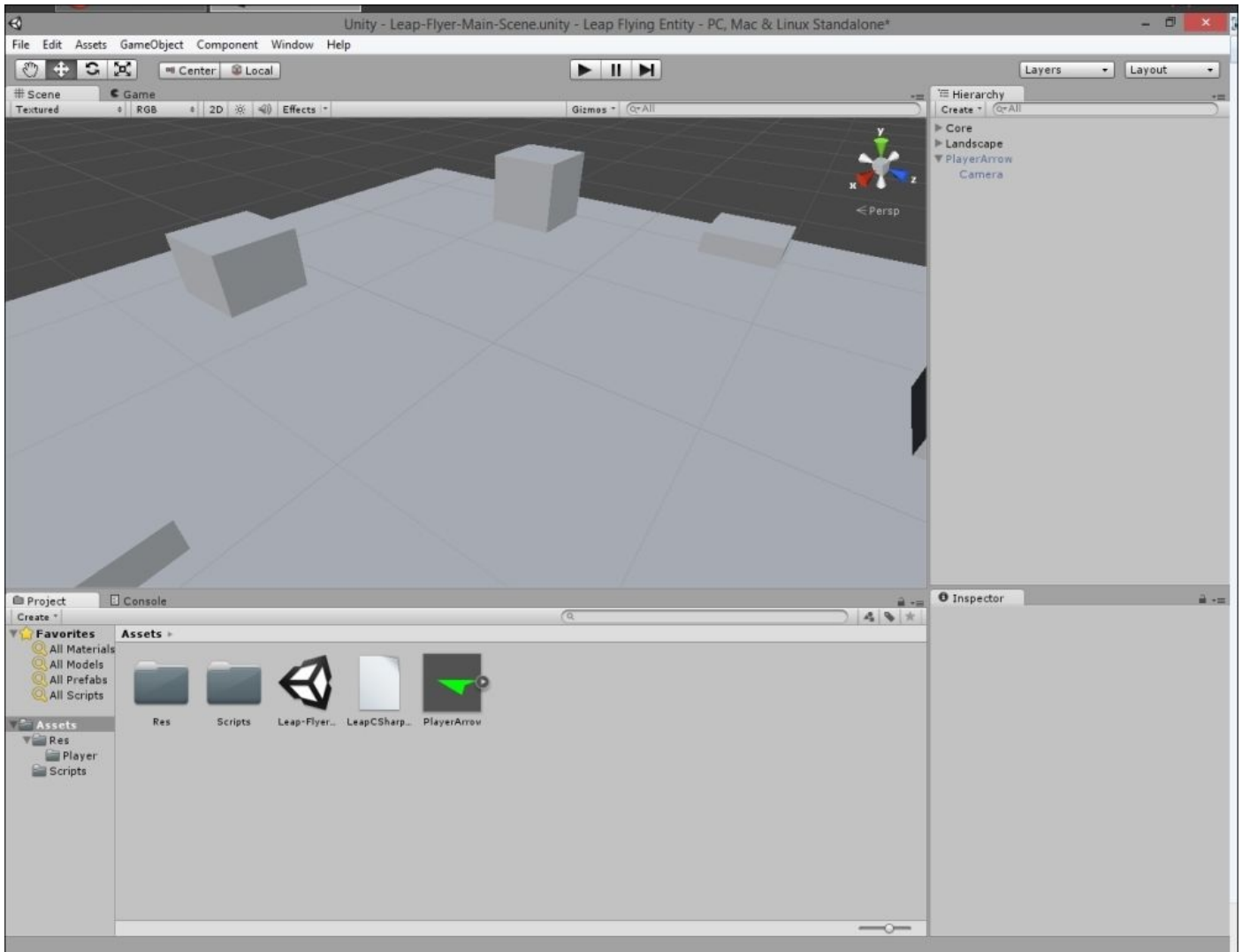
Select our newly downloaded `PlayerArrow.unitypackage` file in the prompt that appears, as shown in the following screenshot. Once you select the package, you will see a dialog window similar to the following one:



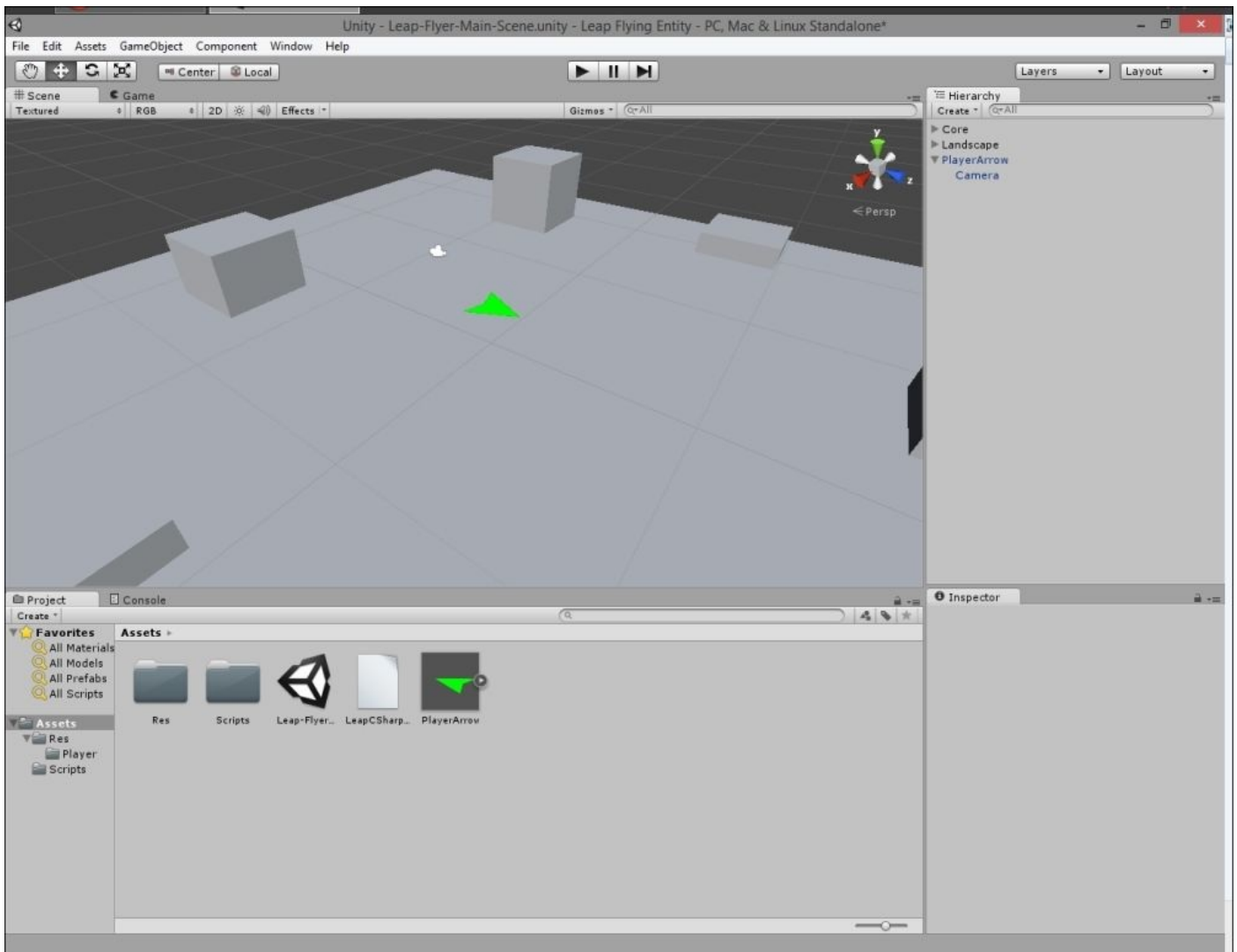
Verify that all the items listed under items to import are checked and then click on the **Import** button.

Adding the PlayerArrow and Rigidbody components

If all goes well, you'll see a new file show up in the root **Assets** window called **PlayerArrow**, as shown here:



Now, the reason Unity packages are so amazing is that you can embed them in the scene with almost no issues; go ahead and drag the **PlayerArrow** file into the **Scene** window and place the resulting green arrow wherever you deem fit! When you're done, you should have a nice two-dimensional arrow in your scene, as shown in the following screenshot:



Once the arrow is in the scene, go to **Transform | Position** and make sure its **Y** option (set via the **Inspector** window) is set to 2. This is to give the arrow the illusion of *flying*.

At this point, we're almost ready to move on to scripting. However, there's still one last step to perform on the arrow before we do so: adding a Rigidbody component.

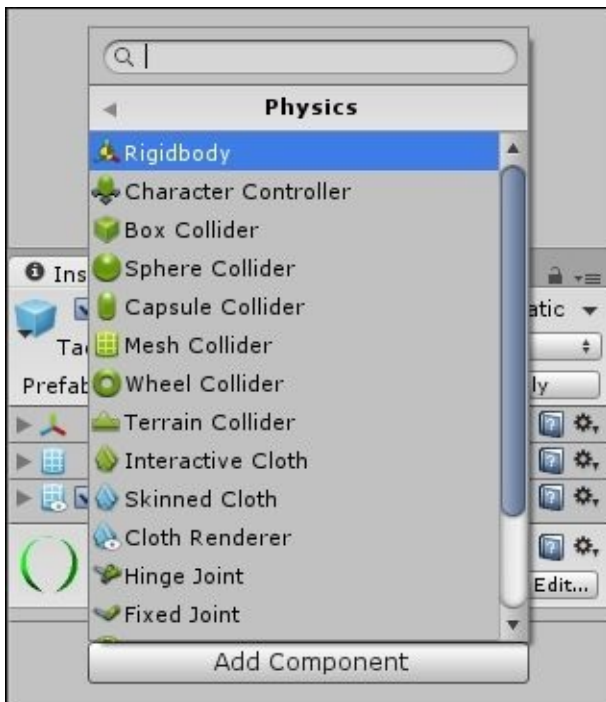
Note

Fun fact

Adding a Rigidbody to a GameObject will allow the object's motion to be controlled by Unity's physics engine. Even if you don't script anything, adding a Rigidbody to a GameObject will cause it to naturally fall with gravity, react to physical collisions with other objects, and so on. A much better write-up on what Rigidbodies are and what they do can be found at <http://docs.unity3d.com/ScriptReference/Rigidbody.html>.

In other words, Rigidbodies make GameObjects move and react to the laws of physics.

Fortunately, adding a Rigidbody is quite easy: simply click on **PlayerArrow** in your **Hierarchy** window, click on the **Add Component** button within the **Inspector** window, and navigate to **Physics | Rigidbody**, as shown in the following screenshot:



Once the Rigidbody component is added to the PlayerArrow GameObject, expand it within the **Inspector** window of PlayerArrow and disable the **Use Gravity** checkbox, as shown in the following screenshot:



Unchecking the **Use Gravity** option will prevent the arrow from falling to the ground when the game starts; after all, we don't want something that flies to plummet ingloriously to the ground.

With this, you're done making the player entity. It's simple, granted, but anything more complex and we'd be spending much more than 50 pages on just 3D modeling and that's

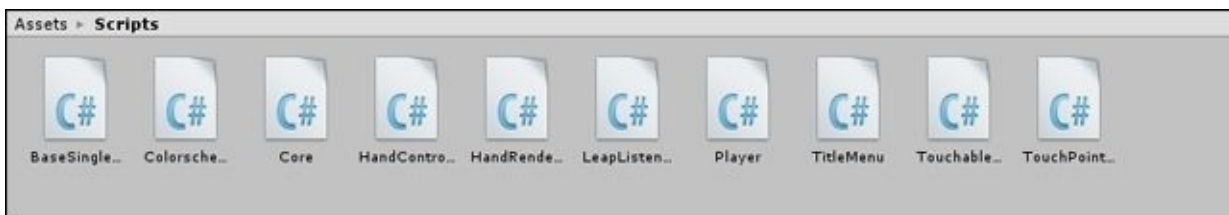
not fun at all! Now to write some code...

Retrieving user input with the HandController class

At long last, only two files stand before us and a completed application. Go ahead and navigate to the Scripts folder in Unity's file browser and create two new C# scripts:

- HandController.cs
- Player.cs

Once these are made, your Scripts folder should look almost identical to the one in the following screenshot:



Now, go ahead and double-click on the HandController.cs file; this should automatically open MonoDevelop. Let's write some code!

As the header of this section suggests, the first class we'll be writing is HandController. This class will take input from the Leap Motion Controller and convert it into an x value and a y value, which will in turn be used to control the movement of our arrow. Kindly copy the following code into your HandController.cs file now:

```
using UnityEngine;
using System.Collections;

class HandController : BaseSingleton<HandController>
{
    //Leap Listener reference.
    public LeapListener listener;

    //Left joystick X/Y. Corresponds to pitch and roll of the hand.
    public int x = 0;
    public int y = 0;

    //Member Function: onAwake
    public override void onAwake() { listener = new LeapListener(); }

    //Member Function: Update
    public void Update()
    {
        //Try to read data from the Leap Motion device.
        try
        {
            //Refresh the Leap data.
            listener.refresh();
        }
    }
}
```

```

//Reset all the variables.
x = y = 0;

//If there are hands in the field of view and the Leap device is
connected, use the Leap's input.
if (LeapListener.connected && listener.hands > 0 && listener.fingers
> 0)
{
    //Right joystick X-Axis. Sensitivity is doubled to increase
responsiveness.
    x = (int) (listener.handRoll * -2);

    //Right joystick Y-Axis. Sensitivity is doubled to increase
responsiveness.
    y = (int) (listener.handPitch * -2);
}
}

//If reading data fails, make sure that X and Y are set to 0.
catch(System.Exception e){x = y = 0;}
}
}

```

This is probably one of the simpler classes we've written thus far; it contains a very simple `onAwake` function that initializes the Leap Motion listener as well as an update function.

The update function is fairly simple; perform the following steps in this order:

1. Refresh the Leap Listener data.
2. Reset all the variables.
3. Verify that there are both hands *and* fingers in the field of view—we wouldn't want to start reacting to a pencil now, would we?
4. Assign the roll and pitch values of the first hand in the field of view to the `x` and `y` variables, respectively. We multiply and invert the raw values from the Leap during this step to make the controller a bit more sensitive (as well as convert it to a roughly 100 to 100 scale) so that the user doesn't have to overexaggerate their hand motions to make our flying entity move.

That's all there is to it. Let's go ahead and move on over to the next class.

Interpreting user input with the Player class

The `Player` class is responsible for controlling our flying entity, although I'm sure the name gave its purpose away. The `Player` class will take input from the `HandController` class and convert it into movement control for whichever `GameObject` it is connected to. Go ahead and open up your `Player.cs` file now and copy the following code into it:

```
using UnityEngine;
using System.Collections;

public class Player : MonoBehaviour
{
    //Reference to the hand controller.
    HandController controller;

    //Member Function: Start
    void Start()
    {
        controller = HandController.GetInstance();
    }

    //Member Function: Update
    void Update ()
    {
        //Only move if the game is unpaused.
        if (!Core.GetInstance().paused)
        {
            //Transform position forward.
            rigidbody.velocity = transform.forward * (controller.y / -2);

            //Rotate.
            transform.Rotate (0, controller.x * 1.25f * Time.deltaTime, 0,
Space.World);
        }

        //If the game is paused, cancel all force vectors.
        else rigidbody.velocity = new Vector3(0, 0, 0);
    }
}
```

This is the simplest class by far. I'll skip straight to breaking down how the `Update` class works, as it uses methods that we have yet to utilize in the prior classes.

`Update` starts by checking whether the game is paused. If the game isn't paused, `Update` will apply force to the `Rigidbody` in the forward direction, equivalent to the `y` axis input from the `HandController` class using the following snippet of code:

```
rigidbody.velocity = transform.forward * (controller.y / -2);
```

The variable, `transform.forward`, contains a three-axis vector that *points* in the forward direction of the `GameObject`, guaranteeing that our velocity is applied in the correct

direction; we wouldn't want our arrow flying straight into the ground, would we now? update will then proceed to rotate the GameObject about its y axis by changing the rotational value of the GameObject itself. If we were to apply sideways velocity to the Rigidbody instead of manually rotating the GameObject, we'd either have to do some extra math or risk overwriting the velocity we just applied to the Rigidbody in the previous line—and neither of these things are any fun.

Note

Fun fact

It's important to note here that the y axis in Unity points up, whereas the y axis in Leap Motion's world points forward. That's why we're applying an x axis value to the y axis, z axis values to the x axis, and so on.

Always keep track of the difference in axes across platforms, or you'll get confused really quickly!!

In the event that the game is paused, update will proceed to cancel out any forces currently acting on the Rigidbody of the GameObject, `Player.cs`, which is attached to by setting them to `0`. This prevents the player from moving around when the menu is open.

With this, the coding is all done. Let's move on to finishing up this application.

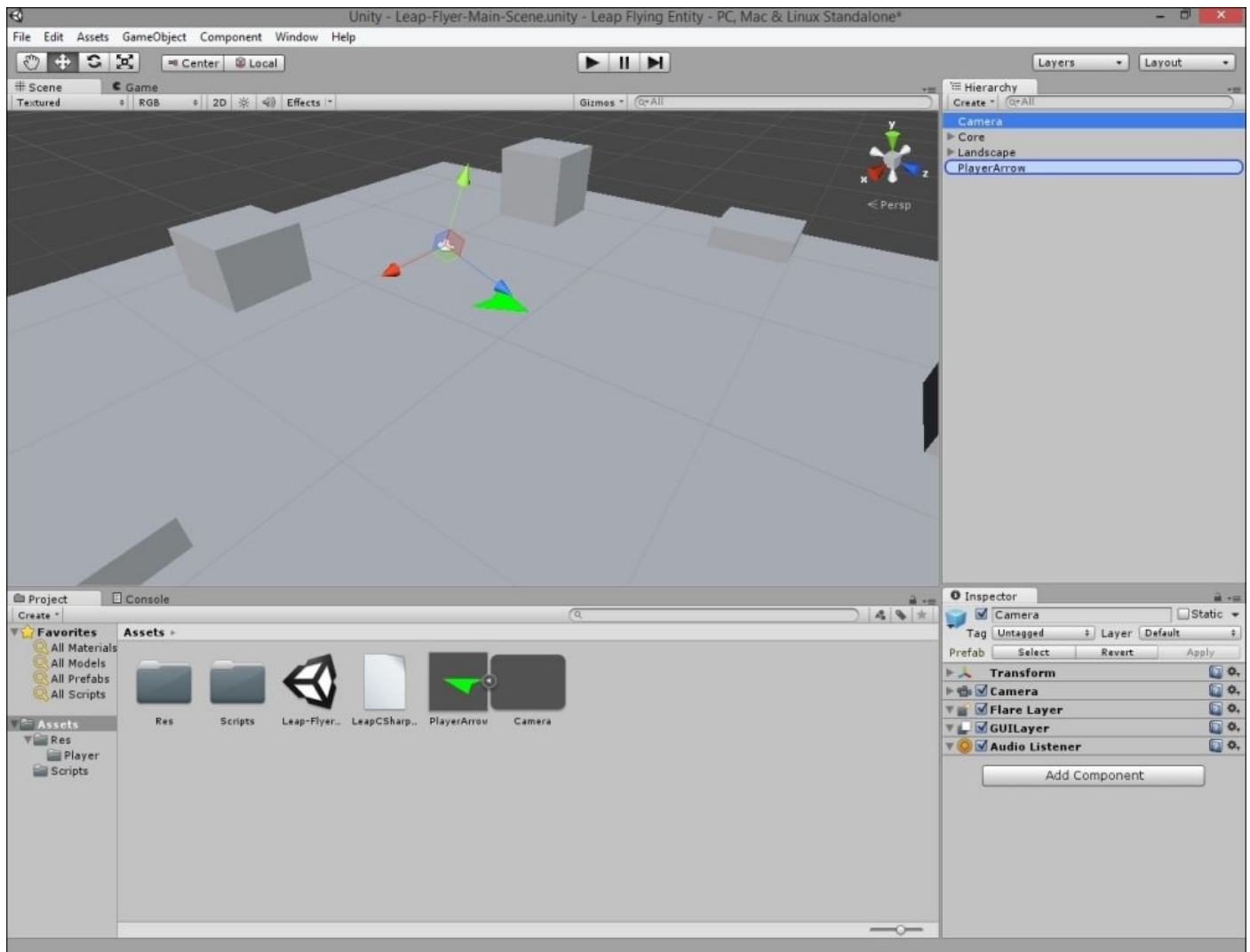
Putting everything together and testing it

Go ahead and open your Unity Editor and load up our project. With the `Player` and `HandController` scripts completed, we can go ahead and attach them to their respective `GameObjects` now:

- Attach the `Player` script to our `PlayerArrow` `GameObject`
- Attach the `HandController` script to our `Core` `GameObject`

At this point, you should be quite familiar with how attaching scripts to `GameObjects` works. As a refresher, simply dragging the desired script onto the target `GameObject` will do the trick. Fortunately, no configuration is required this time around.

With the scripts attached, we need to perform a few last pieces of configuration to make sure everything works. The first thing we'll be doing is attaching the main camera to the `PlayerArrow` `GameObject`; this will allow it to follow the arrow around, just like you'd expect from any third-person game. To do this, simply drag the `Camera` object in the **Hierarchy** window onto the `PlayerArrow` `GameObject`, as shown here:



Now that the camera object is attached to the `PlayerArrow` `GameObject`, its coordinate

system will now use *local* coordinates by default.

Note

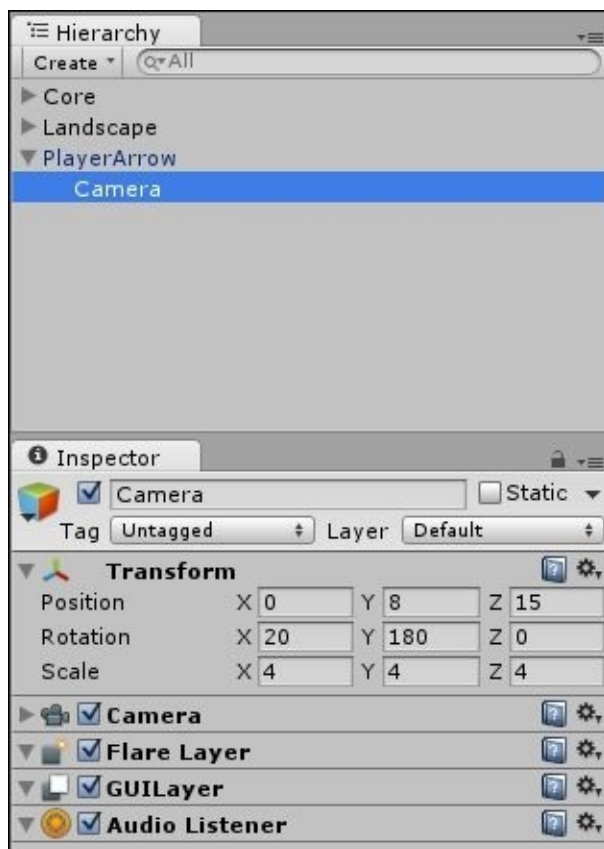
Fun fact

When an object is using local coordinates (a camera in this case), this means that if it has a vector coordinate of 0, 0, 0, it will be placed at the exact center of the GameObject it is attached to and not at the center of the world.

Likewise, if the parent object has a vector coordinate of 0, 10, 0 and the child object has a vector coordinate of 0, 10, 0, the child will actually have a global coordinate of 0, 20, 0, as everything is offset relative to the parent's location. Isn't that cool?

Although local and global coordinates can be extremely confusing, they can also be extremely useful. There are so many cases where it's much easier to think of coordinates in relation to other GameObjects (local) than it is to think of them in relation to the origin of the entire game (global).

In our case, we'll be using local coordinates to ensure the camera always follows the arrow around and maintains a fixed viewpoint. However, right now your camera is most likely looking at the ground or something terrible like that; go ahead and modify your camera's local position and rotation to be equal to the values seen in the editor window here:

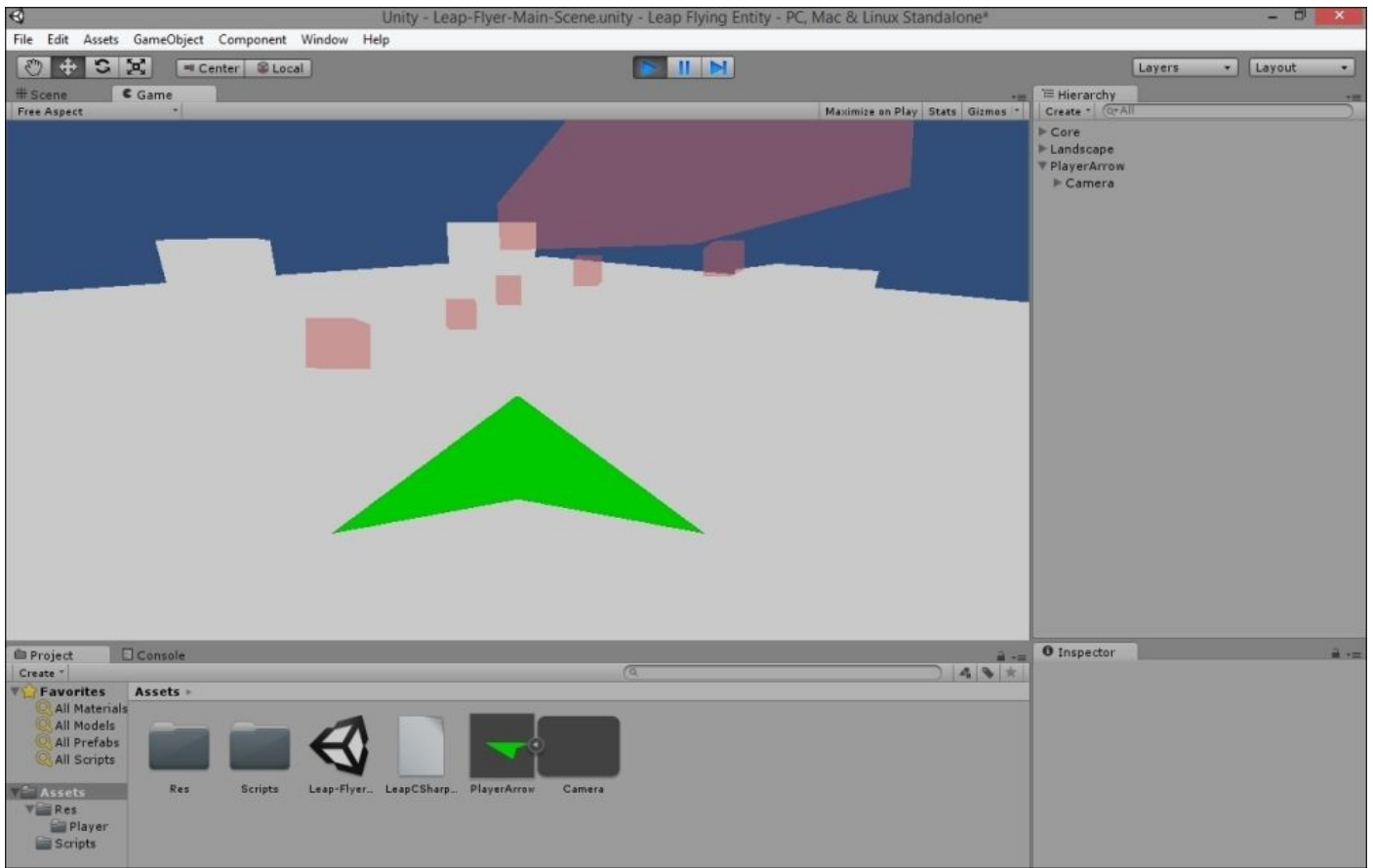


Using these coordinates, the camera will be placed behind the PlayerArrow GameObject and look slightly down at it, similar to most third-person experiences that you might find

in today's games. Before you go and hit play, there's just one last thing to do: we need to make sure that the camera is still being referenced correctly by the HandRenderer script. To do this, head on over to the Core GameObject's **Inspector** window and verify that there is a camera present in the HandRenderer script, as shown here:



At long last, if you did everything correctly (which I'm sure you did), you can now hit the play button in Unity and enjoy the experience of controlling 3D objects with just your hands! Naturally, I'm inclined to include a screenshot of what the final experience should look like:



Congratulations! That's one Unity application down—you can make another or improve this one.

Improving the application

There are many ways you could improve on this app; the most obvious one would be to make it look better (yes, even this author admits it's ugly). This can be achieved by adding more GameObjects, using custom textures and light maps. Unfortunately, the instructions on how to do this far exceeds the scope of this book. I suggest you hit up the official Unity documentation at <http://docs.unity3d.com/> for more information on lighting, texture, and beautification.

Summary

In this chapter, you learned more about Unity's powerful yet complicated coordinate system, its amazing built-in asset manager, and how its physics engine is incorporated into scripting. Indeed, you learned that the y axis is not always *up* and the z axis isn't always *forward*.

We started off by importing a simple Unity package, containing a GameObject for a flying entity that would be controlled by a player. We then proceeded to write two scripts: one that turns Leap Motion input data into fake joystick-like data and another that turns that data into physical forces in the game world, making things move and react to a user's hands. You finished off the chapter by attaching the scripts to their corresponding GameObjects and testing the game.

If everything went well, you'll now have a finished, albeit simple, Unity game to call your own! In the next chapter, we will take a break from writing code and instead talk about something that's slightly less fun, but important nonetheless: troubleshooting, debugging, and optimizing the code.

If you had any trouble during the past few chapters, or just wish to see how this author set up his code, you can view and download the entire Unity Project from GitHub at <https://github.com/Mizumi/Mastering-Leap-Motion-Unity-Project>.

Chapter 8. Troubleshooting, Debugging, and Optimization

If your application gives you trouble, come here for help!

We will cover the following topics in this chapter:

- Making sure your Leap is connected
- Keeping the Leap Motion SDK updated
- Cutting back on Leap Motion API calls
- Handling the `NoSuchMethod` and `NoClassDefFound` errors in Java
- Custom calibration of the Leap Motion Controller

Note

This chapter is sprinkled with periodic *Fun facts* that offer high-level and entry-level factoids about scripting and programming for your reading pleasure.

Making sure your Leap is connected

Well, the absolute first thing you should check when one of your programs is being cruel is whether your device is connected to the computer—and not just physically.

The first thing to do, of course, is to verify that your Leap Motion device is securely plugged into your computer and that the little green light on the front of the Leap device is turned on, as seen in the following image:



Once you verify this, check to see whether your application has started to work as expected.

No? If not, then the next thing we have to do is fire up the oh-so-wonderful Leap Motion Diagnostic Visualizer.

Note

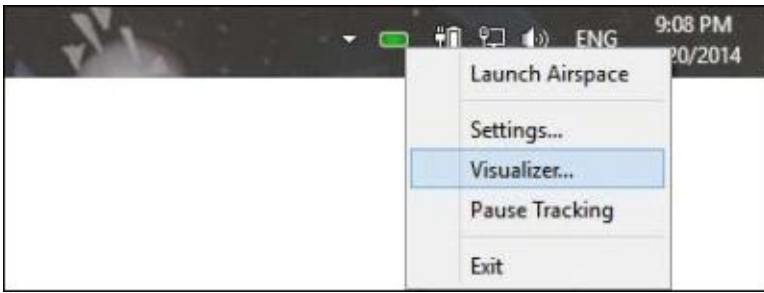
Not-so-fun fact

The following instructions make the assumption that you are on the Windows platform. If this is not the case, do not feel distressed! All of these instructions should work on the OS X and GNU/Linux platforms with minimal differences.

The first thing to do is launch the Leap Motion Control Panel application. Go ahead and do this now, either by going to **Start | All Programs | Leap Motion | Leap Motion Control Panel** (Windows 7) or pressing the Windows key + Q and in the search box, typing Leap Motion Control Panel (Windows 8).

Once the control panel is launched, go to your system tray (the collection of icons on your taskbar), right-click on it, and select the **Visualizer** option, as seen in the following

screenshot:

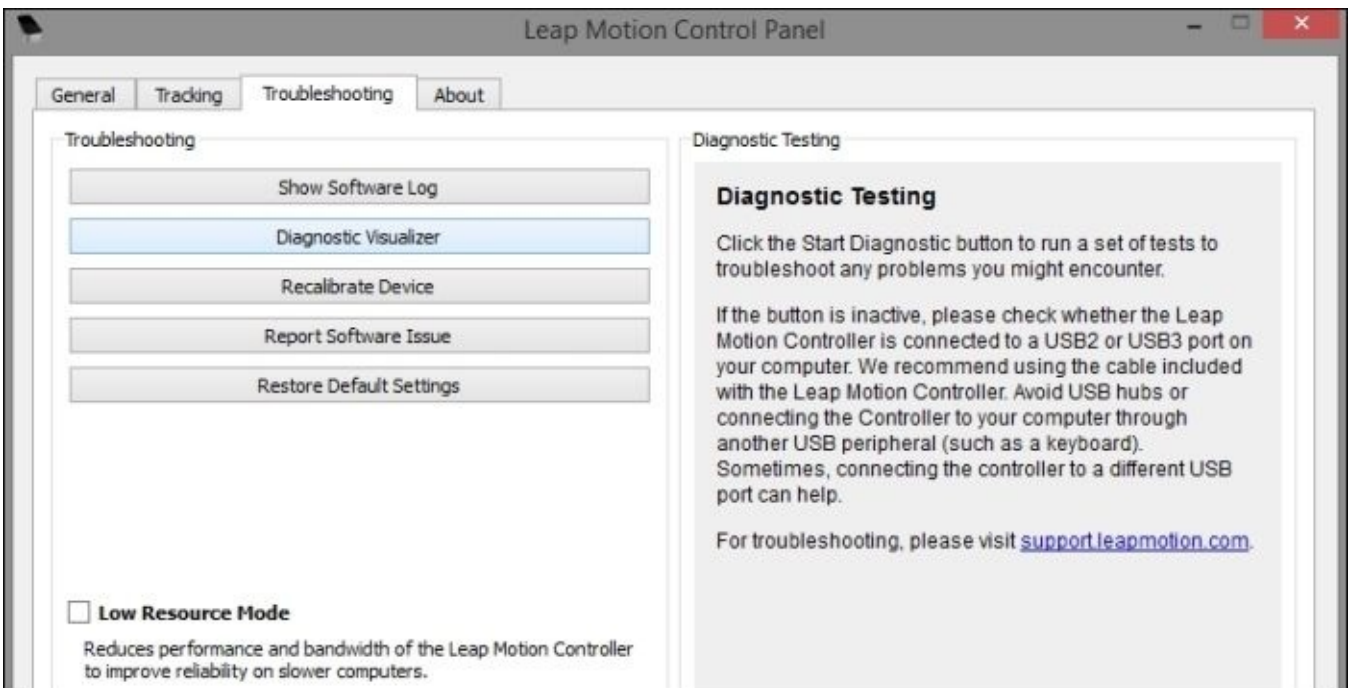


Note

Not-so-fun fact

Depending on your installation of the Leap software, the directions that were previously mentioned will not open the Diagnostic Visualizer. Instead, they'll open a very pretty (but relatively useless) application that shows very fancy versions of your hands.

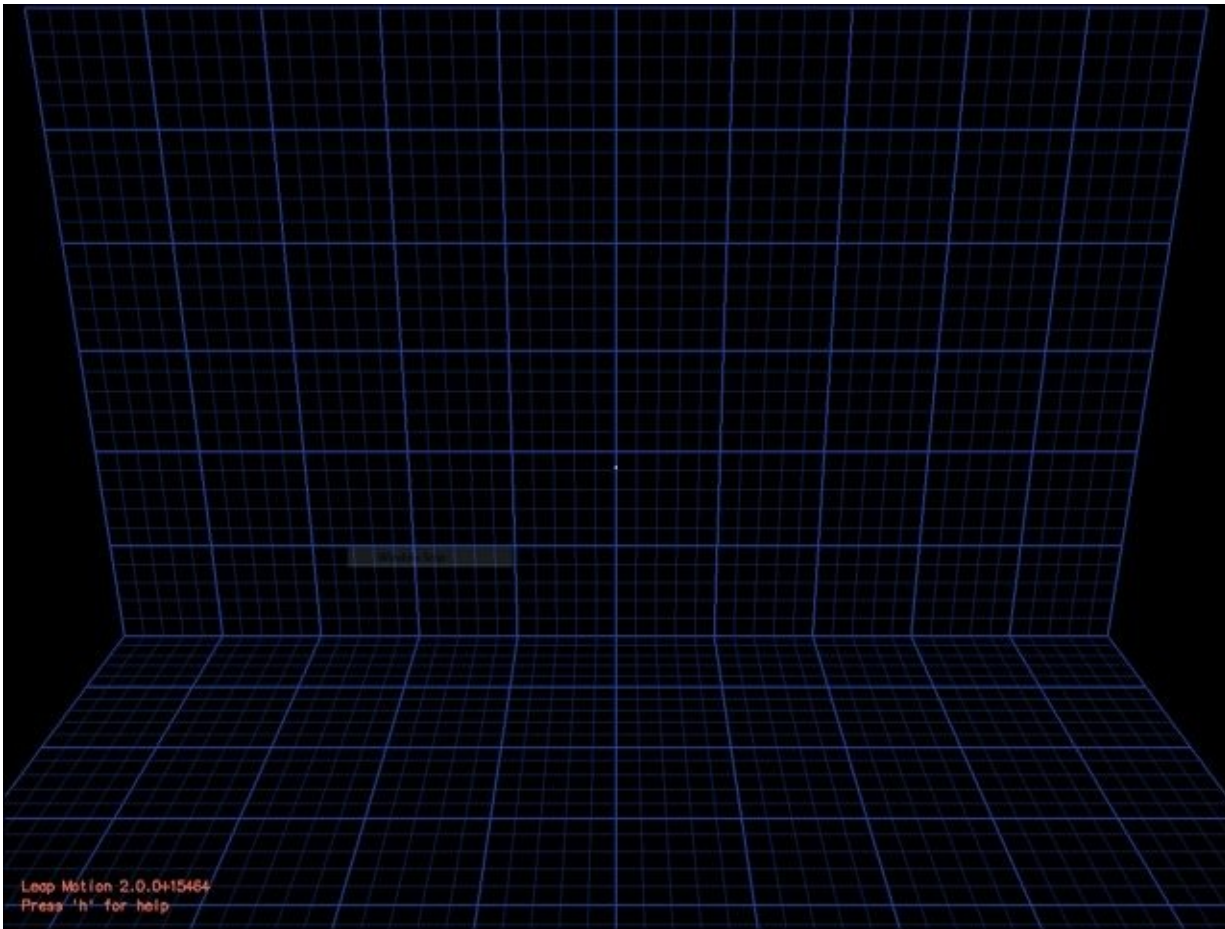
If this happens, you will instead need to right-click on the Leap Motion Control Panel from your system tray (as shown in the preceding screenshot) and click on **Settings**. Then, you'll want to go to **Troubleshooting | Diagnostic Visualizer** in the window that pops up, as seen in the next screenshot.



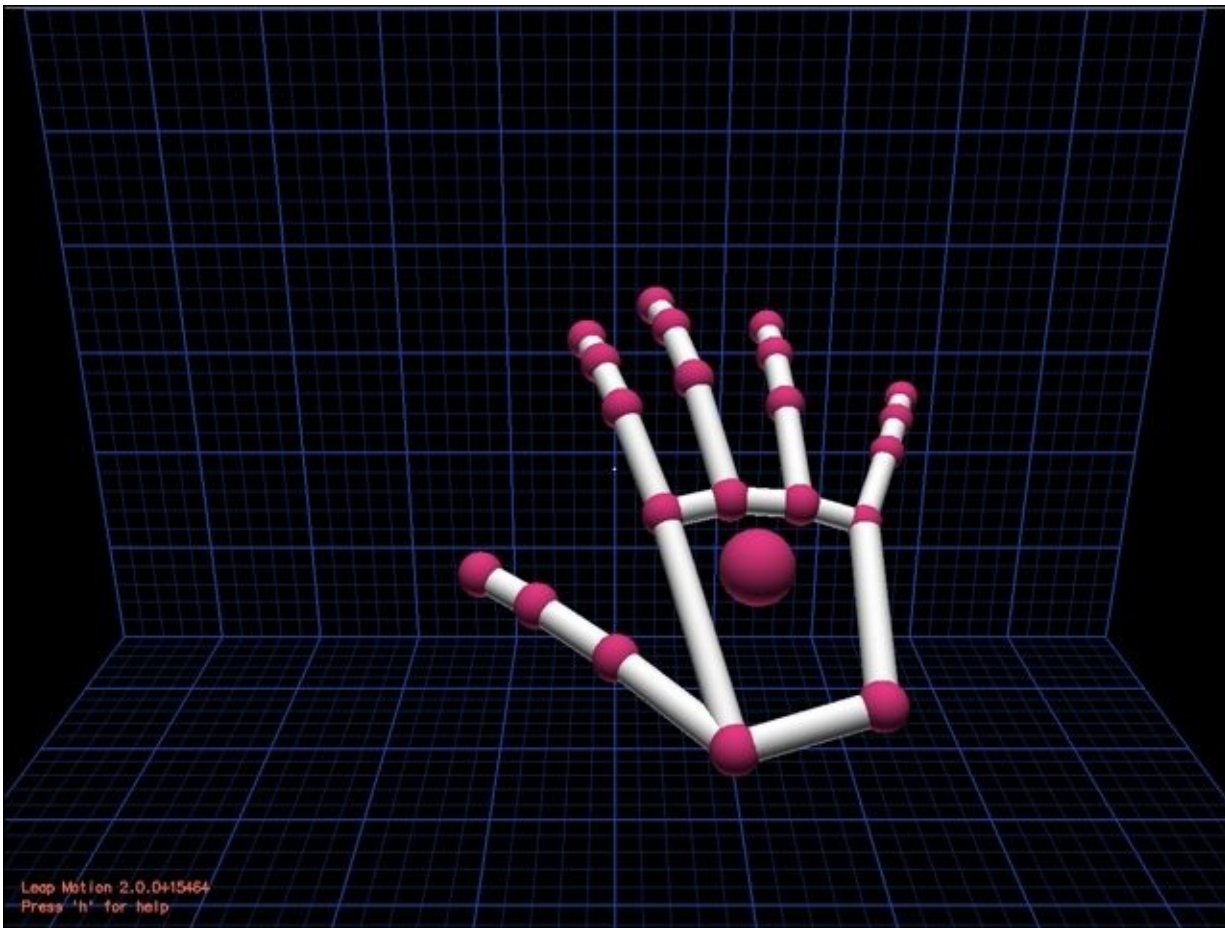
If all goes well, you'll be greeted by the same Diagnostic Visualizer seen next.

The Diagnostic Visualizer

Once the Visualizer is launched, you'll see a relatively empty window pop up that is similar to the one shown here:

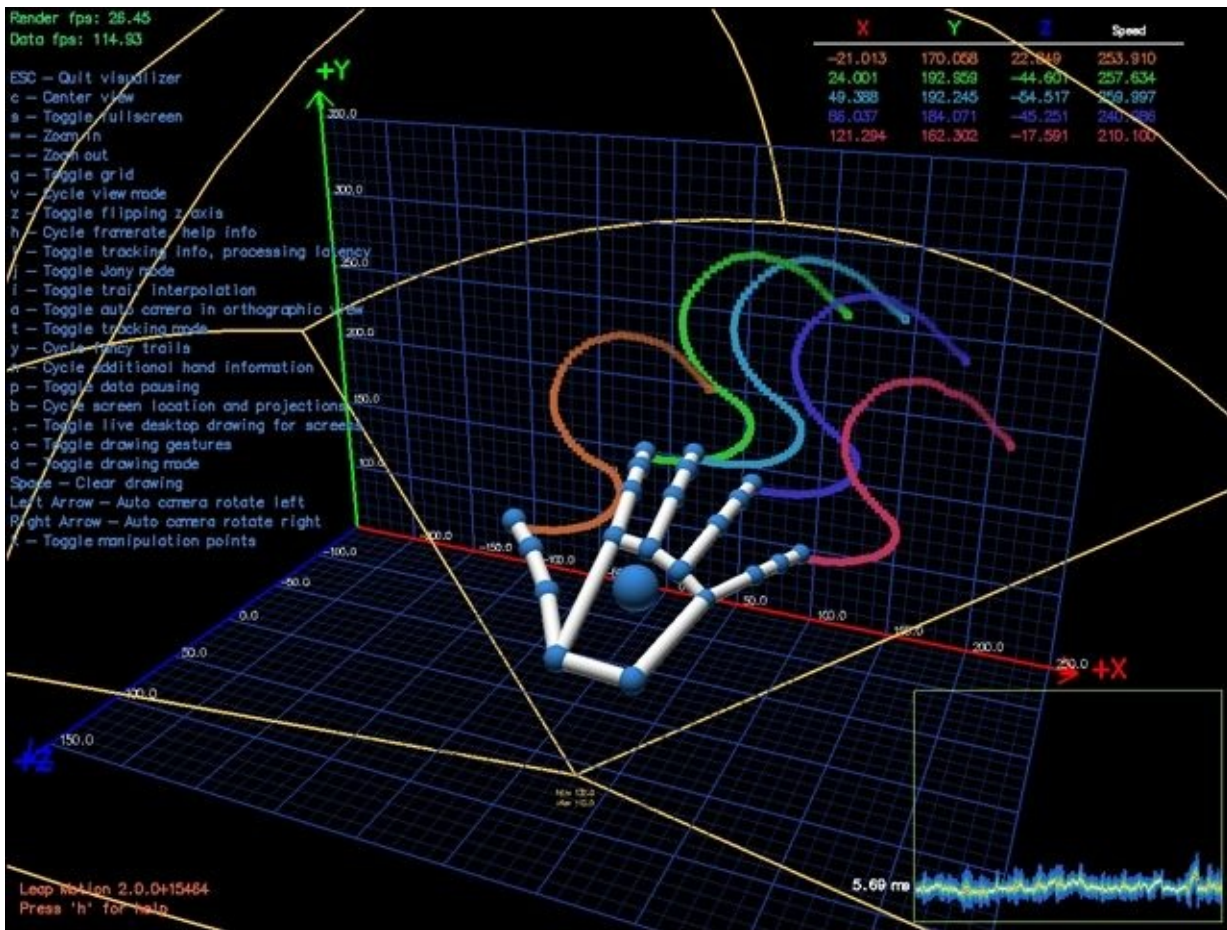


This application is known as the **Diagnostic Visualizer**, which allows developers to see a simple version of what the Leap is currently seeing. Go ahead and try placing your hand into the view—if the Leap is working, you will see a virtual representation of your hand similar to the following screenshot:



If you don't see your hand in the application and yet your Leap Motion device is plugged in, immediately proceed to the next section of this chapter, which is on how to keep the Leap Motion SDK updated. It's possible that you need to get an update for the controller.

The Leap Motion Diagnostic Visualizer is an extremely powerful tool, allowing developers to speed up their development process by being able to easily test the Leap Motion Controller's physical hardware. It is able to provide real-time tracking of data for all the key aspects of every single hand in view as well as provide virtual representations of the Leap's field of view. An example of the amount of data you can get from it is shown here:



In the preceding screenshot, I've turned on tracking info, processing latency, and trail interpolation by hitting the *I* and *L* keys. All of these commands can be listed by hitting the *H* key whenever the Visualizer window is in focus.

Keeping the Leap Motion SDK updated

The second thing to do when your application starts getting buggy or stops working entirely, after you confirm that the device is properly connected, is to update your Leap Motion SDK. Updating the SDK is identical to installing the SDK; all you're doing is overwriting old files with new ones. To do this, simply perform the following steps:

1. Go to <http://developer.leapmotion.com>.
2. Click on the **Downloads** link at the top of the screen.
3. Sign in if prompted.
4. Accept all the agreements and download the most recent SDK for the Windows platform (or a platform of your choice).
5. When the download concludes, extract the .zip archive and run the installer program.
6. When the SDK finishes installing, all you have to do next is replace your current Leap Motion DLLs with the ones from your newly downloaded SDK. This ensures there are no weird issues in the future, even though in most cases, old DLLs will work with new SDKs. To find these, simply navigate to the LeapSDK folder under /lib in your extracted SDK folder and copy the DLLs appropriate for your platform(s).

Cutting back on Leap Motion API calls

Simple as it might sound, cutting back on calls to the Leap API and caching tracking data can (slightly) increase the responsiveness of your code. Take the following (bad) example of an infinite loop that outputs the position of the first hand in the Leap's field of view to the console:

```
while (true)
{
    Controller controller = new Controller();

    if (controller.frame().hands().isEmpty() == false)
    {
        float x = controller.frame().hands().get(0).palmPosition().getX();
        float y = controller.frame().hands().get(0).palmPosition().getY();

        System.out.println("Hand X|Y: " + x + "|" + y);
    }
}
```

This piece of code is pretty simple and short, right? This shortness is actually its downfall, allowing this snippet to demonstrate a few things you can do wrong when writing a piece of Leap code.

Let's break down what's wrong with this snippet:

- We create an entirely new `Controller` reference in each iteration. This is terrible because initialization of a new object is always very expensive processor-wise.
- We retrieve the same frame three times via `controller.frame()`. This is unnecessary abuse of the Leap API, as we can just retrieve the frame once and store it for processing.

This example averages roughly 3-5 milliseconds per iteration on an Intel Core i5 2.4 GHz CPU.

Note

Fun fact

If you continuously grab a new frame from the controller throughout a single segment of the Leap code, like the preceding example, there is a chance that the tracking data in the frame will change between each line of your program—if this happens, it can totally wreck your code. Always, always, always cache the current frame when you're working with tracking data!

So, why don't we fix these issues and see how it looks? Here is the modified snippet:

```
Controller controller = new Controller();

while (true)
{
    Frame frame = controller.frame();
```

```

if (frame.hands().isEmpty() == false)
{
    float x = frame.hands().get(0).palmPosition().getX();
    float y = frame.hands().get(0).palmPosition().getY();

    System.out.println("Hand X|Y: " + x + ":" + y);
}
}

```

So, it's a bit longer, but it's also faster and clearer, now averaging 0 milliseconds and is practically instant! However, there's still one issue: we fetch the exact same hand two times in a row from the Leap API, which uses more time and resources than necessary.

While modern compilers (and Java Virtual Machine) probably inline the method we use to get the hand at runtime, preventing us from truly fetching the same value twice (thus removing any performance detriments from making the call twice), it's also a readability thing—the code just looks nicer if you only get the hand once.

Refer to the snippet here for how the fully optimized code should look:

```

Controller controller = new Controller();

while (true)
{
    Frame frame = controller.frame();

    if (frame.hands().isEmpty() == false)
    {
        Hand hand = frame.hands().get(0);

        System.out.println("Hand X|Y: " + hand.palmPosition().getX() + ":" +
hand.palmPosition().getY());
    }
}

```

While this snippet is even longer, it too averages only 0 milliseconds (again, basically instant) and is the most optimal version of all the snippets we've seen so far. The only change is that we now cache the hand-tracking data by retrieving it once and assigning it to a Hand object, saving on actual code, time, and operations.

Now would also be a good time to point out the biggest problem in our code, which exists mainly to make these examples way simpler: the presence of a `while (true)` loop. What's wrong with a loop, you ask? Inherently, there's nothing wrong. The problem is that when you use a loop that doesn't have any delays or conditional checks, it'll just keep trying to run forever and ever as fast as it possibly can. This behavior results in a loop that's updating faster than its input values, resulting in a huge amount of *wasted* processing power. Therefore, in the final production code, it's best to use loops with fixed intervals (via timers or delays in the loops) instead of loops that run as fast as they can.

However, making a proper game loop with timing is a complicated topic that goes far outside the scope of this chapter, so allow me to leave you with this: don't use `while (true)` loops that try to run as fast as possible!

Note

Fun fact

You might've noticed that throughout this book, there have been examples of Leap Motion programs that define custom `Listener` implementations and other examples that don't. Why is this? Well, you don't actually need to define a `Listener` class (or register one, for that matter!).

A general rule to follow is that you only need to use a `Listener` implementation when you want something to happen every single time the Leap receives a new frame or event. If you instead only need things to happen on your terms, say, an update loop in your main program, you can just define a `Controller` object and manually obtain the current frame from it (as seen in the examples in this section).

Of course, at this level of code (simply printing two values), you won't notice much of a difference. However, if you start writing very complex pieces of Leap code, for example, say, a skeletal tracking system, you will most certainly notice the difference.

Handling the `NoSuchMethod` and `NoClassDefFound` errors in Java

As this book makes extensive use of the Java programming language, this author thought it might be a nice touch to include two errors that you might get from Java if you don't have the Leap Motion SDK installed but try to run a Leap application anyway.

These two errors (or exceptions, as they are referred to in Java) are `java.lang.NoSuchMethodError` and `java.lang.NoClassDefFoundError`. They are caused (or thrown in Java terminology) whenever you try to instantiate a Leap Motion class or call a Leap Motion function without the Leap Motion runtime installed. What's worse, if these errors crop up, they will completely terminate the application with basically no warning whatsoever. This can be potentially disastrous if you're distributing a commercial application to a wide user base and some of them don't have anything to do with Leap Motion installed on their systems yet.

So what do you do in Java to counter this? We catch some exceptions!

Take our example from the previous section, where we print out some hand coordinates to the console. If you were to run it as is on a computer that didn't have the Leap Motion runtime installed, you would receive the `java.lang.NoSuchMethodError` error as the runtime could not be located. Not good. To fix this, we're going to surround everything Leap-related—in other words, the entire snippet—in a try-catch statement, like the example here:

```
try
{
    Controller controller = new Controller();

    while (true)
    {
        Frame frame = controller.frame();

        if (frame.hands().isEmpty() == false)
        {
            Hand hand = frame.hands().get(0);

            System.out.println("Hand X|Y: " + hand.palmPosition().getX() + ":"
+ hand.palmPosition().getY());
        }
    }
}

catch (java.lang.NoSuchMethodError e) { System.out.println("Leap Motion
runtime not found."); }
catch (java.lang.NoClassDefFoundError e) { System.out.println("Leap Motion
runtime not found."); }
```

The good news is that the program won't come to a screeching halt now—it'll just print out a message that says Leap Motion runtime not found. From here, you can add in a

fallback control system, or print a multitude of fancy error graphics informing the user that the Leap Motion software needs to be installed.

Tip

The important thing to take away from here is this: make sure that your application doesn't outright crash if the Leap Motion software isn't installed. Even more importantly, print out an error message telling the user to get it installed! Nothing's worse than a user trying to figure out why their app isn't working, only to realize that they were missing some libraries or executables.

Custom calibration of the Leap Motion Controller

While Leap Motion comes with calibration software to ensure they it is fairly consistent across installs, sometimes it isn't enough. For example, there might be some instances where you need to custom-calibrate the Leap Motion device to make sure that tracking data is consistent and reliable for your particular application.

If you find yourself in such a case, there's a relatively simple method to do this. Although simple, it's still a little lengthy. Peruse the well-documented snippet here for an example of how custom calibration can work:

```
import java.io.IOException;

import com.leapmotion.leap.Controller;
import com.leapmotion.leap.Frame;
import com.leapmotion.leap.Vector;

class CalibratedLeapApp
{
    //Vector that we'll be using for our central coordinates.
    public static Vector center = Vector.zero();

    //Sensitivity of the controller.
    public static final float sensitivity = 50;

    //Member Function: main
    public static void main(String args[]) throws InterruptedException,
    IOException
    {
        //Create the controller and frame objects for use later.
        Controller controller = new Controller();
        Frame frame = null;

        //Print out some nice text telling the user what to do.
        //The program will not proceed until the enter key is pressed.
        System.out.println("Press enter to calibrate the sensor.");
        System.in.read();
        System.out.println("Hold your hand over the sensor for a few
seconds.");

        //Iteration counter.
        int i = 1;

        //Get our base zero point. We don't use a for-loop here since that
would
        //prevent us from "waiting" for a user to place a hand in view.
        while (i <= 10)
        {
            //Retrieve the latest frame from the Leap.
            frame = controller.frame();
```

```

//Only perform operations if hands are in view.
//If a hand isn't in view, the loop will keep going infinitely.
if (frame.hands().count() > 0)
{
    //Get the position of the first hand into the Leap's view.
    Vector handPosition = frame.hands().get(0).palmPosition();

    //If this is the first iteration of the loop, tell the user to
wait.
    if (i == 1)
    {
        System.out.println("Please wait.");
        Thread.sleep(1000);
    }

    //Print out the current iteration and calibrated coordinates.
    System.out.println("Calibrating..." + i + "..." + handPosition);

    //Append the coordinates to our zero position.
    center = center.plus(handPosition);

    i++;
}

//Wait a fifth of a second before getting the next zero.
Thread.sleep(200);
}

//Average out the zero coordinates to get our actual zero position.
center = center.divide(10);

//Print out the final zero coordinates to the screen.
System.out.println("Calibration complete!");
System.out.println("Zero: " + zero);

//Begin tracking hands.
while (true)
{
    //Retrieve the latest frame from the Leap.
    frame = controller.frame();

    //Only perform operations if hands are in view.
    if (frame.hands().count() > 0)
    {
        //Textual position of the hand. We'll start out by assuming it's
centered.
        String posX = "Center";
        String posY = "Center";
        String posZ = "Center";

        //Raw position of the first hand in the Leap Motion's view.
        Vector rawPosition = frame.hands().get(0).palmPosition();

        //If the hand is greater than or less than the X zero coordinate
+/- our tolerance,
        //assign it a value of Left or Right.

```

```

        if (rawPosition.getX() > center.getX() + sensitivity ||
rawPosition.getX() < center.getX() - sensitivity)
        {
            if (rawPosition.getX() < center.getX())
                posX = "Left";

            else
                posX = "Right";
        }

        //If the hand is greater than or less than the Y zero coordinate
+/- our tolerance,
        //assign it a value Up Left or Down.
        if (rawPosition.getY() > center.getY() + sensitivity ||
rawPosition.getY() < center.getY() - sensitivity)
        {
            if (rawPosition.getY() > center.getY())
                posY = "Up";

            else
                posY = "Down";
        }

        //If the hand is greater than or less than the Z zero coordinate
+/- our tolerance,
        //Assign it a value of Front or Back.
        if (rawPosition.getZ() > center.getZ() + sensitivity ||
rawPosition.getZ() < center.getZ() - sensitivity)
        {
            if (rawPosition.getZ() < center.getZ())
                posZ = "Front";

            else
                posZ = "Back";
        }

        //Print out the final position of the hand relative to our zero
coordinates.
        System.out.println("X-Axis Position: " + posX);
        System.out.println("Y-Axis Position: " + posY);
        System.out.println("Z-Axis Position: " + posZ);
    }

    //Wait for half a second to prevent flooding the console with
positions.
    Thread.sleep(500);
}
}
}
}

```

If you were to put this code into a Java file and run it from within Eclipse, you would be presented with a series of prompts that guide you through the basic calibration of a zero position for your hand. Upon completion of the calibration, this example will then continuously print the coordinates of the first hand in the Leap's field of view, relative to the calibrated zero point, to the console.

Personally, this method served me quite well in my early days of working with Leap-Motion-enabled robots. While I've since taken to using pitch, roll, and yaw values for hands, I'm sure there are quite a few places for this kind of custom calibration.

However, wherever possible, I encourage you to use the predefined calibrations from the Leap Motion device, in conjunction with the `InteractionBox` class and the other useful utilities. This will save the user from having to *recalibrate* their Leap as well as possibly allow for a more consistent experience across platforms and installations.

Summary

In this chapter, we covered a variety of topics related to the troubleshooting and debugging of the Leap Motion code. We started off by discussing how to verify a Leap Motion Controller connected using the Diagnostic Visualizer and then making sure your SDK was up to date. We then went over a bad example of the Leap Motion code and talked about how it can be optimized to be much faster. The chapter finished off with an example of how you can create a custom calibration routine for the Leap Motion device.

In the next chapter, we'll review what we've done so far and then look ahead to the future of Leap Motion and how it stacks up with other emerging technologies. Naturally, we'll also discuss a bit about how the Leap Motion Controller can be used with robots. Yes, robots.

Chapter 9. Going beyond the Leap Motion Controller

Armed with all the knowledge to master developing with the Leap, you're finally ready to go beyond the device. In this final chapter, we will cover all sorts of abstract concepts and ideas that go far beyond just software.

Note

This chapter is more of an abstract chapter, retracing what you've learned up to now and then looking at a whole slew of things that are related to, but outside the world of the Leap Motion Controller itself.

This chapter is sprinkled with periodic *Fun facts* that offer high-level and entry-level factoids about scripting and programming for your reading pleasure.

We will cover the following topics in this chapter:

- What you've learned so far
- Moving forward – where the Leap Motion Controller stands next to other emerging technologies
- Concerns regarding the reliability and safety of the device in industrial settings
- Going beyond – ideas to control robots with the Leap Motion Controller

What you've learned so far

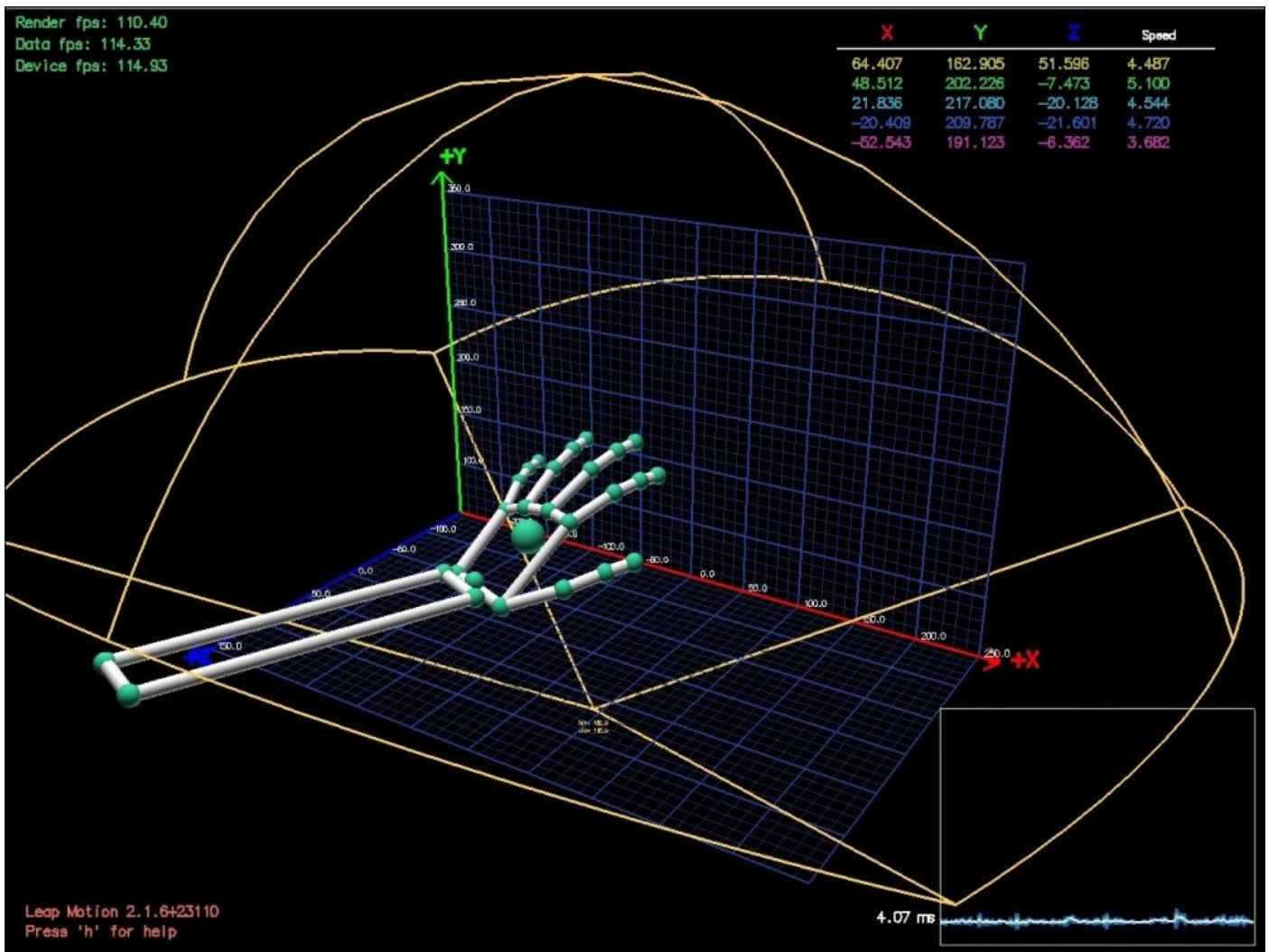
We've covered a lot of things together over the course of the past eight chapters, ranging from the basics of the Leap Motion API all the way up to making a complete three-dimensional application; how about a brief recap of what we've learned?



*Testing out the Visualizer for the first time in [Chapter 1](#), *Introduction to the World of Leap Motion**

In [Chapter 1](#), *Introduction to the World of Leap Motion*, we got the Leap Motion SDK up and running and then covered some basics that surround the Leap Motion Application Programming Interface, or API.

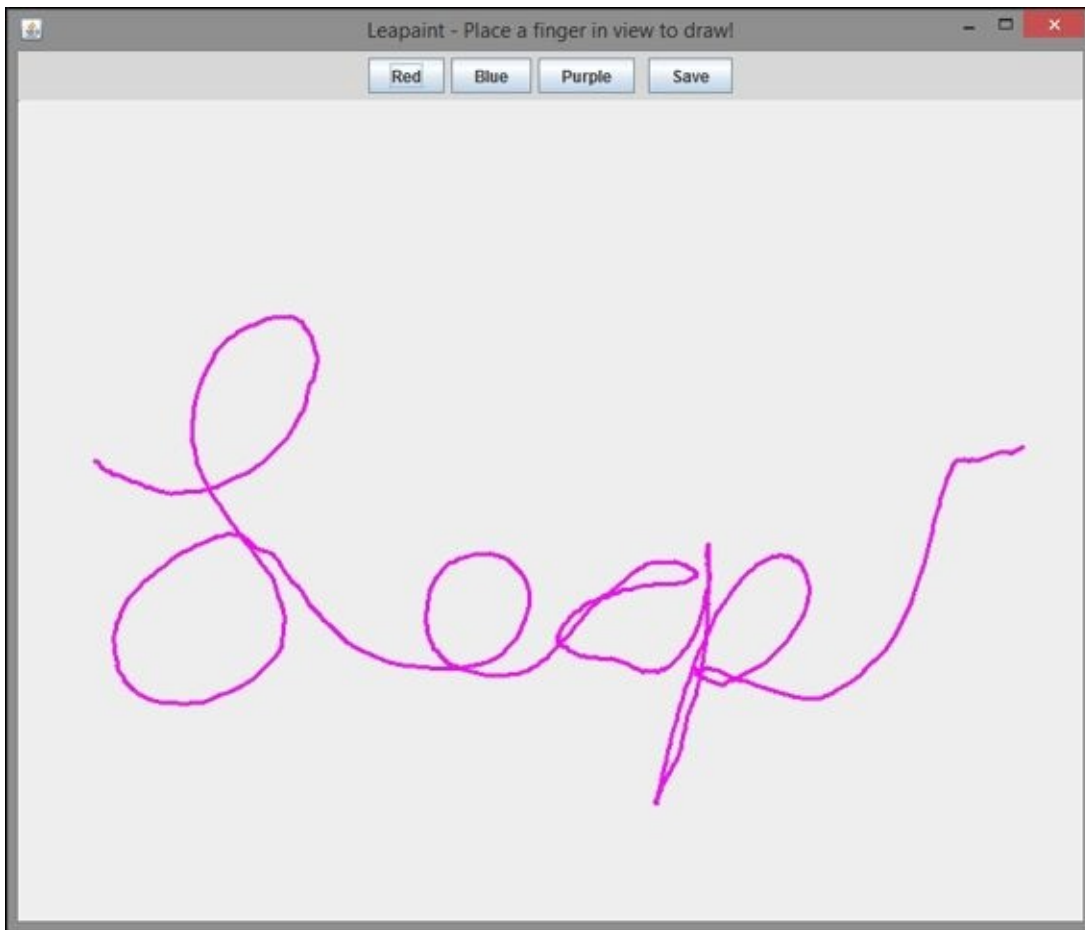
Let's take a look at the Leap Motion Controller's field of view in [Chapter 2](#), *What the Leap Sees – Dealing with Fingers, Hands, Tools, and Gestures*, shown in the following screenshot:



In the second chapter, we covered what the Leap Motion Controller sees, how to detect tools and gestures, while putting particular emphasis on the `InteractionBox` class. We finished this chapter off with a review of some of the limitations.

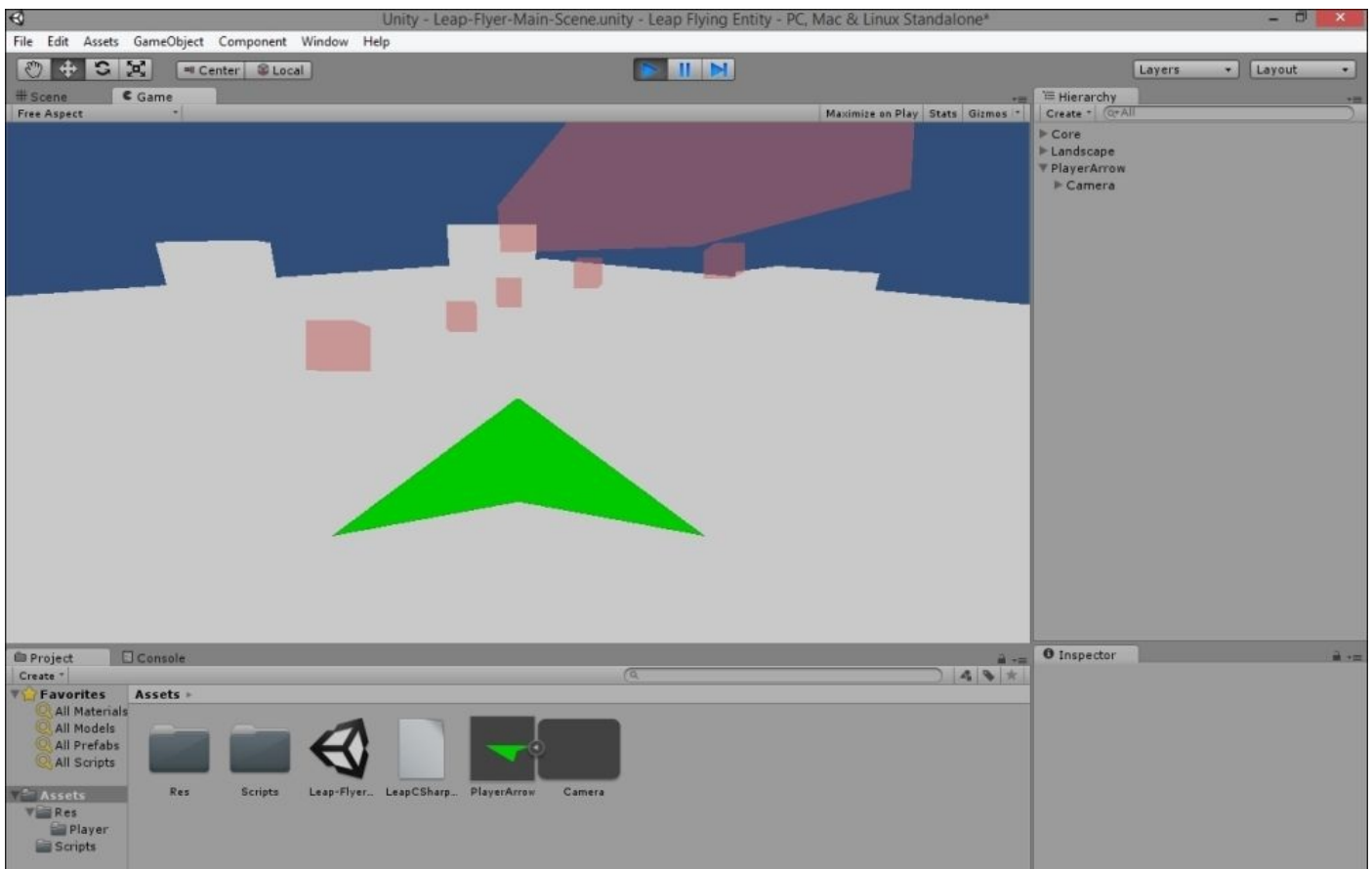
In [Chapter 3](#), *What the User Sees – User Experience, Ergonomics, and Fatigue*, we reviewed a part of the many applications developers often forget: the user experience. We covered topics ranging from ergonomics and fatigue to basic interface design.

Let's have a look at a simple drawing application that we made in [Chapter 4](#), *Creating a 2D Painting Application*, shown in the following screenshot:



In [Chapter 4](#), *Creating a 2D Painting Application*, we created a simple two-dimensional (or 2D) painting application for the Leap Motion Controller. We started by laying out a basic framework for the application. Then, we created the graphical frontend, including responsive buttons and a colorful interface. We then converted Leap's tracking data into input, finishing off the chapter with a test run of the application.

The following screenshot shows the 3D application we made using Unity in [Chapter 5](#), *Creating a 3D Application – a Crash Course in Unity 3D*; [Chapter 6](#), *Creating a 3D Application – Integrating the Leap Motion Device with a 3D Toolkit*; and [Chapter 7](#), *Creating a 3D Application – Controlling a Flying Entity*:



Throughout these three chapters, we created a complete three-dimensional (or 3D) application. We started by learning the basics of the Unity 3D toolkit, followed by the creation of a simple project and 3D scene. After that, we learned how to pull content from the Leap Motion device into the 3D scene; this included the rendering of fingers, hands, and buttons. We concluded [Chapter 6, *Creating a 3D Application – Integrating the Leap Motion Device with a 3D Toolkit*](#), by covering how to react to actions from the user and detect fingers on buttons. We then proceeded to create a simple 2D arrow, or flying entity, to control. After this, we scripted a controller class that used the Leap Motion Controller as an input for the *x* and *y* axes. We then put everything together and tested it, with the result being a fully functional 3D simulation.

Finally, in [Chapter 8, *Troubleshooting, Debugging, and Optimization*](#), we covered a selection of things that have the potential to go wrong when using or developing for the Leap Motion Controller; this included updating the Leap Motion SDK, cutting back on API calls, custom calibration of the Leap Motion Controller, and more.

The Leap Motion Controller standing next to other emerging technologies

Now that you've mastered the essentials of developing with the Leap Motion Controller, let's have a look at some other technologies.

Microsoft's Kinect

It's no secret that the closest competitor out there to the Leap Motion Controller is Microsoft's very own Kinect. In fact, I often find myself using Kinect as a sort of *point of reference* when trying to describe the Leap Motion Controller to other people.



Microsoft's newest iteration of Kinect

In short, Kinect is a motion tracker developed by Microsoft for its Xbox 360 and Xbox One gaming platforms. It was originally designed to track the individual major limbs on a person's body (head, arms, hands, legs, and feet). Since its original release, it has also been used in a number of other projects. For example, I've used it as the primary vision system on a robot, thanks to its advanced suite of range finders and cameras offered at a very competitive price along with a well-documented and widely available API.

However, if Kinect is more mature than the Leap Motion Controller and competitively priced, why would we developers use the Leap instead? Although it's true that the latest advancements in Kinect technology have improved its ability to track people's hands, it still isn't anywhere near as good as what the Leap Motion Controller achieves. The exact metrics have definitely changed over the past year or so, but at launch, the Leap Motion Controller was about 200 times more accurate than Kinect. 200!!

Naturally, Kinect (at the time of writing this) remains the top contender for detecting a user's entire body. This is what it was designed for. It's also a lot better at being used as an advanced vision system for objects such as robots. I've had the pleasure of using one myself, and it's quite amazing. In time, though, the Leap Motion Controller will rise above Kinect in every category.

Oculus VR's Oculus Rift

In contrast to the other technology we just covered, the Kinect, the Oculus Rift is less of a competitor with the Leap Motion Controller and more of a collaborator.



One of Oculus VR's earlier Rift units

In short, the Rift is a very new kind of technology, commonly referred to as either a **head-mounted display (HMD)**, **virtual reality (VR)**, or both. In other words, it's a display... on your head.

However, what does the Rift have to do with the Leap, you ask? They're two totally different technologies...right? Not really. The primary goal of the Leap Motion Controller is to revolutionize the way we control machines, and virtual reality falls under this goal. When you use the Leap with a monitor, a lot of immersion is lost because there's no sense of depth; you can't "reach in" and grab an object when everything is a purely two-dimensional experience.



The Oculus Rift with the Leap Motion Controller mounted to it

This is where the Rift comes into play. Utilizing its dual-display stereoscopic 3D in conjunction with proprietary software, you can truly create an experience that's just like, or very close to, the real world. In fact, Leap Motion has even developed a special mount for the Rift that allows you to move around freely while having your hands in view (shown in the preceding image).

Note

Fun fact

Stereoscopic 3D is a fancy term for, among other things, having a miniature display for each of your eyes, allowing a system to make two 2D images appear truly three-dimensional in every respect. This term also applies to other methods of creating a 3D effect, like those classic red and blue 3D glasses you used to see in movie theaters way back.

Although all of this technology is still very young, it's growing at an exceptional rate. Who knows what the future holds for the Leap Motion Controller?

For more information on Oculus VR, you can visit its official website at <http://www.oculusvr.com/>.

Reliability and safety concerns with the Leap in industrial settings

As is always the case with new and old technologies alike, reliability and safety concerns with the usage of technology in industrial, medical, or high-risk settings have always been the subject of much scrutiny. Such settings include military robots, industrial automation systems, medical systems (such as surgery robots), and more.

The developers over at Leap Motion heavily discourage the usage of the Leap Motion Controller in such settings, even including a clause in the official license (available at <https://central.leapmotion.com/agreements/SdkAgreement>) preventing the distribution (but not the creation and internal usage) of such applications without an additional special license from Leap Motion.

Leap Motion elaborates further on this in its **frequently asked questions (FAQ)** section for the SDK license agreement:

“What is a Specialized Application, and why must I contact Leap Motion if I want to distribute one?”

Basically, a Specialized Application is a Leap Motion-enabled application which is: (i) priced at more than US\$500 (or \$240/year if on a subscription or similar basis); (ii) for use with a system, machine or device (other than a PC), priced at more than US\$500 (or \$240/year if on a subscription or similar basis); or (iii) designed for use with or control of industrial, military, commercial, or medical equipment.

If you would like to distribute a Specialized Application, please contact our business development team at <bizdev@leapmotion.com>.

We have set \$500 (or \$240/year if on a subscription basis) as the price limit for applications that can be distributed under the SDK Agreement because we think that above these prices the use may be so specialized that it is appropriate to have a separate agreement. The same goes for uses with other systems, machines or devices above \$500 / \$240. We believe these thresholds should allow 99% of Leap Motion-enabled applications to be distributed under the SDK Agreement, without needing to get a special distribution license from us.

We do not allow distribution under the SDK Agreement of applications at any price if the application is to control industrial, military, commercial, or medical equipment. Again, we believe that for these uses it is appropriate to have a separate agreement.

Does Leap Motion have a standard agreement for Specialized Applications?

Because of the diverse range of potential Specialized Applications, we do not have a standard agreement. Please contact our business development team at <bizdev@leapmotion.com>.

Does the SDK Agreement restrict development of Specialized Applications?

No. The restriction in the agreement applies to distribution of Specialized Applications. You can still test and develop Specialized Applications.

I'm a maker, and have put together something to control my drone / robot / dragon. What's up?

In general, make away! If you are doing it for yourself, and don't plan to distribute it to others, you are largely free to test and develop as you like. And, if you do want to distribute your app, our restrictions apply to industrial, military, commercial or medical equipment, or to other equipment if you plan to sell it (and then, only if it's priced at more than \$500). (Of course, we don't allow uses where failure of the controller or software could lead to death or serious bodily injury of any person, or to severe physical or environmental damage.)"

So, as you can see, Leap Motion is okay with you making these kinds of applications, but it doesn't want to take responsibility, and it also restricts the distribution of such apps without written permission (for a good reason).

However, why would the Leap Motion Controller be so potentially dangerous in a high-risk environment? Well, straight off, it's a very new technology that's quite unstable and has weekly updates. This is not quite what you'd call *stable*. The sometimes erratic behavior of the Leap Motion Controller could have the potential to throw a robotic arm out of control, severely injuring someone nearby. As the technology develops further and further, it's possible that these restrictions might be lifted; for now, keep any proprietary or high-risk applications to yourself!

To summarize:

- You can make Leap Motion applications for use in industrial, commercial, medical, military, or other high-risk settings
- You cannot redistribute such applications, no matter the price you're charging, without written permission from Leap Motion

Going beyond – ideas to control hardware and robots with the Leap Motion Controller

For this section, I thought it would be nice to focus on what I specialize in—hardware and robotics systems. While normal applications are all fine and good, I find it much more gratifying if a program I write has an impact in the physical world, and I think robots achieve this in the best way possible.

Over the past year and a half since I received my first Leap Motion developer unit, I've been working on various platforms that use the Leap Motion Controller. I thought I might share some of these with you to give you some ideas as to what you can manipulate in the real world with the Leap Motion Controller.

Arduino

One of the most popular hobbyist hardware solutions, as I'm sure you know, is the Arduino. This cute little blue board from Italy brought the power of microcontrollers to the masses. The following image shows the Arduino board:



For one of our last tutorials in this magnificent title, I thought it would be nice if we programmed a physical microcontroller to do something. Don't you agree? The project we're going to do in this section will be relatively simple—we're going to make the built-in LED on an Arduino blink either slower or faster, depending on how far a user's hand is away from the Leap.

Unlike our previous projects, this one will follow the client-server model of programming (as will most other hardware-related endeavors); we'll be writing a simple Java server that will be run from a computer and a C++ client that will run on an Arduino connected to the computer. The server will be responsible for retrieving Leap Motion input and sending it to the client, while the client will be responsible for making an LED blink based on data received from the server.

Note

Before we begin, I'd like to note that you can download the completed (and working) project from GitHub at <https://github.com/Mizumi/Mastering-Leap-Motion-Chapter-9-Project-Leapduino>.

A few things you'll need

Before you begin working on this tutorial, there are a few things you'll need:

- A computer (for obvious reasons)
- An Arduino; this tutorial is based around the Uno model, but other similar models, such as Mega, should work just as well
- A USB cable to connect your Arduino to your computer

Setting up the environment

Before we begin coding an Arduino application, there are two things you'll (besides all the Leap-specific things) need: the **Java Simple Serial Connector (JSSC)** library and the Arduino IDE.

You can download the library JAR for JSSC from GitHub at <https://github.com/scream3r/java-simple-serial-connector/releases>. Once the download completes, extract the JAR file from the downloaded ZIP folder and store it somewhere safe; you'll need it later on in this tutorial.

You can then proceed to download the Arduino IDE from its official website at <http://arduino.cc/en/Main/Software>. If you're on Windows, you will be able to download a Windows installer file, which will automatically install the entire IDE on to your computer. On the other hand, Mac and Linux users will need to instead download the .zip or .tgz file and then extract them manually, running the executable binary from the extracted folder contents.

Setting up the project

To set up our project, perform the following steps:

1. The first thing we're going to do is create a new Java project in Eclipse (this tutorial will assume that you're using Eclipse, as that's what we've used throughout the rest of the book). This can be easily achieved by opening up Eclipse and heading over to **File | New | Java Project**.
2. You will then be greeted by a project creation wizard, where you'll be prompted to choose a name for the project (I used Leapduino).
3. Click on the **Finish** button when you're done.

Note

As we've spent so much time working in Unity, as you work through this section I'll be refreshing you on how to perform some of the basic tasks in Eclipse for your reading pleasure.

Once the project is created, navigate to it in the Package Explorer. Go ahead and perform the following actions:

1. Create a new package for the project by right-clicking on the src folder for your project in the Package Explorer and then navigating to **New | Package** in the

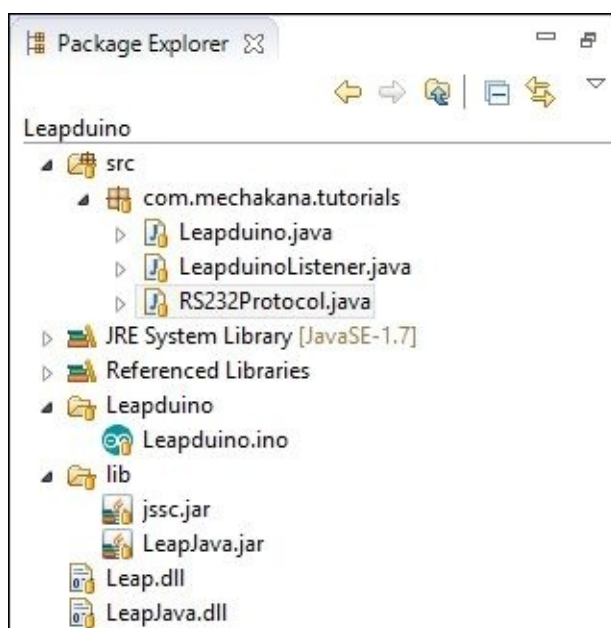
resulting tooltip. You can name it whatever you like; I called mine `com.mechakana.tutorials`.

2. Now, add three files to our newly created package: `Leapduino.java`, `LeapduinoListener.java`, and `RS232Protocol.java`. To create a new file, simply right-click on the package and then navigate to **New | Class**.
3. Create a new folder in your project by right-clicking on the project name in the Package Explorer and then navigating to **New | Folder** in the resulting tooltip. For the purposes of this tutorial, name it `Leapduino`.
4. Now add one file to your newly created folder: `Leapduino.ino`. This file will contain all the code that we're going to upload to the Arduino.

With all of our files created, we need to add the libraries to the project. Go ahead and create a new folder at the root directory of your project, called `lib`. Within the `lib` folder, place the `jssc.jar` file that you downloaded earlier, along with the `LeapJava.jar` file from the Leap Motion SDK. Then, add the appropriate `Leap.dll` and `LeapJava.dll` files for your platform to the root of your project.

Finally, you'll need to *link* the `jssc.jar` and `LeapJava.jar` files to your project. Refer to the *Creating a simple framework program within the Eclipse IDE* section in [Chapter 1, Introduction to the World of Leap Motion](#), for more information on this.

When you're done, your project should look similar to the following screenshot:



You're done; now to write some code!

Writing the Java side of things

With everything set up and ready to go, we can start writing some code. First off, we're going to write the `RS232Protocol` class, which will allow our application to communicate with any Arduino board connected to the computer via a serial (RS-232) connection.

This is where the JSSC library will come into play, allowing us to quickly and easily write code that would otherwise be quite lengthy (and not fun).

Note

Fun fact

RS-232 is a standard for serial communications and the transmission of data. There was a time when it was a common feature on a personal computer and was used for modems, printers, mice, hard drives, and so on. With time, though, the **Universal Serial Bus (USB)** technology replaced RS-232 for many of those roles.

Despite this, today's industrial machines, scientific equipment, and (of course) robots still make heavy usage of this protocol due to its light weight and ease of use; the Arduino is no exception!

Go ahead and open up the `RS232Protocol.java` file that we created earlier and enter the following:

```
package com.mechakana.tutorials;

import jssc.SerialPort;
import jssc.SerialPortEvent;
import jssc.SerialPortEventListener;
import jssc.SerialPortException;

public class RS232Protocol
{
    //Serial port we're manipulating.
    private SerialPort port;

    //Class: RS232Listener
    public class RS232Listener implements SerialPortEventListener
    {
        public void serialEvent(SerialPortEvent event)
        {
            //Check if data is available.
            if (event.isRXCHAR() && event.getEventValue() > 0)
            {
                try
                {
                    int bytesCount = event.getEventValue();
                    System.out.print(port.readString(bytesCount));
                }

                catch (SerialPortException e) { e.printStackTrace(); }
            }
        }
    }

    //Member Function: connect
    public void connect(String newAddress)
    {
        try
        {
```

```

//Set up a connection.
port = new SerialPort(newAddress);

//Open the new port and set its parameters.
port.openPort();
port.setParams(38400, 8, 1, 0);

//Attach our event listener.
port.addEventListener(new RS232Listener());
}

catch (SerialPortException e) { e.printStackTrace(); }
}

//Member Function: disconnect
public void disconnect()
{
    try { port.closePort(); }

    catch (SerialPortException e) { e.printStackTrace(); }
}

//Member Function: write
public void write(String text)
{
    try { port.writeBytes(text.getBytes()); }

    catch (SerialPortException e) { e.printStackTrace(); }
}
}

```

All in all, `RS232Protocol` is a simple class—there really isn’t a whole lot to talk about here! However, I’d love to direct your attention to one interesting part of the class:

```

public class RS232Listener implements SerialPortEventListener
{
    public void serialEvent(SerialPortEvent event) { /*code*/ }
}

```

You might have found it rather odd that we didn’t create a function to read from the serial port—we only created a function to write to it. This is because we’ve opted to utilize an event listener, the nested `RS232Listener` class. Under normal operating conditions, this class’ `serialEvent` function will be called and executed every single time new information is received from the port. When this happens, the function will print all the incoming data out to the user’s screen. Isn’t that nifty?

Moving on, our next class is a familiar one—`LeapduinoListener`, a simple `Listener` implementation. This class represents the meat of our program, receiving Leap Motion tracking data and then sending it over our serial port to the connected Arduino.

Go ahead and open up `LeapduinoListener.java` and enter the following code:

```

package com.mechakana.tutorials;

import com.leapmotion.leap.*;

```

```

public class LeapduinoListener extends Listener
{
    //Serial port that we'll be using to communicate with the Arduino.
    private RS232Protocol serial;

    //Constructor
    public LeapduinoListener(RS232Protocol serial)
    {
        this.serial = serial;
    }

    //Member Function: onInit
    public void onInit(Controller controller)
    {
        System.out.println("Initialized");
    }

    //Member Function: onConnect
    public void onConnect(Controller controller)
    {
        System.out.println("Connected");
    }

    //Member Function: onDisconnect
    public void onDisconnect(Controller controller)
    {
        System.out.println("Disconnected");
    }

    //Member Function: onExit
    public void onExit(Controller controller)
    {
        System.out.println("Exited");
    }

    //Member Function: onFrame
    public void onFrame(Controller controller)
    {
        //Get the most recent frame.
        Frame frame = controller.frame();

        //Verify a hand is in view.
        if (frame.hands().count() > 0)
        {
            //Get some hand tracking data.
            int hand = (int) (frame.hands().frontmost().palmPosition().getY());

            //Send the hand pitch to the Arduino.
            serial.write(String.valueOf(hand));

            //Give the Arduino some time to process our data.
            try { Thread.sleep(30); }
            catch (InterruptedException e) { e.printStackTrace(); }
        }
    }
}

```



```
}
```

At this point, you should be very familiar with what a standard Leap Motion Listener implementation looks like—we've got the basic `onInit`, `onConnect`, `onDisconnect`, `onExit`, and `onFrame` functions.

Our `onFrame` function is fairly straightforward: we get the most recent frame, verify a hand is within view, retrieve its y axis coordinates (the height from the Leap Motion Controller), and then send it off to the Arduino via our instance of the `RS232Protocol` class (which gets assigned during initialization).

And now, for our final class on the Java side of things: `Leapduino`! This class is a super basic main class that simply initializes the `RS232Protocol` class and the `LeapduinoListener`—that's it!

Without further ado, go ahead and open up `Leapduino.java` and enter the following code:

```
package com.mechakana.tutorials;

import com.leapmotion.leap.Controller;

public class Leapduino
{
    //Main
    public static final void main(String args[])
    {
        //Initialize serial communications.
        RS232Protocol serial = new RS232Protocol();
        serial.connect("COM4");

        //Initialize the Leapduino listener.
        LeapduinoListener leap = new LeapduinoListener(serial);
        Controller controller = new Controller();
        controller.addListener(leap);
    }
}
```

Like all the classes so far, there isn't a whole lot to say here. That said, there is one line that you must absolutely be aware of, as it can change depending on how your Arduino is connected:

```
serial.connect("COM4");
```

Depending on which port Windows chose for your Arduino when it is connected to your computer (more on that next), you will need to modify the `COM4` value in the preceding line of code to match the port your Arduino is on. Examples of values you'll probably use are `COM3`, `COM4`, and `COM5`.

With this, the Java side of things is complete. If you run this project right now, most likely all you'll see will be two lines of output: `Initialized` and `Connected`. If you want to see anything else happen, you'll need to move on to the next section and get the Arduino side of things working.

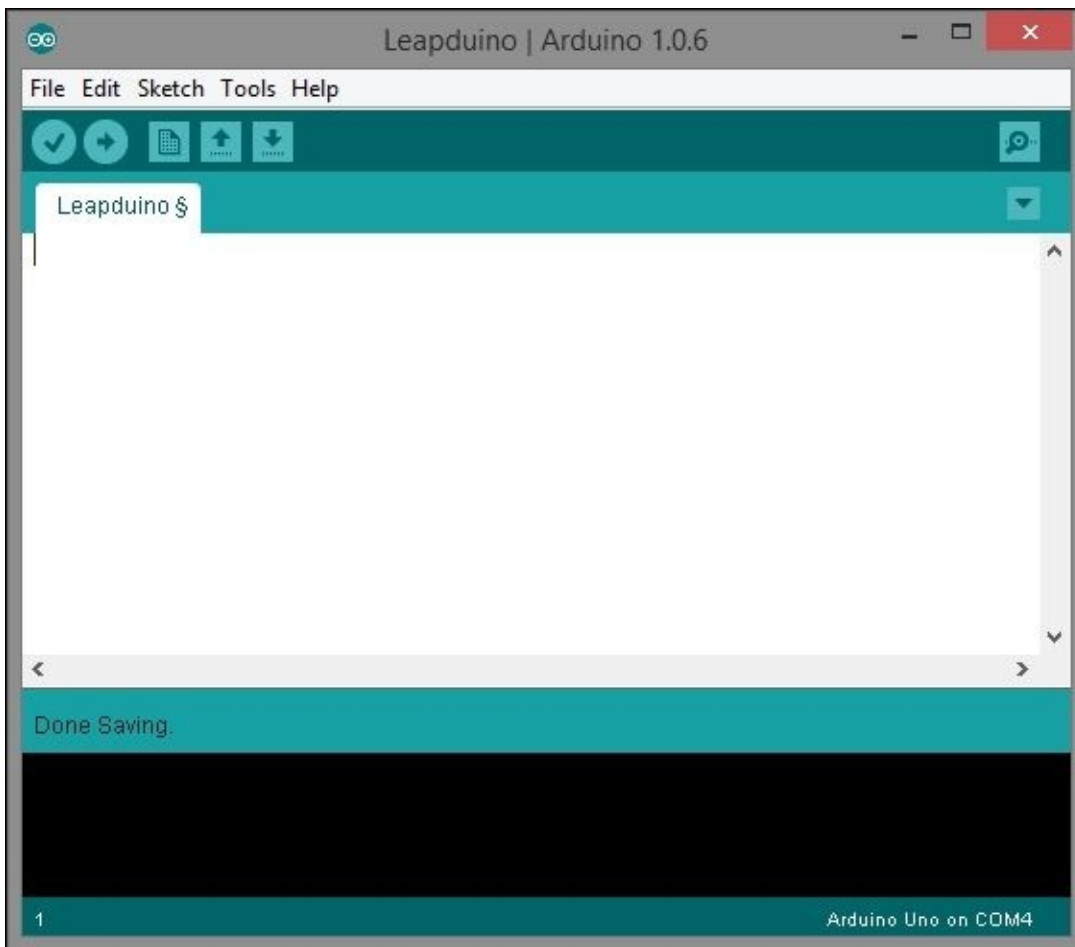
Writing the Arduino side of things

With our Java coding done, it's time to write some good-old C++ for the Arduino. If you were able to use the Windows installer for Arduino, simply navigate to the `Leapduino.ino` file in your Eclipse project explorer and double-click on it.

Note

If you had to extract the entire Arduino IDE and store it somewhere instead of running a simple Windows installer, navigate to it and launch the `Arduino.exe` file. From there, go to **File | Open**, navigate to the `Leapduino.ino` file on your computer, and double-click on it.

You will now be presented with a screen similar to the one here:



This is the wonderful Arduino IDE—a minimalistic and straightforward text editor and compiler for the Arduino microcontrollers.

On the top left of the IDE, you'll find two circular buttons: the check mark verifies (compiles) your code to make sure it works, and the arrow deploys your code to the Arduino board connected to your computer. On the bottom of the IDE, you'll find the compiler output console (the black box), and on the very bottom right you'll see a line of text telling you which Arduino model is connected to your computer, and on what port (I have an **Arduino Uno on COM4** text in the preceding screenshot). As is typical for many IDEs and text editors, the big white area in the middle is where your code will go.

So, without further ado, let's get started with writing some code! Input all the text shown

here into the Arduino IDE:

```
//Most Arduino boards have an LED pre-wired to pin 13.
```

```
int led = 13;
```

```
//Current LED state. LOW is off and HIGH is on.
```

```
int ledState = LOW;
```

```
//Blink rate in milliseconds.
```

```
long blinkRate = 500;
```

```
//Last time the LED was updated.
```

```
long previousTime = 0;
```

```
//Function: setup
```

```
void setup()
```

```
{
```

```
  //Initialize the built-in LED (assuming the Arduino board has one).  
  pinMode(led, OUTPUT);
```

```
  //Start a serial connection at a baud rate of 38,400.
```

```
  Serial.begin(38400);
```

```
}
```

```
//Function: loop
```

```
void loop()
```

```
{
```

```
  //Get the current system time in milliseconds.
```

```
  unsigned long currentTime = millis();
```

```
  //Check if it's time to toggle the LED on or off.
```

```
  if (currentTime - previousTime >= blinkRate)
```

```
  {
```

```
    previousTime = currentTime;
```

```
    if (ledState == LOW) ledState = HIGH;
```

```
    else ledState = LOW;
```

```
    digitalWrite(led, ledState);
```

```
  }
```

```
  //Check if there is serial data available.
```

```
  if (Serial.available())
```

```
  {
```

```
    //Wait for all data to arrive.
```

```
    delay(20);
```

```
    //Our data.
```

```
    String data = "";
```

```
    //Iterate over all the available data and compound it into a string.
```

```
    while (Serial.available())
```

```
      data += (char) (Serial.read());
```

```
    //Set the blink rate based on our newlyread data.
```

```
    blinkRate = abs(data.toInt() * 2);
```

```

    //A blink rate lower than 30 milliseconds won't really be perceptible
    by a human.
    if (blinkRate < 30) blinkRate = 30;

    //Echo the data.
    Serial.println("Leapduino Client Received:");
    Serial.println("Raw Leap Data: " + data + " | Blink Rate (MS): " +
    blinkRate);
  }
}

```

Now, since this is the only Arduino code example in this book, let's go over the contents.

The first few lines are basic global variables, which we'll be using throughout the program (the comments do a good job of describing them, so we won't go into much detail here).

The first function, `setup`, is an Arduino's equivalent of a constructor; it's called only once, when the Arduino is first turned on. Within the `setup` function, we initialize the built-in LED (most Arduino boards have an LED pre-wired to pin 13) on the board. We then initialize serial communications at a baud rate of 38,400 bits per second—this will allow our board to communicate with the computer later on.

Note

Fun fact

The baud rate (abbreviated as Bd in some diagrams) is the unit for symbol rate or modulation rate in symbols or pulses per second. Simply put, on serial ports, the baud rate controls how many bits a serial port can send per second—the higher the number, the faster a serial port can communicate.

The question is, why don't we set a ridiculously high rate? Well, the higher you go with the baud rate, the more likely data loss will occur—and we all know data loss just isn't good. For many applications, though, a baud rate of 9,600 to 38,400 bits per second is sufficient.

Let's move on to the second function, `loop`. It is the *main* function in any Arduino program, which is repeatedly called while the Arduino is turned on. Due to this functionality, many programs will treat any code within this function as if it were inside a `while (true)` loop.

In `loop`, we start off by getting the current system time (in milliseconds) and then comparing it to our ideal blink rate for the LED. If the time elapsed since our last *blink* exceeds the ideal blink rate, we'll go ahead and toggle the LED on or off accordingly.

We then proceed to check whether any data has been received over the serial port. If it has, we'll proceed to wait for a brief period of time, 20 milliseconds, to make sure that all data have been received. At that point, our code will proceed to read in all the data, parse it for an integer (which will be our new blink rate), and then echo the data back out to the serial port for diagnostics purposes.

As you can see, an Arduino program (or *sketch*, as they are formally known) is quite

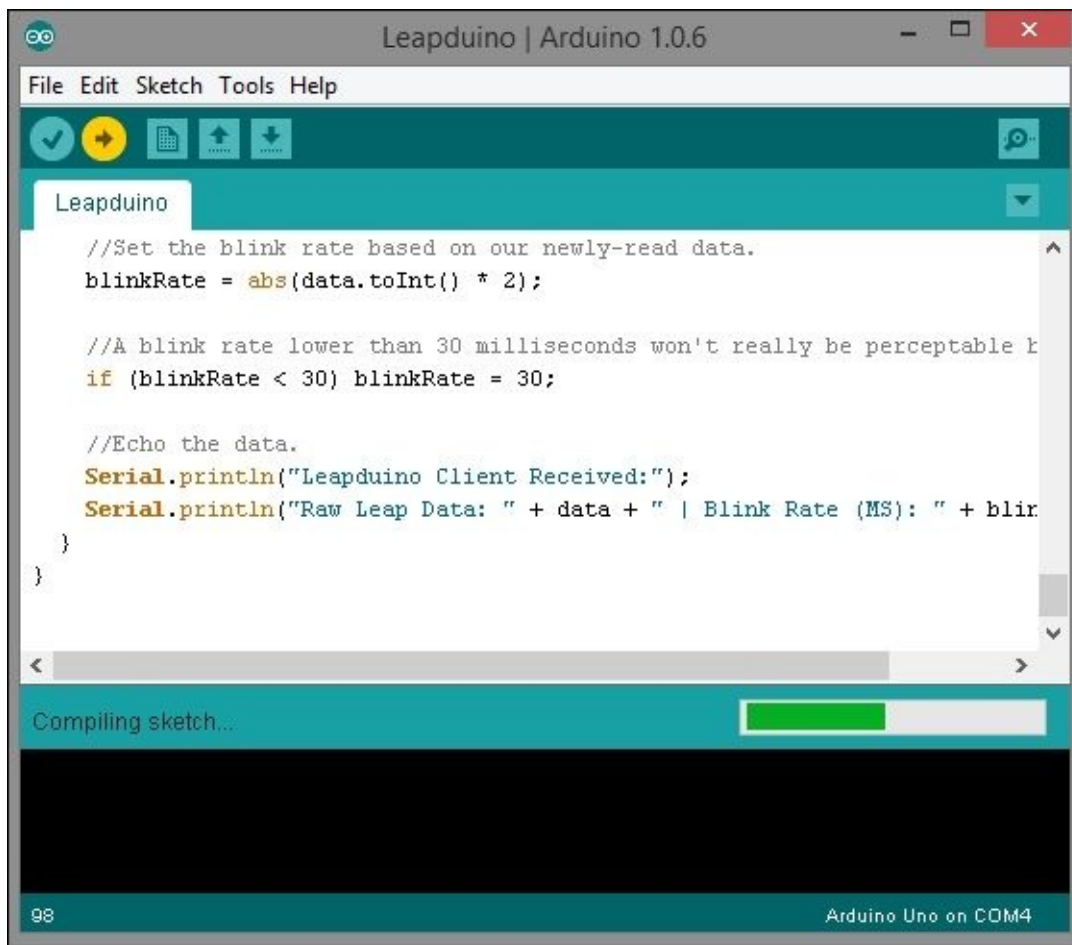
simple. Why don't we test it out?

Deploying and testing the application

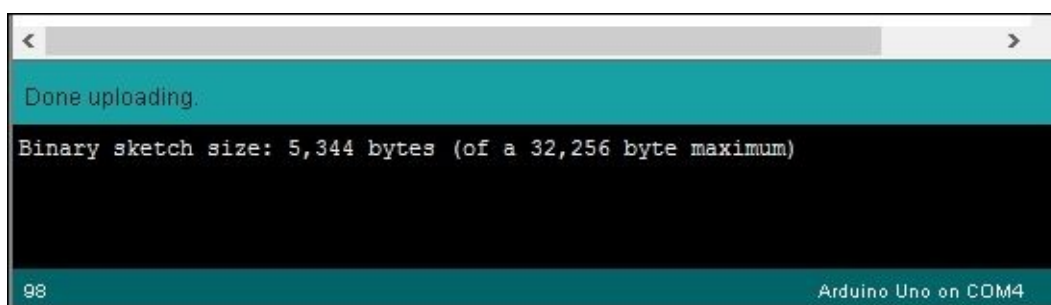
With all the code written, it's time to deploy the Arduino side of things to, well, Arduino.

The first step is to simply open up your `Leapduino.ino` file in the Arduino IDE. Once that's done, navigate to **Tools | Board** and select the appropriate option for your Arduino board. In my case, it's an Arduino Uno. At this point, you'll want to verify that you have an Arduino connected to your computer via a USB cable—after all, we can't deploy to thin air!

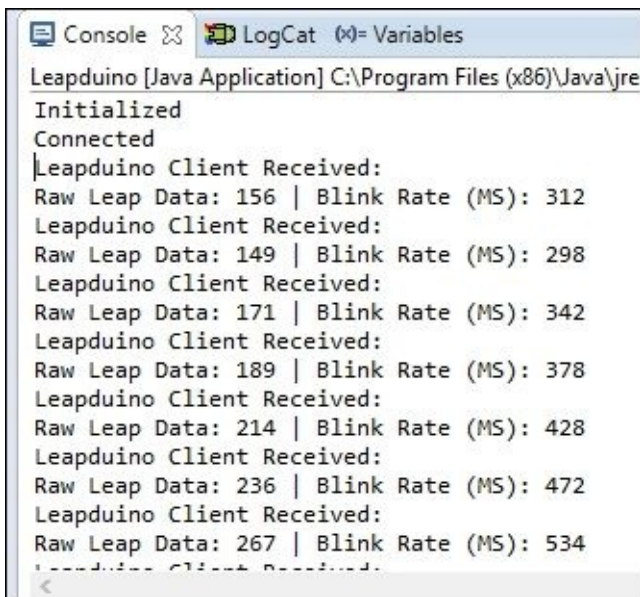
At this point, once everything is ready, simply hit the **Deploy** button in the top-left corner of the IDE, as seen here:



If all goes well, you'll see the following output in the console after 15 or so seconds:

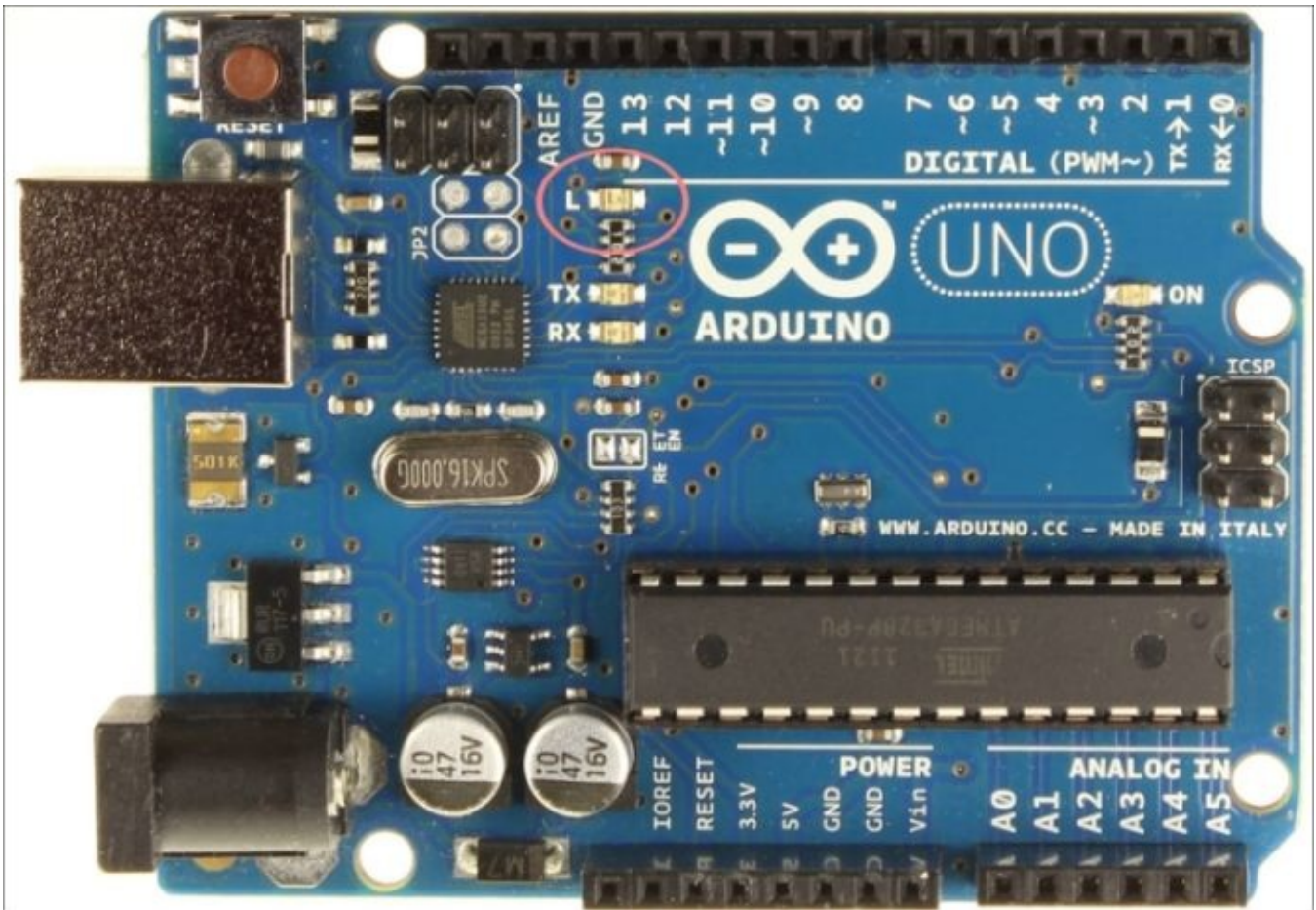


With this, your Arduino is ready to go! How about we test it out? Keeping your Arduino plugged into your computer, go on over to Eclipse and run the project we just made. Once it's running, try moving your hand up and down over your Leap Motion Controller; if all goes well, you'll see the following output from within the console in Eclipse:

A screenshot of the Eclipse IDE's console window. The title bar shows 'Console', 'LogCat', and 'Variables'. The main content area displays the following text:

```
Leapduino [Java Application] C:\Program Files (x86)\Java\jre
Initialized
Connected
Leapduino Client Received:
Raw Leap Data: 156 | Blink Rate (MS): 312
Leapduino Client Received:
Raw Leap Data: 149 | Blink Rate (MS): 298
Leapduino Client Received:
Raw Leap Data: 171 | Blink Rate (MS): 342
Leapduino Client Received:
Raw Leap Data: 189 | Blink Rate (MS): 378
Leapduino Client Received:
Raw Leap Data: 214 | Blink Rate (MS): 428
Leapduino Client Received:
Raw Leap Data: 236 | Blink Rate (MS): 472
Leapduino Client Received:
Raw Leap Data: 267 | Blink Rate (MS): 534
Leapduino Client Received:
```

All of that data is coming directly from Arduino, not your Java program; isn't that cool? Now, take a look at your Arduino while you're doing this; you should notice that the built-in LED (circled in the following image, labelled **L** on the board itself) will begin to blink slower or faster, depending on how close your hand gets to the Leap.



Circled in red is the built-in L LED on an Arduino Uno, wired to pin 13 by default

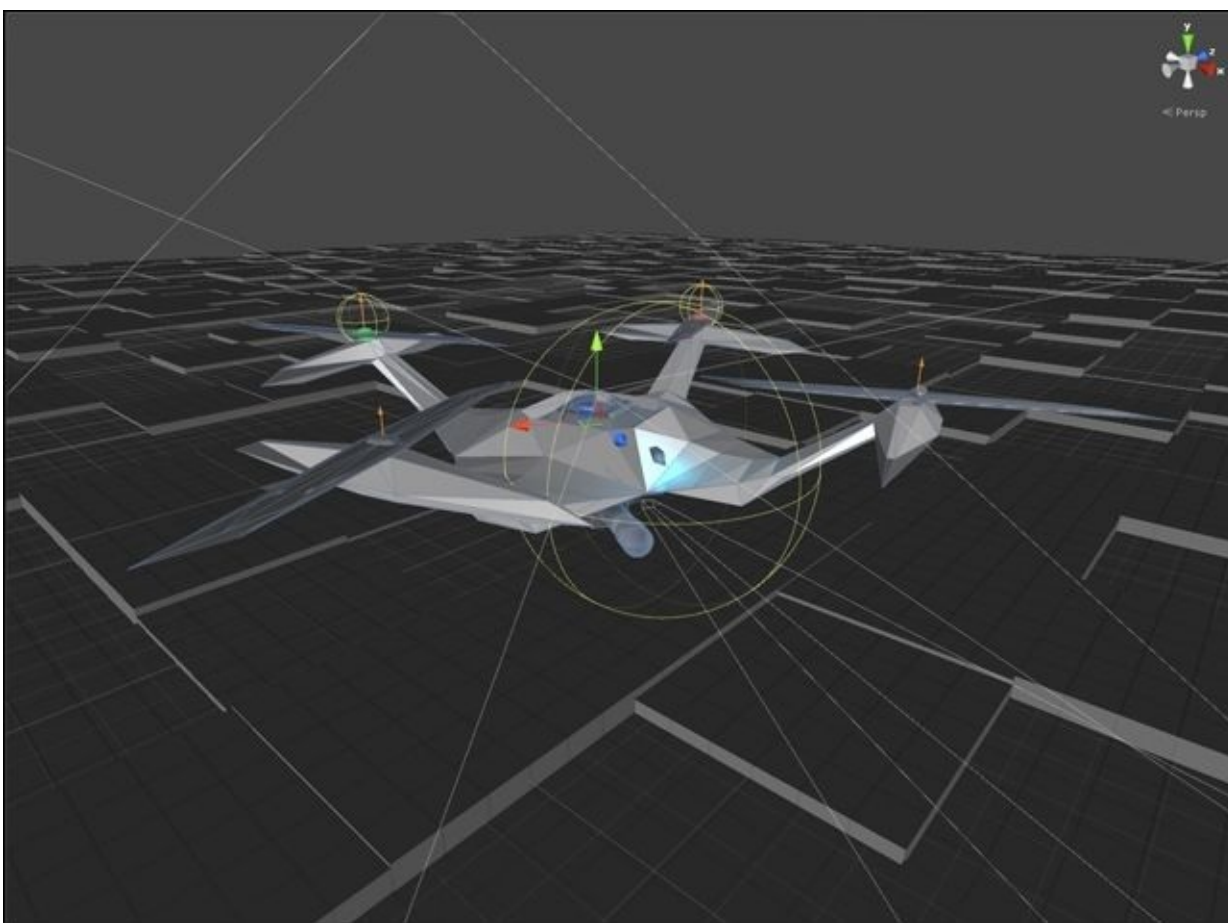
With this, you've created a simple Leap Motion application for use with an Arduino. From here, you could go on to make an Arduino-controlled robotic arm driven by coordinates from the Leap, or maybe an interactive light show. The possibilities are endless, and this is just the (albeit extremely, extremely simple) tip of the iceberg.

Now, finally, how about we take a look at some robots that we can use the Leap with?

Ideas for Leap-driven applications – simulators and robots

Arduinos are all fine and good, but my favorite thing in the entire world is—you guessed it—robots (which more often than not incorporate Arduinos as part of the control system).

Of course, besides the robots themselves, you also have simulators. Not all roboticists make use of these, but they're a great way to test an idea before committing to one (in the case of a very expensive or very hard to implement idea). Simulators have helped me quite a bit when developing for the Leap Motion Controller, as they allowed me to test a Leap Motion Control scheme on a virtual robot before deploying to a real one; this can save lots of time and money. In fact, my first production Leap Motion application was one such simulator: Artemis.



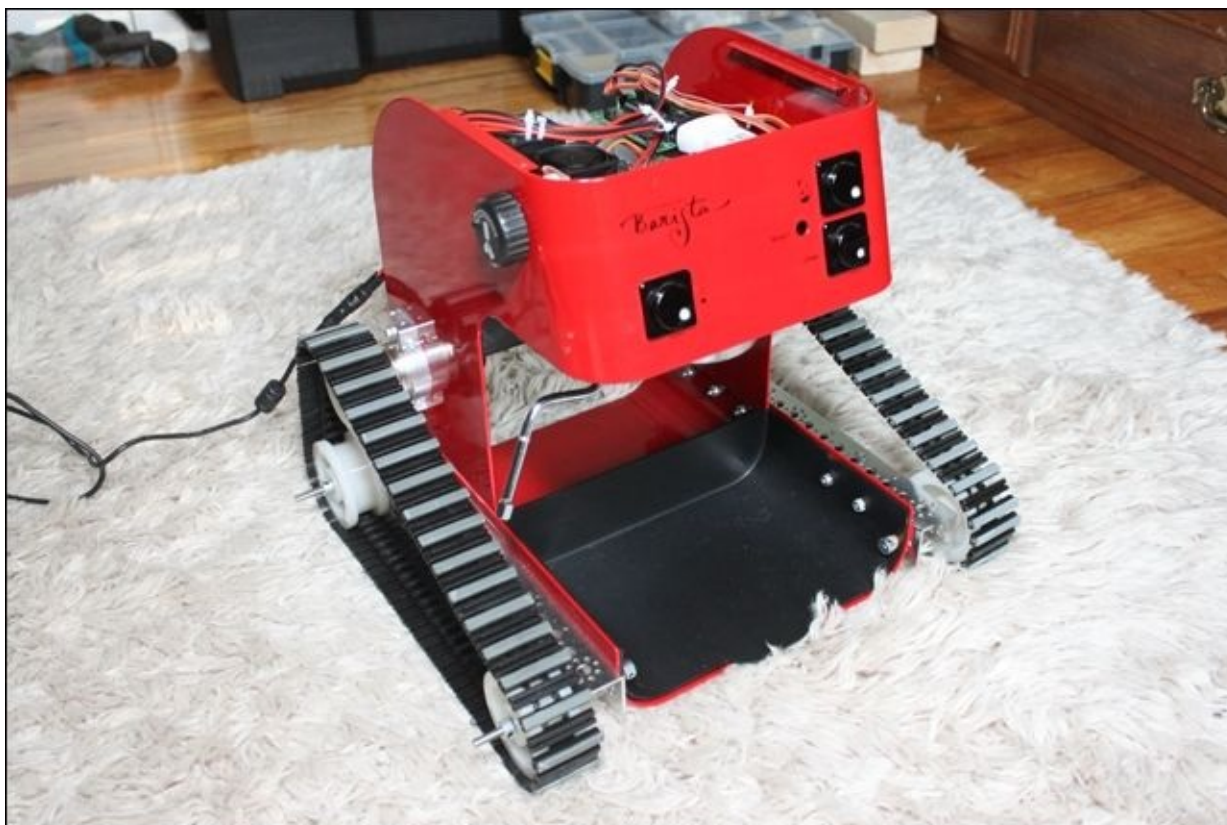
A screenshot from the Artemis Quadrotor Simulator during early development

The application shown in the preceding screenshot, the Artemis Quadrotor Simulator (<http://www.mechakana.com/blog/artemis-a-leap-motion-powered-game-with-quadrotors/>), is one example of a simulator; it was initially designed to allow me to test the Leap Motion Controller with a virtual quadrotor, so as to avoid crashing (and ruining) one of my real ones. I still use it to this day to test new control ideas before deploying them.

Naturally, the nature of a simulator is entirely based on what you're, well, simulating. It

can range anywhere from a simple replica of how a competition robot works, all the way to a full-featured simulation that emulates natural occurrences like wind and thermal updrafts.

There are many methods you can use to write a simulator; I prefer using Unity, as it's very simple and straightforward to develop with. We already covered everything you need to know to get started with writing simulators with the Leap Motion in Unity during [Chapter 5, Creating a 3D Application – a Crash Course in Unity 3D](#); [Chapter 6, Creating a 3D Application – Integrating the Leap Motion Device with a 3D Toolkit](#); and [Chapter 7, Creating a 3D Application – Controlling a Flying Entity](#), so why not hit the Unity documentation and start working on a simulator of your own?



My pet project at the time of writing: a coffeemaker-turned-robot aptly named the Coffeemecha

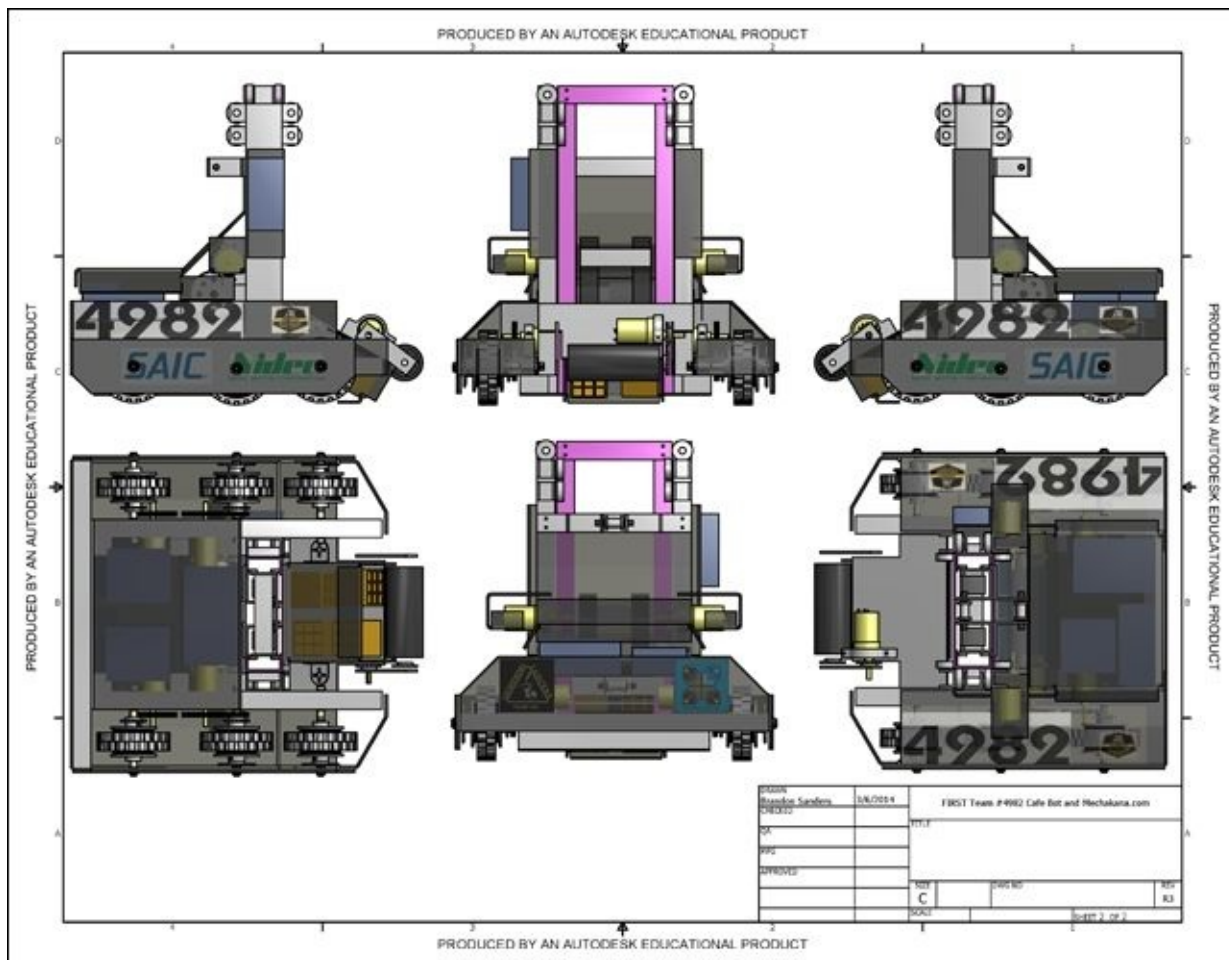
Now, for some actual robots! I've literally been waiting the entire book to talk about these.

One example of a robot that I'm currently working with is my homebrew coffeemaker-turned-robot, the Coffeemecha (shown in the preceding image, and, yes, all the puns are intended). While it features a whole new suite of networked machine learning software, it's also Leap Motion-enabled; nothing is quite as cute as a coffeemaker on treads running around the ground at the command of your hands, right?

As the control systems for these kinds of robots vary wildly, from anywhere to highly integrated systems and Arduino's to Raspberry Pi's with proprietary serial controllers, I will not be covering how to program these—instead, I leave it to your imagination.

FIRST Robotics Competition Robots

My latest FTC robot design and a 2014 World Champion is shown here:



Competition robots are a great place to start with Leap Motion and robots if you were (or are) on a competitive robotics team with a few spare robots lying around.

The competition robots I've worked with are exclusive to the **FIRST Robotics Competition (FRC)** and **FIRST Tech Challenge (FTC)** programs.

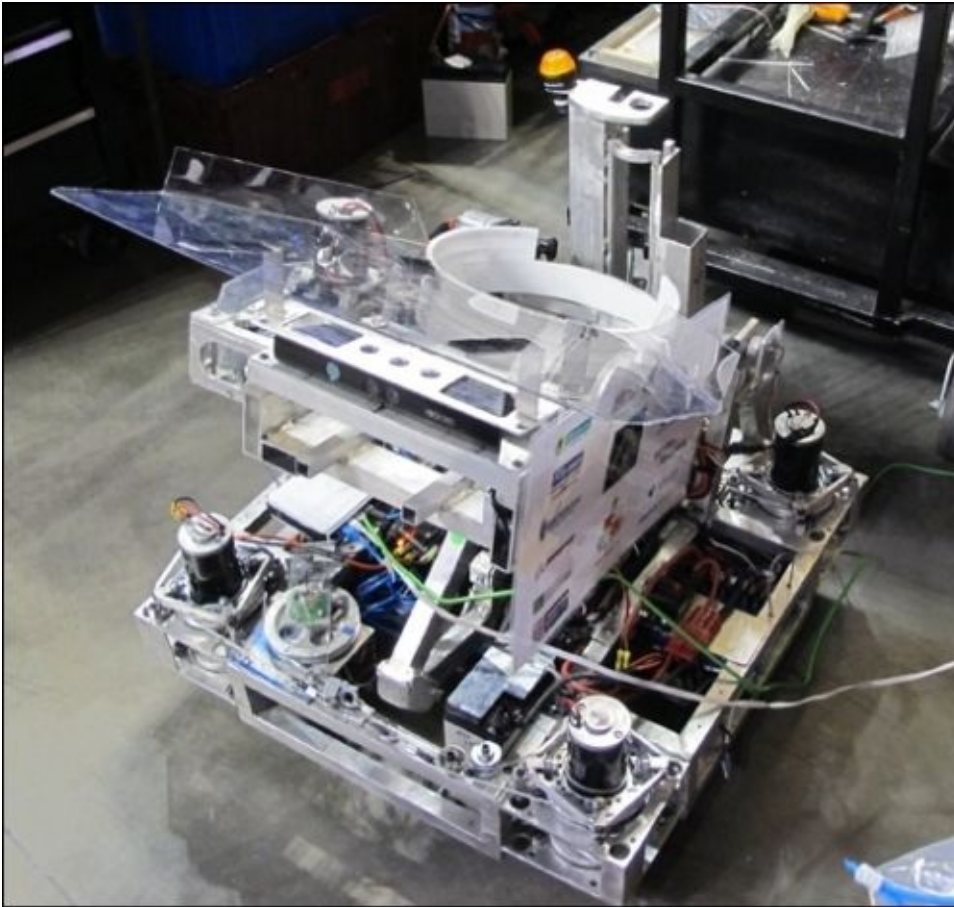
These programs engage students from all around the world in **STEM (Science, Technology, Engineering, and Mathematics)** learning driven by complex robots—I happen to be one such student, having competed (and won) at the World Championship level and received numerous awards and accolades.

In both of these programs, teams will always work with other teams to achieve success while learning and having fun—thanks to FIRST's core values, centered around a term known as Gracious Professionalism®, fierce competition and mutual gain are anything but separate notions.

You can learn more about FIRST at <http://www.usfirst.org>.

The FIRST Robotics Competition

Now, let's go ahead and cover one of the kinds of competition robots I've worked on—FRC robots.



One of the earlier FIRST Robotics Competition robots I worked on

These large-scale robots, ranging anywhere from three feet to ten foot high, are participants in the FIRST Robotics Competition. In this competition, students design, build, and program a robot that must complete a predefined set of challenges that change every year, just like FTC. The challenges for FRC can be anywhere from placing giant inner tubes on pegs to shooting Frisbees into goals using Kinect targeting systems.

FRC teams only have six weeks (starting on the first Saturday in January of every year) to complete their robots. This makes strong time management and organizational skills key to the success of FRC teams—nothing is worse than only having five minutes to program a robot before it has to go into the shipping crate on the last day of week six!



The FIRST Robotics Competition Ultimate Ascent field

To give you an idea of what a game might look like, the FRC robot shown in the preceding image competed in the 2013 FIRST Robotics Competition *Ultimate Ascent*. In this challenge, robots had to retrieve Frisbees from human players on the far ends of the field, or the ground, and then shoot them into small goals raised a good amount off the ground. During the last 30 seconds of the game, robots were awarded bonus points for climbing up their alliance-colored pyramids—the higher they climbed, the more points they got.

There are two identical sides to the field: red and blue. These colors represent two alliances of teams that comprise three randomly selected teams (and their robots). This makes for a very exciting, if not hectic, game!

You can read more about the FIRST Robotics Competition at <http://www.usfirst.org/roboticsprograms/frc>.

Controlling an FRC robot with the Leap Motion Controller

Of course, at this point, you're probably wondering "Great! But what does this have to do with the Leap Motion Controller?" Fear not, for that is exactly what I'm going to go over now: interfacing the Leap Motion Controller with the big FRC robots. It's shockingly easy (I've not attempted to do so yet with the smaller FTC robots, so we won't be discussing those).

Unfortunately, due to Field Management System (which is responsible for connecting the FRC driver stations to the robots during a game) that is in place during an official competition, the Leap Motion Controller is only a viable option during an off-season event or practice tournament. However, don't let this get you down; the time you'll spend working (playing) with your FRC robot during the off-season will be much bigger than the time you spend with it during the actual season (more than eight months, on average).

Note

Fun fact

In FRC, each team supplies both a robot and a custom-made driver station at each event. The driver station is a computer that is typically embedded in a custom control panel containing a dozen or more switches, LEDs, and joysticks. Though you probably guessed this by the name, the driver station is what teams use to control their robots during an event—it's a very important piece of hardware!

The question of how to make an FRC robot integrate with a Leap Motion has been asked many times on the Leap Motion developer forums, and I thought it would be nice to include an answer to it here. I cannot include full example code as the exact implementation varies wildly depending on the robot and control system in question, but I will do my best to explain it in a logical fashion.

Ultimately, you will need to write two applications—a server and a client.

The **server** runs on the computer that has a Leap Motion Controller connected to it, interpreting tracking data and then forwarding it to the client.

The **client** runs directly on the FRC robot's required control system (a National Instruments CompactRIO, as of the 2014 rules). It receives and interprets the data from the server, turning it into whatever actions are required on the robot itself.

Historically, I've achieved a working Leap Motion control scheme using two simple Java applications and network sockets. Usually, I forward the data from the server to the client in the form of a simple colon-delimited string, similar to the format of `x-axis:y-axis:z-axis`. In practice, one of these strings will usually look something like `0:100:0` or `40:90:0`, and so on. The client will then unpack the string and react accordingly.

That's all there is to it! If you have an FRC robot on hand, or know someone who does, give this a try and see how it works for you. There's a certain joy in controlling a

complicated several-thousand-dollar competition robot without ever having to touch any joysticks or buttons.

Making a robot of your own!

Ok. This would literally go far outside the scope of our title. I mean, way outside the scope. You could write an entire series of books on making robots...literally.

Instead, there are some facts that I'd like to point you to if you're truly interested in building robots:

- Did you find the FIRST robots interesting, even though you might not be old (young) enough to participate? No worries—even I'm too old now! One website (well, a forum really) in particular is a great resource for asking questions about the robots and programs—there's also a ton of examples of robots teams have made. This website can be found at <http://www.chiefdelphi.com/>.

You can also take a look at my own team's website at <http://www.cafebot.org/> if you're interested in what I'm up to.

- Want to make your own Coffeemecha? Well, you're going to need a classic Starbucks Barista. Have one of those? Head on over to my first blog post about the Coffeemecha and how I put mine together at <http://www.mechakana.com/blog/introducing-project-coffeemecha/>.

Note

Disclaimer: The preceding blog post is *NOT* a tutorial—it's more of a log of what I did and a good guide to follow if you want to do so!

- Perhaps you want to make a robot, but don't know where to get started? Well...drones sure are popular these days, aren't they?

There's a wonderful and helpful online community of hobbyist, do-it-yourself drone builders who are always ready to help people with their ideas at <http://diydrones.com/>. If you want to get started with robots (and Leap Motion with robotics), DIY Drones could be a great place to start.

So, get out there and make a robot with some groovy Leap Motion integration—make me proud!

As this chapter draws to a close, I hope you took something awesome with you from the book. If not, send an e-mail to me at <brandon@mechakana.com> and tell me what could've been better. If you found the robots fascinating, visit <http://www.mechakana.com> for even more, well, robots.

Summary

In this final chapter, we started by reviewing everything that you have learned and covered up to this point. We then took a look at some other emerging technologies and how they impact the Leap Motion Controller, including Microsoft's Kinect and the Oculus VR technology. This chapter, and the book, was finished off by taking a lengthy look at some things you can do with the Leap Motion Controller and robots.

From here, you can take everything we've learned and looked at and create something entirely new—how about a robot that uses a Kinect to see, but is operated by a Leap Motion Controller connected to an Oculus Rift?

Remember, the only limitation when developing with the Leap Motion is your own imagination (and user fatigue).

Index

A

- API
 - about / [Structure of the Leap Motion Application Programming Interface \(API\)](#)
- application, Leapaint
 - improving / [Improving the application](#)
- Arduino
 - about / [Arduino](#)
 - side of things. writing / [Writing the Arduino side of things](#)
- Arduino IDE
 - URL / [Setting up the environment](#)
- Arduino side of things
 - deploying / [Deploying and testing the application](#)
 - testing / [Deploying and testing the application](#)
- Artemis / [Why would you ever want to use something like the interaction box?](#)
- Artemis quadrotor simulator
 - about / [A case study – the Artemis Quadrotor Simulator](#)
 - URL / [Ideas for Leap-driven applications – simulators and robots](#)
- asynchronous / [TouchableButton – surely, the name is self-explanatory](#)
- Awake function / [A quick summary – the fundamentals of Unity scripts](#)

B

- BaseSingleton
 - about / [BaseSingleton – a custom singleton pattern](#)
- Bone class
 - about / [A new API class – Bones](#)
- Bone object
 - isValid() function / [A new API class – Bones](#)
 - invalid() functions / [A new API class – Bones](#)
 - length() function / [A new API class – Bones](#)
 - prevJoint() function / [A new API class – Bones](#)
 - nextJoint() function / [A new API class – Bones](#)
 - type() function / [A new API class – Bones](#)
- bounds, graphical frontend
 - getting / [Getting our bounds](#)
- button presses
 - detecting / [Rendering buttons and detecting button presses](#)
- buttons
 - rendering / [Rendering buttons and detecting button presses](#)

C

- carpal tunnel syndrome / [The Leap Motion user experience guidelines](#)
- circle gestures / [Detecting gestures](#)
- Colorscheme
 - about / [Colorscheme – a utility class to keep track of colors](#)
- constructor, graphical user interface
 - constructing / [Constructing a constructor](#)
- Controller class / [The Controller class](#)
- core class
 - about / [Core – the main class, if Unity had main classes](#)
- Cursor.png / [Putting it all together](#)

D

- Diagnostic Visualizer
 - about / [The Diagnostic Visualizer](#)
- DIY Drones
 - URL / [Making a robot of your own!](#)

E

- Eclipse
 - URL / [Setting up your IDE](#), [Setting up the project](#)
- Eclipse IDE
 - framework program, creating / [Creating a simple framework program within the Eclipse IDE](#)
- Eclipse Java project
 - creating / [Setting up the project](#)
- environment
 - setting up / [Setting up the environment](#)
- Ergonomics
 - about / [Ergonomics](#)

F

- field of view (FOV) / [The Leap's field of view](#)
- Finger class / [The Finger class](#)
- FIRST
 - URL / [FIRST Robotics Competition Robots](#)
- FIRST Robotics Competition Robots
 - about / [FIRST Robotics Competition Robots](#)
 - working / [The FIRST Robotics Competition](#)
 - URL / [The FIRST Robotics Competition](#)
 - controlling, with Leap Motion Controller / [Controlling an FRC robot with the Leap Motion Controller](#)
- FIRST Tech Challenge (FTC) / [FIRST Robotics Competition Robots](#)
- flying entity
 - about / [Creating the flying entity](#)
 - PlayerArrow component, adding / [Adding the PlayerArrow and RigidBody components](#)
 - RigidBody component, adding / [Adding the PlayerArrow and RigidBody components](#)
 - user input, retrieving with HandController class / [Retrieving user input with the HandController class](#)
 - user input, interpreting with Player class / [Interpreting user input with the Player class](#)
 - testing / [Putting everything together and testing it](#)
 - improving / [Improving the application](#)
- Frame class / [The Frame class](#)
- framework program, Eclipse IDE
 - creating / [Creating a simple framework program within the Eclipse IDE](#)
 - project, setting up / [Setting up the project](#)
 - code, writing / [Let's write some code!](#)
 - code, testing / [Trying it out](#)
- frequently asked questions (FAQ)
 - learnings / [Reliability and safety concerns with the Leap in industrial settings](#)

G

- GameObject
 - script, attaching to / [Attaching a script to a GameObject](#)
- GameObjects
 - about / [GameObjects](#)
- gestures
 - about / [Gestures](#)
 - detecting / [Detecting gestures](#)
 - circle gestures / [Detecting gestures](#)
 - swipe gestures / [Detecting gestures](#)
 - screen tap gestures / [Detecting gestures](#)
 - key tap gestures / [Detecting gestures](#)
 - types / [Detecting gestures](#)
- GitHub
 - URL / [Summary](#), [Arduino](#)
- grabStrength() function / [Pinching and grabbing are now much easier](#)
- graphical frontend
 - creating / [Creating the graphical frontend](#)
 - responsive button, creating / [Making a responsive button – the LeapButton class](#)
 - LeapButton class / [Making a responsive button – the LeapButton class](#)
 - bounds, getting / [Getting our bounds](#)
 - user responding to, visually / [Visually responding to the user](#)
 - Leap data, interpreting / [Interpreting Leap data to render on the graphical frontend](#)
- graphical user interface
 - creating / [Making a graphical user interface](#)

H

- Hand class / [The Hand class](#)
- HandController class
 - used, for retrieving user input / [Retrieving user input with the HandController class](#)
- HandRenderer.cs
 - about / [HandRenderer.cs](#)
- hand rendering
 - about / [Rendering hands](#)
 - LeapListener.cs / [LeapListener.cs](#)
 - HandRenderer.cs / [HandRenderer.cs](#)
 - scene, preparing for / [Preparing the scene for hand rendering](#)
 - testing / [Testing out the Hand Renderer](#)
- hands and fingers
 - about / [Handling hands and fingers](#)
 - Leap's field of view / [The Leap's field of view](#)
 - limitations / [Some \(albeit minor\) limitations to keep in mind](#), [Needing too many hands is a bad thing](#), [Differentiating fingers can be fun!](#)
- head-mounted display (HMD) / [Oculus VR's Oculus Rift](#)

I

- IDE
 - setting up / [Setting up your IDE](#)
- images, graphical user interface
 - saving / [Saving images](#)
- InteractionBox class
 - about / [The InteractionBox class](#)
 - working / [How the interaction box works](#)
 - need for / [Why would you ever want to use something like the interaction box?](#)

J

- JAR
 - URL / [Setting up the environment](#)
- Java
 - NoClassDefFound error, handling / [Handling the NoSuchMethod and NoClassDefFound errors in Java](#)
 - side of things. writing / [Writing the Java side of things](#)
- Java JDK
 - installing / [Installing the Java JDK](#)
- Java side of things
 - writing / [Writing the Java side of things](#)
- Java Simple Serial Connector (JSSC)
 - about / [Setting up the environment](#)

K

- key tap gestures / [Detecting gestures](#)

L

- Leap
 - using / [When to use the Leap \(and more importantly, when not to\)](#)
- Leap-Driven applications
 - ideas / [Ideas for Leap-driven applications – simulators and robots](#)
- Leapaint
 - framework, laying out / [Laying out the framework for Leapaint](#)
 - LeapButton.java / [LeapButton.java](#)
 - Listener.java / [LeapaintListener.java](#)
 - .java / [Leapaint.java](#)
 - testing / [Testing it out](#)
 - improving / [Improving the application](#)
- Leapaint.java
 - about / [Leapaint.java](#)
- LeapaintListener.java
 - about / [LeapaintListener.java](#)
- LeapaintListener class / [Laying out the framework for Leapaint](#)
- LeapButton.java
 - about / [LeapButton.java](#)
- LeapButton class / [Laying out the framework for Leapaint](#)
 - about / [Making a responsive button – the LeapButton class](#)
- Leap connection
 - checking / [Making sure your Leap is connected](#)
 - Diagnostic Visualizer / [The Diagnostic Visualizer](#)
- Leap data
 - interpreting, to render on graphical front end / [Interpreting Leap data to render on the graphical frontend](#)
- LeapListener.cs
 - about / [LeapListener.cs](#)
- Leap Motion
 - URL / [LeapaintListener.java](#)
 - learnings / [What you've learned so far](#)
- Leap Motion API
 - structure / [Structure of the Leap Motion Application Programming Interface \(API\)](#)
- Leap Motion API calls
 - cutting back / [Cutting back on Leap Motion API calls](#)
- Leap Motion Controller
 - limitations / [Some \(albeit minor\) limitations to keep in mind, Needing too many hands is a bad thing, Differentiating fingers can be fun!, Lack of support for custom gestures](#)
 - custom calibration / [Custom calibration of the Leap Motion Controller](#)
 - Microsoft's Kinect / [Microsoft's Kinect](#)

- Oculus VR's Oculus Rift / [Oculus VR's Oculus Rift](#)
- settings / [Reliability and safety concerns with the Leap in industrial settings](#)
- hardware controlling, ideas / [Going beyond – ideas to control hardware and robots with the Leap Motion Controller](#)
- robots controlling, ideas / [Going beyond – ideas to control hardware and robots with the Leap Motion Controller](#)
- used, for controlling FIRST Robotics Competition Robots / [Controlling an FRC robot with the Leap Motion Controller](#)
- Leap Motion device
 - setting up / [Setting up the Leap Motion device](#)
 - URL / [Setting up the Leap Motion device](#)
 - SDK, setting up / [Installing the Leap Motion Developers' SDK](#)
- Leap Motion input
 - receiving, scene set up for / [Setting up the scene to receive Leap Motion input](#)
- Leap Motion Listener system / [The Listener class](#)
- Leap Motion SDK
 - updating / [Keeping the Leap Motion SDK updated](#)
- Leap Motion User Experience
 - guidelines / [The Leap Motion user experience guidelines](#)
- Listener class / [The Listener class](#)

M

- Microsoft's Kinect / [Microsoft's Kinect](#)
- MonoBehaviour class
 - Awake function / [A quick summary – the fundamentals of Unity scripts](#)
 - OnEnable function / [A quick summary – the fundamentals of Unity scripts](#)
 - Start function / [A quick summary – the fundamentals of Unity scripts](#)
 - Update function / [A quick summary – the fundamentals of Unity scripts](#)
 - OnGUI function / [A quick summary – the fundamentals of Unity scripts](#)
 - OnDisable function / [A quick summary – the fundamentals of Unity scripts](#)
 - OnDestroy function / [A quick summary – the fundamentals of Unity scripts](#)
 - URL / [A quick summary – the fundamentals of Unity scripts](#)

N

- NoClassDefFound error
 - in Java, handling / [Handling the NoSuchMethod and NoClassDefFound errors in Java](#)
- normalize / [The InteractionBox class](#)
- NoSuchMethod error
 - in Java, handling / [Handling the NoSuchMethod and NoClassDefFound errors in Java](#)

O

- Oculus VR's Oculus Rift / [Oculus VR's Oculus Rift](#)
- onConnect function / [LeapaintListener.java](#)
- OnDestroy function / [A quick summary – the fundamentals of Unity scripts](#)
- OnDisable function / [A quick summary – the fundamentals of Unity scripts](#)
- onDisconnect function / [LeapaintListener.java](#)
- OnEnable function / [A quick summary – the fundamentals of Unity scripts](#)
- onExit function / [LeapaintListener.java](#)
- onFrame function / [LeapaintListener.java](#)
- OnGUI function / [A quick summary – the fundamentals of Unity scripts](#)
- onInit function / [LeapaintListener.java](#)
- Oracle
 - URL / [Improving the application](#)

P

- packages / [Creating the flying entity](#)
- pinchStrength() function / [Pinching and grabbing are now much easier](#)
- PlayerArrow component
 - adding / [Adding the PlayerArrow and Rigidbody components](#)
- player character / [Creating the flying entity](#)
- Player class
 - used, for interpreting user input / [Interpreting user input with the Player class](#)
- play testing
 - advantages / [Play testing and why you should do it](#)
- project
 - setting up / [Setting up the project](#)
- project, Unity
 - creating / [Creating a project](#)
 - hierarchy window / [Creating a project](#)
 - scene window / [Creating a project](#)
 - project window / [Creating a project](#)
 - inspector window / [Creating a project](#)

R

- responsive button
 - creating / [Making a responsive button – the LeapButton class](#)
- Rigidbodies
 - URL / [Adding the PlayerArrow and Rigidbody components](#)
- Rigidbody component
 - adding / [Adding the PlayerArrow and Rigidbody components](#)
- robots / [Ideas for Leap-driven applications – simulators and robots](#)
 - creating / [Making a robot of your own!](#)

S

- saveImage function / [Saving images](#)
- scene
 - setting up, to receive Leap Motion input / [Setting up the scene to receive Leap Motion input](#)
 - preparing, for hand rendering / [Preparing the scene for hand rendering](#)
- scene, Unity
 - URL / [Scenes](#)
 - setting / [Setting the scene](#)
- screen tap gestures / [Detecting gestures](#)
- script
 - attaching, to GameObject / [Attaching a script to a GameObject](#)
 - framework, laying out / [Laying out a framework of scripts](#)
- scripts
 - about / [Scripts](#)
- SDK, Leap Motion device
 - setting up / [Installing the Leap Motion Developers' SDK](#)
- simulators / [Ideas for Leap-driven applications – simulators and robots](#)
- Skeletal Tracking API
 - about / [Looking forward – the Skeletal Tracking API](#)
 - finger / [Different fingers? Not a problem](#)
 - handedness / [Handedness is no longer an issue](#)
 - isLeft() function / [Handedness is no longer an issue](#)
 - isRight() function / [Handedness is no longer an issue](#)
 - confidence rating / [Having confidence in tracking data](#)
 - pinching / [Pinching and grabbing are now much easier](#)
 - grabStrength() function / [Pinching and grabbing are now much easier](#)
 - pinchStrength() function / [Pinching and grabbing are now much easier](#)
 - Bone class / [A new API class – Bones](#)
- Start function / [A quick summary – the fundamentals of Unity scripts](#)
- STEM (Science, Technology, Engineering, and Mathematics) / [FIRST Robotics Competition Robots](#)
- structure, Leap Motion API
 - about / [Structure of the Leap Motion Application Programming Interface \(API\)](#)
 - Vector class / [The Vector class](#)
 - Finger class / [The Finger class](#)
 - Hand class / [The Hand class](#)
 - Frame class / [The Frame class](#)
 - Controller class / [The Controller class](#)
 - Listener class / [The Listener class](#)
- swipe gestures / [Detecting gestures](#)
- synchronous / [TouchableButton – surely, the name is self-explanatory](#)

T

- TitleMenu
 - about / [TitleMenu – a simple main menu](#)
- tools
 - detecting / [Detecting and using tools](#)
- TouchableButton
 - about / [TouchableButton – surely, the name is self-explanatory](#)
- TouchPointer class
 - about / [TouchPointer – let's draw some cursors on the screen](#)

U

- Unity
 - about / [A brief introduction to Unity](#)
 - jargon / [Common jargon found in Unity](#)
 - documentation, URL / [Improving the application](#)
- Unity, jargon
 - scenes / [Scenes](#)
 - GameObjects / [GameObjects](#)
 - scripts / [Scripts](#)
- Unity 3D
 - installing / [Installing and setting up Unity 3D](#)
 - URL / [Installing and setting up Unity 3D](#)
 - setting up / [Installing and setting up Unity 3D](#)
 - working / [Putting it all together](#)
- Unity 3D application
 - about / [A brief introduction to Unity](#)
- Unity scripts
 - about / [A quick summary – the fundamentals of Unity scripts](#)
- Universal Serial Bus (USB) / [Writing the Java side of things](#)
- up-to-date API documentation, Leap Motion device
 - URL / [Structure of the Leap Motion Application Programming Interface \(API\)](#)
- Update function / [A quick summary – the fundamentals of Unity scripts](#), [Core – the main class, if Unity had main classes](#)
- user experience (UX) / [The Leap Motion user experience guidelines](#)
- user fatigue
 - about / [User fatigue](#)
- user input
 - retrieving, HandController class used / [Retrieving user input with the HandController class](#)
 - interpreting, Player class used / [Interpreting user input with the Player class](#)

V

- Vector class / [The Vector class](#)
- virtual reality (VR) / [Oculus VR's Oculus Rift](#)
- visual feedback
 - providing / [Providing as much visual feedback as possible, That's it – for now!](#)