

Ben Tristem
Mike Geig



In Full Color

Second Edition

Features
Unity 5

Sams **Teach Yourself**

Unity® Game Development

in **24**
Hours

SAMS

Praise for *Sams Teach Yourself Unity Game Development in 24 Hours*, Second Edition

“Rapid prototyping is one of the most valuable skills in the industry, and this book will help you get up and running with enough time left over to finish a weekend game jam. Despite being a long time Unity user, I learned a dozen new time-saving tricks in the first half of this book alone!”

—**Andy Moore**, Captain, Radial Games

“24 hours, 3 games, and a plethora of lessons on not only how to build games in Unity but how to be a game designer, programmer, and developer. *Sams Teach Yourself Unity Game Development in 24 Hours*, 2/e is a great foundation for budding game builders.”

—**Tim J. Harrington**, EdD, Higher Education Games and Social Learning Specialist

“*Sams Teach Yourself Unity Game Development in 24 Hours*, 2/e provides a terrific and thorough introductory look at the Unity development environment, game terminology, and game-making process, with plenty of hands-on examples, exercises and quizzes that will have readers creating their own games in no time!”

—**Dr. Kimberley Voll**, Game Developer/Researcher, ZanyT Games

“This is the book we have been waiting for! Ben and Mike don’t just explain how to use Unity, they explain how to use it properly so you won’t get stuck later. Every Unity developer should carry this around in their back pocket.”

—**Efraim Meulenberg**, Co-Founder, Tornado Twins

“Unity’s fun to play with and fun to learn. It’s become extremely popular as a platform for game studios ranging in size from one to one hundred people. Game engines are only as good as the games they enable; as a developer you need to ship games. That’s where this book will help you. I especially enjoyed the starter 2D and 3D games developed in this book. They gather the material learned in previous chapters and show you how the parts fit together into a working whole. Reading this book will inspire you to create your own experiences and share them with the world.”

—**Jeff Somers**, Developer on Rock Band, Guitar Hero, Phase and Dance Central

“This book will make all of your dreams come true, provided your dreams exclusively revolve around game development in Unity. Plus, I’m British, so it must be true.”

—**Will Goldstone**, Unity Technologies

“*Sams Teach Yourself Unity Game Development in 24 Hours*, 2/e is a comprehensive primer for learning Unity3D akin to eating dessert first—you get to the fun quickly!”

—**Elliott Mitchell**, Co-founder, Vermont Digital Arts/Boston Unity Group

This page intentionally left blank

Ben Tristem
Mike Geig

Sams **Teach Yourself**
Unity® Game Development

Second Edition

in **24**
Hours



800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself Unity® Game Development in 24 Hours, Second Edition

Copyright © 2016 by Pearson Education

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

Unity is a registered trademark of Unity technologies.

Kinect is a trademark of Microsoft®.

PlayStation and PlayStation Move are trademarks of Sony®.

Wii is a trademark of Nintendo®.

ISBN-13: 978-0-672-33751-2

ISBN-10: 0-672-33751-7

Library of Congress Control Number: 2015913726

Printed in the United States of America

First Printing December 2016

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com

For questions about sales outside the U.S., please contact

international@pearsoned.com.

Editor-in-Chief

Mark Taub

Executive Editor

Laura Lewin

Senior Development Editor

Chris Zahn

Managing Editor

Kristy Hart

Project Editor

Andy Beaster

Copy Editor

Cenveo® Publisher Services

Indexer

Cenveo Publisher Services

Proofreader

Cenveo Publisher Services

Technical Editors

Tim Harrington
Jeff Somers

Publishing Coordinator

Olivia Basegio

Interior Designer

Gary Adair

Cover Designer

Mark Shirar

Composition

Cenveo Publisher Services

Contents at a Glance

Preface	xiii
HOUR 1 Introduction to Unity	1
HOUR 2 Game Objects	21
HOUR 3 Models, Materials, and Textures	35
HOUR 4 Terrain	49
HOUR 5 Environments	63
HOUR 6 Lights and Cameras	81
HOUR 7 Game 1: <i>Amazing Racer</i>	103
HOUR 8 Scripting—Part 1	119
HOUR 9 Scripting—Part 2	141
HOUR 10 Collision	161
HOUR 11 Game 2: <i>Chaos Ball</i>	173
HOUR 12 Prefabs	189
HOUR 13 2D Games Tools	201
HOUR 14 User Interfaces	217
HOUR 15 Game 3: <i>Captain Blaster</i>	237
HOUR 16 Particle Systems	257
HOUR 17 Animations	275
HOUR 18 Animators	291
HOUR 19 Game 4: <i>Gauntlet Runner</i>	317
HOUR 20 Audio	339
HOUR 21 Mobile Development	353
HOUR 22 Game Revisions	365
HOUR 23 Polish and Deploy	379
HOUR 24 Wrap Up	393
Index	399

Table of Contents

Preface	xiii
HOUR 1: Introduction to Unity	1
Installing Unity	1
Getting to Know the Unity Editor	4
Navigating the Unity Scene View	17
Summary	19
Q&A	19
Workshop	19
Exercise	20
HOUR 2: Game Objects	21
Dimensions and Coordinate Systems	21
Game Objects	25
Transforms	26
Summary	33
Q&A	33
Workshop	33
Exercise	34
HOUR 3: Models, Materials, and Textures	35
The Basics of Models	35
Textures, Shaders, and Materials	41
Summary	46
Q&A	46
Workshop	47
Exercise	47
HOUR 4: Terrain	49
Terrain Generation	49
Terrain Textures	57
Summary	61
Q&A	61

Workshop	61
Exercise	62
HOUR 5: Environments	63
Generating Trees and Grass	63
Environment Effects	71
Character Controllers	75
Summary	78
Q&A	78
Workshop	79
Exercise	79
HOUR 6: Lights and Cameras	81
Lights	81
Cameras	91
Layers	95
Summary	100
Q&A	100
Workshop	100
Exercise	101
HOUR 7: Game 1: <i>Amazing Racer</i>	103
Design	103
Creating the Game World	106
Gamification	108
Playtesting	114
Summary	116
Q&A	116
Workshop	116
Exercise	117
HOUR 8: Scripting—Part 1	119
Scripts	120
Variables	128
Operators	130
Conditionals	133

Iteration	136
Summary	137
Q&A	137
Workshop	138
Exercise	138
HOUR 9: Scripting—Part 2	141
Methods	141
Input	146
Accessing Local Components	151
Accessing Other Objects	153
Summary	158
Q&A	158
Workshop	158
Exercise	159
HOUR 10: Collision	161
Rigidbody	161
Collision	163
Triggers	167
Raycasting	169
Summary	171
Q&A	171
Workshop	172
Exercise	172
HOUR 11: Game 2: Chaos Ball	173
Design	173
The Arena	175
Game Entities	179
The Control Objects	183
Improving the Game	187
Summary	187
Q&A	187
Workshop	188
Exercise	188

HOUR 12: Prefabs	189
Prefab Basics	189
Working with Prefabs	192
Summary	198
Q&A	198
Workshop	198
Exercise	199
HOUR 13: 2D Games Tools	201
The Basics of 2D Games	201
Orthographic Cameras	204
Adding Sprites	205
Draw Order	209
2D Physics	212
Summary	214
Q&A	215
Workshop	215
Exercise	215
HOUR 14: User Interfaces	217
Basic UI Principles	217
The Canvas	218
UI Elements	223
Canvas Render Modes	230
Summary	232
Q&A	233
Workshop	233
Exercise	233
HOUR 15: Game 3: Captain Blaster	237
Design	237
The World	238
Controls	247
Improvements	255
Summary	255
Q&A	255

Workshop	256
Exercise	256
HOUR 16: Particle Systems	257
Particle Systems	257
Particle System Modules	259
The Curve Editor	270
Summary	273
Q&A	273
Workshop	273
Exercise	273
HOUR 17: Animations	275
Animation Basics	275
Animation Types	277
Animation Tools	281
Summary	288
Q&A	288
Workshop	289
Exercise	289
HOUR 18: Animators	291
Animator Basics	291
Configuring Your Assets	296
Creating an Animator	305
Scripting Animators	314
Summary	315
Q&A	315
Workshop	316
Exercise	316
HOUR 19: Game 4: Gauntlet Runner	317
Design	317
The World	318
The Entities	321

The Controls	329
Room for Improvement	336
Summary	336
Q&A	336
Workshop	336
Exercise	337
HOUR 20: Audio	339
Audio Basics	339
Audio Sources	341
Audio Scripting	346
Summary	349
Q&A	349
Workshop	349
Exercise	350
HOUR 21: Mobile Development	353
Preparing for Mobile	353
Accelerometers	357
Summary	361
Q&A	362
Workshop	362
Exercise	362
HOUR 22: Game Revisions	365
Cross-Platform Input	365
Amazing Racer	368
Chaos Ball	372
Captain Blaster	374
Gauntlet Runner	375
Summary	376
Q&A	376
Workshop	376
Exercise	377

HOUR 23: Polish and Deploy	379
Managing Scenes	379
Persisting Data and Objects	381
Unity Player Settings	384
Building Your Game	387
Summary	391
Q&A	391
Workshop	391
Exercise	392
HOUR 24: Wrap Up	393
Accomplishments	393
Where to Go from Here	395
Resources Available to You	396
Summary	397
Q&A	397
Workshop	397
Exercise	398
Index	399

Preface

The Unity game engine is an incredibly powerful and popular choice for professional and amateur game developers alike. This book has been written to get readers up to speed and working in Unity as fast as possible (about 24 hours to be exact) while covering fundamental principles of game development. Unlike other books that only cover specific topics or spend the entire time teaching a single game, this book covers a large array of topics while still managing to contain four games! Talk about a bargain. By the time you are done reading this book, you won't have just theoretical knowledge of the Unity game engine. You will have a portfolio of games to go with it.

Who Should Read This Book

This book is for anyone looking to learn how to use the Unity game engine. Whether you are a student or a development expert, there is something to learn in these pages. It is not assumed that you have any prior game development knowledge or experience, so don't worry if this is your first foray into the art of making games. Take your time and have fun. You will be learning in no time.

How This Book Is Organized and What It Covers

Following the Sam's Teach Yourself approach, this book is organized into 24 chapters that should take approximately 1 hour each to work through. The chapters include the following:

- ▶ Hour 1, "Introduction to Unity"—This hour gets you up and running with the various components of the Unity game engine.
- ▶ Hour 2, "Game Objects"—Hour 2 teaches you how to use the fundamental building blocks of the Unity game engine—the game object. You also learn about coordinate systems and transformations.
- ▶ Hour 3, "Models, Materials, and Textures"—In this hour, you learn to work with Unity's graphical asset pipeline as you apply shaders and textures to materials. You also learn how to apply those materials to a variety of 3D objects.

- ▶ Hour 4, “Terrain”—In Hour 4, you learn to sculpt game worlds using Unity’s terrain system. Don’t be afraid to get your hands dirty as you dig around and create unique and stunning landscapes.
- ▶ Hour 5, “Environments”—In this hour, you learn to apply environmental effects to your sculpted terrain. Time to plant some trees!
- ▶ Hour 6, “Lights and Cameras”—Hour 6 covers lights and cameras in great detail.
- ▶ Hour 7, “Game 1—Amazing Racer”: Time for your first game. In Hour 7, you create *Amazing Racer*, which requires you to take all the knowledge you have gained so far and apply it.
- ▶ Hour 8, ‘Scripting Part 1”—In Hour 8, you begin your foray into scripting with Unity. If you’ve never programmed before, don’t worry. We go slowly as you learn the basics.
- ▶ Hour 9, “Scripting Part 2”—In this hour, you expand on what you learned in Hour 8. This time, you focus on more advanced topics.
- ▶ Hour 10, “Collision”—Hour 10 walks you through the various collision interactions that are common in modern video games. You learn about physical as well as trigger collisions. You also learn to create physical materials to add some variety to your objects.
- ▶ Hour 11, “Game 2—Chaos Ball”—Time for another game! In this hour, you create *Chaos Ball*. This title certainly lives up to its name as you implement various collisions, physical materials, and goals. Prepare to mix strategy with twitch reaction.
- ▶ Hour 12, “Prefabs”—Prefabs are a great way to create repeatable game objects. In Hour 12, you learn to create and modify prefabs. You also learn to build them in scripts.
- ▶ Hour 13, “2D Game Tools”—In Hour 13, you learn about Unity’s powerful tools for creating 2D games, including how to work with sprites and Box2D physics.
- ▶ Hour 14, “User Interfaces”—In this hour, you learn how to use Unity’s powerful User Interface system, and how to create a menu for your game.
- ▶ Hour 15, “Game 3—Captain Blaster”—Game number 3! In this hour, you make *Captain Blaster*, a retro-style spaceship shooting game.
- ▶ Hour 16, “Particle Systems”—Time to learn about particle effects. In this chapter, you experiment with Unity’s particle system to create cool effects, and apply them to your projects.
- ▶ Hour 17, “Animations”—In Hour 17, you get to learn about animations and Unity’s animation system. You experiment 2D and 3D animation, and some powerful animation tools.

- ▶ Hour 18, “Animators”—Hour 18 is all about Unity’s Mecanim animation system. You learn how to use the powerful state machine, and how to blend animations.
- ▶ Hour 19, “Game 4—Gauntlet Runner”—Lucky game number 4 is called *Gauntlet Runner*. This game explores a new way to scroll backgrounds and how to implement animator controllers to build complex blended animations.
- ▶ Hour 20, “Audio”—Hour 20 has you adding important ambient effects via audio. You learn about 2D and 3D audio and their different properties.
- ▶ Hour 21, “Mobile Development”—In this hour, you learn how to build games for mobile devices. You also learn to utilize a mobile device’s built-in accelerometer and multitouch display.
- ▶ Hour 22, “Game Revisions”—It’s time to go back and revisit the four games you have made. This time you modify them to work on a mobile device. You get to see which control schemes translate well to mobile and which don’t.
- ▶ Hour 23, “Polish and Deploy”—Time to learn how to add multiple scenes and persist data between scenes. You also learn about the deployment settings and playing your games.
- ▶ Hour 24, “Wrap Up”—Here, you look back and summarize the journey you went on to learn Unity. This hour provides useful information about what you have done and where to go next.

Thank you for reading my preface! We hope you enjoy this book and learn much from it. Good luck on your journey with the Unity game engine!

Companion Files

Bonus files include full source code listings from every chapter with author comments, all third party art assets (textures, fonts, models), and all third party sound assets.

To gain access to the companion files:

- 1.** Register your product at informit.com/register.
- 2.** Log in or create an account.
- 3.** Enter the product ISBN: 9780672337512, click submit and answer any challenge questions.

Once the process is complete, you can find any available bonus content under “Registered Products.”

About the Authors

Ben Tristem is an internet entrepreneur, focusing on teaching technical subjects to beginners. Ben has been passionate about using computers since the days of the ZX81, and is now a world-class technology trainer. At the time of writing, Ben has over 60,000 students and more than 1,200 5-star reviews on his online courses. In previous lives, Ben has been an RAF pilot, financial trader, stunt man, helicopter pilot, franchise creator, and more. Now that he has two kids, Toby and Lucy, he has settled down to focus on what he loves—teaching.

Mike Geig is both an experienced teacher and game developer, with a foot firmly in both camps. Mike is a Trainer for Unity Technologies where he develops and delivers recorded, live, and onsite learning content. He enjoys loitering and accordions. His Pearson video series, *Game Development Essentials with Unity 4 LiveLessons*, is a key title on Unity and rumor has it that people really enjoyed the first edition of *Sams Teach Yourself Unity Game Development in 24 Hours*. Mike was once set on fire and has over a million “likes” on Facebook.

Dedication

From Ben:

To Lizzie: For being an amazing wife, enabling me to thrive.

From Mike:

To Dad: Everything worth learning, I learned from you.

Acknowledgments

From Ben:

I've had so much support in writing this book, thank you.

Firstly to Mike for writing the first edition of the book. Having this to work from was an amazing starting point for this second edition. You have been fantastic to work with, and I'm grateful for your time.

Thanks to Laura, our editor, for making it easy for me to write my first book. Thank you also for keeping us all on track so that it got written on time.

Thanks to my beautiful wife, Lizzie, and to my kids, Lucy and Toby, for your patience as I worked late to get the book finished. I'm very grateful for your understanding.

Last but not least to my Mum, without her I probably wouldn't be writing this!

From Mike:

A big "thank you" goes out to everyone who helped me write this book.

First and foremost, thank you Kara for keeping me on track. I don't know what we'll be talking about when this book comes out, but whatever it is, you are probably right. Love ya babe.

Link and Luke: We should take it easy on mommy for a little while. I think she's about to crack.

Thanks to my parents. As I am now a parent myself, I recognize how hard it was for you not to strangle or stab me. Thanks for not strangling or stabbing me.

Thanks to Angelina Jolie. Due to your role in the spectacular movie Hackers (1995), I decided to learn how to use a computer. You underestimate the impact you had on 10-year-olds at the time. You're elite!

To the inventor of beef jerky: History may have forgotten your name, but definitely not your product. I love that stuff. Thanks!

Thank you to our technical editors: Tim and Jeff. Your corrections and insights played a vital role in making this a better product.

Thank you Laura for convincing me to write this book. Also thank you for buying me lunch at GDC. I feel that lunch, the best of all three meals, specifically enabled me to finish this.

Finally, a “thank you” is in order for Unity Technologies. If you never made the Unity game engine, this book would be very weird and confusing.

We Want to Hear from You

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write directly to let us know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that we cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail we receive, we might not be able to reply to every message.

When you write, please be sure to include this book's title and author, as well as your name and contact information.

Email: feedback@samspublishing.com

Mail:
Sams Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at **www.informit.com/register** for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

HOUR 1

Introduction to Unity

What You'll Learn in This Hour:

- ▶ How to install Unity
- ▶ How to create a new project or open an existing project
- ▶ How to use the Unity editor
- ▶ How to navigate inside the Unity Scene view

This hour focuses on getting you ready to rock and roll in the Unity environment. We start by looking at the different Unity licenses, choosing one, and then installing it. Once that is installed, you learn how to create new projects as well as open existing ones. You open the powerful Unity editor, and we examine its various components. Finally, you learn to navigate a scene using mouse controls and keyboard commands. This chapter is meant to be hands-on, so download Unity while reading and follow along.

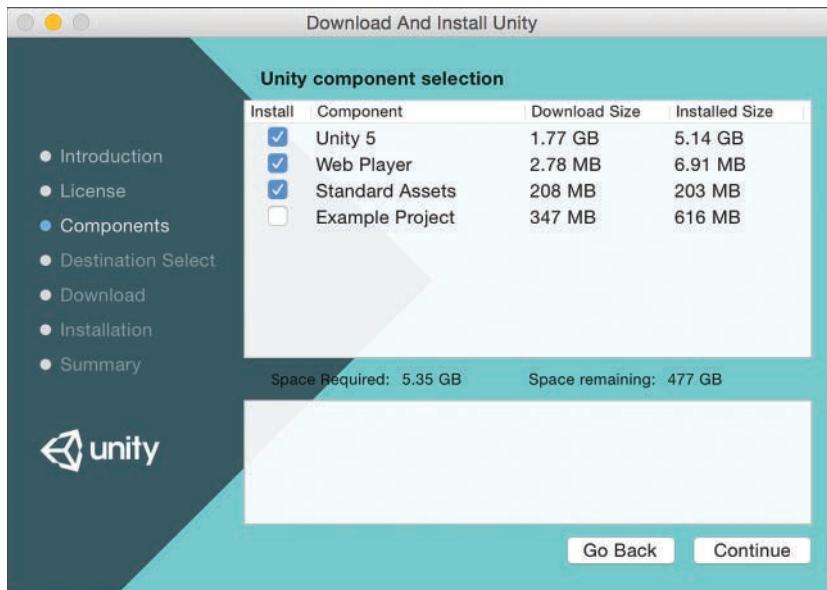
Installing Unity

To begin using Unity, you first need to download and install it. Software installation is a pretty simple and straightforward process these days, and Unity is no exception. Before we can install anything, though, we need to look at the two available Unity licenses: Unity Personal and Unity Professional. Unity Personal is free and more than sufficient to complete all the examples and projects in this book. In fact, Unity Personal contains everything you need to make games commercially, up to an annual revenue of \$100,000! If you're lucky enough to start earning more than this, or you want access to Unity Pro's advanced features (mainly aimed at teams), then you can always upgrade in the future.

Downloading and Installing Unity

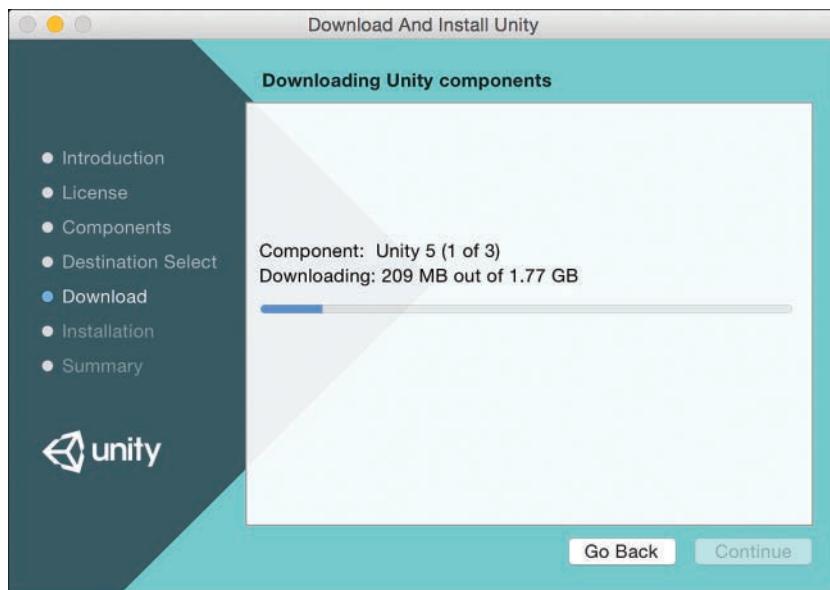
For the purposes of this chapter, we will assume you are sticking with the Unity Personal license. If you went with the Professional version, the process will be very similar, only deviating when it comes to time to choose the license. When you are ready to begin downloading and installing Unity, follow these steps:

1. Download the Unity installer from the Unity download page at <http://unity3d.com/get-unity/download>.
2. Run the installer and follow the prompts as you would with any other piece of software.
3. When prompted, be sure to leave the *Unity 5*, *Web Player*, and *Standard Assets* check boxes checked (see Figure 1.1). It is OK to install the Example Project if you have space; it won't affect your experience of the book.

**FIGURE 1.1**

Prompt to choose the installed components.

4. Choose an install location for Unity. It is recommended that you leave the default unless you know what you are doing.
5. Unity 5 will take some time to download, during which time you'll see a download screen (see Figure 1.2).

**FIGURE 1.2**

Be patient while Unity 5 downloads.

6. If you already have a Unity account, you may be asked to login with it. If you don't yet have a Unity account, follow the instructions to create one. You will need access to your email to verify your address.
7. That's it! Unity installation is now complete.

NOTE**Supported Operating Systems and Hardware**

To use Unity, you must be using a Windows PC or a Macintosh computer. Although it is possible to build your projects to run on a Linux machine, the Unity editor itself will not. Your computer must also meet the minimum requirements outlined here (taken from the Unity website at the time of writing):

- ▶ Windows: XP SP2 or later. Mac OS X: Intel CPU and Snow Leopard 10.8 or later. Note that Unity was not tested on server versions of Windows and OS X.
- ▶ Graphics card with DirectX 9 (Shader Model 2.0) capabilities. Any card made since 2004 should work.
- ▶ Using occlusion culling requires a GPU with occlusion query support (some Intel GPUs do not support that).

Note that these are **minimum** requirements.

CAUTION**Internet Links**

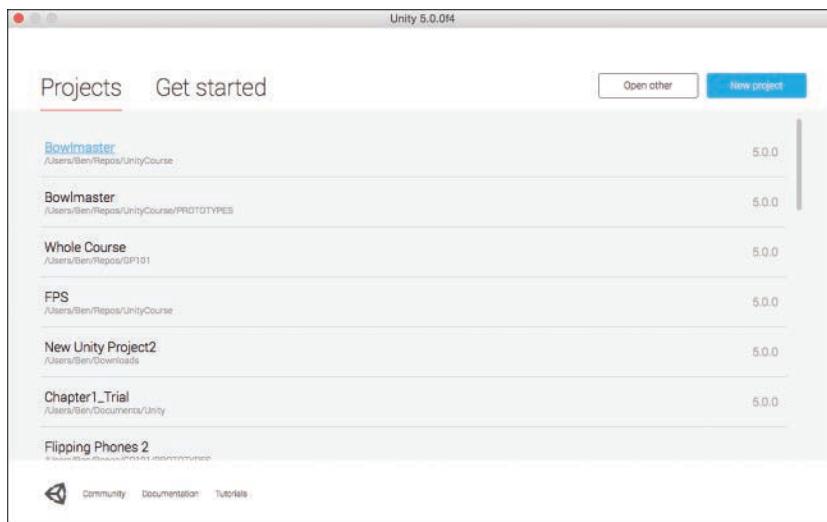
All Internet links are current as of the time of this writing. Web locations do change sometimes, though. If the material you are looking for is no longer provided at the links we give you, a good Internet search should turn up what you are looking for.

Getting to Know the Unity Editor

Now that you have Unity installed, you can begin exploring the Unity editor. The Unity editor is the visual component that enables you to build your games in a “what you see is what you get” fashion. Because most interaction we have is actually with the editor, we often just refer to it as Unity. The next portion of this chapter examines all the different elements of the Unity editor and how they fit together to make games.

The Project Dialog

When opening Unity for the first time, you will see the Project dialog (see Figure 1.3). This window is what we use to open recent projects, browse for projects that have already been created, or start new projects.

**FIGURE 1.3**

The Project dialog (Mac version shown, the Windows version is similar).

If you have created a project in Unity already, whenever you open Unity, it will go directly into that project. To get back to the Project dialog, you go (from inside Unity) to **File > New Project** to get to the Create New Project dialog, or you go to **File > Open Project** to get to the Open Project dialog.

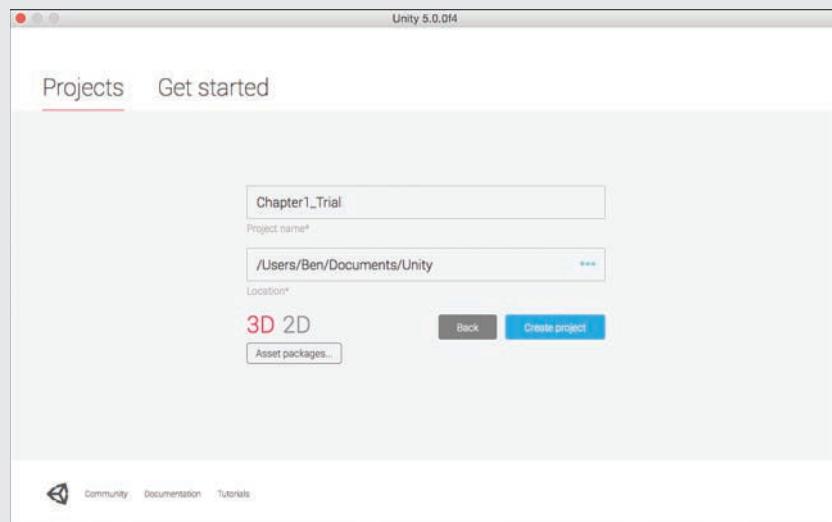
TIP**Opening the Project Dialog**

When you run Unity, the Project dialog will show each time. If you want last project to open automatically instead, you can set this in **Edit > Preferences (Unity > Preferences on a Mac)** and check the box **Load Previous Project on Startup**.

TRY IT YOURSELF ▼**Creating Our First Project**

Let's go ahead and create a project now. You want to pay special attention to where you save the project so that you can find it easily later if necessary. Figure 1.4 shows you what the dialog window should look like before creating the project:

1. Open the New Project dialog.
2. Select a location for your project. We recommend you create a folder called **Unity** to keep all your book projects together. If you are unsure where to put your project, you can leave the default location.
3. Name your project **Chapter 1_Trial**. Unity will create a folder with the same name as the project, in the Location specified.
4. Leave **3D** selected, and ignore the **Asset Packages . . .** button for now.
5. Click **Create Project**.

**FIGURE 1.4**

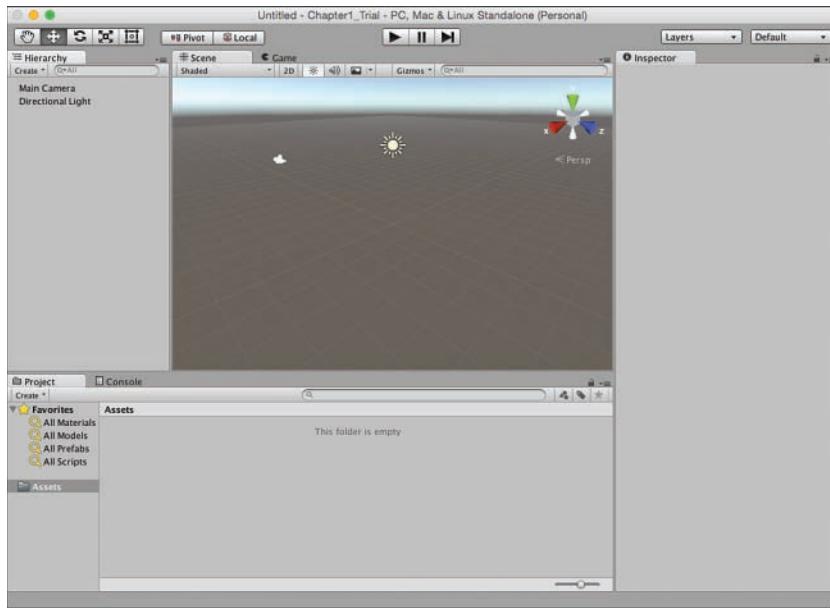
The settings used for our first project.

CAUTION**Projects and Packages**

At first, you might be tempted to select a bunch of “Asset packages” in the Create New Project dialog. We want to caution you against frivolously adding packages to your project, however, because unneeded items can add size and lag. Unused packages just take up space and provide no real benefit. With that in mind, it is better to wait until you actually need a package to import it. Even then, only import the parts of the package that you intend to use.

The Unity Interface

So far, we have installed Unity and looked at the Project dialog. Now it is time to dig in and start playing around. When you open a new Unity project for the first time, you will see a collection of gray windows (called **views**), and everything will be rather empty (see Figure 1.5). Never fear, we will quickly get this place hopping. In the following sections, we look at each of the unique views one by one. First, though, we want to talk about the layout as a whole.

**FIGURE 1.5**

The Unity interface.

For starters, Unity allows the user to determine exactly how they want to work. This means that any of the views can be moved, docked, duplicated, or changed. For instance, if you click the word **Hierarchy** (on the left) to select the Hierarchy view and drag it over to the Inspector (on the right), you can tab the two views together. You can also place your cursor on any line

between views and resize the windows. In fact, why don't you take a moment to play around and move things so that they are to your liking. If you end up with a layout that you don't much care for, never fear. You can quickly and easily switch back to the built-in default view by going to **Window > Layouts > Default Layout**. While we are on the topic of built-in layouts, go ahead and try out a few of the other layouts (we're a fan of the Wide layout). If you create a custom layout you like, you can always save it by going to **Window > Layouts > Save Layout**. Now if you accidentally change your layout, you can always get it back.

NOTE

Finding the Right Layout

No two people are alike, and likewise, no two ideal layouts are alike. A good layout will help you work on your projects and make things much easier for you. Be sure to take the time to fiddle around with the layout to find the one that works best for you. You will be working a lot with Unity. It pays to set your environment up in a way that is comfortable.

If you would like to duplicate a view, it is a fairly straightforward process as well. You can simply right-click any view tab (the **tab** is the part sticking up with the views name on it), hover the mouse cursor over **Add Tab**, and a list of views will pop up for you to choose from (see Figure 1.6). You may wonder why you would want to duplicate a view. It is possible that in your

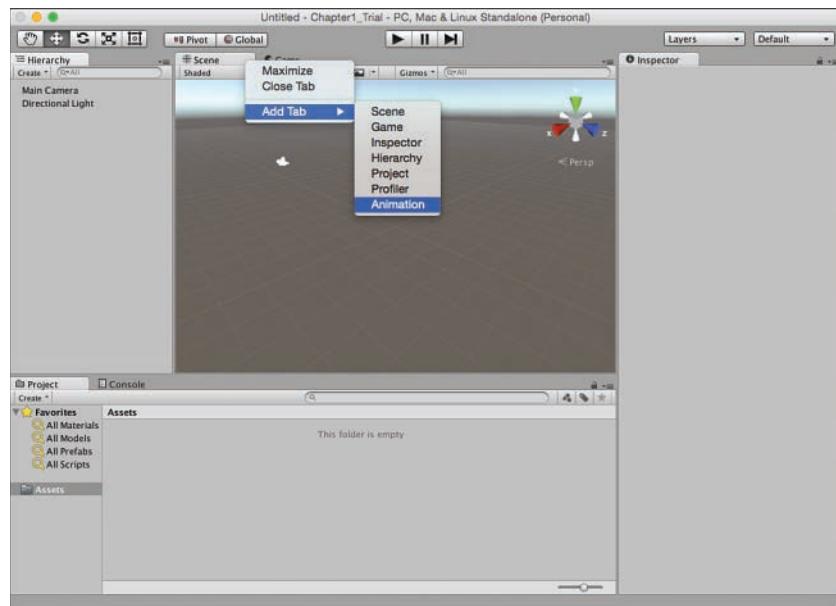


FIGURE 1.6
Adding a new tab.

view-moving frenzy, you accidentally closed the view. Re-adding the tab will give it back to you. Also, consider the capability to create multiple Scene views. Each Scene view could align with a specific element or axis within your project. If you want to see this in action, check out the four Split built-in layout by going to **Window > Layouts > 4 Split**. (If you created a layout that you like, be sure to save it first.)

Now, without further ado, let's look at the specific views themselves.

The Project View

Everything that has been created for a project (files, scripts, textures, models, and so on) can be found in the Project view (see Figure 1.7). This is the window into which all the assets and organization of our project go. When you create a new project, you will notice a single folder item called Assets. If you go to the folder on your hard drive where you save the project, you will also find an Assets folder. This is because Unity mirrors the Project view with the folders on the hard drive. If you create a file or folder in Unity, the corresponding one appears in the explorer (and vice versa). You can move items in the Project view simply by dragging and dropping. This enables you to place items inside folders or reorganize your project on the fly.

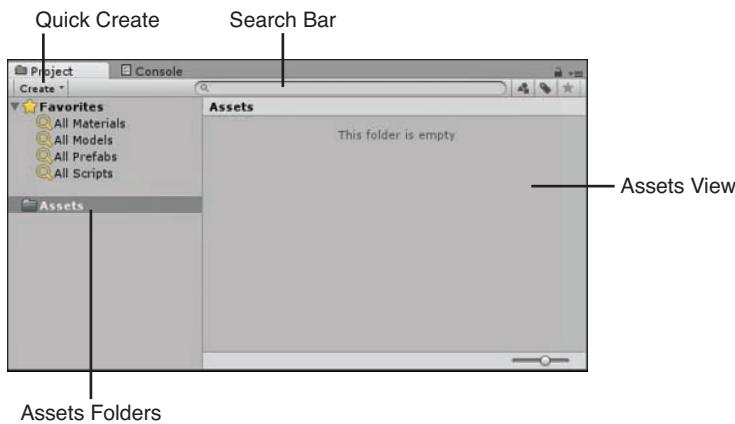


FIGURE 1.7
The Project view.

NOTE

Assets and Objects

An **asset** is any item that exists as a file in your assets folder. All textures, meshes, sound files, scripts, and so on are considered assets. In contrast, if you create a game object, but it doesn't create a corresponding file, it is not an asset.

CAUTION**Moving Assets**

Unity maintains links between the various assets associated with projects. As a result, moving or deleting items outside of Unity could cause potential problems. As a general rule, it is a good idea to do all of your asset management inside Unity.

Whenever you click a folder in the Project view, the contents of the folder will be displayed under the Assets section on the right. As you can see in Figure 1.7, the Assets folder is currently empty, and therefore nothing is appearing on the right. If you would like to create assets, you can do so easily by clicking the Create drop-down menu. This menu enables you to add all manner of assets and folders to your project.

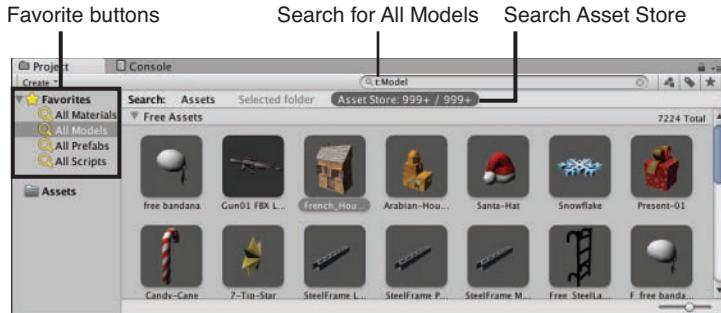
TIP**Project Organization**

Organization is extremely important for project management. As your projects get bigger, the number of assets will start to grow until finding anything can be a chore. You can help prevent a lot of frustration by employing some simple organization rules:

- ▶ Every asset type (scenes, scripts, textures, and so on) should get its own folder.
- ▶ Every asset should be in a folder.
- ▶ If you are going to use a folder inside another folder, make sure that the structure makes sense. Folders should become more specific and not be vague or generalized.

Following these few, simple rules will really make a difference.

Favorites buttons enable you to quickly select all assets of a certain type. This makes it possible for you to get an “at a glance” view of your assets quickly. When you click one of the Favorites buttons (**All Models**, for instance) or perform a search with the built-in search bar, you will see that you can narrow down the results between Assets and Asset Store. If you click **Asset Store**, you will be able to browse the assets that fit your search criteria from the Unity Asset Store (see Figure 1.8). You can further narrow your results down by free and paid assets. This is a fantastic addition because it enables you to go and grab assets that you need for your project without ever leaving the Unity interface.

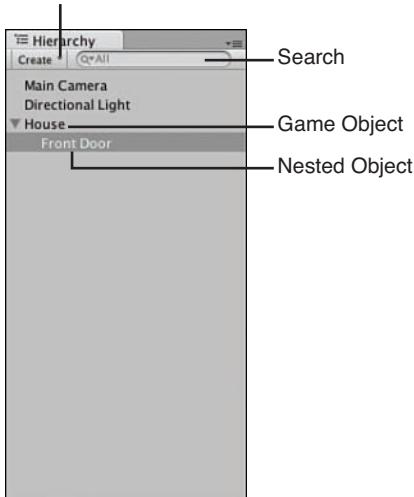
**FIGURE 1.8**

Searching the Unity Asset Store.

The Hierarchy View

In many ways, the Hierarchy view (see Figure 1.9) is a lot like the Project view. The difference is that the Hierarchy view shows all the items in the current scene instead of the entire project. When you first create a project with Unity, you get the default scene, which has just two items in it, the Main Camera and a Directional Light. As you add items to your scene, they will appear in the Hierarchy view. Just like with the Project view, you can use the Create menu to quickly add items to your scene, search using the built-in search bar, and click and drag items to organize and “nest” them.

Quick Create

**FIGURE 1.9**

The Hierarchy view.

TIP**Nesting**

Nesting is the term for establishing a relationship between two or more items. In the Hierarchy view, clicking and dragging an item onto another item will nest the dragged item under the other. This is commonly known as a parent–child relationship. In this case, the object on top is the parent, and any objects below it are children. You will know when an object is nested because it will become indented. As you will see later, nesting objects in the Hierarchy view can affect how they behave.

TIP**Scenes**

A scene is the term Unity uses to describe what you might already know as a level. As you develop a Unity project, each collection of objects and behaviors should be its own scene. Therefore, if you were building a game with a snow level and a jungle level, those would be separate scenes. You will see the words scene and level used interchangeably as you look for answers on the Internet.

TIP**Scene Organization**

The first thing you should do when working with a new Unity project is create a Scenes folder under Assets in the Project view. This way, all your scenes (or levels) will be stored in the same place. Be sure to give your scenes a descriptive name. Scene1 may sound like a great name now, but when you have 30 scenes, it can get confusing.

The Inspector View

The Inspector view enables you to see all of the properties of a currently selected item. Simply click any asset or object from the Project or Hierarchy view, and the Inspector view automatically propagates with information.

In Figure 1.10, we can see the Inspector view after the Main Camera object was selected from the Hierarchy view.

Let's break down some of this functionality:

- ▶ If you click the check box next to the object's name, it will become disabled and not appear in the project.
- ▶ Drop-down lists (such as the Layer or Tag lists; more on those later) are used to select from a set of predefined options.
- ▶ Text boxes, drop-downs, and sliders can have their values changed, and the changes will be automatically and immediately reflected in the scene—even if the game is running!

- ▶ Each game object acts like a container for different components (such as Transform, Camera, and GUIlayer in Figure 1.10). You can disable these components by unchecking them or remove them by right-clicking and selecting **Remove Component**.
- ▶ Components can be added by clicking the **Add Component** button.

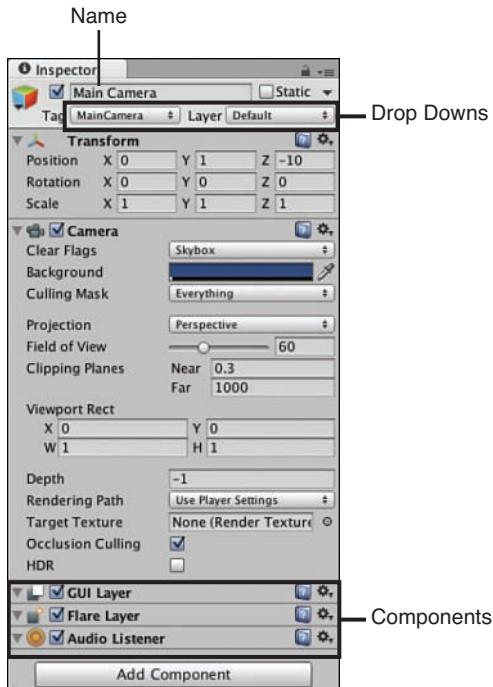


FIGURE 1.10
The Inspector view.

CAUTION

Changing Properties While Running a Scene

The capability to change the properties of an object and seeing those changes reflected immediately in a running scene is very powerful. It enables you to tweak things like movement speed, jumping height, collision power, and so on, all on the fly without stopping and starting the game. Be wary, though. Any changes you make to the properties of an object while the scene is running will be changed back when the scene finishes. If you make a change and like the result, be sure to remember what it was so that you can set it again when the scene is stopped.

The Scene View

The Scene view is the most important view you work with because it enables you to see your game visually as it is being built (see Figure 1.11). Using the mouse controls and a few hotkeys, you can move around inside your scene and place objects where you want them. This gives you an immense level of control.

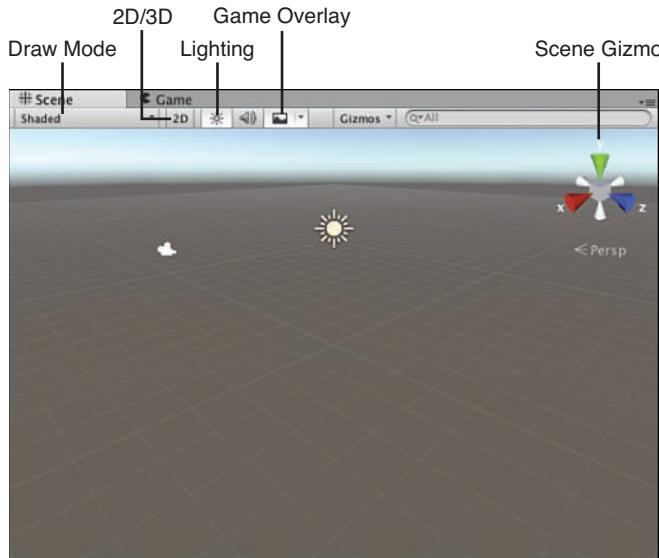


FIGURE 1.11
The Scene view.

In a little bit, we will talk about moving around within a scene, but, first, let's focus on the controls that are a part of the Scene view:

- ▶ **Drawmode:** This controls how the scene is drawn. By default, it is set to Shaded, which means objects will be drawn with their textures in full color.
- ▶ **2D/3D view:** This control changes from a 3D view, to a 2D view. Note in 2D view the scene gizmo does not show.
- ▶ **Scene lighting:** This control determines whether objects in the Scene view will be lit by default ambient lighting, or only by lights that actually exist within the scene. The default is to include the built-in ambient lighting.
- ▶ **Audition mode:** This control sets whether an audio source in the Scene view functions or not.

- ▶ **Game overlay:** This determines whether items like skyboxes, fog, and other effects appear in the Scene view.
- ▶ **Gizmo selector:** This control enables you to choose which “gizmos” appear in the Scene view. A gizmo is an indicator that gives visual debugging or aids in setup. This also controls whether the placement grid is visible.
- ▶ **Scene gizmo:** This control serves to show you which direction you are currently facing and to align the Scene view with an axis.

NOTE

The Scene Gizmo

The scene gizmo gives you a lot of power over the Scene view. As you can see, the control has an X, Y, and Z indicator that aligns with the three axes. This makes it easy to tell exactly which way you are looking in the scene. We discuss axes and 3D space more in a later chapter. The gizmo also gives you active control over the scene alignment. If you click one of the gizmo’s axes, you will notice that the Scene view immediately snaps to that axis and gets set to a direction like top or left. Clicking the box in the center of the gizmo toggles you between Iso and Persp modes.

Iso stands for Isometric and is the 3D view with no perspective applied. Inversely, Persp stands for Perspective and is the 3D view with perspective applied. Try it out for yourself and see how it affects the Scene view. You’ll notice the icon before the word change from parallel lines for isometric and diverging lines like crow’s feet for perspective.

The Game View

The last view to go over is the Game view. Essentially, the Game view allows you to “play” the game inside the editor by giving you a full simulation of the current scene. All elements of a game will function in the Game view just as they would if the project were fully built. Figure 1.12 shows you what a Game view looks like. Note that although the Play, Pause, and Step buttons are not technically a part of the Game view, they control the Game view and therefore are included in the image.

**FIGURE 1.12**

The Game view.

TIP**Missing Game View**

If you find that the Game view is hidden behind the Scene view, or that the Game view tab is missing entirely, don't worry. As soon as you click the **Play** button, a Game view tab will appear in the editor and begin displaying the game.

The Game view comes with some controls that assist us with testing our games:

- ▶ **Play:** The Play button enables you to play your current scene. All controls, animations, sounds, and effects will be present and working. Once a game is running, it will behave just like the game would if it were being run in a standalone player (such as on your PC or mobile device). To stop the game from running, click the Play button again.
- ▶ **Pause:** The Pause button pauses the execution of the currently running Game view. The game will maintain its state and continue exactly where it was when paused. Clicking the Pause button again will continue running the game.
- ▶ **Step:** The Step button works while the Game view is paused and causes the game to execute a single frame of the game. This effectively allows you to "step" through the game slowly and debug any issues you might have. Pressing the Step button while the game is running will cause the game to pause.

- ▶ **Aspect drop-down:** From this drop-down menu, you can choose the aspect ratio you want the Game view window to display in while running. The default is Free Aspect, but you can change this to match the aspect ratio of the target platform you are developing for.
- ▶ **Maximize on Play:** This button determines whether the Game view takes up the entirety of the editor when run. By default, this is off, and a running game will only take up the size of the Game view tab.
- ▶ **Mute Audio:** This button turns off the sounds when playing the game. Handy when the person sitting next to you is getting tired of hearing your repeated play-testing!
- ▶ **Stats:** This button determines whether rendering statistics are displayed on the screen while the game is running. These statistics can be useful for measuring the efficiency of your scene. This button is set to off by default.
- ▶ **Gizmos:** This is both a button and a drop-down menu. The button determines whether gizmos are displayed while the game is running. The button is set to off by default. The drop-down menu (the small arrow) on this button determines which gizmos appear if gizmos are turned on.

NOTE

Running, Paused, and Off

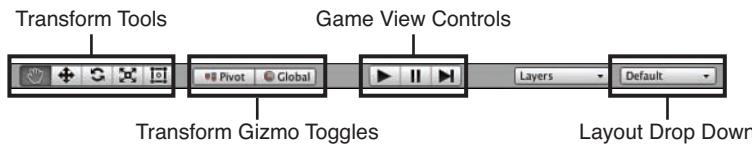
It can be difficult at first to determine what is meant by the terms **running**, **paused**, and **off**. When the game is not executing in the Game view, the game is said to be off. When a game is off, the game controls do not work and the game cannot be played. When the Play button is pressed and the game begins executing, the game is said to be running. Playing, executing, and running all mean the same thing. If the game is running and the Pause button is pressed, the game stops running but still maintains its state. At this point, the game is paused. The difference between a paused game and an off game is that a paused game will resume execution at the point it was paused, while an off game will begin executing at the beginning.

Honorable Mention: The Toolbar

Although not a view, the toolbar is an essential part of the Unity editor. Figure 1.13 shows the toolbar components:

- ▶ **Transform tools:** These buttons enable you manipulate game objects and are covered in greater detail later. Pay special attention to the button that resembles a hand. This is the Hand tool and is described later in this chapter.
- ▶ **Transform gizmo toggles:** These toggles manipulate how gizmos appear in the Scene view. Leave these alone for now.

- ▶ **Game view controls:** These buttons control the Game view.
- ▶ **Layers drop-down:** This menu determines which object layers appear in the Scene view. By default, everything appears in the Scene view. Leave this alone for now. Layers are covered in a later chapter.
- ▶ **Layout drop-down:** This menu allows you to quickly change the layout of the editor.

**FIGURE 1.13**

The toolbar.

Navigating the Unity Scene View

The Scene view gives you a lot of control over the construction of your game. The ability to place and modify items visually is very powerful. None of this is very useful though if you cannot move around inside the scene. This section covers a couple of different ways to change your position and navigate the Scene view.

The Hand Tool

The Hand tool (hotkey: Q) provides you a simple mechanic to move about the Scene view with the mouse (see Figure 1.14). This tool proves especially useful if you are using a mouse with only a single button (because other methods require a two-button mouse). Table 1.1 briefly explains each of the Hand tool controls.

**FIGURE 1.14**

The Hand tool.

TABLE 1.1 The Hand Tool Controls

Action	Effect
Click-drag	Drags the camera around the scene
Hold Alt and click-drag	Orbits the camera around the current pivot point
Hold Ctrl (Command on Mac) and right-click-drag	Zooms the camera

You can find all the Unity hotkeys here:

<http://docs.unity3d.com/Manual/UnityHotkeys.html>

CAUTION

Different Cameras

When working in Unity, you will be dealing with two types of cameras. The first is the standard game object camera. You can see that you already have one in your scene (by default). The second type is more of an imaginary camera. It is not a camera in the traditional sense. Instead, it is what determines what we can see in the Scene view. In this chapter, when the camera is mentioned, it is the second type that is being referred to. You will not actually be manipulating the game object camera.

Flythrough Mode

Flythrough mode enables you to move about the scene using a tradition first-person control scheme. This mode will feel right at home for anyone who plays first-person 3D games (such as the first-person shooter genre). If you don't play those games, this mode might take a little getting used to. Once you become familiar with it, though, it will be second nature.

Holding down the right mouse button will put you into Flythrough mode. All the actions laid out for you in Table 1.2 require that the right mouse button be held down.

TABLE 1.2 Flythrough Mode Controls

Action	Effect
Move the mouse	Causes the camera to pivot, which gives the feeling of "looking around" within the scene.
Press the WASD keys	The WASD keys move you about the scene. Each key corresponds with a direction: forward, left, back, and right, respectively.
Press the QE keys	The QE keys move you up and down, respectively, within the scene.
Hold Shift while pressing WASD or QE keys	Has the same effect as before, but it is much faster. Consider Shift to be your "sprint" button.

TIP

Zoom

Regardless of what method you are using for navigation, scrolling the mouse wheel will always zoom the view within a scene. By default, the scene zooms in and out of the center of the Scene view. If you hold **Alt** while scrolling, however, you zoom in and out of wherever the mouse is currently pointing. Go ahead and give it a try!

TIP**Snap Controls**

You have many ways to attain precious control over the scene navigation. Sometimes, you just want to quickly get around the scene though. For times like these, it is good to use what we call **snap controls**. If you want to quickly navigate to, and zoom in on, a game object in your scene, you can do so by highlighting the object in the Hierarchy view and pressing F. You will notice that the scene “snaps” to that game object. Another snap control is one you have seen already. The scene gizmo allows you to quickly snap the camera to any axis. This way, you can see an object from any angle without have to manually move the scene camera around. Be sure to learn the snap controls and navigating your scene quickly with snap!

Summary

In this hour, you took our first look at the Unity game engine. You started off by downloading and installing Unity. From there, you learned how to open and create projects. Then you learned about all the different views that make up the Unity editor. You also learned how to navigate around the Scene view.

Q&A

Q. Are assets and game objects the same?

A. Not exactly. Basically the big difference is that assets have a corresponding file or group of files on the hard drive, whereas a game object does not. An asset may or may not contain a game object.

Q. There are a lot of different controls and options. Will I need to memorize them all right away?

A. Not at all. Most controls and options will already be set to a default state that covers most situations. As your knowledge of Unity grows, you can continue to learn more about the different controls that you have available to you. This chapter is just meant to show you what's there and to give you some level of familiarity.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. True or False: You must purchase Unity Professional to make commercial games.
2. Which view enables us to manipulate objects in a scene visually?

3. True or False: You should always move your asset files around within Unity and not use the operating system's file explorer.
4. True or False: When creating a new project, you should include every asset that you think is awesome.
5. What mode do you enter in the Scene view when you hold down the right mouse button?

Answers

1. False. Up to \$100,000 of revenue you can use the free Personal addition.
2. The Scene view
3. True. This helps Unity keep track of the assets.
4. False. This will take up space, and slow your project down.
5. Flythrough mode

Exercise

Take a moment and practice the concepts studied in this chapter. It is important to have a strong foundational understanding of the Unity editor because everything you will learn from here on out will utilize it in some way. To complete this exercise, do the following:

1. Create a new scene by going to **File > New Scene** or by pressing **Ctrl+N** (**Command+N** on a Mac).
2. Create a Scene folder under Assets in the Project view.
3. Save your scene by going to **File > Save Scene** or by pressing **Ctrl+S** (**Command+S** on a Mac). Be sure to save the scene in the Scenes folder you created and name it something descriptive.
4. Add a cube to your scene. To do this, click the **GameObject** menu at the top, place your mouse over **Create Other**, and select **Cube** from the pop-up menu.
5. Select the newly added cube in the Hierarchy view and experiment with its properties in the Inspector view.
6. Practice navigating around the Scene view using Flythrough mode, the Hand tool, and snap controls. Use the cube as a point of reference to help you navigate.

HOUR 2

Game Objects

What You'll Learn in This Hour:

- ▶ How to work with 2D and 3D coordinates
- ▶ How to work with game objects
- ▶ How to work with transforms

Game objects are the foundational components of a Unity game project. Every item that exists in a scene is, or is based on, a game object. In this hour, you learn about game objects within Unity. Before you can start working with objects in Unity, however, you must first learn about the 2D and 3D coordinate systems. Then you will begin working with the built-in Unity game objects, and you will wrap up the hour by learning about the various game object transformations. Information gained in this hour is foundational to everything else in this book. Be sure to take your time and learn it well.

Dimensions and Coordinate Systems

For all of their glitz and glamour, video games are mathematical constructs. All the properties, movements, and interactions can be boiled down to numbers. Luckily for you, a lot of the groundwork has already been laid. Mathematicians have been toiling away for centuries to discover, invent, and simplify different processes so that you can more easily build your games with modern software. You may think the objects in a game just exist in space randomly, but really every game space has dimensions, and every object is placed in a coordinate system (or grid).

Putting the D in 3D

As mentioned previously, every game uses some level of dimensions. The most common dimension systems, the ones you are most likely familiar with, are 2D and 3D (short for two-dimensional and three-dimensional). A 2D system is a flat system. In a 2D system, you deal only with vertical and horizontal elements (or to put it another way: up, down, left, and right). Games like *Tetris*, *Pong*, and *Pac Man* are good examples of 2D games. A 3D system is like a 2D system, but it obviously has one more dimension. In a 3D system, you have not only horizontal and vertical (up, down, left, and right), but also depth (in and out). Figure 2.1 does a good job of illustrating the difference between a 2D square and a 3D cube, otherwise known as a *cube*. Notice how the inclusion of the depth axis in the 3D cube makes it seem to “pop out.”

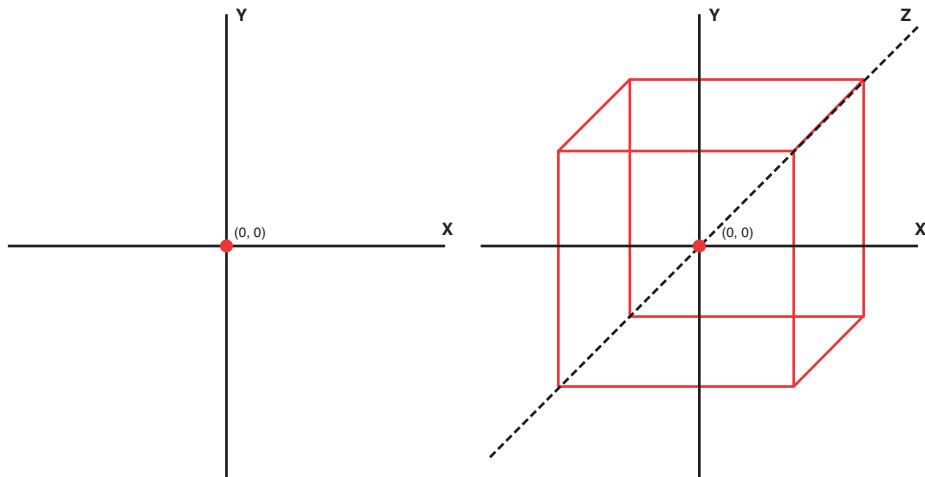


FIGURE 2.1

2D square versus 3D cube.

NOTE

Learning About 2D and 3D

Unity is a 3D engine. Therefore, all the projects made with it will inherently use all three dimensions. You might be wondering why we bother to cover 2D systems at all. The truth is that even in 3D projects, there are still a lot of 2D elements. Textures, screen elements, and mapping techniques all use a 2D system. Unity has a powerful set of tools to work with 2D games, and 2D systems aren't going away any time soon.

Using Coordinate Systems

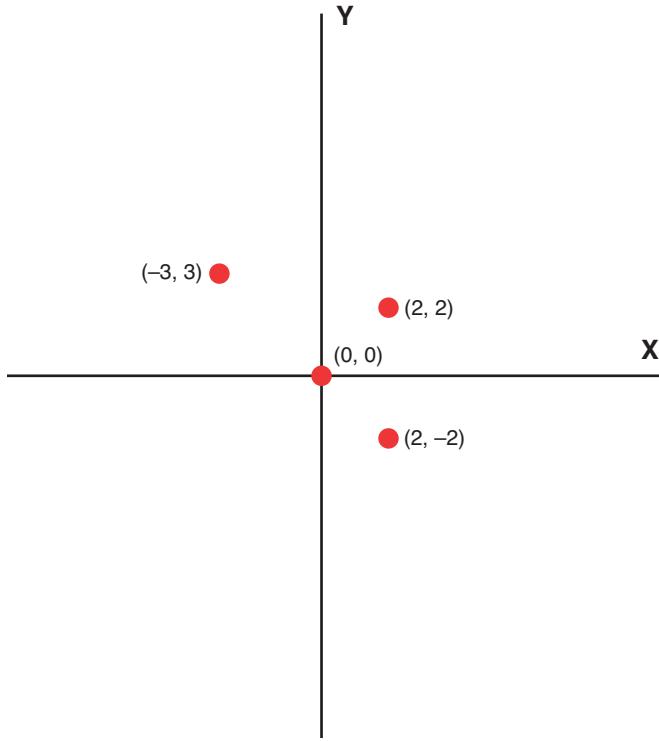
The mathematical equivalent of a dimension system is a coordinate system. A coordinate system uses a series of lines, called *axes* (the plural of *axis*), and locations, called *points*. These axes correspond directly with the dimensions that they mimic. For instance, a 2D coordinate system has the *x* axis and *y* axis, which represent the horizontal and vertical directions, respectively. If an object is moving horizontally, we say it is moving “along the *x* axis.” Likewise, the 3D coordinate system uses the *x* axis, the *y* axis, and the *z* axis for horizontal, vertical, and depth, respectively.

NOTE

Common Coordinate Syntax

When referring to an object's position, you will generally list its coordinates. Saying that an object is 2 on the *x* axis and 4 on the *y* axis can be a little cumbersome. Luckily, a shorthand way of writing coordinates exists. In a 2D system, you write coordinates like (x, y) , and in a 3D system, you write them like (x, y, z) . Therefore, this example would instead be written as $(2, 4)$. If that object were also 10 on the *z* axis, it would be written as $(2, 4, 10)$.

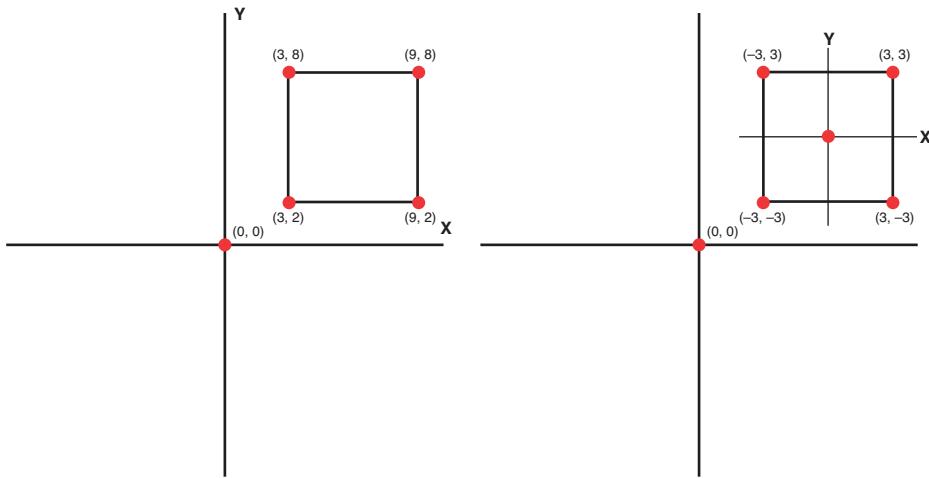
Every coordinate system has a point where all the axes intersect. This point is called the *origin*, and the coordinates for the origin are always $(0, 0)$ in a 2D system and $(0, 0, 0)$ in a 3D system. This origin point is very important because it is the basis by which all other points are derived. The coordinates for any other point are simply the distance of that point from the origin along each axis. A point's coordinates will get larger as it moves away from the origin. For example, as a point moves to the right, its *x* axis value gets larger. When it moves left, the *x* axis value gets smaller until it passes through the origin. At that time, the *x* value of the point begins getting larger again, but it also becomes negative. Consider Figure 2.2. This 2D coordinate system has three points defined. The point $(2, 2)$ is 2 units away from the origin in both the *x* and *y* directions. The point $(-3, 3)$ is 3 units to the left of the origin and 3 units above the origin. The point $(2, -2)$ is 2 units to the right of the origin and 2 units below the origin.

**FIGURE 2.2**

Points in relation to the origin.

World Versus Local Coordinates

You have now learned about the dimensions of a game world and about the coordinate systems that compose them. What you have been working with so far is considered the *world* coordinate system. At any given time, there is only a single x, y, and z axis in the world coordinate system. Likewise, there is only one origin that all objects share. What you might not know is that there is also something called the *local* coordinate system. This system is unique to each object, and it is completely separate from other objects. This local system has its own axes and origin that other objects don't use. Figure 2.3 illustrates the world versus local coordinate systems by showing the four points that make of a square for each.

**FIGURE 2.3**

World coordinates versus local coordinates.

You might be wondering what the local coordinate system is for if the world coordinate system is used for the position of objects. Later in this hour, you will look at transforming game objects and at parenting game objects. Both of these require the local coordinate system.

Game Objects

Every shape, model, light, camera, particle system, and so on in a Unity game all have one thing in common: They are all game objects. The game object is the fundamental unit of any scene. Even though they are simple, they are very powerful. At their root, game objects are little more than a transform (as discussed in greater detail later in the hour) and a container. This container exists to hold the various components that make objects more dynamic and meaningful. What you add to your game objects is up to you. There are many components, and they add a huge amount of variety. Throughout the course of this book, you will be learning to use many of these components.

NOTE

Built-In Objects

Not every game object you use will start as an empty object. Unity has several built-in game objects available to use right out of the box. You can see the large amount of items available by clicking the **GameObject** menu item at the top of the Unity editor. A large portion of learning to use Unity is learning to work with built-in and custom game objects.

▼ TRY IT YOURSELF

Creating Some Game Objects

Let's take some time now to work with game objects. You will be creating a few basic objects and examining their different components:

1. Create a new project or create a new scene in a project you already have.
2. Add an empty game object by clicking the **GameObject** menu item and selecting **Create Empty**. (Note: You could also create an empty game object by pressing **Ctrl+Shift+N** for PC users or **Command+Shift+N** for Mac users.)
3. Look in the Inspector view and notice how the game object you just created has no components other than a transform. All game objects have a transform. Clicking the **Add Component** button in the Inspector will show you all the components you could add to the object. Don't select any components at this time.
4. Add a cube to your project by clicking the **GameObject** menu item, hovering the cursor over **3D Object** and selecting **Cube** from the list.
5. Notice the various components the cube has that the empty game object doesn't. The mesh components make the cube visible, and the collider makes it able to interact with other objects.
6. Finally, add a point light to your project by clicking the **Create** drop-down in the Hierarchy view and selecting **Light > Point Light** from the list.
7. You can see that the point light only shares the transform component with the cube. It is instead focused entirely upon emitting light. The light emitted by this point light adds to the directional light already in the scene.

Transforms

At this point, you have learned and explored the different coordinate systems and experimented with some game objects. It is time to put the two together. When dealing with 3D objects, you will often hear the term *transform*. Depending on the context, transform is either a noun or a verb. All objects in 3D space have a position, a rotation, and a scale. If you combine them all together, you get an object's transform (noun). Alternatively, *transform* can be a verb if it refers to changing an object's position, rotation, or scale. Unity combines the two meanings of the word with the *transform component*.

You will recall that the transform component is the only component that every game object has to have. Even empty game objects have transforms. Using this component, you can both see the current transform of the object and change (or transform) the transform of the object. It might sound confusing now, but it is fairly simple. You will get the hang of it in no time. Because the

transform is made up of the position, rotation, and scale, it stands to reason that there are three separate methods (called *transformations*) of changing the transform: translation, rotation, and scaling (respectively). These transformations can be achieved using either the Inspector or the transform tools. Figures 2.4 and 2.5 illustrate which Inspector components and tools correlate with which transforms.

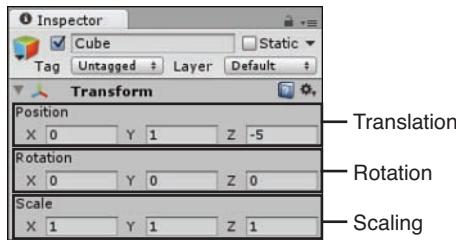


FIGURE 2.4
Transform options in the Inspector.

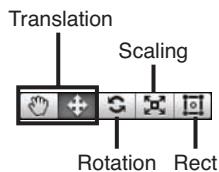


FIGURE 2.5
The transform tools.

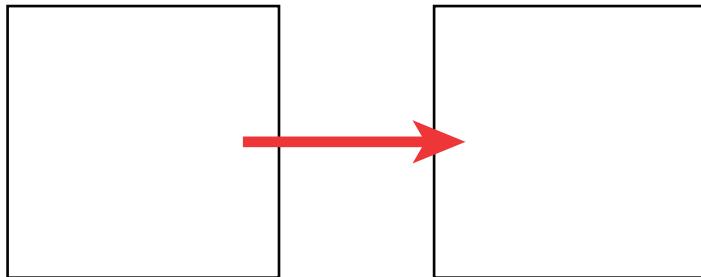
TIP

About Rect Transforms

2D objects in Unity have a different type of transform, called a rect transform. We cover this later in the book in more detail; for now, just understand that they are the 2D equivalent of the other three tools.

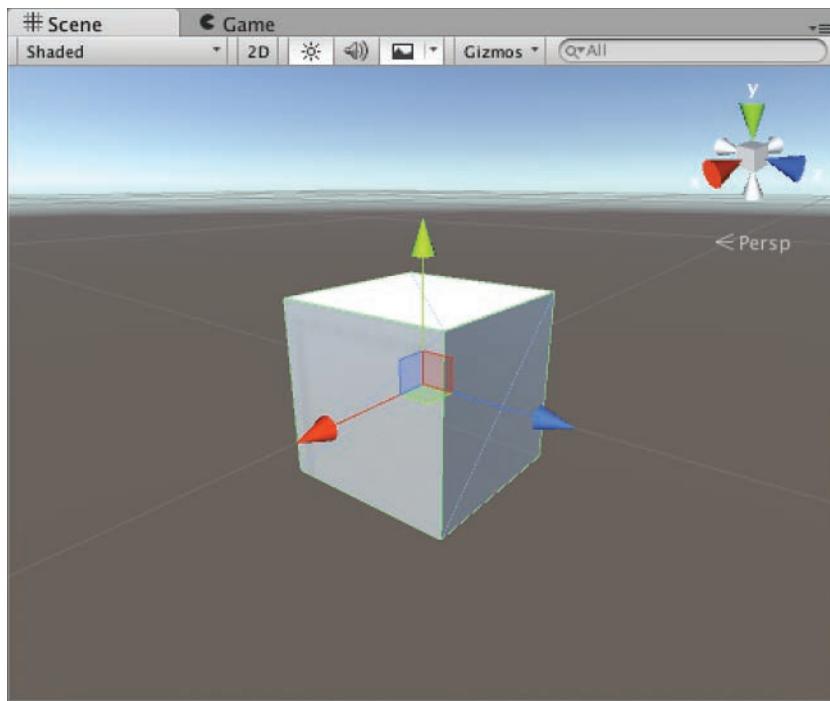
Translation

Changing the coordinate position of an object in a 3D system is called *translation*, and it is the simplest transform that you can apply to an object. When you translate an object, it is shifted along an axis. Figure 2.6 demonstrates a square being translated along the x axis.

**FIGURE 2.6**

Sample translation.

When you select the Translate tool (hotkey: **w**), you will notice that whatever object you have selected will change slightly in the Scene view. More specifically, you will see three arrows appear pointing away from the center of the object along the three axes. These are translation gizmos, and they help you move your objects around in the scene. Clicking on and holding any of these axis arrows causes them to turn yellow. Then, if you move your mouse, the object will move along that axis. Figure 2.7 shows you what the translation gizmos look like. Note that the gizmos appear only in the Scene view; if you are in the Game view, you will not see them.

**FIGURE 2.7**

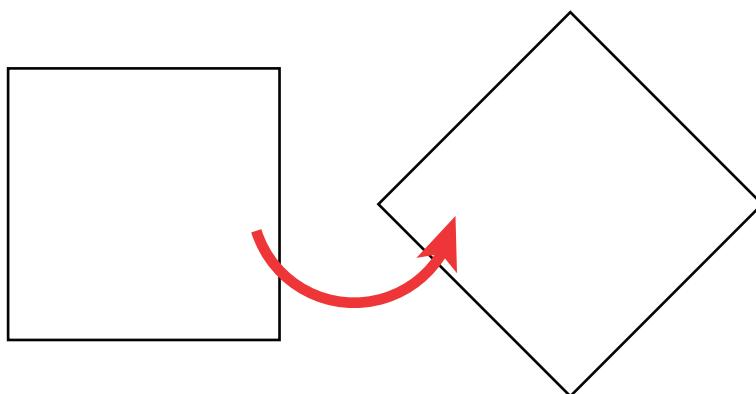
Translation gizmos.

TIP**The Transform Component and Transform Tools**

Unity provides two ways to manage the transform of your objects. Knowing when to use each is important. You will notice that when you change an object's transform in the Scene view with a transform tool, the transform data also changes in the Inspector view. It is often easier to make large changes to an object's transform using the Inspector view because you can just change the values to what they need to be. The transform tools, however, are more useful for quick, small changes. Learning to use both together will greatly improve your workflow.

Rotation

Rotating an object does not move it in space. Instead, it changes the object's relationship to that space. More simply stated, rotation enables you to redefine which direction the x, y, and z axes point for a particular object. When an object rotates around an axis, it is said to be rotating *about* that axis. Figure 2.8 shows a square being rotated about the z axis.

**FIGURE 2.8**

Rotation about the z axis.

TIP**Determining the Axis of Rotation**

If you are unsure which axis you need to rotate an object about to get a desired effect, you can use a simple mental method. One axis at a time, pretend that the object is stuck in place by a pin that is parallel with that axis. The object can only spin around the pin stuck in it. Now, determine which pin allows the object to spin the way you want. That is the axis you need to rotate the object about.

Just as with the Translate tool, selecting the Rotate tool (hotkey: **e**) causes rotation gizmos to appear around your object. These gizmos are circles representing the object's rotation path about the axes. Clicking and dragging on any of these circles turns them yellow and rotates the object about that axis. Figure 2.9 shows you what the rotation gizmos look like.

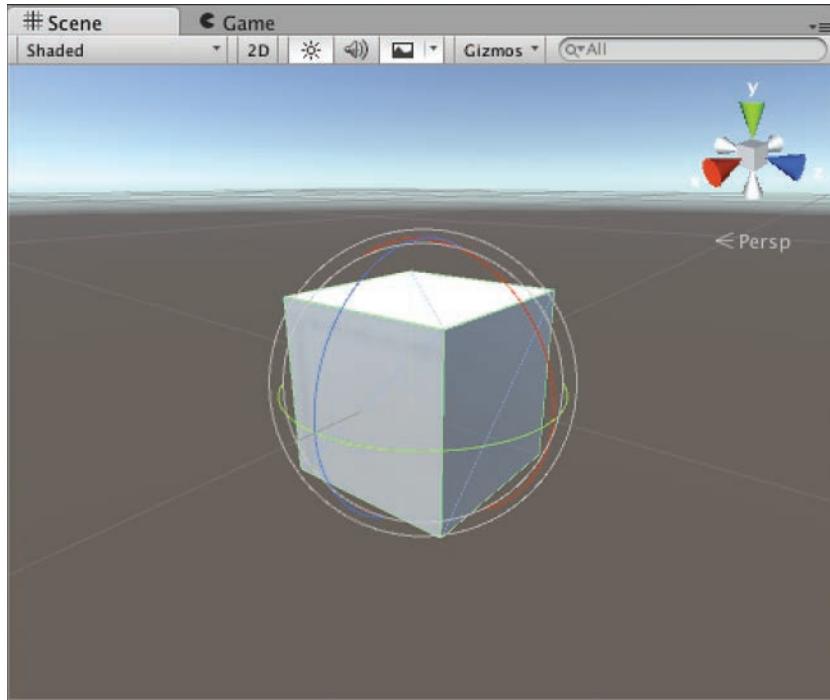
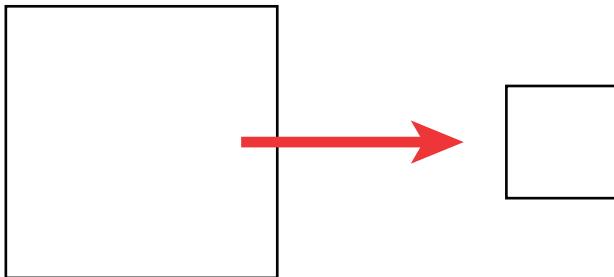


FIGURE 2.9

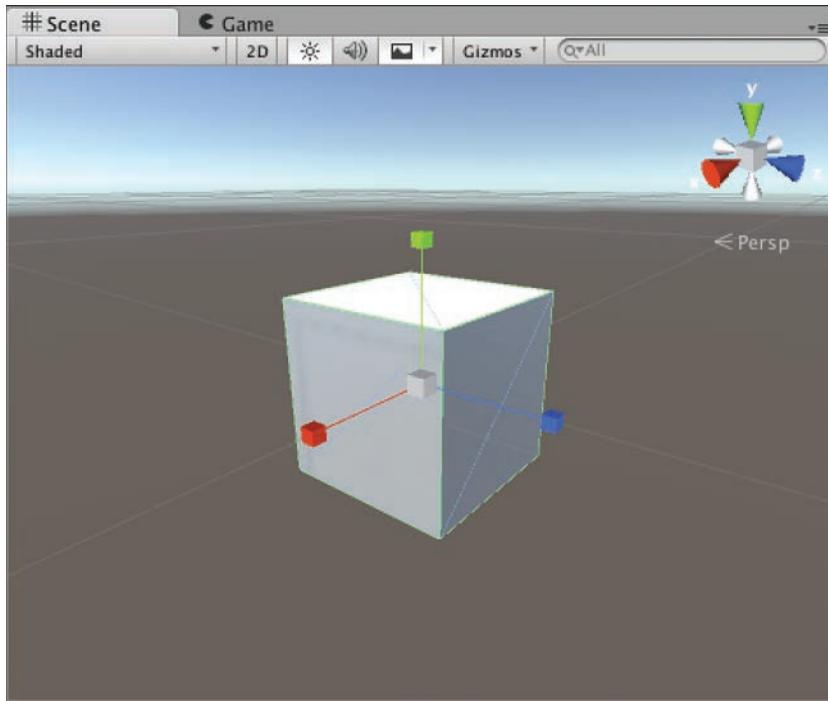
The Rotate tool gizmos.

Scaling

Scaling causes an object to grow or shrink within a 3D space. This transform is really straightforward and simple in its use. Scaling an object on any axis causes its size to change on that axis. Figure 2.10 demonstrates a square being scaled down on the x and y axes. Figure 2.11 shows you what the scaling gizmos look like when you select the Scaling tool (hotkey: **r**).

**FIGURE 2.10**

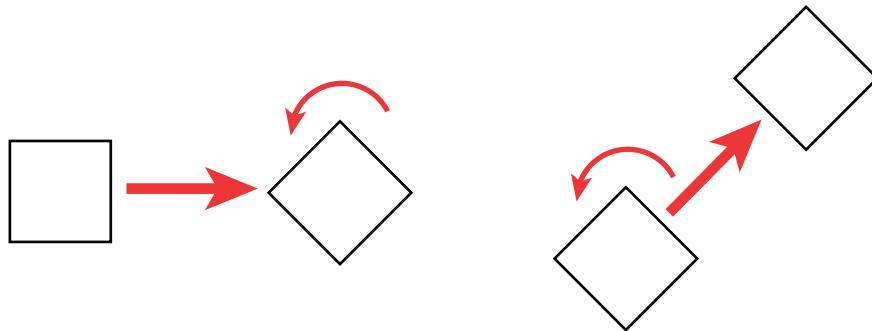
Scaling on the x and y axes.

**FIGURE 2.11**

The scaling gizmos.

Hazards of Transformations

As mentioned before, transformations use the local coordinate system. Therefore, the changes that are made can potentially impact future transformations. Consider Figure 2.12. Notice how the same two transformations, when applied in reverse order, have very different effects.

**FIGURE 2.12**

Effects of transformation order.

As you can see, not paying attention to transformation order can have unexpected consequences. Luckily, the transformations have consistent effects that can be planned on:

- ▶ **Translation:** Translation is a fairly inert transformation. That means that any changes applied after it generally won't be affected.
- ▶ **Rotation:** Rotation changes the orientation of the local coordinate system axes. Any translations applied after a rotation would cause the object to move along the new axes. If you were to rotate an object 180 degrees about the z axis, for example, and then move in the positive y direction, the object would appear to be moving down instead of up.
- ▶ **Scaling:** Scaling effectively changes the size of the local coordinate *grid*. Basically, when you scale an object to be larger, you are really scaling the local coordinate system to be larger. This causes the object to seem to grow. This change is multiplicative. For example, if an object is scaled to 1 (its natural, default size) and then translated 5 units along the x axis, the object appears to move 5 units to the right. If the same object were to be scaled to 2, however, then translating 5 units on the x axis would result in the object appearing to move 10 units to the right. This is because the local coordinate system is now double the size and 5 times 2 equals 10. Inversely, if the object were scaled to .5 and then moved, it would appear to only move 2.5 units ($.5 \times 5 = 2.5$).

Once you understand these rules, determining how an object will change with a set of transformations becomes easy. These effects depend on whether we have Local or Global selected, and take some getting used to, so the best thing is to experiment.

Transforms and Nested Objects

In Hour 1, “Introduction to Unity,” you learned how to nest game objects in the Hierarchy view (drag one object onto another one). Recall that when you have an object nested inside another one, the top-level object is the parent, and the other object is the child. Transformations applied

to the parent object work as normal. The object can be moved, scaled, and rotated. What's special is how the child object behaves. Once nested, a child object's transform is relative to that of the parent object, not the world. Therefore, a child object position is not based on its distance from the origin, but the distance from the parent object. If the parent object is rotated, the child object would move with it. If you looked at the child's rotation, however, it would not register that it had rotated at all. The same goes for scaling. If you scale the parent object, the child also changes in size. The scale of the child object would remain unchanged. You might be confused by why this is. Remember, when a transformation is applied, it is not applied to the object, but to the object's coordinate system. An object isn't rotated, its coordinate system is. The effect is that the object turns. When a child object's coordinate system is based on the local coordinate system of the parent, any changes to the parent system will directly change the child (without the child knowing about it).

Summary

In this hour, you learned all about game objects in Unity. You started off by learning all about the differences between 2D and 3D. From there, you looked at the coordinate system and how it breaks the "world" concepts down mathematically. You then began working with game objects, including some of the built-in ones. You ended by learning all about transforms and the three transformations. You got to try out the transforms, learn about some of the hazards, and how they affect nested objects.

Q&A

Q. Is it important to learn both the 2D and 3D concepts?

A. Yes. Even games that are entirely 3D still utilize some of the 2D concepts on a technical level.

Q. Should I learn to use all the built-in game objects right away?

A. Not necessarily. There are many game objects, and it can be overwhelming to attempt to learn them all right away. Take your time and learn about the objects as they are covered here.

Q. What is the best way to get familiar with transforms?

A. Practice. Keep working with them; eventually, they will become quite natural.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What does the *D* in 2D and 3D stand for?
2. How many transformations are there?
3. True or False: Unity has no built-in objects and you must create your own.
4. If you want an object to be “lying on its side” 5 units to the right of its current position, does it matter whether you translate then rotate, or the other way round?

Answers

1. Dimension.
2. Three.
3. False. Unity provides many built-in objects for you.
4. No, you can do this in either order.

Exercise

Take a moment to experiment with the way transformations work in a parent/child object scenario. You will get a better feel for exactly how the coordinate systems change the way things are oriented.

1. Create a new scene or project.
2. Add a cube to the project and place it at $(0, 2, -5)$. Remember the shorthand notation for coordinates. The cube should have an *x* value of 0, a *y* value of 2, and a *z* value of -5 . You can set these values easily in the Transform component in the Inspector view.
3. Add a sphere to your scene. Pay attention to the sphere’s *x*, *y*, and *z* values.
4. Nest the sphere under the cube by dragging the sphere in the Hierarchy view onto the cube. Notice how the position values changed. The sphere is now located relative to the cube.
5. Place the sphere at $(0, 1, 0)$. Notice how it doesn’t go to right above the origin and instead sits right above the cube.
6. Now experiment with the various transformations. Be sure to try them on the cube as well as the sphere and see how differently they behave for a parent versus a child object.

HOUR 3

Models, Materials, and Textures

What You'll Learn in This Hour:

- ▶ The fundamentals of models
- ▶ How to import custom and premade models
- ▶ How to work with materials and shaders

In this hour, you learn all about models and how they are used in Unity. You start by looking at the fundamental principles of meshes and 3D objects. From there, you learn how to import your own models or use ones acquired from the Asset Store. You finish this hour by examining Unity's material and shader functionality.

The Basics of Models

Video games wouldn't be very *video* without the graphical components. In 2D games, the graphics consist of flat images called *sprites*. All you needed to do was change the x and y positions of these sprites and flip several of them in sequence and the viewer's eye was fooled into believing that it saw true motion and animation. In 3D games, however, things aren't so simple. In worlds with a third axis, objects need to have volume to fool the eye. Because games use a large number of objects, the need to process things quickly was very important. Enter the mesh. A mesh, at its most simple, is a series of interconnected triangles. These triangles build off of each other in strips to form basic to very complex objects. These strips provide the 3D definitions of a model and can be processed very quickly. Don't worry, though; Unity handles all of this for you so that you don't have to manage it yourself. Later in this hour, you'll see just how triangles can make up various shapes in the Unity Scene view.

NOTE**Why Triangles?**

You might be asking yourself why 3D objects are made up entirely of triangles. The answer is simple. Computers process graphics as a series of points, otherwise known as *vertices*. The fewer vertices an object has, the faster it can be drawn. Triangles have two properties that make them desirable. The first is that whenever you have a single triangle, you need only one more vertex to make another. To make one triangle, you need three vertices, two triangles take only four, and three triangles require only five. This makes them very efficient. The second is that by using this practice of making strips of triangles, you can model any 3D object. No other shape affords you that level of flexibility and performance.

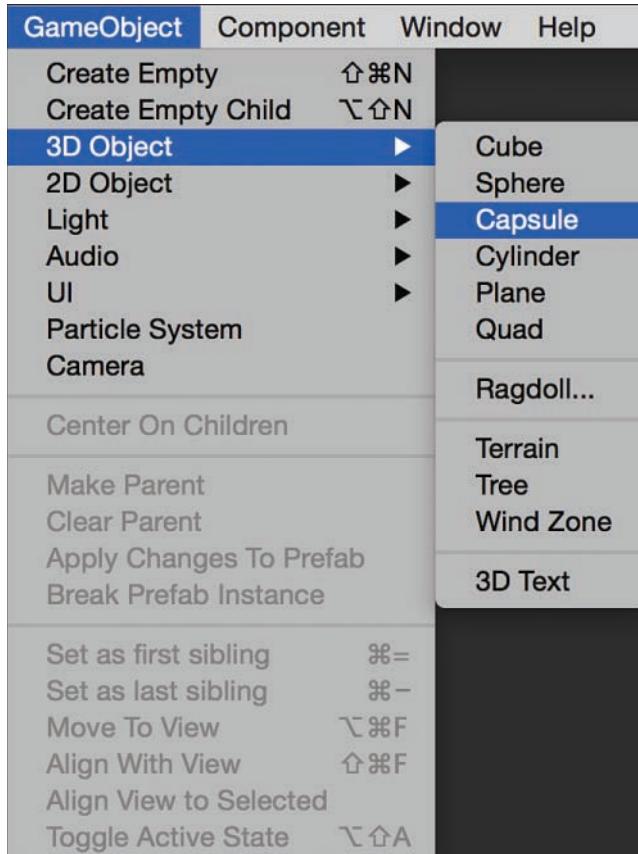
NOTE**Model or Mesh?**

The terms *model* and *mesh* are similar, and you can often use them interchangeably. There is a difference, however. A mesh contains all the points and lines that defines the 3D shape of an object. When you refer to the shape or form of a model, you are really referring to a mesh. A model, therefore, is an object that contains a mesh.

A model has a mesh to define its dimensions, but it can also contain animations, textures, materials, shaders, and other meshes. A good general rule is this: If the item in question contains anything other than vertex information, it is a model; otherwise, it is a mesh.

Built-In 3D Objects

Unity comes with a few basic built-in meshes (or primitives) for you to work with. These tend to be simple shapes that serve simple utilities or can be combined to make more complex objects. Figure 3.1 shows the available built-in meshes. (You worked with the cube and sphere in the previous hours.)

**FIGURE 3.1**

The built-in meshes in Unity.

TIP**Modeling with Simple Meshes**

Do you need a complex object in your game but you can't find the right type of model to use? Nesting objects in Unity enables you to easily make simple models using the built-in meshes. Just place the meshes near each other so that they form the rough look you want. Then nest all the objects under one central object. This way, when you move the parent, all the children move, too. This might not be the prettiest way to make models for your game, but it will do in a pinch!

Importing Models

Having built-in models is nice, but most of the time your games will require art assets that are a little more complex. Thankfully, Unity makes it rather easy to bring your own 3D models into

your projects. Just placing the file containing the 3D model in your Assets folder is enough to bring it into the project. From there, dragging it into the scene or hierarchy builds a game object around it. Natively, Unity supports .fbx, .dae, .3ds, .dxf, and .obj files. This enables you to work with just about any 3D modeling tool.

TRY IT YOURSELF

Importing Your Own 3D Model

Let's walk through the steps required to bring custom 3D models into a Unity project:

1. Create a new Unity project or scene.
2. In the Project view, create a new folder named **Models** under the Assets folder. (Right-click the Assets folder and select **Create > Folder**.)
3. Locate the Torus.fbx file provided for you in the Hour 3 folder of the book files.
4. With both the operating system's file browser and the Unity editor open and side by side, click and drag the Torus.fbx file from the file browser into the Models folder that you created in step 2. In Unity, click the **Models** folder to see the new Torus file. If done correctly, your Project view will resemble Figure 3.2. Notice the Materials folder that was added for you. You will learn more about this later.

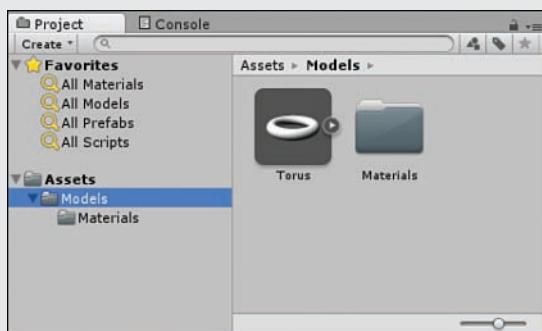


FIGURE 3.2

The Project view after the Torus model was added.

5. Click the **Torus** asset in the Models folder and look at the Inspector view. Change the value of the scale factor from 1 to 100. Now click the **Apply** button at the bottomright of the inspector.
6. Drag the Torus asset from the Models folder onto the Scene view. Notice how a Torus game object was added to the scene containing a mesh filter and mesh renderer. These allow the Torus to be drawn to the screen. If you click the **Torus** object, you see how it is made up of many connected triangles.

CAUTION

Default Scaling of Meshes

Most of the Inspector view options for meshes are advanced and are not covered right now. The property you are interested in is the scale factor. By default, Unity imports meshes scaled down. By changing the value of the scale factor from .01 to 1, you are telling Unity to allow the model to enter the scene as the same size as it was created.

Models and the Asset Store

You don't have to be an expert modeler to make games with Unity. The Asset Store provides a simple and effective way to find premade models and import them into your project. Generally speaking, models on the Asset Store are either free or paid and come alone or in a collection of similar models. Some of the models come with their own textures, and some of them are simply the mesh data.

TRY IT YOURSELF ▼

Downloading Models from the Asset Store

Let's learn how to find and download models from Unity's Asset Store. We will be acquiring a model named Robot Kyle and importing it into our scene:

1. Create a new scene (click **File > New Scene**). In the Project view, type **Robot Kyle: Model** into the search bar (see Figure 3.3).



FIGURE 3.3

Steps to locate a model asset.

2. In the search filter section, click the **Asset Store** button (see Figure 3.3). If these words aren't visible, you may need to resize your editor window or Project view window. You will also need to be connected to the Internet.
3. Locate **Robot Kyle** and select any one of the three free assets.
4. In the Inspector view, click **Import Package**. At this point, you may be prompted to provide your Unity account credentials. If you have any trouble, we have also provided the file in this hour's book files as a Unity package; simply double-click this to import.
5. When the Importing Package dialog opens, leave everything checked and select **Import**.
6. There will now be a new asset folder called Robot Kyle. Locate the robot model under **Assets > Robot Kyle > Model** and drag it into the Scene view (see Figure 3.4). Note that the model will be fairly small in the Scene view; you might need to move closer to see it.



FIGURE 3.4
The Unity project with Robot Kyle added.

Textures, Shaders, and Materials

Applying graphical assets to 3D models can be daunting for beginners, if you are not familiar with it. Unity uses a simple and specific workflow that gives you a lot of power when determining exactly how you want things to look. Graphical assets are broken down into textures, shaders, and materials. Each of these is covered individually in its own section, but Figure 3.5 shows you how they fit together. Notice that textures are not applied directly to models. Instead, textures and shaders are applied to materials. Those materials are in turn applied to the models. This way, the look of a model can be swapped or modified quickly and cleanly without a lot of work.

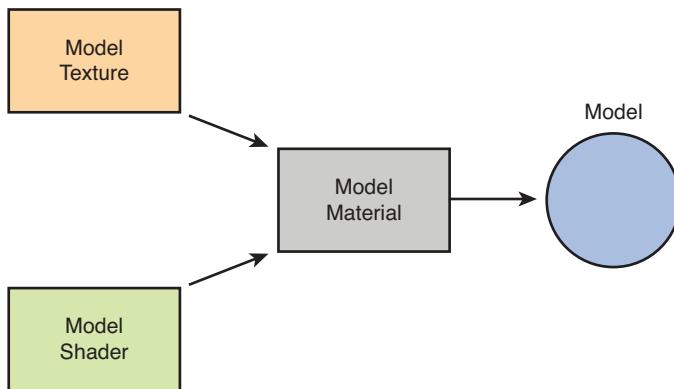


FIGURE 3.5

The model asset workflow.

Textures

Textures are flat images that get applied to 3D objects. They are responsible for models being colorful and interesting instead of blank and boring. It can be strange to think that a 2D image can be applied to a 3D model, but it is a fairly straightforward process once you are familiar with it. Think about a soup can for a moment. If you were to take the label off of the can, you would see that it is a flat piece of paper. That label is like a texture. After the label was printed, it was then wrapped around the 3D can to provide a more pleasing look.

Just like all other assets, adding textures to a Unity project is easy. Start by creating a folder for your textures; a good name would be `Textures`. Then drag any textures you want in your project into the `Textures` folder you just created. That's it!

NOTE**That's an Unwrap!**

Imagining how textures wrap around cans is fine, but what about more complex objects? When creating an intricate model, it is common to generate something called an *unwrap*. The unwrap is somewhat akin to a map that shows you exactly how a flat texture will wrap back around a model. If you look in the Robot Kyle > Textures folder from earlier this hour, you notice the Robot_Color texture. It looks strange, but that is the unwrapped texture for the model. The generation of unwraps, models, and textures is an art form to itself and is not covered in this text. A preliminary knowledge of how it works should suffice at this level.

CAUTION**Weird Textures**

Later in this hour, you will apply some textures to models. You might notice that the textures warp a bit or get flipped in the wrong direction. Just know that this is not a mistake or an error. This problem occurs when you take a basic rectangular 2D texture and apply it to a model. The model has no idea which way is correct, so it applies the texture however it can. If you want to avoid this issue, use textures specifically designed for (unwrapped for) the model that you are using.

Shaders

If the texture of a model determines what is drawn on its surface, the shader is what determines *how* it is drawn. Here's another way to look at this: A material contains properties and textures, and shaders dictate what properties and textures a material can have. This might seem nonsensical right now, but later when we create materials you will begin to understand how they work. Much of the information about shaders is covered later this hour, because you cannot create a shader without a material. In fact, much of the information to be learned about materials is actually about the material's shader.

TIP**Thought Exercise**

If you are having trouble understanding how a shader works, consider this scenario: Imagine you have a piece of wood. The physicality of the wood is its mesh; the color, texture, and visible element are its texture. Now take that piece of wood and pour water on it. The wood still has the same mesh. It still is made of the same substance (wood). It looks different, though. It is slightly darker and shiny. The water in this example is the shader. The shader took something and made it look a little different without actually changing it.

Materials

As mentioned earlier, materials are not much more than containers for shaders and textures that can be applied to models. Most of the customization of materials depends on which shader is chosen for it, although all shaders have some common functionality.

To create a new material, start by making a Materials folder. Then right-click the folder and select **Create > Material**. Give your material some descriptive name and you are done. Figure 3.6 shows two materials with different shader settings. Notice how they both use the same Standard shader. Each has a base Albedo color of white. A Smoothness setting of zero is a rough surface, and the lighting looks very flat as the light bounces in a lot of directions. A higher setting leads to a shinier look. There is also a preview of the material (blank now because there is no texture).

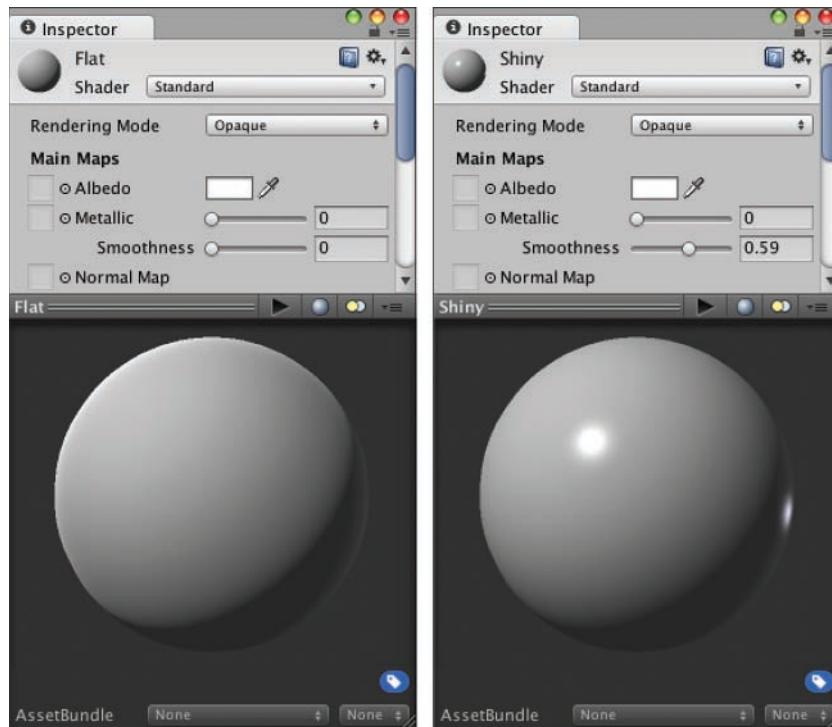


FIGURE 3.6

Two materials with different shader settings.

Shaders Revisited

Now that you understand textures, models, and shaders, it is time to look at how it all comes together. Unity 5 has a very powerful Standard shader, which we will be focusing on in this book. Table 3.1 describes the common shader properties.

TABLE 3.1 Common Shader Properties

Property	Description
Albedo	The Albedo property defines the base color of the object. With Unity 5's powerful new Physically Based Shaders, this color interacts with light like a real object would. For example, a yellow Albedo will look yellow in white light, but green under blue light. You can even attach a texture to set the Albedo at different points on the object.
Metallic	This setting does what it says on the tin; it changes how metallic the material looks. This setting can also take in an image as a "map" of how metallic different parts of the model appear. For realistic results, set this to either 0 or 1.
Smoothness	Smoothness is a key element in Physically Based Shader materials. It contributes variation, imperfections, and detail to surfaces, and helps represent their state and age. This can also have the effect of making the model look more or less shiny. For realistic results, avoid extreme values like 0 and 1.
Normal Map	The Normal Map block contains the normal map that will be applied to your model. A normal map can be used to apply relief to a model. This is useful when calculating lighting to give the model more detail than it would otherwise have.
Tiling	The Tiling property defines how often a texture can repeat on a model. It can repeat in both the x and y axes.
Offset	The Offset property defines whether a gap will exist between edges of the object and the texture.
Other Settings	There are many other settings of the Standard shader for you to explore; however, we'll be focusing on mastering the settings above in this book.

This might seem like a lot of information to take in, but once you become more familiar with the few basics of textures, shaders, and materials, you'll find this a smooth process.

Unity has several other shaders which we won't cover in this book. The Standard Shader is very flexible, and will cover most of your basic needs.

TRY IT YOURSELF ▼

Applying Textures, Shaders, and Materials to Models

Let's put all of our knowledge of textures, shaders, and materials together and create a decent-looking brick wall:

1. Start a new project or scene. Note that creating a new project will close and reopen the editor.
2. Create a Textures and a Materials folder.
3. Locate the Brick_Texture.png file in the book files and drag it into the Textures folder created in step 2.
4. Add a cube to the scene. Position it at (0, 1, -5). Give it a scale of (5, 2, 1). See Figure 3.7 for the cube properties.

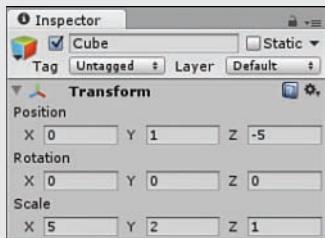
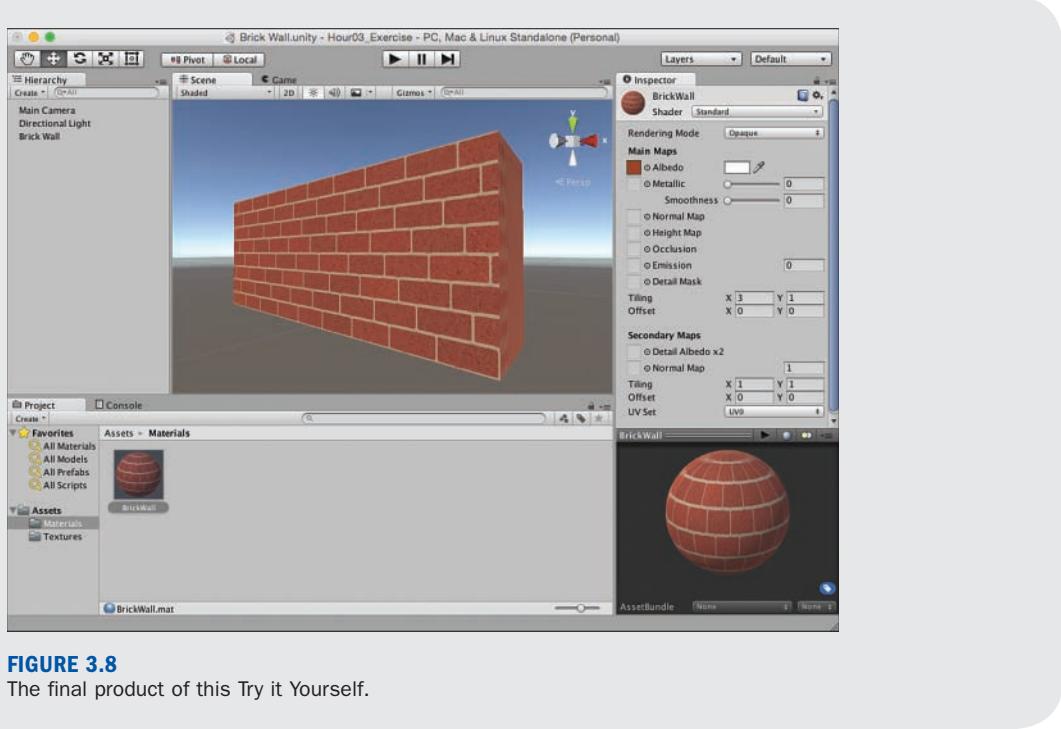


FIGURE 3.7

The properties of the cube.

5. Create a new material (right-click the Materials folder and select **Create > Material**) and name it **BrickWall**.
6. Leave the shader as Standard, and under Main Maps click the little circle to the left of the word Albedo. Select **Brick_Texture** from the pop-up window.
7. Click and drag the brick wall material from the Project view onto the cube in the Scene view.
8. Notice how the texture is stretched across the wall a little too much. With the material selected, change the value of the x tiling to be **3**. Make sure you do this in the Main Maps section, not the Secondary Maps. Now the wall looks much better.
9. You now have a textured brick wall in your scene. Figure 3.8 contains the final product.

**FIGURE 3.8**

The final product of this Try it Yourself.

Summary

In this hour, you learned all about models in Unity. You started by learning about how models are built with collections of vertices called meshes. Then, you discovered how to use the built-in models, import your own models, and download models from the Asset Store. You then learned about the model art workflow in Unity. You experimented with textures, shaders, and materials. You finished by creating a textured brick wall.

Q&A

Q. Will I still be able to make games if I'm not an artist?

A. Absolutely. Using free online resources and the Unity Asset Store, you can find various art assets to put in your games.

Q. Will I need to know how to use all the built-in shaders?

A. Not necessarily. Many shaders are very situational. Start with the shader covered in this chapter and learn more if a game project requires it.

- Q.** If there are paid art assets in the Unity Asset Store, does that mean I can sell my own art assets?
- A.** Yes, it does. In fact, it is not limited to only art assets. If you can create high-quality assets, you can certainly sell them in the store.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. True or False: Because of their simple nature, squares make up meshes in models.
2. What file formats does Unity support for 3D models?
3. True or False: Only paid models can be downloaded from the Unity Asset Store.
4. Explain the relationship between textures, shaders, and materials.

Answers

1. False, meshes are made up of triangles.
2. .fbx, .dae, .3ds, .dxf, and .obj files.
3. False. There are many free models.
4. Materials contain textures and shaders. Shaders dictate the properties that can be set by the material and how the material gets rendered.

Exercise

Let's experiment with the effects shaders have on the way models look. You will use the same mesh and texture for each model; only the shaders will be different. The project created in this exercise is named Hour 3_Exercise and is available in the Hour 3 book files.

1. Create a new scene or project.
2. Add a Materials and a Textures folder to your project. Locate the files Brick_Normal.png and Brick_Texture.png in the Hour 3 book files and drag them into the Textures folder.
3. In the Project view, select **Brick_Texture**. In the Inspector view, change the aniso level to 3 to increase the texture quality for curves. Click **Apply**.
4. In the Project view, select **Brick_Normal**. In the Inspector view, change the texture type to **Normal Map**. Click **Apply**.
5. Select the Directional Light in your Hierarchy, and give it a position of (0, 10, -10) with a rotation of (30, -180, 0).
6. Add four spheres to your project. Scale them each to (2, 2, 2). Spread them out by giving them positions of (1, 2, -5), (-1, 0, -5), (1, 0, -5), and (-1, 2, -5).

7. Create four new materials in the Materials folder. Name them **DiffuseBrick**, **SpecularBrick**, **BumpedBrick**, and **BumpedSpecularBrick**. Figure 3.9 contains all the properties of the four materials. Go ahead and set their values.

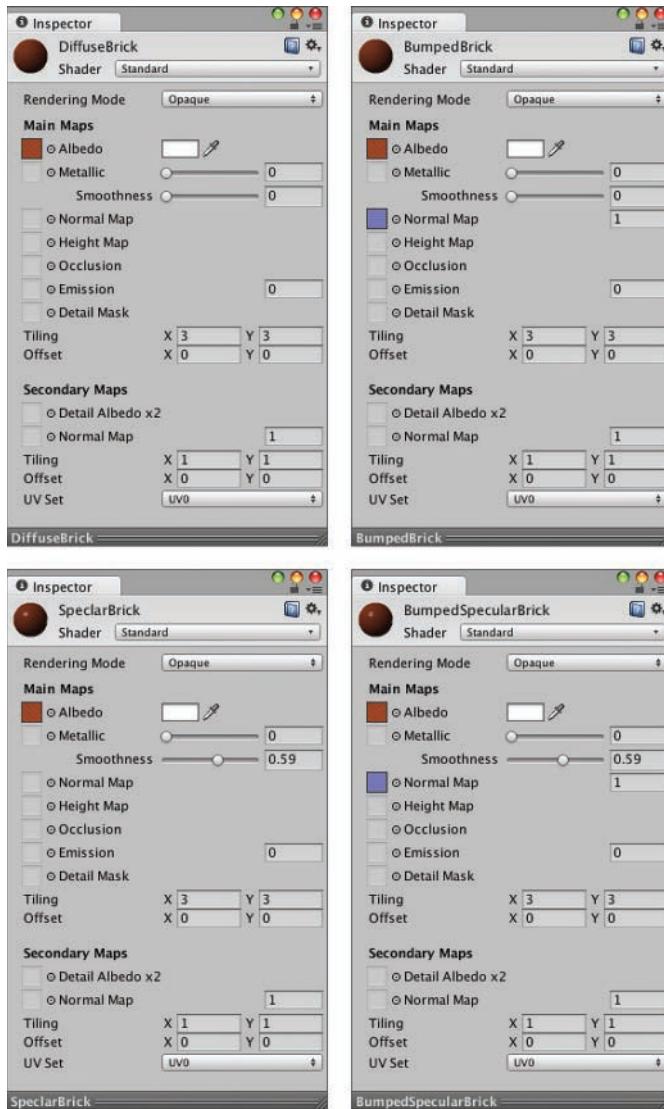


FIGURE 3.9
Material properties.

8. Click and drag each of the materials onto one of the four spheres. Notice how the light and the curvature of the spheres interact with the different shaders. Remember that you can move about the Scene view to see the spheres at different angles.

HOUR 4

Terrain

What You'll Learn in This Hour:

- ▶ The fundamentals of terrain
- ▶ How to sculpt terrain
- ▶ How to decorate your terrain with textures

In this hour, you learn about terrain generation. You learn what terrain is, how to create it, and how to sculpt it. You will also get hands-on experience with texture painting and fine-tuning. In addition, you learn to make large, expansive, and realistic-looking terrain pieces for your games.

Terrain Generation

All 3D game levels exist in some form of a world. These worlds can be highly abstract or realistic. Often, games with expansive “outdoor” levels are said to have a terrain. The term *terrain* refers to any section of land that simulates a world’s external landscape. Tall mountains, far plains, or dank swamps are all examples of possible game terrain.

In Unity, terrain is a flat mesh that can be sculpted into many different shapes. It may help to think of terrain as the sand in a sandbox. You can dig into the sand or raise sections of it up. The only thing basic terrain cannot do is overlap. This means that you cannot make things like caves or overhangs. Those items have to be modeled separately. Also, just like any other object in Unity, terrain has a position, rotation, and scale (although they aren’t usually changed).

Adding Terrain to Your Project

Creating a flat terrain in a scene is an easy task with some basic parameters. To add terrain to a scene, just click the menu items **GameObject > 3D Object > Terrain**. You will see that an object called Terrain has been added. If you navigate around in your Scene view, you may also notice that the terrain piece is very large. In fact, the piece is much larger than we could possibly need right now. Therefore, we need to modify some of the properties of this terrain.

To make this terrain more manageable, you need to change the terrain resolution. By modifying the resolution, you can change the length, width, and maximum height of your piece of terrain.

The reason the term *resolution* is used will become more apparent later when you learn about heightmaps. To change the resolution of the terrain piece, follow these steps:

1. Select your terrain in the Hierarchy view. In the Inspector view, locate and click the **Terrain Settings** button (see Figure 4.1).
2. Locate the Resolution settings.
3. Currently, the terrain width and length are set to 500. Set these values both to **50**.

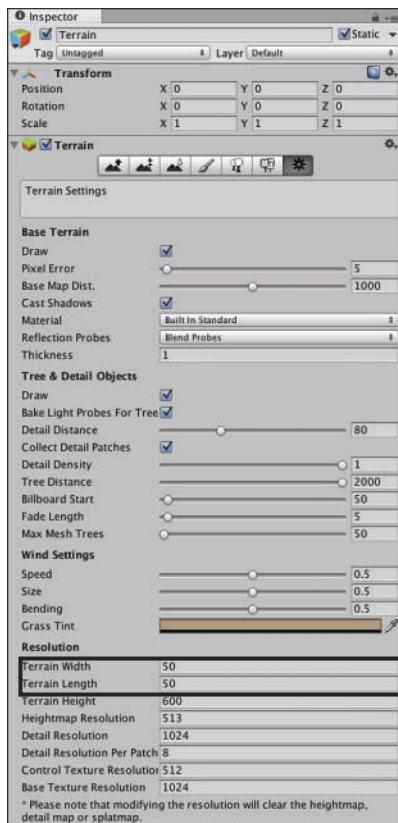


FIGURE 4.1

The Resolution settings.

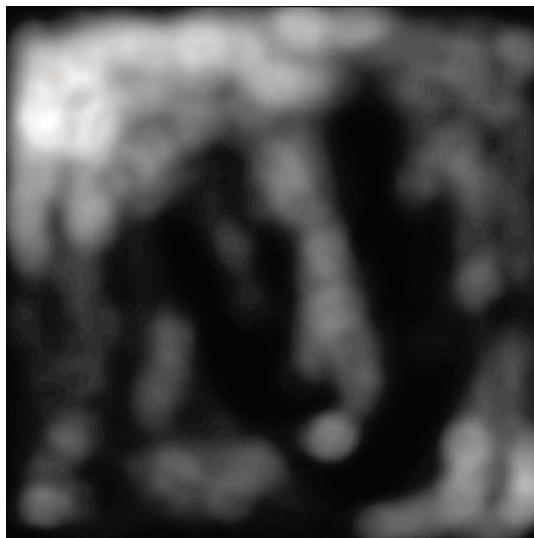
The other options in the Resolution settings modify how textures are drawn and the performance of your terrain. Leave these alone for now. After you change the width and length, you will see that the terrain is much smaller and manageable. Now it is time to start sculpting.

CAUTION**Terrain Size**

Currently, you are going to be working with terrain that is 50 units long and wide. This is purely for manageability while learning the various tools. In a real game, the terrain would probably be a bigger size to fit your needs. It is also worth noting that if you already have a heightmap (covered in the next section), you will want the terrain ratio (the ratio of length and width) to match the ratio of the heightmap.

Heightmap Sculpting

Traditionally, 256 shades of gray are available in 8-bit images. These shades range from 0 (black) to 255 (white). Knowing this, you can take a black-and-white image, often called a *grayscale* image, and use it as something called a *heightmap*. A heightmap is a grayscale image that contains elevation information similar to a topographical map. The darker shades can be thought of as low points, and the lighter shades are high points. Figure 4.2 is an example of a heightmap. It might not look like much, but a simple image like that can produce some dynamic scenery.

**FIGURE 4.2**

A simple heightmap.

Applying a heightmap to your currently flat terrain is simple. You simply start with a flat terrain and import the heightmap onto it, as follows.

▼ TRY IT YOURSELF

Applying a Heightmap to Terrain

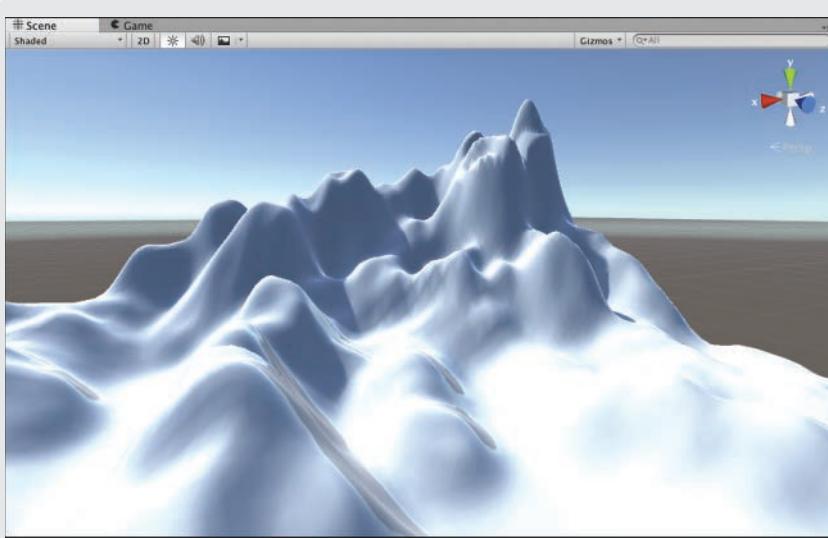
Let's walk through the steps of importing and applying a heightmap:

1. Locate the terrain.raw file in the Hour 4 files and put it somewhere you can easily find it.
2. With your terrain selected in the Hierarchy view, click the **Terrain Settings** button. (See Figure 4.1 if you don't remember where that is.) In the Heightmap section, click **Import Raw**.
3. The Import Raw Heightmap dialog will open. Locate the terrain.raw file from step 1 and click **Open**.
4. The Import Heightmap dialog will open (see Figure 4.3). Leave all options as they appear and click **Import**. Right about now, your terrain is looking strange. The problem is that when you set the length and width of your terrain to be more manageable, you left the height at 600. This is obviously much too high for your current needs.



FIGURE 4.3
Import Heightmap dialog.

5. Change the terrain resolution by going back to the Resolution section in the Terrain Settings in the Inspector view. This time, change the height value to **60**. The result should be something much more pleasant (see Figure 4.4).

**FIGURE 4.4**

Terrain after you import a heightmap.

TIP**Calculating Height**

So far, the heightmap might seem random, but it is actually quite easy to figure out. Everything is based on a percentage of 255 and the maximum height of the terrain. The max height of the terrain defaults to 600 but is easily changeable. If you apply the formula of $(\text{gray shade})/255 \times (\text{max height})$, you can easily calculate any point on the terrain.

For instance, black has a value of 0, and so any spot that is black will be 0 units high ($0/255 \times 600$). White has a value of 255 and therefore produces spots that are 600 units high ($255/255 \times 600$). If you have a medium gray with a value of 125, any spots of that color will produce terrain that is about 294 units high ($125/255 \times 600$).

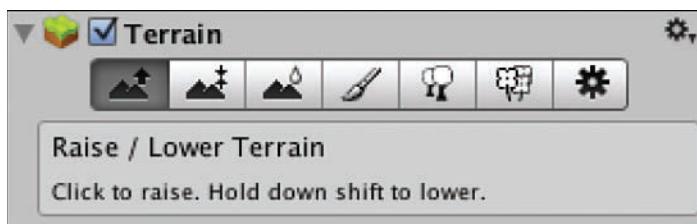
NOTE**Heightmap Formats**

In Unity, heightmaps must be grayscale images in the .raw format. There are many ways to generate these types of images; you can use a simple image editor or even Unity itself. If you create a heightmap using an image editor, try to make the map the same length and width ratio as your terrain. Otherwise, some distortion will be apparent. If you sculpt some terrain using Unity's sculpting tools and you want to generate a heightmap for it, you can do so by going to the Heightmap section in the Terrain Settings in the Inspector view and clicking **Export Raw**.

Generally for large terrains, or where performance is important, we recommend you export your heightmap and convert your terrain to a mesh in another program. You can then add caves, overhangs, etc., before re-importing to Unity.

Unity Terrain Sculpting Tools

Unity gives you multiple tools for hand sculpting your terrain. You can see these tools in the Inspector view under the component Terrain (Script). These tools all work under the same premise: You use a brush with a given size and opacity to “paint” terrain. In effect, what you are doing behind the scene is painting a heightmap that is translated into changes for the 3D terrain. The painting effects are cumulative, which means that the more you paint an area, the stronger the effect is on that area. Figure 4.5 shows these tools. Using these tools, you can generate pretty much any landscape you can imagine.

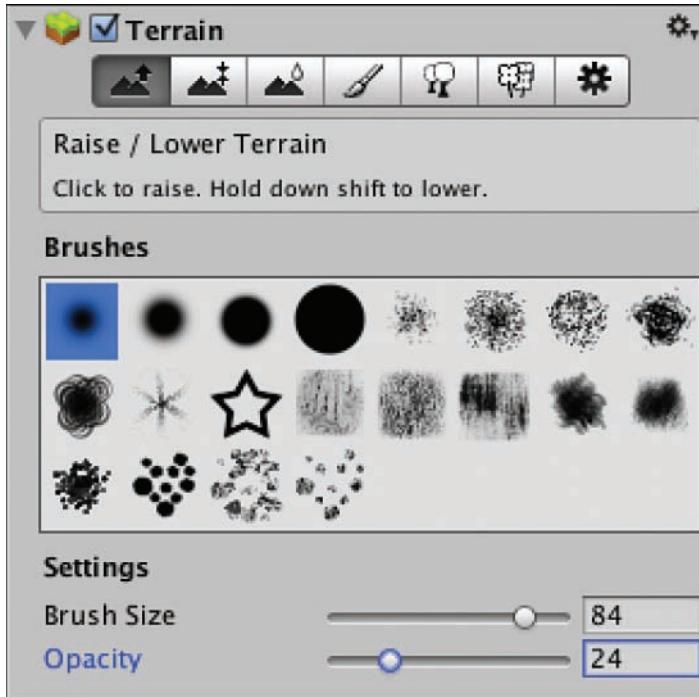
**FIGURE 4.5**

The terrain sculpting tools.

The first tool you will learn to use is the Raise/Lower tool. This tool, just as it sounds, enables you to raise or lower the terrain wherever you paint. To sculpt with this tool, follow these steps:

1. Select a brush. Brushes determine the size and shape of the sculpting effect.
2. Choose a brush size and opacity. The opacity determines how strong the sculpting effect is.
3. Click and drag over the terrain in the Scene view to raise the terrain. Holding Shift when you click and drag will instead lower the terrain.

Figure 4.6 illustrates some good starting options for sculpting given the terrain size 50×50 with a height of 60.

**FIGURE 4.6**

Good starting properties for sculpting.

The next tool is the Paint Height tool. This tool works almost exactly as the Raise/Lower tool except that it paints your terrain to a specified height. If the specified height is higher than the current terrain, painting raises the terrain. If the specified height is lower than the current terrain, however, the terrain is lowered. This proves useful for creating mesas or other flat structures in your landscape. Go ahead and try it out!

TIP**Flattening Terrain**

If, at any time, you want to reset your terrain back to being flat, you can do so by going to the Paint Height tool and clicking **Flatten**. One added benefit of this is that you can flatten the terrain to a height other than its default 0. If your maximum height is 60 and you flatten your heightmap to 30, you have the ability raise the terrain by 30 units, but you can also lower it by 30 units. This makes it easy to sculpt valleys into your otherwise flat terrain.

The final tool you will use is the Smooth Height tool. This tool doesn't alter the terrain in highly noticeable ways. Instead, it removes a lot of the jagged lines that appear when sculpting terrain. Think of this tool as a polisher. You will really only use it to make minor tweaks after your major sculpting is done.

TRY IT YOURSELF

Sculpting Terrain

Now that you have learned about the sculpting tools, let's practice using them. In this exercise, you attempt to sculpt a specific piece of terrain:

1. Create a new project or scene and add a terrain. Set the resolution of the terrain to 50×50 and give it a height of 60.
2. Flatten the terrain to a height of 20 by clicking the **Paint Height** tool, changing the height to **20**, and clicking **Flatten**.
3. Using the sculpting tools, attempt to create a landscape similar to Figure 4.7.
4. Continue to experiment with the tools to try to add unique features to your terrain.

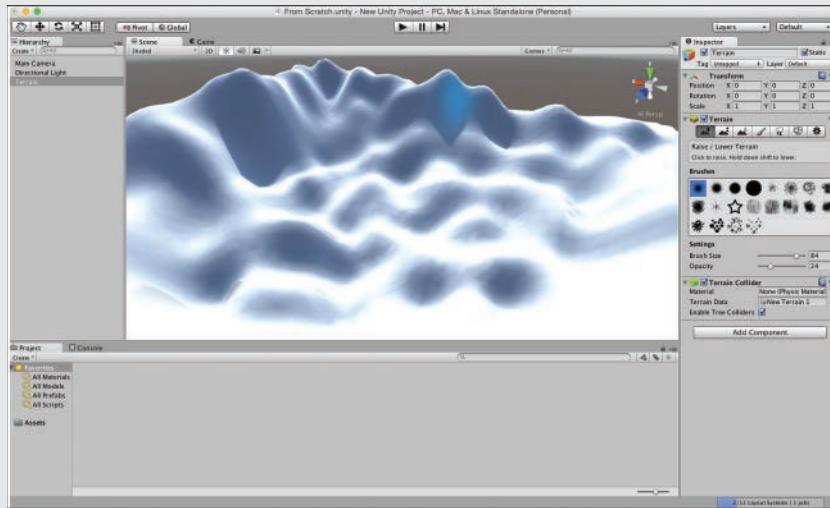


FIGURE 4.7
A sample terrain.

TIP

Practice, Practice, Practice

Developing strong, compelling levels is an art form in itself. Much thought has to be given to the placement of hills, valleys, mountains, and lakes. Not only do the elements need to be visually satisfying, but also they need to be placed in such a way as to make the level playable. This type of skill doesn't develop overnight. Be sure to practice and refine your level-building skills to make exciting and memorable levels.

Terrain Textures

You now know how to make the physical dimensions of a 3D world. Even though there may be a lot of features to your landscape, it is still bland and difficult to navigate. It is time to add some character to your level. In this section, you learn how to texture your terrain to give it an engaging look.

Importing Terrain Assets

Like sculpting terrain, texturing terrain works a lot like painting. You select a brush and a texture and paint it onto your world. Before you can begin painting the world with textures, however, you need some textures to work with. Unity has some terrain assets available to you, but you need to import them first. To load these assets, click **Assets > Import Package > Environment**. The Importing Package dialog will appear (see Figure 4.8). This dialog is where you specify

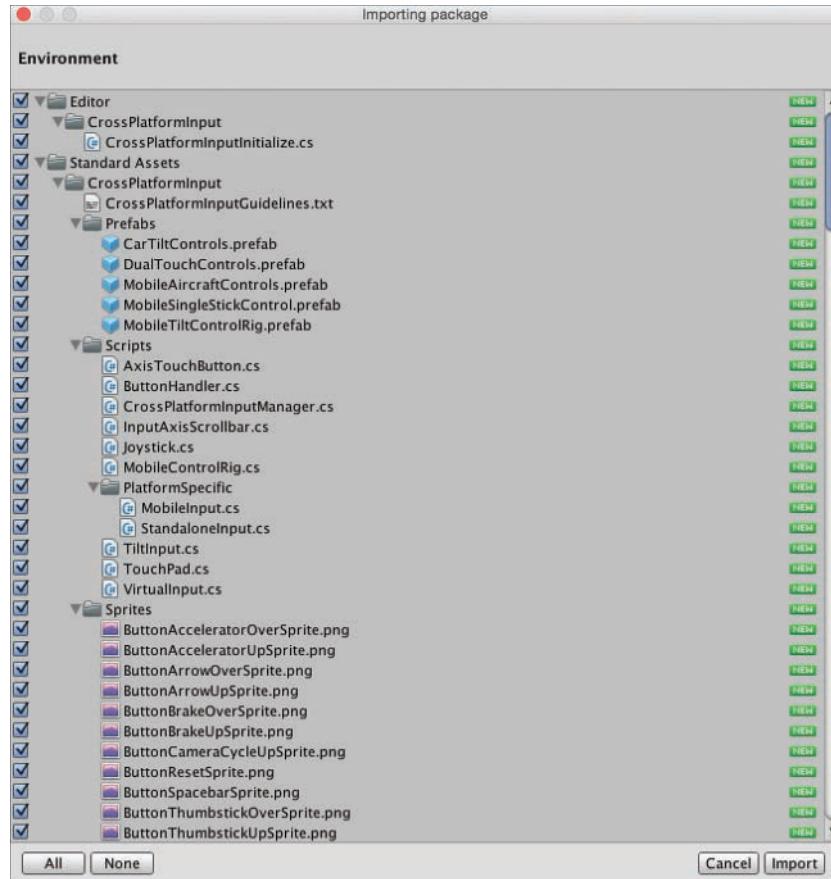


FIGURE 4.8

The Importing Package dialog.

exactly which assets you want to import. Deselecting unneeded items is a good idea if you want to keep your project size down. For now, just leave all options checked and click **Import**. You should now have a new folder under Assets in the Project view called Standard Assets. This folder contains all the terrain assets you will be using in the rest of this hour.

Texturing Terrain

The terrain texturing procedure is simple in Unity and works a lot like the sculpting. The first thing you need to do is load a texture. Figure 4.9 illustrates the texturing tool in the Inspector, which you access by selecting the Terrain in your Hierarchy. Pay attention to the three numeric properties: brush size, opacity, and target strength. You should be familiar with the first two properties, but the last one is new. The target strength is the maximum opacity that it achievable through constant painting. Its value is a percentage, with 1 being 100%. Use this as a control to prevent painting your textures on too strongly.

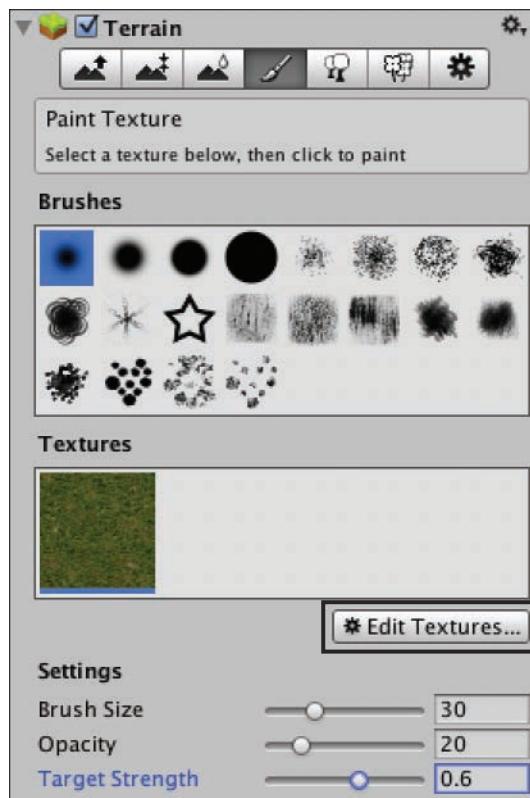


FIGURE 4.9

Terrain texture tool and properties.

To load a texture, follow these steps:

1. Click **Edit Textures > Add Texture** in the Inspector (not the Unity menus).
2. The Add Terrain Texture dialog will appear. Click **Select** in the Texture box (see Figure 4.10) and select the **GrassHillAlbedo** texture.
3. Click **Add** (there is no need to add a “normal map”).

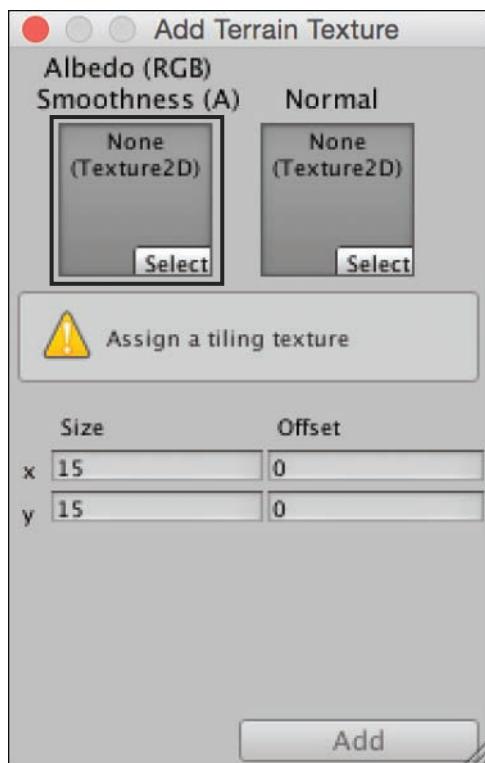


FIGURE 4.10

The Add Terrain Texture dialog.

At this point, your entire terrain should be covered in patchy grass. This looks better than the white terrain previously, but it is still far from realistic. Now, you will actually begin painting and making your terrain look better.

TRY IT YOURSELF

Painting Textures onto Terrain

Let's apply a new texture to your terrain to give it a more realistic two-tone effect:

1. Using the steps listed earlier, add a new texture. This time, load the **GrassRockyAlbedo** texture. Once you have loaded it, be sure to select it by clicking it. (A blue bar appears under it if it is selected.)
2. Set your brush size to **30**, your opacity to **20**, and your target strength to **0.6**.
3. Sparingly, paint (click and drag) on the steep parts and crevices of your terrain. This gives the impression that grass isn't growing on the sides of steep grades and in between hills (see Figure 4.11).
4. Continue experimenting with texture painting. Feel free to load the texture **CliffAlbedoSpecular** and apply it to steeper parts or the texture **SandAlbedo** and make a path.

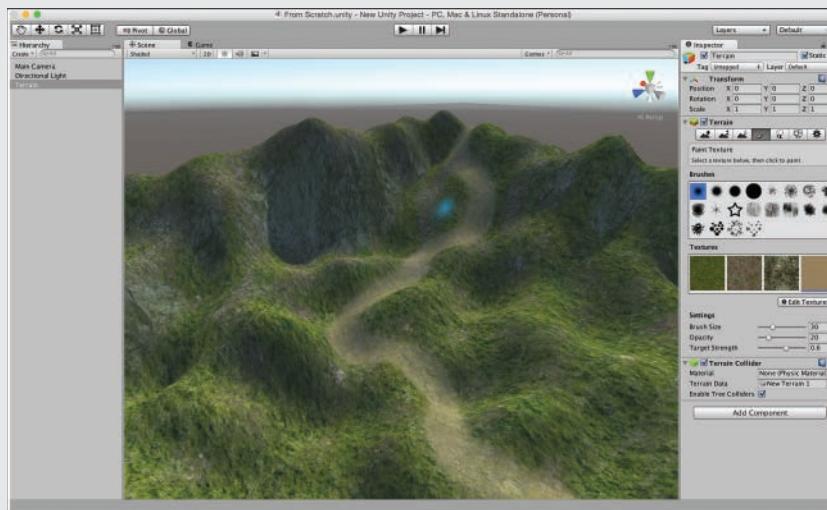


FIGURE 4.11

Example of a two-toned textured cliff, with a sand path.

You can load as many textures as you want in this fashion and achieve some realistic effects. Be sure to practice texturing to determine the best-looking patterns.

NOTE**Creating Terrain Textures**

Game worlds are often unique and require custom textures to fit within the context of the games they are created for. You can follow some general guidelines when making your own textures for terrain. The first is to always try to make the pattern repeatable. This means that the textures can be tiled seamlessly. The larger the texture, the less obvious a repeating pattern is. The second guideline is to make the texture square. The last guideline is to try to make the texture dimension a power of 2 (32, 64, 128, 512, and so on). The last two guidelines effect the compression of the texture and the texture's efficiency. With a little practice, you will be making brilliant terrain textures in no time.

TIP**Subtlety Is the Best Policy**

When texturing, remember to keep your effects subtle. Most of nature fades from one element to another without many harsh transitions. Your texturing efforts should also reflect that. If you can zoom out away from a piece of terrain and tell the exact point where one texture starts, your effect is too sudden. It is often better to work in many small and subtle applications of a texture rather than with one broad application.

Summary

In this hour, you learned all terrains in Unity. You started by learning what terrains are and how to add them to your scene. From there, you looked at sculpting the terrain with both a height-map and Unity's built-in sculpting tools. Finally, you learned how to make your terrains look more appealing by applying textures in a realistic fashion.

Q&A

Q. Does my game have to have terrain?

A. Not at all. Many games take place entirely inside modeled rooms or in abstract spaces.

Q. My terrain doesn't look very good. Is that normal?

A. It takes a while to build up proficiency with the sculpting tools. With some practice, your levels will begin looking much better. True quality comes from play testing a level, which we cover in the next hour.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. True or False: You can make caves out of the Unity terrain.
2. What is a grayscale image containing terrain elevation information called?
3. True or False: Sculpting terrain in Unity is a lot like painting.
4. How do you access Unity's available terrain textures?

Answers

1. False. Unity's terrain cannot overlap.
2. A heightmap.
3. True.
4. You import the terrain assets by going to **Assets > Import Package > Environment**.

Exercise

Let's practice terrain sculpting and texturing. Sculpt a terrain that contains the following elements:

- ▶ Lakebed
- ▶ Beach
- ▶ Mountain range
- ▶ Flat plains

Once you have sculpted these, apply textures to your terrain in the following manner. You can find all textures listed here in the Terrain Assets package:

- ▶ The beach should use the texture SandAlbedo and should fade into GrassRockyAlbedo.
- ▶ Plains and all flat areas should be textured with GrassHillAlbedo.
- ▶ The texture GrassHillAlbedo should smoothly transition into GlassRockyAlbedo as the terrain gets steeper.
- ▶ The texture GlassRockyAlbedo should transition into Cliff at its steepest and highest points.

Be as creative as you want with this exercise. Build a world that makes you proud.

HOUR 5

Environments

What You'll Learn in This Hour:

- ▶ How to add trees and grass to your terrain
- ▶ How to add environment effects to your terrain
- ▶ How to navigate your world with a character controller

In the preceding hour, you learned to sculpt and texture terrain for your game. In this hour, you add environment effects that will really give character to your world. You start by learning how to add vegetation like trees and grass to your terrain. From there, you learn to apply environment effects like water, sky, fog, and lens flares. You finish by adding a character controller to your scene and moving around inside your world.

Generating Trees and Grass

A world with only flat textures would be boring. Almost every natural landscape has some form of plant life. In this section, you learn how to add and customize trees and grass to give your terrain an organic look and feel.

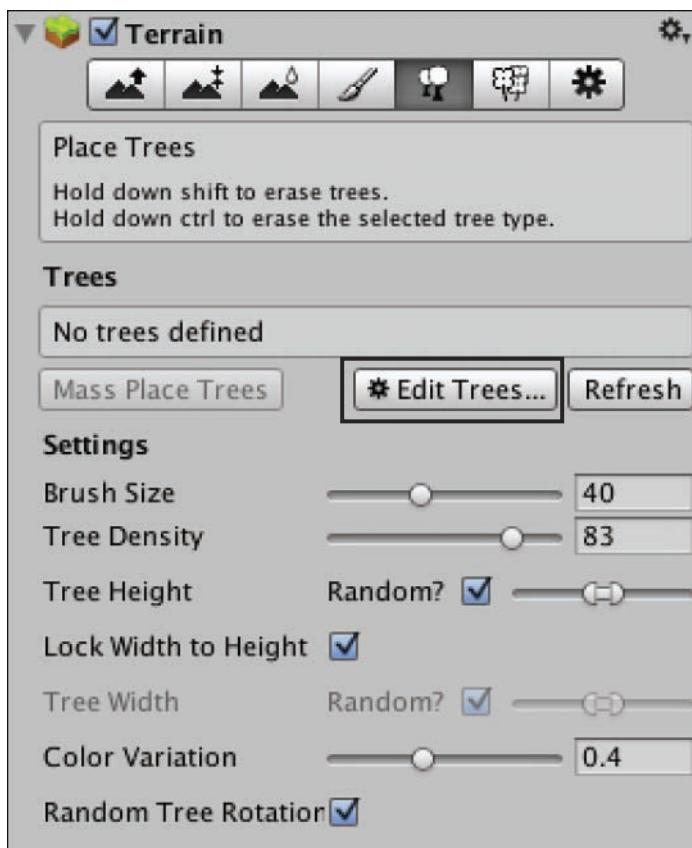
Painting Trees

Adding trees to your terrain works just like the sculpting and texturing from the preceding hour; the whole process is very similar to painting. The basic premise is to load a tree model, set the properties for the trees, and then paint the area you want the trees to appear in. Based on the options you choose, Unity will spread the trees out and vary them to give a more natural and organic look.

TIP**Terrain Assets**

To follow along with the rest of this section, you need the Environment asset package loaded into your project. If you do not have it, refer to the preceding hour for instructions on how to import it into your project.

You use the Place Trees tool to spread trees out over the terrain. Once the terrain has been selected in the scene, the Place Trees tool is accessed in the Inspector view as part of the Terrain (Script) component. Figure 5.1 shows the Place Trees tool and its standard properties.

**FIGURE 5.1**

The Place Trees tool.

Table 5.1 describes the tree tool's properties.

TABLE 5.1 The Place Tree Tool's Properties

Property	Description
Brush Size	The size of the area that trees are added to when painting.
Tree Density	How densely the trees will be able to be packed.
Tree Height/Width, Color Variation, etc.	These properties allow all the trees to be slightly different from each other. This gives the impression of many different trees instead of the same tree repeated.

TRY IT YOURSELF ▼

Placing Trees on a Terrain

Let's walk through the steps to place trees onto your terrain using the Paint Trees tool. This exercise assumes that you have created a new scene and have already added a terrain. The terrain should be set to a length and width of 100. It will look better if the terrain has some sculpting and texturing done already:

1. Click **Edit Trees > Add Tree** to pull up the Add Tree dialog (refer to Figure 5.1).
2. Clicking the circle icon to the right of the Tree text box on the Add Tree dialog pulls up the Tree Selector dialog (see Figure 5.2).

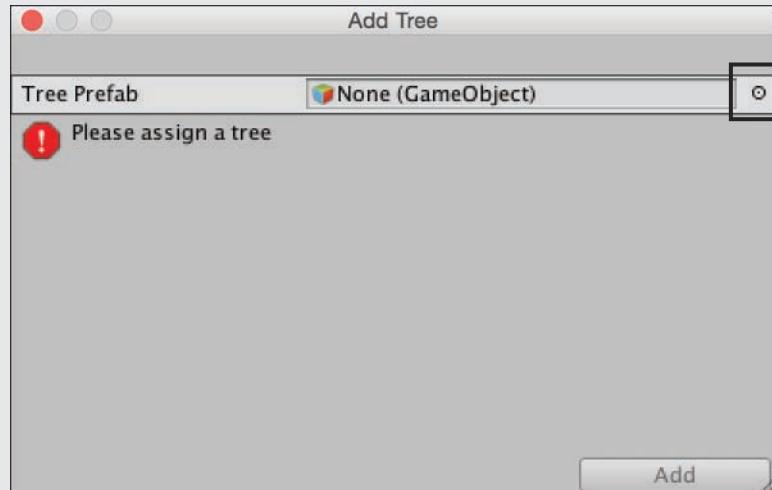


FIGURE 5.2
The Add Tree dialog.

- ▼
3. Select the **Palm/Desktop** and click **Add**.
 4. Set your brush size to **10**, your tree density to **70**. Leave the tree height/width set to Random, and choose whichever variation properties you want.
 5. Paint trees on the terrain by clicking and dragging over the areas where you want trees. Holding the Shift key while click-dragging removes trees. If you can't paint, go to Terrain Settings > Tree & Detail Objects and check the Draw checkbox is ticked.
 6. Continue to experiment with different brush sizes, densities, and tree sizes/variations.

Painting Grass

Now that you have learned how to paint trees, you learn how to apply grass or other small plant life to your world. Grass and other small plants are called *details* in Unity. Therefore, the tool used to paint grass is the Paint Details tool. Unlike trees, which are 3D models, details are billboards (see note below). Just like you have seen over and over by now, details are applied to a terrain using a brush and a painting motion. Figure 5.3 illustrates the Paint Details tool and some of its properties.

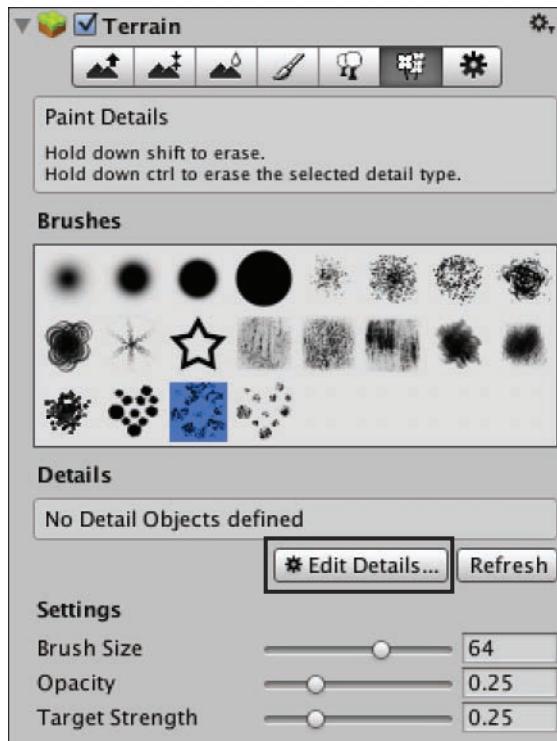


FIGURE 5.3

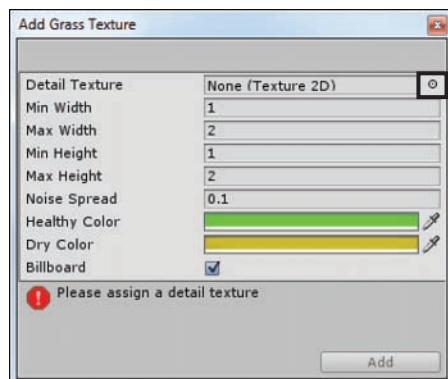
The Paint Details tool.

NOTE**Billboards**

Billboards are a special type of visual component in a 3D world that give the effect of a 3D model without actually being a 3D model. Models exist in all three dimensions. Therefore, when moving around one, you can see the different sides. Billboards, however, are flat images that always face the camera. When you attempt to go around a billboard, the billboard turns to face your new position. Common uses for billboards are grass details, particles, and onscreen effects.

Applying grass to your terrain is a fairly straightforward process. You first need to add a grass texture. To add a grass texture:

1. Click **Edit Details** in the Inspector view and select **Add Grass Texture**.
2. In the Add Grass Texture dialog, click the **circle** icon next to the **Texture** text box (see Figure 5.4). Select the **GrassFrond01AlbedoAlpha** texture. You can search for grass to help you find it.

**FIGURE 5.4**

The Add Grass Texture dialog.

3. Set your texture properties to whatever values you want. Pay special attention to the color properties because those establish the range of natural colors for your grass.
4. When done, click **Add**.

After you have your grass loaded, you just need to choose a brush and your brush properties. You are now ready to begin painting grass.

TIP**Realistic Grass**

You may notice that when you begin painting grass it does not look realistic. You need to focus on a few things when adding grass to your terrain. The first is to pay attention to the colors you set for the grass texture. Try to keep them darker and more earth toned. The next thing you need to do is choose a nongeometric brush shape to help break up hard edges (refer to Figure 5.3 for a good brush to use). Finally, keep your opacity and target strength properties very low. A good setting to start with is .02 for each. If you need more grass, you can just keep painting over the same area. You can also go back to Edit Details and change the grass texture properties.

CAUTION**Vegetation and Performance**

The more trees and grass you have in a scene, the more processing is required to render it. If you are concerned with performance, you need to keep the amount of vegetation low. There are some properties that you look at later this hour that will help you manage this, but as an easy rule, try to add trees and grass only to areas where it is really needed.

TIP**Disappearing Grass**

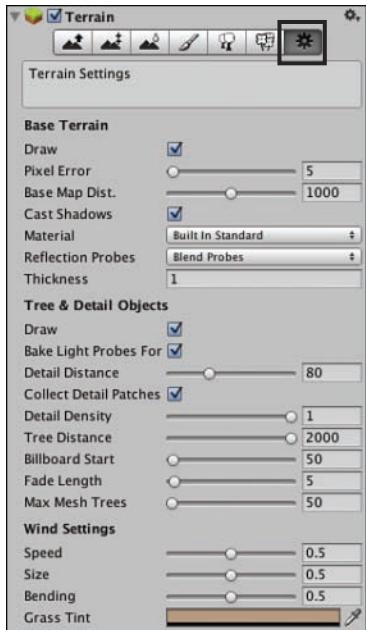
As with trees, grass is affected by its distance from the viewer. Whereas trees revert to a lower quality when the viewer is far away, grass is just not rendered. The result is a ring around the viewer beyond which no grass is visible. Again, you can modify this distance by properties you look at later this hour.

Terrain Settings

The last button on this list of terrain tools in the Inspector view is for the Terrain Settings tool.

These settings control how the terrain, texture, trees, and details look and function overall.

Figure 5.5 shows all the terrain settings.

**FIGURE 5.5**

The Terrain Settings tool.

The first grouping of settings is for the overall terrain. Table 5.2 describes the various settings.

TABLE 5.2 Base Terrain Settings

Setting	Description
Pixel Error	Number of allowable errors when displaying terrain geometry. The higher the value, the lower the detail of the terrain.
Base Map Dist.	The maximum distance that high-resolution textures will be displayed. When the viewer is farther than the given distance, textures degrade to a lower resolution.
Cast Shadows	Determines whether terrain geometry casts shadows.
Material	This slot is for assigning a custom material capable of rendering terrain. The material must contain a shader capable of rendering terrain.
Reflection Probes	How reflection probes are used on terrain. Only effective when using built-in standard material or a custom material that supports rendering with reflection. An advanced setting we won't be covering in this text.
Thickness	How much the terrain collision volume should extend along the negative y axis. Objects are considered colliding with the terrain from the surface to a depth equal to the thickness. This helps prevent high-speed moving objects from penetrating into the terrain without using expensive continuous collision detection.

In addition, some settings directly affect the way trees and details (like grass) behave in your terrain. Table 5.3 describes these settings.

TABLE 5.3 Tree and Detail Object Settings

Setting	Description
Draw	Determines whether trees and details are rendered in the scene.
Bake Light Probes For	An advanced performance setting related to making real-time lighting more realistic and efficient.
Detail Distance	The distance from the camera where details will no longer be drawn to the screen.
Collect Detail Patches	Preloads the terrain details to prevent hiccups when moving around terrain, at the cost of memory usage.
Detail Density	The number of detail/grass objects in a given unit of area. The value can be set lower to reduce rendering overhead.
Tree Distance	The distance from the camera where trees will no longer be drawn to the screen.
Billboard Start	The distance from the camera where 3D tree models will begin to transition into lower-quality billboards.
Fade Length	The distance over which trees will transition between billboards to higher-quality 3D models. The higher the setting, the smoother the transition.
Max Mesh Trees	The total number of trees able to be drawn simultaneously as 3D meshes and not billboards.

The next settings are for the wind. Because you haven't had a chance to actually run around inside your world yet (you will later this hour), you might be wondering what that means. Basically, Unity simulates a light wind over your terrain. This light wind causes the grass to bend and sway and livens up the world. Table 5.4 describes the wind settings. The Resolution settings were briefly introduced in the last hour.

TABLE 5.4 Wind Settings

Setting	Description
Speed	The speed, and therefore the strength, of the wind effect.
Size	The size of the area of grass affected by the wind at the same time.
Bending	The amount of sway the grass will have due to wind.
Grass Tint	Although not a wind setting, this setting controls the overall coloration of all grass in your level.

Environment Effects

At this point, you have sculpted, textured, and added trees and grass to your terrain. It is safe to say that it is looking much better than when it was just a flat white square. In this section, you learn about adding environment details to really make your game world as complete as possible.

Skyboxes

You might have noticed that while your terrain is full of texture and detail, the sky is a bland solid color. What you need to do is to add a skybox to your world. A skybox is a large box that goes around your world. Even though it is a cube consisting of six flat sides, it has inward-facing textures to make it look round and infinite. You can create your own skyboxes or use one of Unity's standard skyboxes. In this book, you will use the built-in ones.

To use the standard skyboxes, you need to import the Environment asset package into your project, if you haven't already.

There are two ways to add skyboxes to your world: You can add the skybox to your camera or add it to the environment using the Lighting window.

Adding a Skybox to the Camera

You can add a skybox to your camera so that whatever the camera sees beyond your game world will be replaced with sky. A camera skybox overrides any scene skybox, when viewing the game through that camera. To add a skybox to your camera, follow these steps:

1. Select the **Main Camera** in the Hierarchy view.
2. Add a skybox component by clicking **Component > Rendering > Skybox**.
3. In the Inspector view, locate the Skybox component and click the **circle** icon next to the **Custom Skybox** field (see Figure 5.6). In the Select Material dialog, select the **Default-Skybox**.
4. Run your scene to see the skybox applied to the camera.



FIGURE 5.6
The Skybox component.

NOTE

Multiple Skyboxes

The reason there is an option to add the skybox to a specific camera is so that you can have different skyboxes (or no skyboxes) on different cameras. This enables you to make the world look different to different viewers. If you want to have flexibility in the way your world looks, add the skybox to the camera. If you want your world to look uniform to everyone, add it to the scene (as covered next).

Changing the Default Scene Skybox

By default, scenes come with the Skybox setting set to Default-Skybox. To change this for the entire scene, follow these steps:

1. Drag the **Stars1024** folder from the book files into your Assets window.
2. Click **Window > Lighting** to open the lighting tab.
3. Locate the Skybox field and click the circle icon to the right of it (see Figure 5.7).

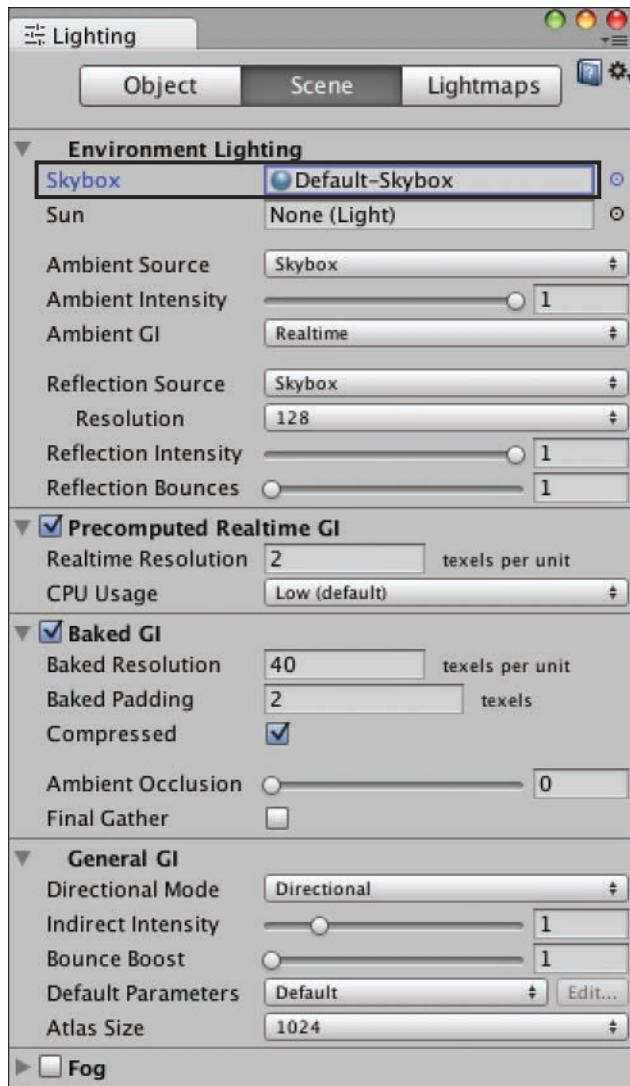


FIGURE 5.7

Choosing a skybox for the scene.

4. Choose the **Stars1024** we just imported. Notice how the Scene view changes, but the game view doesn't change if you have set a specific camera skybox.

Fog

In Unity, you can add fog to a scene. You can use this fog to simulate many different natural occurrences, such as haze, an actual fog, or the fading of objects over great distances. You can also use fog to give new and alien appearances to your world.

TRY IT YOURSELF ▼

Adding Fog to a Scene

Let's add fog to your scene and learn about the different properties that affect it:

1. Click **Window > Lighting**. The lighting window will open.
2. Turn on fog by checking the **Fog** check box (see the bottom of Figure 5.7).
3. Experiment with the different fog densities and colors. Table 5.5 describes the various fog properties.

Several properties impact how fog looks in a scene. Table 5.5 describes these properties.

TABLE 5.5 Fog Properties

Setting	Description
Fog Color	The color of the fog effect.
Fog Mode	This controls how the fog is calculated. The three modes are Linear, Exponential, and Exponential Squared. For mobile, Linear works best.
Density	How strong the fog effect is. This property is used only if the fog mode is set to Exponential or Exponential Squared.
Start & End	These control how close to the camera the fog starts and how far from the camera it ends. These properties are only used in Linear mode.

Lens Flares

A lens flare is a visual deformity that occurs whenever a camera looks at a bright light source. It is the result of light bouncing around inside the glass of a lens. A lens flare can also be experienced when you attempt to look into a bright source like the sun (not recommended). In Unity, you can add flares to light sources to give them a more realistic effect and make it seem like they are very bright.

▼ TRY IT YOURSELF

Adding a Lens Flare to Your Scene

It will be easier to see how lens flares are placed in a scene if you follow along step by step. Adding a flare is pretty simple, but it uses some new items you might not be familiar with yet. Before you can add a flare to your scene, you need to have a light source and some flare assets. These will all be taken care of in the following steps:

1. If there isn't already directional light in your scene, add one by clicking **GameObject > Light > Directional Light**. Directional lights are covered in greater detail in a later hour. For now, just understand that a directional light is a parallel light, just like the sun.
2. Once the light is added to your scene, rotate it so that it gives the desired light effect on your terrain.
3. Next, you need light flare assets. Import the Unity light flare assets by clicking **Assets > Import Package > Effects**. In the Import dialog, leave everything checked and click **Import**.
4. Select the directional light in the Hierarchy view and locate the Flare property in the Inspector view.
5. Click the **circle** icon next to the Flare property and choose the **50mm Zoom** flare from the Select Flare dialog.

At this point, your flare is on the light and will be picked up by the camera. This is because the Main Camera of the scene has a Flare Layer component by default. Any cameras without that component will not be able to see the lens flares.

TIP

Where's the Flare?

You might not be able to see the lens flare yet because your camera is not pointed at your directional light. Don't bother trying to make the camera point at the light just yet. However, you can rotate the Scene view to look towards the sun, and Unity 5 will preview the lens flare effect for you. If you can't see it, check your Game Overlay settings in the Scene window (refer to Figure 1.11, from Hour 1).

Water

The last environment effect you look at adding is water. In Unity, water is an asset that needs to be imported, and with Unity 5 you get great-looking water for free. In the scene, water is a flat plane that looks like the top surface area of a pond or lake. Note that the water is just an effect. If a player jumps into a lake with water, the player will fall right through the water and down into the hole that was sculpted for it.

TRY IT YOURSELF ▼

Creating a Lake and Adding Water

To add water, you need some part of your terrain to contain the water. In this exercise, you sculpt a lake and add water to it:

1. Create a new terrain or work with an existing terrain. Sculpt a lakebed down into the terrain.
2. Water is part of the **Environment** package, so import this if you haven't already.
3. Locate the **Environment > Water > Water > Prefabs** folder (see Figure 5.8).
4. Drag the **WaterProDaytime** asset onto the Scene view and into the lakebed you created. Scale and move the water as necessary until it fills up the lakebed properly.

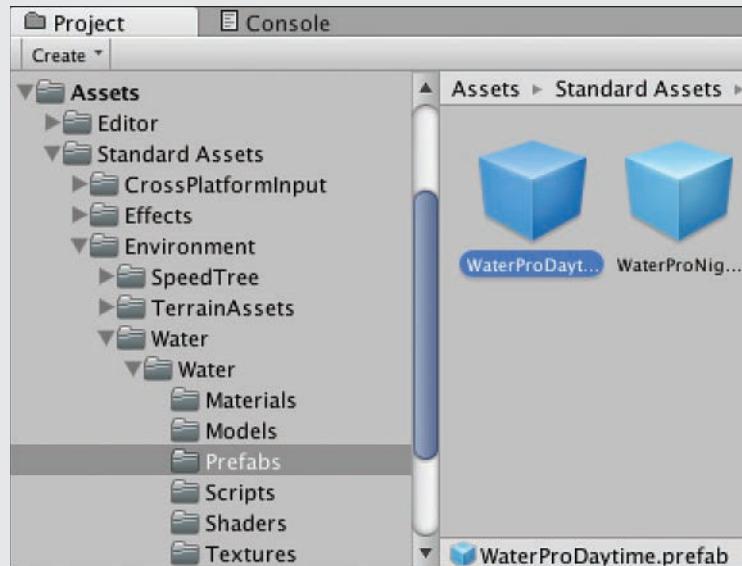


FIGURE 5.8
The Water (Basic) folder and assets.

Character Controllers

At this point, you have finished your terrain. It has been sculpted and textured; had trees and grass added and has been given a sky; and it has a fog, lens flare, and water. It is now time to get into your level and “play” it. Unity provides two basic character controllers to easily get right into your scene without a lot of work on your end. Basically, you drop a controller into your scene and then move around with the control scheme common to most first-person games.

Adding a Character Controller

To add a character controller to your scene, you first need to import the asset. Click **Assets > Import Package > Characters**. In the Import Package dialog, leave everything checked and click **Import**. A new folder named **Characters** should have been added to your Project view under the Standard Assets folder. Because you don't have a 3D model to use as the player, we are going to use the first-person controller. Locate the **FPSController** asset in the Character Controllers folder (see Figure 5.9) and drag it onto your terrain in the Scene view.

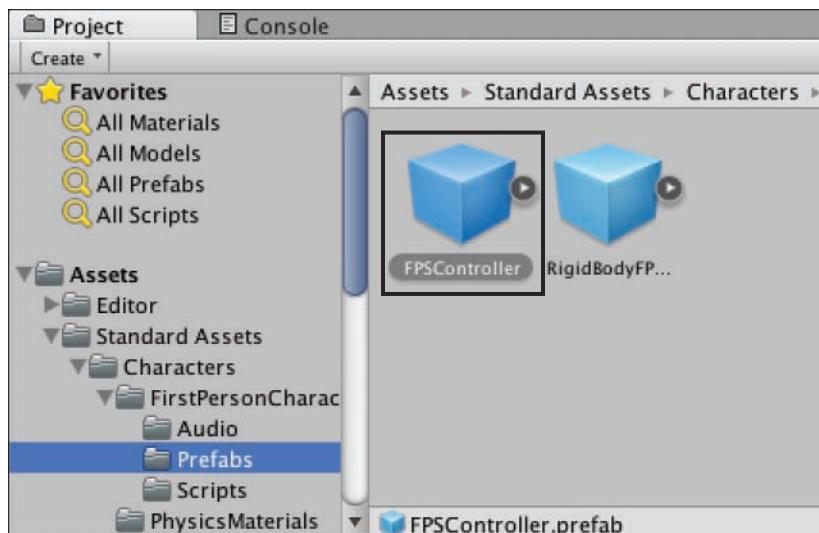


FIGURE 5.9

The First Person character controller.

Now that the character controller has been added to your scene, you can move around in the terrain you created. When you play your scene, you will notice that you can now see from where the controller was placed. You can use the WASD keys to move around, the mouse to look around, and the spacebar to jump. Play around with the controls if they feel a bit unusual to you and enjoy experiencing your world!

If you select the **FPSController** in the hierarchy, you will see a lot of settings in the inspector. One of these settings is **Slope Limit**, which determines how steep a hill the character can climb. Try changing this and other values in the Inspector and see the range of behavior you can achieve with this powerful built-in component. Table 5.6 describes all the properties of the character controller.

TABLE 5.6 Character Controller Properties

Property	Description
Slope Limit	Determines the steepest slope a controller can climb. Any slopes steeper than the value indicated here will be impassable to the controller.
Step Offset	Any steps closer to the ground than the specified value will be climbable. Any steps further than the value specified will be impassable
Skin Width	Determines how deeply a collider can penetrate the controller's collider before a collision is detected. Too little width causes the controller to jitter. Too much causes the controller to get stuck. A good general setting is 10% of the controller's radius.
Min Move Distance	Determines the minimum distance a controller can be told to move. This can be used to reduce jitter, but setting it too high can cause the controls to feel unresponsive. A good rule is to set this at 0 and only increase it if needed.
Center	The center of the capsule collider belonging to the character controller.
Radius	The radius of the capsule collider belonging to the character controller.
Height	The height of the capsule collider belonging to the character controller.

TIP**“2 Audio Listeners”**

When you added the character controller to the scene, you might have noticed a message at the bottom of the editor that said, “There are 2 audio listeners in the scene.” This is because the Main Camera (the camera that exists by default) has an audio listener component and so does the character controller that you added. Because the cameras represent the player’s perspective, only one can listen for audio. You can fix this by removing the audio listener component from the Main Camera. You can even delete the Main Camera game object altogether if you wish, as the FPSController has its own camera.

TIP**Falling Through the World**

If you find the camera falling through the world whenever you run your scene, chances are that your character controller is stuck partially in the ground. Try raising your character controller up a little bit above the ground. When the scene starts, the camera should fall just a little bit until it hits the ground and stops.

NOTE**Importing Assets**

In this hour, you imported a lot of asset packages. When you imported them, you left everything checked in the Import Package dialog. This caused every asset in that package to be added to your project. Doing this can make the project files very large, although Unity is smart enough to only include assets you actually use in the final game build. Still, it's a good habit to only import the assets that you need to use, as this will speed up Unity, especially when you come to switch your target platform (e.g., to mobile or console). You can simply uncheck all the assets that you don't need. Remember, you can always import them later if needed!

Fixing Your World

Now that you can enter your world and see it close up, it is time to refine some of the smaller details. You might notice that some areas that you built to be a path are too steep to walk on. You may also see some areas where the textures are not placed quite right. Now is the time to smooth out the world and fix any errors you find. It can be difficult to see all the places that need to be fixed from the Scene view. It is not until you are on the ground moving around your world that you really get a chance to experience it.

One fix that is worth looking at is the lens flare added previously. This was added to the Directional Light, and showed up in the old Main Camera because the camera had a Flare Layer component on it. Inspect the FirstPersonCharacter object, which is a child of the FPSController object. You will notice it has a camera, but no Flare Layer component. Add Component > Flare Layer to the FirstPersonCharacter game object, and you'll be back in business. Cross one more detail off your list.

Summary

In this hour, you learned all about environment details in Unity. You started by learning to add trees and grass to your scene. Next, you added ambient effects like the sky, fog, and lens flares. From there, you worked with Unity's water assets. You finished this hour by adding a character controller to your scene and actually playing around in your world.

Q&A

Q. Do trees and grass greatly impact performance?

A. It depends on how much of it you have on scene at once. It also depends on the power of the computer you are running it on. A good rule is to have grass and trees if they positively impact your scene.

Q. Can I make my own skyboxes?

A. Yes, you can. You should consider it mostly if you are building a custom world or a world with specific details not present in the available skyboxes.

Q. There are a lot of properties to the character controller. Will I need to know them all?

A. Not really. The character controllers are easy to use as they are. If you need to make easy changes to the movement, you can do that, but it shouldn't be necessary for most uses.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What setting controls how much trees sway in the wind?
2. What's the name of the ambient effect that can simulate haze or fading of colors at a distance?
3. This object is a cube that fits around your world and is textured to look like a sky.
4. What character controller did you add to your scene: First Person or Third Person?

Answers

1. That's a trick question. Trees don't sway in the wind, grass does. The setting that controls it is Bending (under Wind Settings).
2. Fog.
3. A skybox.
4. First Person.

Exercise

In this exercise, you have a chance to finish the terrain you began making at the end of Hour 4, “Terrain.” You will be adding the rest of the environmental effects to give your world a better level of realism.

Open the project or scene with the terrain you created in Hour 4. You need to do the following to it:

- ▶ Add water to the lakebed you created.
- ▶ Add some sparse grass around the edge of the lakebed and more grass over the flat plains.

- ▶ Add some palm trees to the grassy area where the grass texture meets the sand texture of the beach.
- ▶ Add a skybox to your scene.
- ▶ Add a fog effect to your scene. Change the settings so that it realistically makes the mountain range look cloudy.
- ▶ Add a directional light and a lens flare to the light. Ensure that the light lines up with a sun image if it is present in the skybox.
- ▶ Add a character controller and test drive your level. Ensure that everything is properly placed and looks realistic.

HOUR 6

Lights and Cameras

What You'll Learn in This Hour:

- ▶ How to work with lights in Unity
- ▶ The core elements of cameras
- ▶ How to work with multiple cameras in a scene
- ▶ How to work with layers

In this hour, you learn to use lights and cameras in Unity. You start by looking at the main features of lights. You then explore the different types of lights and their unique uses. Once you are finished with lights, you begin working with cameras. You learn how to add new cameras, place them, and generate interesting effects with them. You finish by learning about layers in Unity.

Lights

In any form of visual media, lights go a long way in defining how it is to be perceived. Bright, slightly yellow light can make a scene look sunny and warm. Take the same scene and give it a low-intensity blue light, and it will look eerie and disconcerting. The color of the lights will also mix with the color of the skybox, to give even more realistic looking results.

Most scenes that strive for realism or dramatic effect employ at least one light (and often many). In the past, you have briefly worked with lights to highlight other elements. In this section, you work with lights more directly.

NOTE

Repeat Properties

The different lights share many of the same properties. If a light has a property that has already been covered under a different light type, it won't be covered again. Just remember that if two different light types have properties with the same names, those properties do the same thing.

NOTE**What Is a Light?**

In Unity, lights are not objects themselves. Instead, lights are a component. This means that when you add a light to a scene, you are really just adding a game object with the Light component. This light component can be any of the types of light you can use.

Point Lights

The first light type you will be working with is the point light. Think of a point light as a light bulb. All light is emitted from one central location out in every direction. The point light is also the most common type of light for illuminating interior areas.

To add a point light to a scene, click **GameObject > Light > Point Light**. Once in the scene, the point light game object can be manipulated just like any other. Table 6.1 describes the point light properties.

TABLE 6.1 Point Light Properties

Property	Description
Type	The Type property is the type of light that the component gives off. Because this is a point light, the type should be Point. Changing the Type property changes the type of light it is.
Baking	Determines when the effect of this light on surrounding textures is calculated. See the Note about baking below for more details.
Range	The Range property dictates how far the light shines. Illumination will fade evenly from the source of light to the range dictated.
Color	The color the light shines. Color is additive, which means that if you shine a red light on a blue object, it will end up purple.
Intensity	The Intensity property dictates how brightly a light will shine. Note that the light will still shine only as far as the Range property dictates.
Bounce Intensity	Unity 5 supports Global Illumination, meaning it calculates the results of bouncing light. This control determines how bright the light is after it bounces off objects.
Shadow Type	The Shadow Type property is how shadows are calculated for this source in a scene. Hard shadows are more accurate but more performance intensive.
Cookie	The Cookie property accepts a cubemap (like a skybox) that dictates a pattern for the light to shine through. Cookies are covered in more detail later.
Draw Halo	The Draw Halo toggle determines whether a glowing halo will appear around your light. Halos are covered in more detail later.

Property	Description
Flare	The Flare property accepts a light flare asset and simulates the effect of a bright light shining into a camera lens. You have worked with light flares in previous hours to simulate a sun effect.
Render Mode	The Render Mode property determines the importance of this light. The three settings are Auto, Important, and Not Important. An important light is rendered in higher quality, whereas a less-important light is rendered more quickly. Use the Auto setting for now.
Culling Mask	The Culling Mask property determines what layers are affected by the light. By default, everything is affected by the light. Layers are covered in detail later.

NOTE

Baking

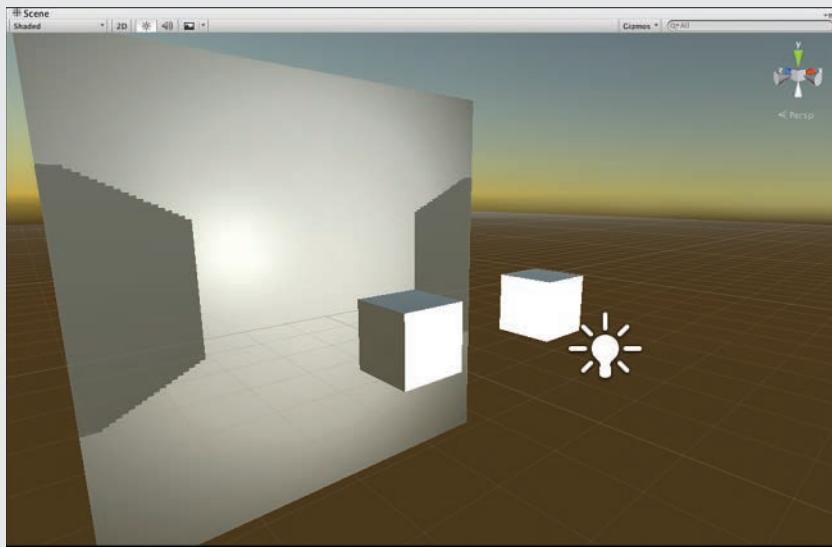
Baking refers to the process of adding light and shadow to textures and objects during creation. You can do this with Unity or with a graphical editor. For instance, if you were to make a wall texture with a dark spot on it that resembled a human shadow, and then put a human model next to the wall it was on, it would seem like the model was casting a shadow on the wall. The truth is, though, that the shadow was “baked” into the texture. Baking can make your games run much more quickly because the engine won’t have to calculate light and shadow every single frame. That’s a big deal!

TRY IT YOURSELF ▼

Adding a Point Light to a Scene

Let’s build a scene with some dynamic point lighting. The completed version of this project is available as Hour6_Lights in the book assets under Hour 6:

1. Create a new scene, save it as Point Light, and delete the Directional Light.
2. Add a plane to the scene (**GameObject > 3D Object > Plane**). Ensure that the plane is positioned at (0, 0.5, 0) and it rotated (270, 0, 0). The plane should be visible to the camera, but only from one side in the Scene view.
3. Add two cubes to the scene. Position them at (-1.5, 1, -5) and (1.5, 1, -5).
4. Add a point light to the scene (**GameObject > Light > Point Light**). Position the point light at (0, 1, -7). Notice how the light illuminates the inner sides of the cubes and the background plane (see Figure 6.1).
5. Set the light’s Shadow Type to Hard Shadows, and try moving it around. Continue exploring the light properties. Be sure to experiment with the light color, range, and intensity.

**FIGURE 6.1**

The results of the exercise.

Spotlights

Spotlights work a lot like the headlights in a car or flashlights. The light of a spotlight begins in at a central spot and then radiates out in a cone. In other words, spotlights illuminate whatever is in front of them while leaving everything else in the dark. Unlike a point light, which sends light in every direction, you can aim spotlights.

To add a spotlight to your scene, click **GameObject > Create Other > Spotlight**. Alternatively, if you already have a light in your scene, you can change its type to **Spot**. It will then become a spotlight.

Spotlights have only one property not already covered: **Spot Angle**. The **Spot Angle** property determines the radius of the cone of light emitted by the spotlight.

TRY IT YOURSELF ▼

Adding a Spotlight to a Scene

You now have a chance to work with spotlights in Unity. For brevity, this exercise uses the project created in the previous Try It Yourself for point lights. If you have not completed that, do so to continue with this exercise. The completed version of this project is available as Hour6_Lights in the book assets under Hour 6, in the Spotlight Scene:

1. Duplicate the Point Light scene from the previous project (Edit > Duplicate). Name this new scene Spotlight.
2. Right-click the **Point light** in the Hierarchy view and select **Rename**. Rename the object to **Spotlight**. In the Inspector, change the Type property to **Spot**. Place the light object at (0, 1, -13).
3. Experiment with the properties of the spotlight. Notice how the range, intensity, and spot angle shape and change the effect of the light.

Directional Lights

The last light type you work with in this section is the directional light. The directional light is similar to the spotlight in that it can be aimed. Unlike the spotlight, though, the directional light illuminates the entire scene. You can think of a directional light as a sun. In fact, you used a directional light already as a sun in the previous hours working with terrain. The light from a directional light radiates evenly in parallel lines across a scene.

New scenes come with a directional light by default. To add a new directional light to your scene, click **GameObject > Light > Directional Light**. Alternatively, if you already have a light in your scene, you can change its type to Directional. It will then become a directional light.

Directional lights have one additional property that hasn't been covered yet: Cookie Size. Cookies are covered later, but basically this property controls how big a cookie is and thus how many times it is repeated across a scene.

TRY IT YOURSELF ▼

Adding a Directional Light to a Scene

We will now add a directional light to a Unity scene. Once again, this exercise builds off of the previous project created in the Try It Yourself for spotlights. If you have not completed that, do so to continue with this exercise. The completed version of this project is available as Hour6_Lights in the book assets under Hour 6, in the Directional Light Scene:

1. Duplicate the Spotlight scene from the previous project (Edit > Duplicate). Name this new scene Directional Light.

- 
2. Right-click the **Spotlight** in the Hierarchy view and select **Rename**. Rename the object to **Directional Light**. In the Inspector, change the Type property to **Directional**. Change the object's rotation to be (75, 0, 0).
 3. Notice how the light looks on the objects in the scene. Now change the light's position to be (50, 50, 50). Notice how the light does not change. Because the directional light comes in parallel lines, the position of it does not matter. Only the rotation of a directional light matters.
 4. Experiment with the properties of the directional light. There is no range (range is infinite), but see how the color and intensity affect the scene.

NOTE

Area Lights and Emissive Materials

There are two more light types that are not being covered in this text: the area light and emissive materials.

An area light is a feature that exists for a process called lightmap baking. These topics are more advanced than the aim of this text and aren't needed for basic game projects. If you want to learn more about this, Unity has a wealth of online documentation.

The other type is an emissive material—a material applied to an object that actually transmits light. This could be very useful for a TV screen, indicator lights, etc.

Creating Lights Out of Objects

Because lights in Unity are components, any object in a scene can be a light. To add a light to an object, first select the object. Then in the Inspector view, click the **Add Component** button. A new list should pop up. Select **Rendering** and then **Light**. Now your object has a light component. An alternative way to add a light to an object is to select the object and click **Component > Rendering > Light** in the menu.

Note a couple of things about adding lights to objects. The first is that the object will not block the light. This means that putting a light inside a cube will not stop the light from radiating. The second is that adding a light to an object does not make it glow. The object itself will not look like it is giving off light, but it is.

Halos

Halos are glowing circles that appear around lights in foggy or cloudy conditions (see Figure 6.2). They occur because light is bouncing off of small particles all around the light source. In Unity, you can easily add halos to your lights. Each light has a check box called Draw Halo. If it is checked, a halo will be drawn for the light. If you can't see the halo, you may be too close to the light. Try backing up a bit.

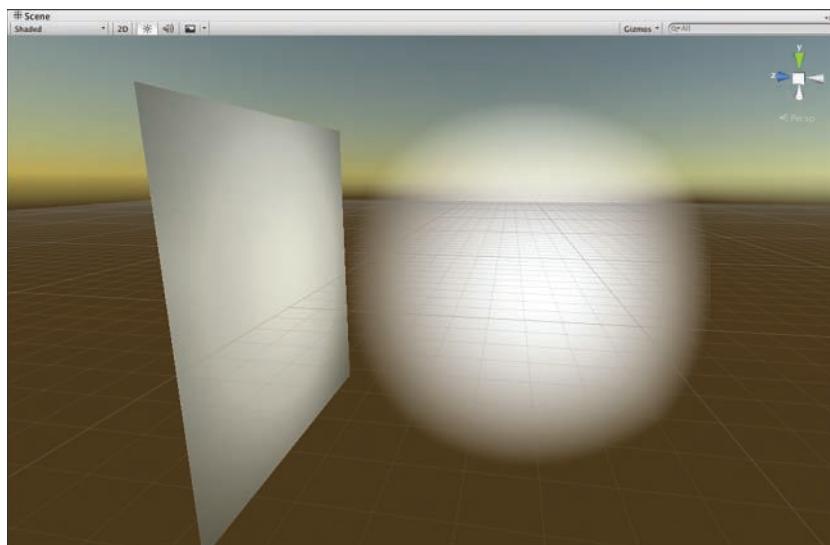
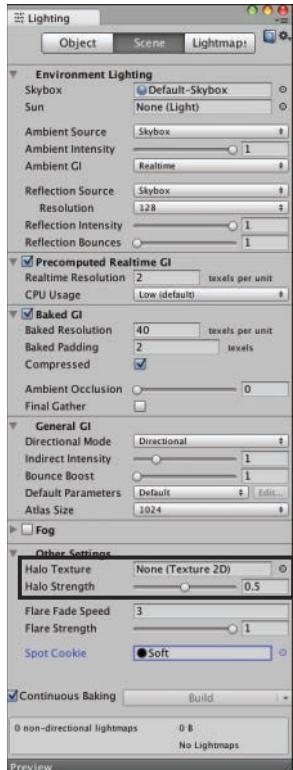


FIGURE 6.2

A halo around a light.

The size of a halo is determined by the light's range. The bigger the range, the bigger the halo. Unity also provides a few properties that apply to all halos in a scene. You can access these properties by clicking **Window > Lighting > Scene**. Expand Other Settings, and the settings will then appear in the Inspector view (see Figure 6.3).

**FIGURE 6.3**

The scene lighting settings.

The Halo Strength property determines how big the halo will be based off of the light's range. For instance, if a light has a range of 10 and the strength is set to 1, the halo will extend out all 10 units. If the strength were set to .5, then the halo would extend out only 5 units ($10 \times .5 = 5$). The Halo Texture property allows you to specify a different shape for your halo by providing a new texture. If you do not want to use a custom texture for your halo, you can leave it blank and the default circular one will be used.

Cookies

If you have ever shone a light on a wall and then put your hand in between the light and the wall, you probably noticed that your hand blocked some of the light, leaving a hand-shaped shadow on the wall. You can simulate this effect in Unity with cookies. Cookies are special textures that you can add to lights to dictate how the light radiates. Cookies differ a little for point, spot, and directional lights. Spotlights and directional lights both use black-and-white flat textures for cookies. Spotlights don't repeat the cookies, but directional lights do. Point lights also use black-and-white textures, but they must be placed in a cubemap. A cubemap is six textures placed together to form a box (like a skybox).

Adding a cookie to a light is a fairly straightforward process. You simply apply a texture to the Cookie property of the light. You can set up the texture correctly in Unity, and then change its properties in the Inspector window. Figure 6.4 shows the correct properties for a point cookie, a spot cookie, and a directional cookie.

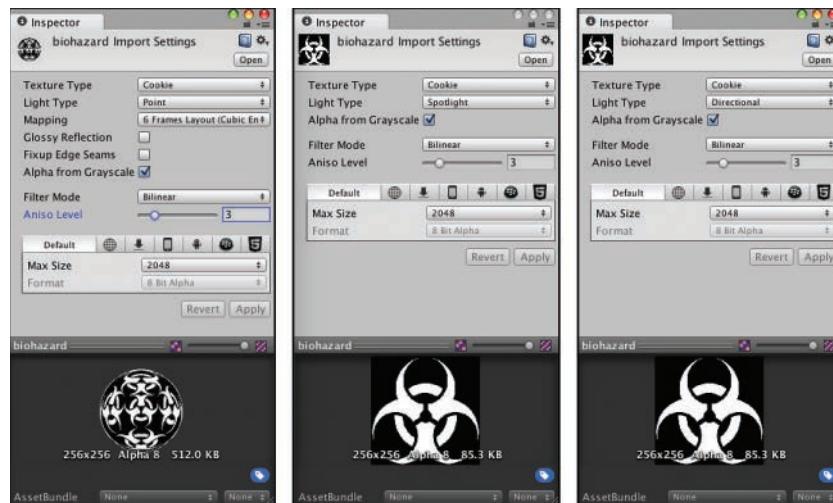


FIGURE 6.4

The texture properties of cookies for point, spot, and directional lights.

TRY IT YOURSELF

Adding a Cookie to a Spotlight

Let's add a cookie to a spotlight so that you can see the process from start to finish. This exercise requires the biohazard.png image in the book assets for Hour 6:

1. Create a new project or scene. Add a plane to the scene and position it at (0, 1, 0) with a rotation of (270, 0, 0).
2. Add a spotlight to the Main Camera by selecting the **Main Camera** and then clicking **Component > Rendering > Light** and changing the type to **Spot**. Set the range to **18**, the spot angle to **40**, and the intensity to **3**.
3. Drag the biohazard.png texture from the book assets into your Project view. Select the texture, and in the Inspector view change the texture type to **Cookie**. Check the **Alpha from Grayscale** check box. This makes the cookie block light where it's black. If you are unsure whether you have the correct settings, check the spot settings in Figure 6.4.
4. With the Main Camera selected, click and drag the biohazard texture into the **Cookie** property of the light component. You should see the biohazard symbol projected onto the plane (see Figure 6.5).
5. Experiment with different ranges and intensities of the light. Rotate the plane and see how the symbol warps and distorts.

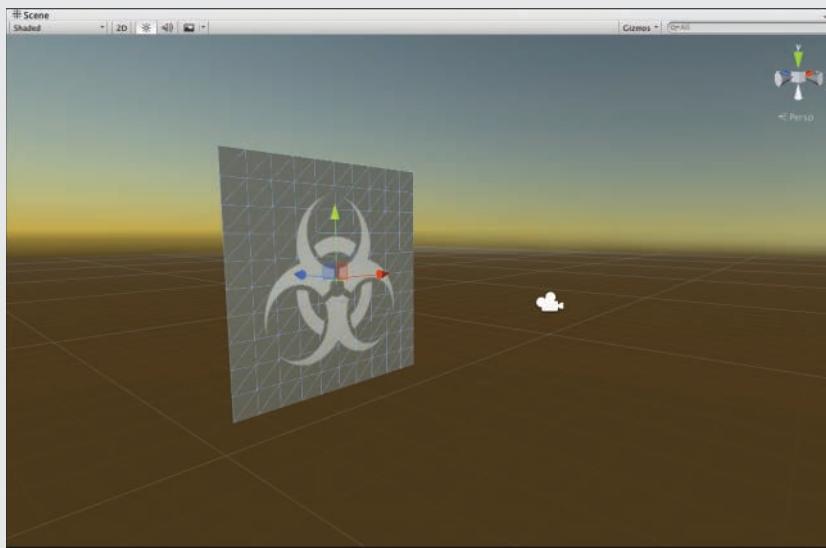


FIGURE 6.5

Spotlight with a cookie.

Cameras

The camera is the player's view into the world. It provides their perspective and controls how things appear to them. All games in Unity have at least one camera. In fact, a camera is always added for you whenever you create a new scene. The camera always appears in the hierarchy as Main Camera. In this section, you learn all about cameras and how to use them for interesting effects.

Anatomy of a Camera

All cameras share the same set of properties that dictate how they behave. Table 6.2 describes all the camera properties.

TABLE 6.2 Camera Properties

Property	Description
Clear Flags	The Clear Flags property determines what the camera displays in the areas where there are no game objects. The default is Skybox. If there is no skybox, the camera defaults to a solid color. Depth Only should be used only when there are multiple cameras. Don't Clear causes streaking and should be used only if writing a custom shader.
Background	The Background property dictates the background color if there is no skybox present.
Culling Mask	The Culling Mask property determines what layers are picked up by the camera. By default, the camera sees everything. You can uncheck certain layers (more on layers later), and they won't be visible to the camera.
Projection	The Projection property determines how the camera sees the world. The two options are Perspective and Orthographic. Perspective cameras perceive the world in 3D, where closer objects are larger and farther objects are smaller. This is the setting to use if you want depth in your game. The Orthographic camera setting ignores depth and treats everything as flat.
Field of View	The Field of View property specifies how wide of an area the camera can see.
Clipping Planes	The Clipping Planes property dictates the range where objects are visible to the camera. Objects that are closer than the near plane or farther than the far plane will not be seen.
View Port Rect	The Normalized View Port Rect property establishes what part of the actual screen the camera is projected on, and is short for View Port Rectangle. By default, the x and y are both set to 0, which causes the camera to start in the lower left of the screen. The width and height are both set to 1, which causes the camera to cover 100% of the screen vertically and horizontally. This is covered in more detail later.

Property	Description
Depth	The Depth property dictates the priority for multiple cameras. Lower numbers are drawn first, which means that higher numbers may be drawn on top and effectively hide them.
Rendering Path	The Rendering Path property determines how the camera renders. It should be left as Use Player Settings.
Target Texture	The Target Texture property enables you to specify a texture for the camera to draw to instead of the screen.
Occlusion Culling	Occlusion Culling is a feature that disables rendering of objects when they are not currently seen by the camera because they are obscured (occluded) by other objects.
HDR	The HDR (Hyper-Dynamic Range) property determines whether Unity's internal light calculations are limited to the basic color range. The property allows for advanced visual effects. For now, leave this unchecked.

Cameras have many properties, but you can set most and forget about them. Cameras also have a few extra components. The GUI Layer allows the camera to see GUI elements (as covered later in this book). The Flare Layer allows the camera to see the lens flares of lights. Finally, the audio listener allows the camera to pick up sound. If you add more cameras to a scene, you need to remove their audio listeners. There can be only one audio listener per scene.

Multiple Cameras

Many effects in modern games would not be possible without multiple cameras. Thankfully, you can have as many cameras as you want in a Unity scene. To add a new camera to a scene, click **GameObject > Camera**. Alternatively, you can add the camera component to a game object already in your scene. To do that, select the object and click **Add Component** in the Inspector. Select **Rendering > Camera** to add the camera component. Remember that adding a camera component to an existing object will not automatically give you the GUI Layer, Flare Layer, or audio listener.

CAUTION

Multiple Audio Listeners

As mentioned earlier, a scene can have only a single audio listener. In older versions of Unity, having two or more listeners would cause an error and prevent a scene from running. Having multiple listeners will just display a warning message, although audio might not be heard correctly. This topic is covered in detail in a later hour.

TRY IT YOURSELF ▼

Working with Multiple Cameras

The best way to understand how multiple cameras interact is to work with them hands on. This exercise focuses on basic camera manipulation:

1. Create a new project or scene and add two cubes. Place the cubes at $(-2, 1, -5)$ and $(2, 1, -5)$. Leave the directional light to the scene.
2. Move the Main Camera to $(-3, 1, -8)$ and change its rotation to $(0, 45, 0)$.
3. Add a new camera to the scene (click **GameObject > Camera**) and position it at $(3, 1, -8)$. Change its rotation to $(0, 315, 0)$. Be sure to disable the audio listener for the camera by unchecking the box next to the component.
4. Run the scene. Notice how the second camera is the only one displayed. This is because the second camera has a higher depth than the Main Camera. The Main Camera is drawn to the screen first, and then the second camera is drawn overtop of it. Change the Main Camera Depth to 1 and then run the scene again. Notice how the Main Camera is now the only one visible.

Split Screen and Picture in Picture

As you saw earlier, having multiple cameras in a scene doesn't do much good if one simply draws over the other. In this section, you learn to use the Normalized View Port Rect property to achieve split screen and picture-in-picture effects.

The normalized view port basically treats the screen as a simple rectangle. The lower-left corner of the rectangle is $(0, 0)$ and the upper-right corner is $(1, 1)$. This does not mean that the screen has to be a perfect square. Instead, think of the coordinates as percentages of the size. So, a coordinate of 1 means 100%, and a coordinate of .5 means 50%. With this in mind, placing cameras on the screen becomes easy. By default, cameras project from $(0, 0)$ with a width and height of 1 (or 100%). This causes them to take up the entire screen. If you were to change those numbers, however, you would get a different effect.

TRY IT YOURSELF ▼

Creating a Split-Screen Camera System

Let's walk through creating a split-screen camera system. This type of system is common in two-player games where the players have to share the same screen. This exercise builds off of the previous Try It Yourself for multiple cameras earlier this hour:

1. Open the previously created project.

- ▼
2. Ensure that the Main Camera has a depth of **-1**. Ensure the X and Y properties of the camera's View Port Rect property are both **0**. Set the W and H properties to **1** and **.5**, respectively (100% of the width and 50% of the height).
 3. Ensure that the second camera also has a depth of **-1**. Set the X and Y properties of the view port to **(0, .5)**. This will cause the camera to begin drawing halfway down the screen. Set the W and H properties to **1** and **.5**, respectively.
 4. Run the scene and notice how both cameras are now projecting on the screen at the same time (see Figure 6.6). You can split the screen like this as many times as you want.

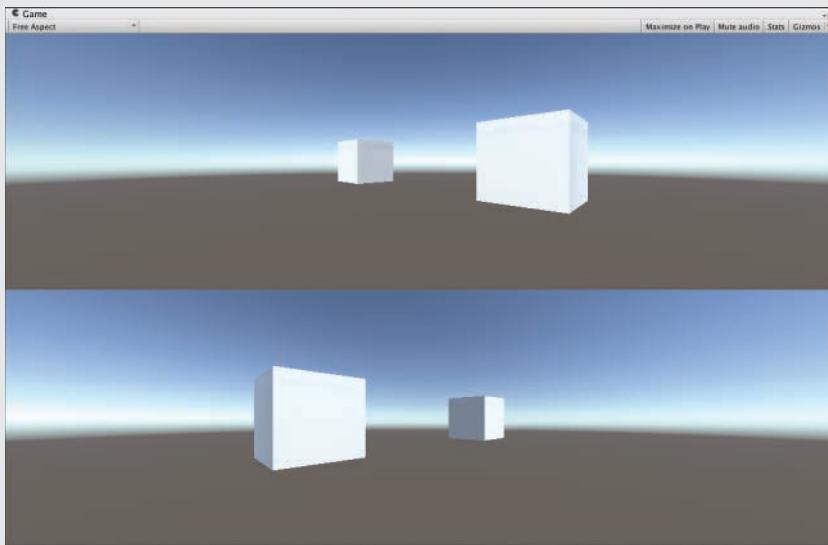


FIGURE 6.6
The split-screen effect.

▼ TRY IT YOURSELF

Creating a Picture-in-Picture Effect

Picture in picture is a common way to create effects like minimaps. With this effect, one camera is going to draw over another one in a specific area. This exercise will build off of the previous Try It Yourself for multiple cameras earlier in this hour:

1. Open the previously created project.
2. Ensure that the Main Camera has a depth of **-1**. Ensure that the X and Y properties of the camera's Normalized View Port Rect property are both **0** and the W and H properties both **1**.

3. Ensure that the depth of the second camera is 0. Set the X and Y property of the view port to (.75, .75) and set the W and H values to .2 each.
4. Run the scene. Notice how the second camera appears in the upper-right corner of the screen (see Figure 6.7). Experiment with the different view port settings to get the camera to appear in the different corners.

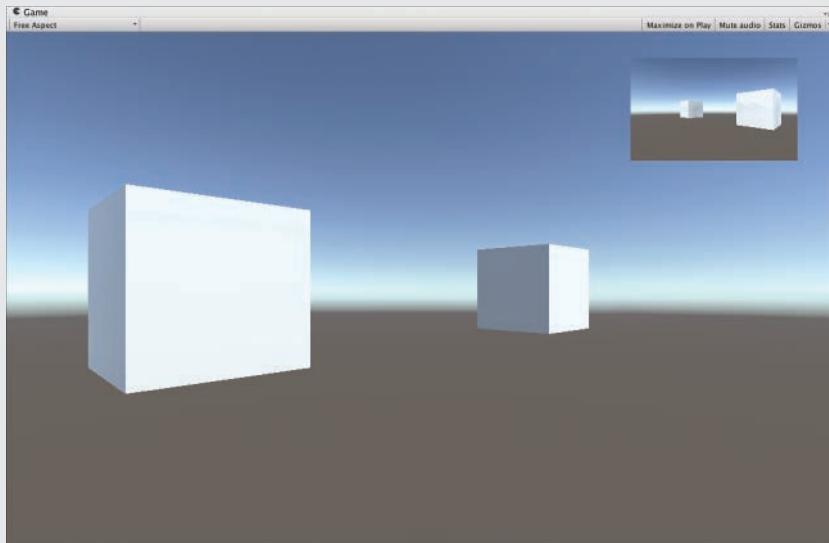


FIGURE 6.7

The picture-in-picture effect.

Layers

With so many objects in a project and in a scene, it can often be difficult to organize them. Sometimes you want items to be viewable by only certain cameras or illuminated by only certain lights. Sometimes you want collision to occur only between certain types of objects. Unity's answer to this organization is layers. Layers are groupings of similar objects so that they can be treated a certain way. By default, there are 8 built-in layers and 24 layers for the user to define.

CAUTION

Layer Overload!

Adding layers can be a great way to achieve complex behaviors without doing a lot of work. A word of warning, though: Do not create layers for items unless you need to. Too often, people arbitrarily create layers when adding objects to a scene with the thinking that they might need them later. This approach can lead to an organizational nightmare as you try to remember what each layer is for and what it does. In short, add layers when you need them. Don't try to use layers just because you can.

Working with Layers

Every game object starts in the Default layer. That is, the object has no specific layer to belong to and so it is lumped in with everything else. You can easily add an object to a layer in the Inspector view. With the object selected, click the **Layer** drop-down in the Inspector and choose a new layer for the object to be a part of (see Figure 6.8). By default, there are five layers to choose from: Default, TransparentFX, Ignore Raycast, Water, and UI. You can safely ignore most of these for now because they are not very useful to you at this point.

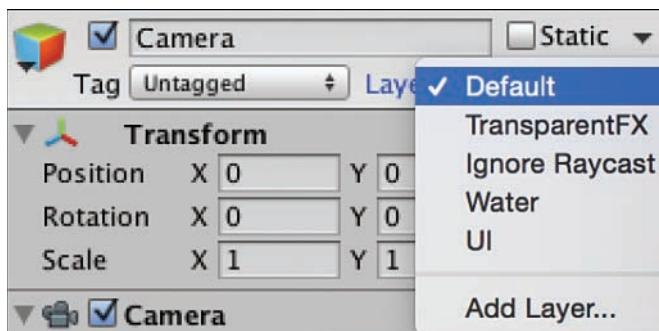


FIGURE 6.8

The Layer drop-down menu.

Although the current built-in layers aren't exactly useful to you, you can easily add new layers. You add layers in the Tag and Layers Manager, and there are three ways to open it:

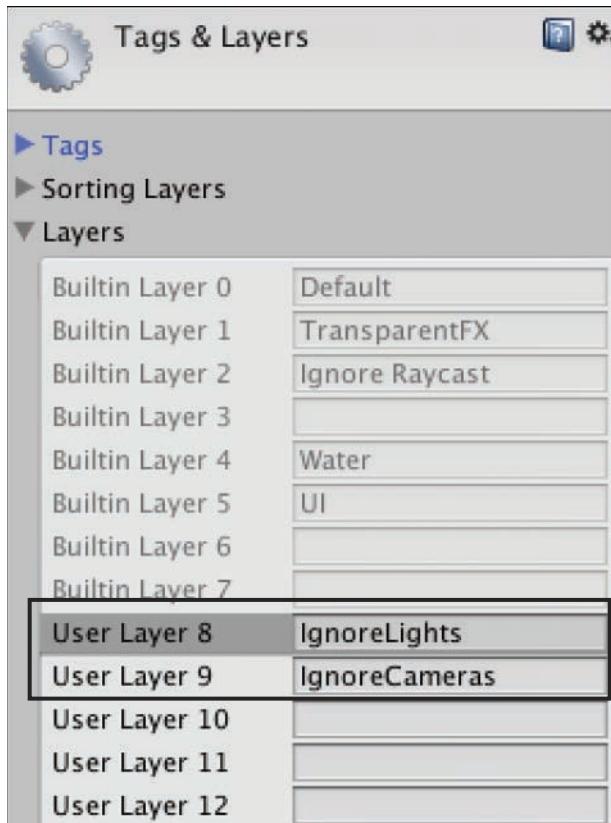
- ▶ With an object selected, click the **Layer** drop-down and select **Add Layer** (see Figure 6.8).
- ▶ In the menu at the top of the editor, click **Edit > Project Settings > Tags and Layers**.
- ▶ Click the **Layers** selector in the scene toolbar and choose **Edit Layers** (see Figure 6.9).



FIGURE 6.9

The Layers selector in the scene toolbar.

Once in the Tags and Layers Manager, just click to the right of one of the user layers to give it a name. Figure 6.10 illustrates this process and shows two new layers being added. (They are added for this picture, and you won't have them unless you add them yourself.)

**FIGURE 6.10**

Adding new layers to the Tag Manager.

Using Layers

There are many uses for layers. The usefulness of layers is limited only by what you can think to do with them. This section covers four common uses.

The first is the ability to hide layers from the Scene view. By clicking the Layers selector in the Scene view toolbar (see Figure 6.9), you can choose which layers appear in the Scene view and which don't. By default, the scene is set up to show everything.

TIP

Invisible Scene Items

One common mistake for people who are new to Unity is accidentally changing the layers visible in the Scene view. If you are not familiar with the ability to make layers invisible, this can be quite confusing. Just note that if at any time items are not appearing in the Scene view when they should, check the Layers selector to ensure that it is set to show everything.

The second utility of layers is to use them to exclude objects from being illuminated by light. This can prove useful if you are making a custom user interface, shadowing system, or are using a complex lighting system. To prevent a layer from being illuminated by a light, select the light. Then, in the Inspector view, click the **Culling Mask** property and deselect any layers that you want ignored (see Figure 6.11).

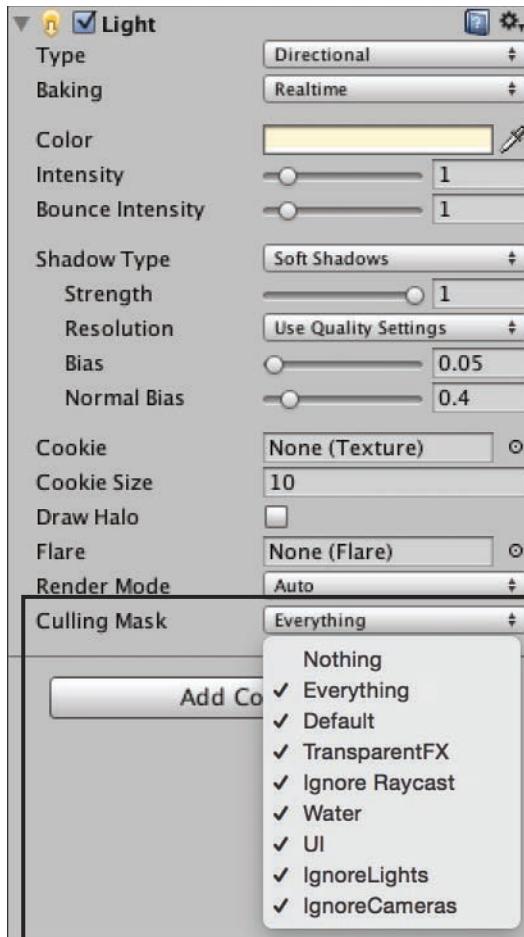
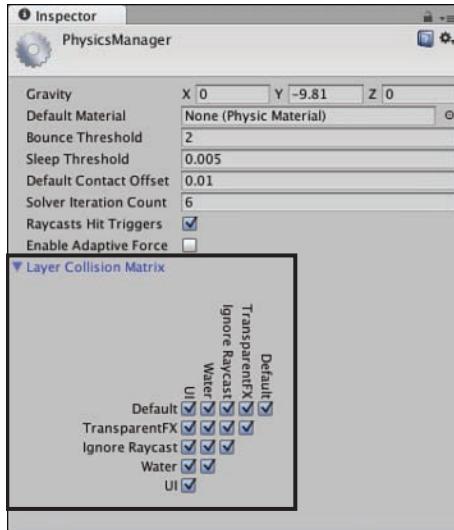


FIGURE 6.11
The Culling Mask property.

The third thing you can use layers for is to tell Unity which physics objects interact with which other. You can choose this in **Edit > Project Settings > Physics** and look for the Layer Collision Matrix (see Figure 6.12).

**FIGURE 6.12**

The Layer Collision Matrix.

The last thing to know about layers is that you can use them to determine what a camera can and cannot see. This is useful if you want to build a custom visual effect using multiple cameras for a single viewer. Just as previously described, to ignore layers simply click the **Culling Mask** drop-down on the camera component and deselect anything you don't want to appear.

TRY IT YOURSELF ▾

Ignoring Light and Cameras

Let's take a moment to work with layers for both lights and cameras:

1. Create a new project or scene. Add two cubes to the scene and position them at $(-2, 1, -5)$ and $(2, 1, -5)$.
2. Enter the Tag Manager using any of the three methods listed earlier and add two new layers: **IgnoreLights** and **IgnoreCameras** (see Figure 6.10).
3. Select one of the cubes and add it to the **IgnoreLights** layer. Select the other cube and add it to the **IgnoreCameras** layer.
4. Add a point light to the scene and place it at $(0, 1, -7)$. Set the light's intensity to 2. In the Culling Mask property for the light, deselect the **IgnoreLights** layer. Notice now how only one of the cubes is illuminated. The other one has been ignored because of its layer.
5. Select the Main Camera and remove the **IgnoreCameras** layer from its Culling Mask property. Run the scene and notice how only one nonilluminated cube appears. The other one has been ignored by the camera.

Summary

In this hour, you learned about lights and cameras. You worked with the different types of lights. You also learned to add cookies and halos to the lights you had in the scene. From there, you got hands on with cameras. You learned all about the basics of cameras and about adding multiple cameras to create a split-screen and picture-in-picture effect. You wrapped up the hour by learning about layers in Unity.

Q&A

- Q. I noticed we skipped lightmapping. Is it important to learn?**
- A.** Lightmapping is a useful technique for optimizing the lighting of a scene. It's slightly more advanced, and you don't need to know how to use it to make your scenes look great.
- Q. How do I know if I want a perspective or orthographic camera?**
- A.** As mentioned in the text, a general rule of thumb is that you want perspective for 3D games and effects and orthographic for 2D games and effects.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

- 1.** If you want to illuminate an entire scene with one light, which type should you use?
- 2.** How many cameras can be added to a scene?
- 3.** How many user-defined layers can you have?
- 4.** What property determines which layers are ignored by lights and cameras?

Answers

- 1.** A directional light is the only light that is applied evenly to an entire scene.
- 2.** You can have as many as you want.
- 3.** 24.
- 4.** The Culling Mask property.

Exercise

In this exercise, you have a chance to work with multiple cameras and lights. You have a bit of leeway in the construction of this exercise, so feel free to be creative:

1. Create a new scene or project. Delete the directional light. Add a sphere to the scene and place it at (0, 0, 0).
2. Add four point lights to your scene. Place them at (-4, 0, 0), (4, 0, 0), (0, 0, -4), and (0, 0, 4). Give each of them their own color. Set the ranges and intensities to create the visual effect on the sphere that you want.
3. Delete the Main Camera from your scene (by right-clicking the **Main Camera** and selecting **Delete**). Add four cameras to the scene. Disable the audio listener on three of them. Position them at (2, 0, 0), (-2, 0, 0), (0, 0, 2), and (0, 0, -2). Rotate each of them about the y axis until they are facing the sphere.
4. Change the view port settings on the four cameras so that you achieve a split-screen effect with all four cameras. You should have a camera displaying in each corner of the screen taking up a quarter of the screen's size (see Figure 6.13). This step is left for you to complete. If you get stuck, a completed version of this exercise called Hour6_Exercise is available in the Hour 6 assets.

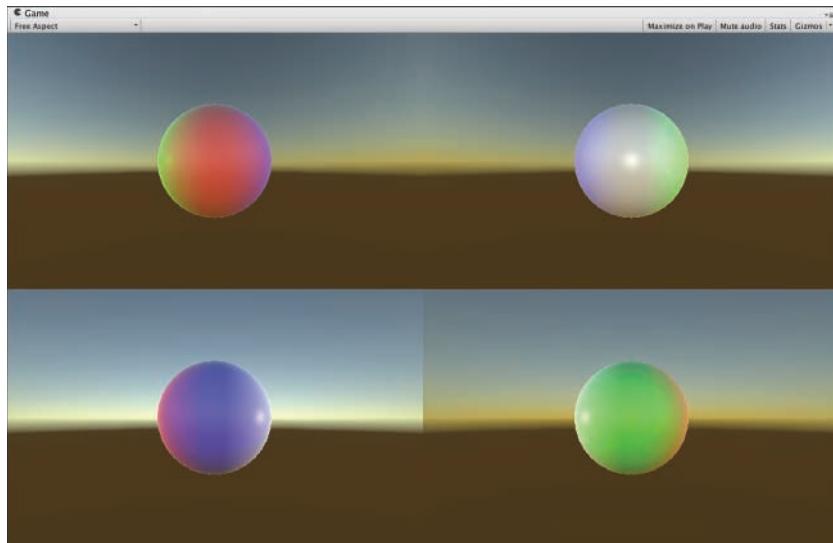


FIGURE 6.13
The completed exercise.

This page intentionally left blank

HOUR 7

Game 1: *Amazing Racer*

What You'll Learn in This Hour:

- ▶ How to design a basic game
- ▶ How to apply your knowledge of terrains to build a game-specific world
- ▶ How to add objects to a game to provide interactivity
- ▶ How to playtest and tweak a finished game

In this hour, you take what you have learned so far and use it to build your first Unity game. You start by covering the basic design elements of the game. From there, you build the world that the game will take place in. Then you add some interactivity objects to make the game playable. You finish by playing the game and making any necessary tweaks to improve the experience.

TIP

Completed Project

Be sure to follow along in this hour to build the complete game project. If you get stuck, you can find a completed copy of the game in the book assets for Hour 7. Take a look at it if you need help or inspiration!

Design

The design portion of game development is where you plan ahead of time all the major features and components of a game. You can think of it as laying down the blueprint so that the actual construction process is much smoother. When making a game, a lot of time is normally spent working through the design. Because the game you are making in this hour is fairly basic, the design phase will go faster. You need to focus on three areas of planning to make this game: the concept, the rules, and the requirements.

The Concept

The idea behind this game is simple. You start at one end of an area and run quickly to the other side. There will be hills, trees, and obstacles in your path. Your goal is to see how fast you can make it to the finish zone. This game concept was chosen for your first game because it highlights all the sections you have worked on so far. Also, because you have not learned scripting in Unity yet, you cannot add very elaborate interactions. Future games will be more complex.

The Rules

Every game must have a set of rules. The rules serve two purposes. First, they tell you how the player will actually play the game. Second, because software is a process of permission (see the Process of Permission note), the rules dictate the actions available to the players to overcome challenges. The rules for *Amazing Racer* are as follows:

- ▶ There is no win or loss condition; only a completed condition. The game is completed when the player enters the finish zone.
- ▶ The player will always spawn in the same spot. The finish zone will always be in the same spot.
- ▶ There will be water hazards present. Whenever the player falls into a water hazard, that player is moved back to the spawn point.
- ▶ The objective of the game is to try to get the fastest time possible. This is an implicit rule and is not specifically built into the game. Instead, cues will be built into the game as hints to the player that this is the goal. The idea is that the players will intuit the desire for a faster time based on the signals given to them.

NOTE

Process of Permission

Something to always remember when making a game is that software is a process of permission. What this means is that unless you specifically allow something, it will be unavailable to the player. For instance, if the player wants to climb a tree, but you have not created any way for the player to climb a tree, that action will not be permitted. If you do not give players the ability to jump, they can't jump. Everything that you want the player to be able to do must be explicitly built in. Remember that you cannot assume any action and must plan for everything! Also remember that players can combine actions in inventive ways, like stacking blocks, then jumping from the top block, if you make that possible.

NOTE**Terminology**

Some new terms are used in this hour:

- ▶ **Spawn:** Spawning is the process by which a player or entity enters a game.
- ▶ **Spawning point:** A spawning point is the place where a player or entity spawns. There can be one or many of these. They can be stationary or moving around.
- ▶ **Condition:** A condition is a form of trigger. A win condition is the event that will cause the player to win the game (such as accumulating enough points). A loss condition is the event that will cause the player to lose the game (such as losing all of your click points).
- ▶ **Game Controller:** The game controller dictates the rules and flow of a game. It is responsible for knowing when the game is won or lost (or just over). Any object can be designated as the game controller as long as it is always in the scene. Often, an empty object or the Main Camera is designated as the game controller.
- ▶ **Playtesting:** The process of playing a game that is still in development, to see how actual players react to the game so it can be improved accordingly.

The Requirements

Another important step in the design process is determining which assets will be required for the game. Generally speaking, a game development team is made up of several individuals. Some will be designing, and others program or make art. Every member of the team needs something to do to be productive during every step of the development process. If everyone waited until something was needed to begin working, there would be a lot of starting and stopping. Instead, you determine your assets ahead of time so that things can be created before they are needed.

Here is a list of all of the requirements for *Amazing Racer*:

- ▶ A piece of rectangular terrain. The terrain needs to be big enough to present a challenging race. The terrain should have obstacles built in as well as a designated spawn and finish point (see Figure 7.1).
- ▶ Textures and environment effects for the terrain. These are provided in the Unity standard assets.
- ▶ A spawn point object, a finish zone object, and a water hazard object. These will be generated in Unity.
- ▶ A character controller. This is provided by the Unity standard assets.
- ▶ A graphical user interface (GUI). This will be provided for you in the book assets. Note we are using the old-style GUI here which works purely from script, for simplicity. In your projects, use the new UI system as introduced in Hour 14.
- ▶ A game controller. This will be created in Unity.

**FIGURE 7.1**

The general terrain layout for the game *Amazing Racer*.

Creating the Game World

Now that you have the basic idea of the game on paper, it is time to start building it. There are many places to begin building a game. For this project, you begin with the world. Because this is a linear racing game, the world will be longer than it is wide (or wider than it is long, depending on how you look at it). Many of the Unity standard assets will be used to rapidly create the game.

Sculpting the World

There are many ways you can create this terrain. Everyone will probably have a different vision for it in his or her head. To streamline the process, a heightmap has been provided for you. This is to ensure that everyone will have the same experiences during this hour. To sculpt the terrain, follow these steps:

- 1.** Create a new project in a folder named *Amazing Racer*. Add a terrain to the project, and position its Transform at (0, 0, 0) in the Inspector.
- 2.** Locate the file *terrain.raw* in the book assets for Hour 7. Import the *terrain.raw* file as a heightmap for the terrain (by clicking **Import Raw** in the Heightmap section of the Terrain Settings, which you find in the Inspector).

3. Leave the Depth, Width, Height, and Byte Order settings as they are. Set the Terrain Size to **200** wide by **100** long and **100** tall.
4. Create a **Scenes** folder under assets and save the current scene as **Main**.

The terrain should now be sculpted to match the world in the book. Feel free to make minor tweaks and changes to your liking.

CAUTION

Building Your Own Terrain

In this hour, you are building a game based on a heightmap given to you. The heightmap has been prepared for you so that you can quickly get through the process of game development. You may, however, choose to build your own custom world to make this game truly unique and yours. If you do that, however, be warned that some of the coordinates and rotations provided for you might not match up. If you want to build your own world, pay attention to intended placement of objects and position them in your world accordingly.

Adding the Environment

At this point, you can begin texturing and adding the environment effects to your terrain. You need to import the Environment package (click **Assets > Import Package**).

You now have a bit of freedom to decorate the world, however, you would like. The following suggestions are guidelines. Feel free to do things in a manner that looks good to you:

- ▶ Rotate the directional light to suit your preference.
- ▶ Texture the terrain. The sample project uses the following textures: **GrassHillAlbedo** for flat parts, **CliffAlbedoSpecular** for the steep parts, and **GrassRockyAlbedo** for the areas in between, and **MudRockyAlbedoSpecular** for inside the pits.
- ▶ Add trees to your terrain. Trees should be placed sparsely and mostly on flat surfaces.
- ▶ Add some basic water to your scene. See the Hour 5 for how to do this. Place the water (at 88, 29, 49) and scale it (50, 1, 50).

The terrain should now be prepared and ready to go. Be sure to spend a good amount of time on texturing to make sure that you have a good blend and a realistic look. There are some additional things not present in the sample project but that you may want to add, including the following:

- ▶ Fog.
- ▶ Grass around the water hazards. This may obscure them a bit and add to the difficulty.
- ▶ Light flares for the directional light to simulate the sun.

The Character Controller

At this stage of development, you want to add a character controller to your terrain:

1. Import the standard character controllers by clicking **Assets > Import Package > Characters**.
2. Drag a **FPSCControllerPerson** controller asset from the **Assets\Standard Assets\Characters\FirstPersonCharacter\Prefabs** folder into your scene.
3. Position the First Person Shooter Controller (it will be named Player and be blue) at (160, 32, 64). If the player doesn't sit on the terrain, ensure the terrain is positioned at (0, 0, 0) as per the previous exercise. Now rotate the controller 260 on the y axis so that it faces the correct direction. Rename the object Player.
4. As the FPS Controller has its own camera, you can now delete the Main Camera from the scene.

Once the character controller is in your scene and positioned, play the scene. Be sure to move around and look for any areas that need fixed or smoothed. Pay attention to the borders. Look for any areas where you are able to escape the world. Those places will need to be raised so that the player cannot fall off of the map. This is the stage where you generally fix any basic problems with your terrain.

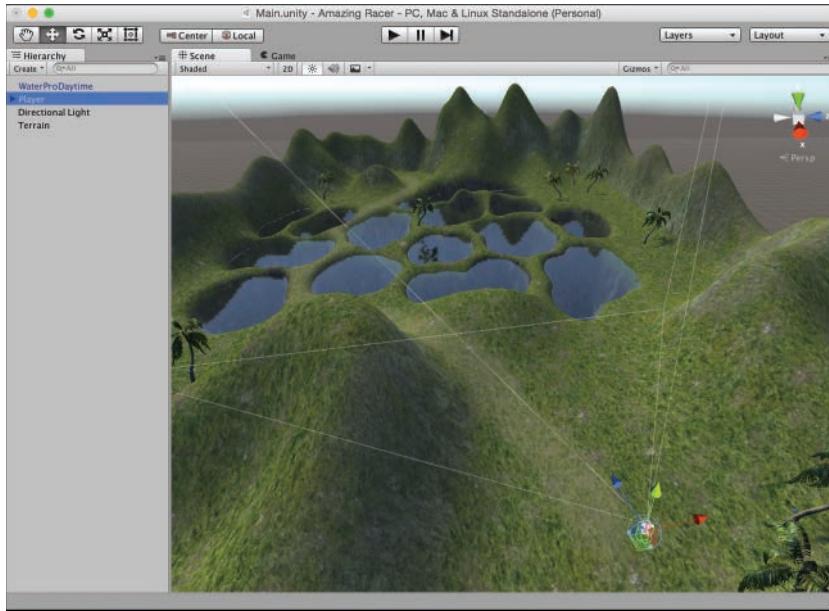
TIP

Falling Off of the World

Generally, game levels will have walls or some other obstacle in place to prevent the player from exiting the developed area. If the game employs gravity, the player may fall off of the side of the world. You always want to create some way to prevent players from going somewhere they shouldn't. This game project uses a tall berm to keep the players in the play area. The heightmap provided to you in the book's assets for Hour 7 intentionally has a few places where the player can climb out. See if you can find and correct them. You can also set a Slope Limit, and other for the FPSCController in the Inspector.

Gamification

You now have a world in which your game can take place. You can run around and experience the world to an extent. The piece that is missing is the game itself. Right now, what you have is considered a toy. It is something that you can play with. What you want is a game, which is a toy that has rules and a goal. The process of turning something into a game is called *gamification*, and that's what this section is all about. If you followed the previous steps, your game project should now look something like Figure 7.2. The next few steps are to add game control objects for interaction, apply game scripts to those objects, and connect them to each other.

**FIGURE 7.2**

The current state of the *Amazing Racer* game.

NOTE

Scripts

Scripts are pieces of code that define behaviors for game objects. You have not yet learned about scripting in Unity. To make an interactive game, however, scripts are a must. With this in mind, the scripts needed to make this game have been provided for you. An effort has been made to make the scripts as minimal as possible so that you can understand most of this project. Feel free to open the scripts in a text editor and read what they are doing. Scripts are covered in greater detail in Hour 8, “Scripting Part 1,” and Hour 9, “Scripting Part 2.”

Adding Game Control Objects

As defined in your requirements section earlier, you need four specific game control objects. The first object will be a spawning point. This will be a simple game object that exists solely to tell the game where to spawn the player. To create the spawning point, follow these steps:

1. Add an empty game object to the scene (click **GameObject > Create Empty**) and position it (at 160, 32, 64).
2. Rename the empty object to **SpawnPoint** in the Hierarchy view.

Next, you want to create the water hazard detector. This will be a simple plane that will sit just below the water. The plane will have a trigger collider (as covered in more detail later in this book), which will detect when a player has fallen in the water. To create the detector, follow these steps:

1. Add a plane to the scene (click **GameObject > 3D Object > Plane**) and position it (at 86, 27, 51). Scale the plane (10, 1, 10).
2. Rename the plane to **WaterHazardDetector** in the Hierarchy view.
3. Check the **Convex** and **Is Trigger** checkboxes on the Mesh Collider component in the Inspector view (see Figure 7.3).



FIGURE 7.3

The Inspector view of the WaterHazardDetector object.

Next you want to add the finish zone to your game. This zone will be a simple object with a point light on it so that the player knows where to go. The object will have a capsule collider attached to it so that it will know when a player can enter the zone. To add the finish zone object, follow these steps:

1. Add an empty game object to the scene and position it at (26, 32, 24).
2. Rename the object to **Finish** in the Hierarchy view.
3. Add a light component to the finish object. (With the object selected, click **Component > Rendering > Light**.) Change the type to **Point** if it isn't already and set the range to 35 and intensity to 3.
4. Add a capsule collider to the finish object by selecting the object and clicking **Component > Physics > Capsule Collider**. Check the **Is Trigger** check box and change the Radius property to 9 in the Inspector view (see Figure 7.4).

**FIGURE 7.4**

The Inspector view of the Finish object.

The final object you need to create is the game control object. This object doesn't technically need to exist. You could instead just apply its properties to some other persistent object in the game world such as the Main Camera. You generally create its own object to prevent any accidental deletion, though. During this phase of development, the game control object is very basic. It will be used more later on. To create the game control object, follow these steps:

1. Add an empty game object to the scene.
2. Rename the game object to **GameControl** in the Hierarchy view.

Adding Scripts

As mentioned earlier, scripts specify behaviors for your game objects. In this section, you apply scripts to your game objects. At this point, it is not important for you to understand what these scripts do. The first thing you need to do is add the scripts to your project:

1. Create a **Scripts** folder under Assets in the Project view.
2. Locate the Scripts folder in the book assets for Hour 7.
3. Click and drag the scripts from the book asset's Scripts folder into the Scripts folder in Unity. There should be three scripts: FinishScript, GameControlScript, and RespawnScript.

Once the scripts are in your project, applying them is easy. To apply a script, simply drag it from the Project view onto whatever object you want to apply it to (see Figure 7.5). Apply the following scripts:

- ▶ Apply the FinishScript to the Finish game object.
- ▶ Apply the GameControlScript to the GameControl object.
- ▶ Apply the RespawnScript to the WaterHazardDetector object.

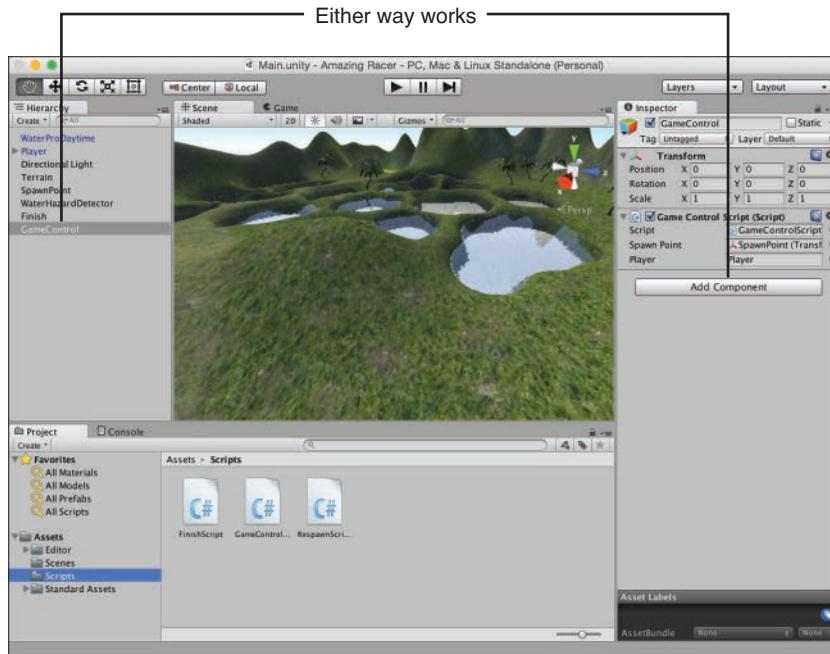


FIGURE 7.5

Applying scripts by dragging them onto game objects.

TRY IT YOURSELF ▼

Import and Attach Scripts

Now let's import the scripts from the book files, and attach them to the correct objects.

1. Locate the three scripts in your book files, and drag them into your Assets folder.
2. Drag the FinishScript.cs script from Assets onto the Finish game object in the hierarchy.
3. Select the GameControl object in the hierarchy. Click Add Component in the Inspector then select the GameControl.cs. This is an alternative way of adding a script component to a game object.
4. Drag the RespawnScript.cs from Assets onto the WaterHazardDetector in the hierarchy.

Connecting the Scripts Together

If you read through the scripts, you noticed that they all have placeholders for other objects. These placeholders allow one script to talk to another script. You see that for every placeholder that existed in the scripts, there is a property in the component for that script in the Inspector view. Just like with scripts, you apply the objects to the placeholders by clicking and dragging (see Figure 7.6).

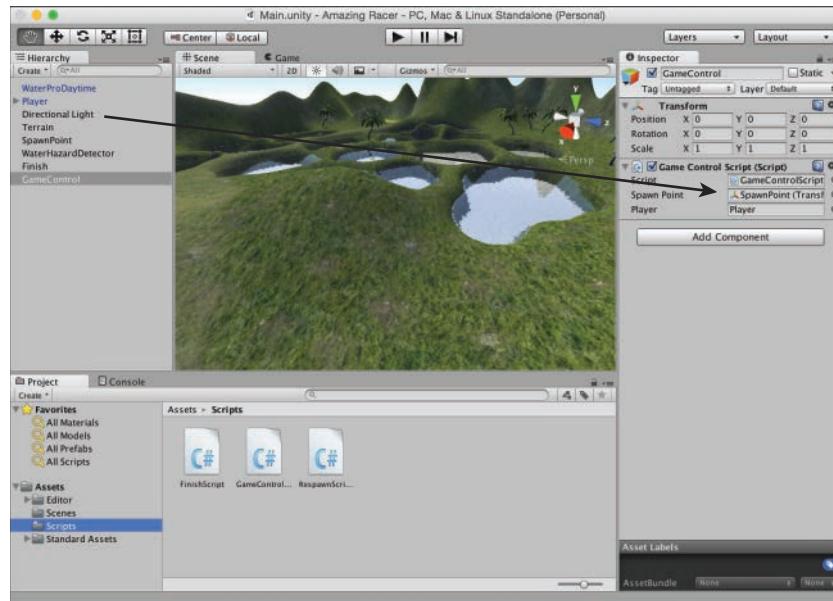


FIGURE 7.6
Moving game objects onto placeholders.

You start connecting objects with the WaterHazardDetector first. Select the **WaterHazardDetector** in the Hierarchy view and notice how it has the Respawn Script component. This is the result of applying the respawn script in the previous section. You also notice that the respawn component has a Respawn Point property. This property is a placeholder for the SpawnPoint game object you made previously. With the WaterHazardDetector object selected, click and drag the **SpawnPoint** object from the Hierarchy view onto the Respawn Point property of the Respawn Script component. Now, whenever players fall into the water hazard, they will get moved back to the spawn point at the beginning of the level.

The next object to set up is the Finish game object. With the Finish game object selected, click and drag the **GameControl** object from the Hierarchy view onto the Game Control Script property of the Finish Script component in the Inspector view. Now, whenever the player enters the finish zone, the game control will be notified.

The last object you need to set up is the GameControl. Click and drag the **SpawnPoint** object onto the Spawn Point property of the Game Control Script component of the GameControl. Finally click and drag the **Player** object (this is the character controller) onto the Player property.

That's all there is to connecting the game objects. Your game is now completely playable! Some of this might not make sense right now, but the more you study it and work with it, the more intuitive it becomes.

▼ TRY IT YOURSELF

Attach Game Objects To Scripts

Let's give these scripts the game objects they need to function correctly.

1. With the WaterHazardDetector object selected, click and drag the **SpawnPoint** object from the Hierarchy view onto the Respawn Point property of the Respawn Script component.
2. With the Finish game object selected, click and drag the **GameControl** object from the Hierarchy view onto the Game Control Script property of the Finish Script component.
3. With the GameControl object selected, click and drag the **SpawnPoint** object onto the Spawn Point property of the Game Control Script component.
4. Still with the GameControl object selected, click and drag the **Player** object (this is the character controller) onto the Player property.

Playtesting

Your game is now done, but it is not time to rest just yet. Now you have to begin the process of playtesting. Playtesting is where you play a game with the intention of finding errors or things that just aren't as fun as you thought they would be. A lot of times, it can be beneficial to have other people playtest your games so that they can tell you what makes sense to them and what they found enjoyable.

If you followed all the steps previously described, there shouldn't be any errors (commonly called *bugs*) for you to find. The process of determining what parts are fun, however, is completely at the discretion of the person making the game. Therefore, this part will be left up to you. Play the game and see what you don't like. Take notes on the things that aren't enjoyable to you. Don't just focus on the negative, though. Also find the things that you like. Your ability to change these things may be limited at the moment, so write them down. Plan on how you would change the game for the better if given the opportunity.

One simple thing you can tweak right now to make the game more enjoyable is the player's speed. If you have played the game a couple of times, you might have noticed that the character moves too slowly, and that can make the game feel very long and drawn out. To make the character fast, you need to modify the First Person Controller (Script) component on the Player object. Expand the **Movement** property in the Inspector view and change run speed (see Figure 7.7). The sample project has this set at 10. Try that and see how you like it. Try faster or slower speeds and pick one you enjoy. You did notice that holding Shift makes you run, right?

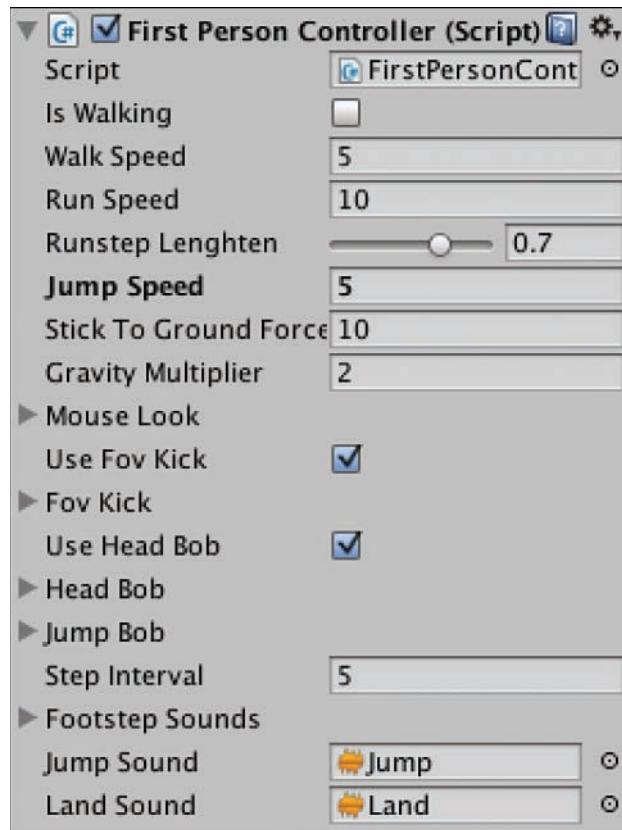


FIGURE 7.7

Changing the player's walk and run speed.

Summary

In this hour, you made your first game in Unity. You started by designing the various aspects of the game's concept, rules, and requirements. From there, you built the game world and added environment effects. Then, you added the game objects required for interactivity. You applied scripts to those game objects and connected them together. Finally, you playtested your game and noted the things you liked and didn't like.

Q&A

- Q. This seems over my head. Am I doing something wrong?**
- A.** Not at all! This process can feel very alien to someone who is not used to it. Keep reading and studying the materials and it will all begin to come together. The best thing you can do is pay attention to how the objects connect to each other through the scripts.
- Q. You didn't cover how to build and deploy the game. Why not?**
- A.** Building and deployment is its own hour later on. There are many things to consider when building a game, and at this point you should just focus on the concepts required to develop it.
- Q. Why couldn't we make a game without scripts?**
- A.** As mentioned earlier, scripts define the behavior of objects. It is very difficult to have a coherent game without some form of interactive behavior. The only reason you are building a game in Hour 7 before learning scripting in Hours 8 and 9 is that you should reinforce the topics you have already learned before moving on to something different.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

- 1.** What are a game's requirements?
- 2.** What is the win condition of this game?
- 3.** Which object is responsible for controlling the flow of the game?
- 4.** Why do we playtest a game?

Answers

1. The requirements are the list of assets that will need to be created to make the game.
2. Trick question! There is no explicit win condition for this game. It is assumed that the player wins when he or she gets a better time than previous attempts. This is not built into the game in any way, though.
3. The game controller. In this game, it was called GameControl.
4. To discover bugs and determine what parts of the game work the way we want them to.

Exercise

The best part about making games is that you can get to make them the way you want. Following a guide can be a good learning experience, but you don't get the satisfaction of making a custom game. In this exercise, you have an opportunity to modify the game a little to make something more unique. Exactly how you want to change the game is up to you. Some suggestions are listed here:

- ▶ Try to add multiple finish zones. See whether you can place them in a way that offers the players more choices.
- ▶ Modify the terrain to have more or different hazards. As long as the hazards are built like the water hazard (including the script), they will work just fine.
- ▶ Try having multiple spawn locations. Make it so some of the hazards move you to a second or third spawn point.
- ▶ Modify the sky and textures to create an alien world. Make the world experience unique.

This page intentionally left blank

HOUR 8

Scripting—Part 1

What You'll Learn in This Hour:

- ▶ The basics of scripts in Unity
- ▶ How to use variables
- ▶ How to use operators
- ▶ How to use conditionals
- ▶ How to use loops

You have so far learned how to make objects in Unity. However, those objects have been a bit boring. How useful is a cube that just sits there? It would be much better to give the cube some custom action to make it interesting in some way. What you need are scripts. Scripts are files of codes that are used to define complex or nonstandard behaviors for objects. In this hour, you learn about the basics of scripting. You begin by looking at how to start working with scripts in Unity. You learn how to create scripts and use the scripting environment. Then, you learn about the various components of a scripting language. These components include variables, operators, conditionals, and loops.

TIP

Sample Scripts

Several of the scripts and coding structures mentioned in this hour are available in the book assets for Hour 8. Be sure to check them out for additional learning.

CAUTION

New to Programming

If you have never programmed before, this might all seem strange and confusing. As you work through this hour, try your best to focus on how things are structured and why they are structured that way. Remember that programming is purely logical. If a program is not doing something you want it to, it is because you have not told it how to do it correctly. Sometimes it is up to you to change the way you think. Take this hour slowly, and be sure to practice.

Scripts

As mentioned earlier, scripts are a way to define behavior. They attach to objects in Unity just like other components and give them interactivity. There are generally three steps to working with scripts in Unity:

1. Create the script.
2. Attach the script to one or more game objects.
3. If the script requires it, populate any properties with values or other game objects. (This step is talked about later.)

Creating Scripts

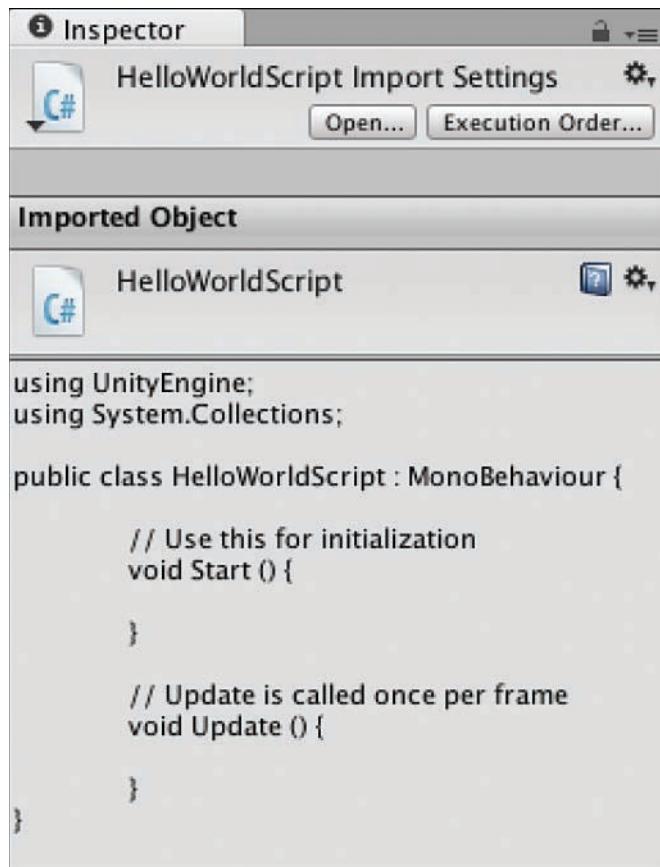
Before creating scripts, it is best to create a Scripts folder under the Assets folder in the Project view. Once you have a folder to contain all of your scripts, simply right-click the folder and select **Create > C# Script**. Once created, you need to give your script a name before continuing.

NOTE

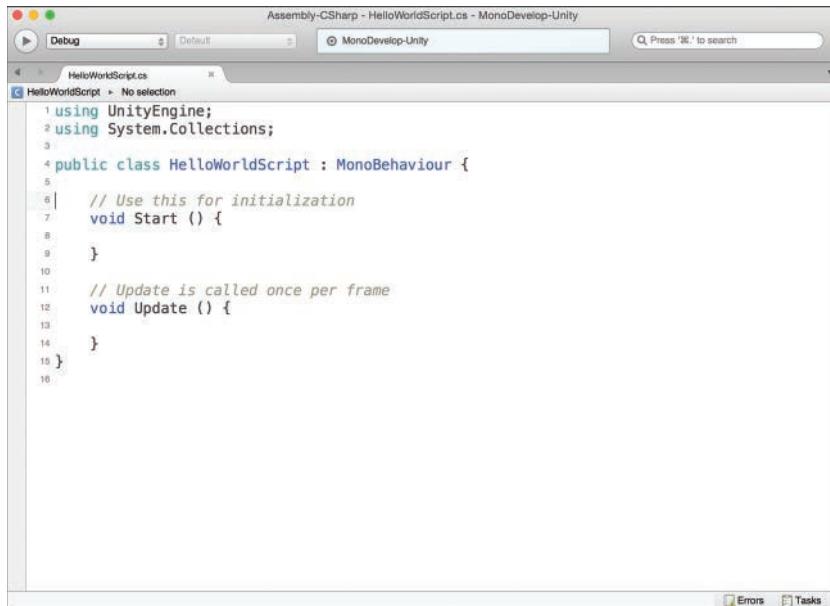
Scripting Language

Unity allows you to write scripts in C# or JavaScript. This book uses the C# language for all scripts because it is a little more versatile and powerful. If you have a preference for a different language, feel free to work in that language.

Once the script is created, you can view and modify it. Clicking the script in the Project view will enable you to see the contents of the script in the Inspector view (see Figure 8.1). Double-clicking the script in the Project view opens your default editor, which will enable you to add code to the script. Assuming that you have installed the default components and haven't changed anything, double-clicking a file will open the MonoDevelop development software (see Figure 8.2).

**FIGURE 8.1**

The Inspector view preview of a script.

**FIGURE 8.2**

The MonoDevelop software with the editor window showing.

▼ TRY IT YOURSELF

Creating a Script

Let's create a script for you to use in this section:

1. Create a new project or scene. Add a **Scripts** folder to the Project view.
2. Right-click the Scripts folder and choose **Create > C# Script**. Name the script **HelloWorldScript**.
3. Double-click the new script file and wait for MonoDevelop to open. In the editor window of MonoDevelop (refer to Figure 8.2 for the editor window), erase all the text and replace it with the code from this listing:

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

```
public class HelloWorldScript : MonoBehaviour {  
  
    // Use this for initialization  
    void Start () {  
        print ("Hello World");  
    }  
  
    // Update is called once per frame  
    void Update () {  
  
    }  
}
```

4. Save your script by clicking **File > Save** or by pressing **Ctrl+S (Command+S on a Mac)**. Back in Unity, confirm in the Inspector view that the script has been changed and run the scene. Notice how nothing happens. The script was created, but it does not work until it is attached to an object. That is covered next.

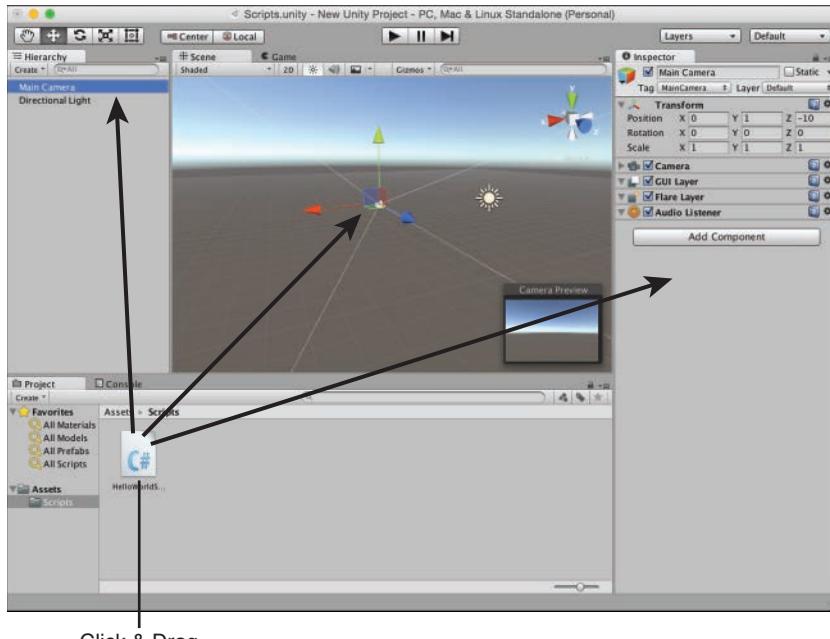
NOTE

MonoDevelop

MonoDevelop is a robust and complex piece of software that is bundled with Unity. It is not actually a part of Unity. Therefore, we do not cover it in any depth. The only part of MonoDevelop you need to be familiar with right now is the editor window. If there is anything else you need to know about MonoDevelop, it is covered in the hour where it is needed.

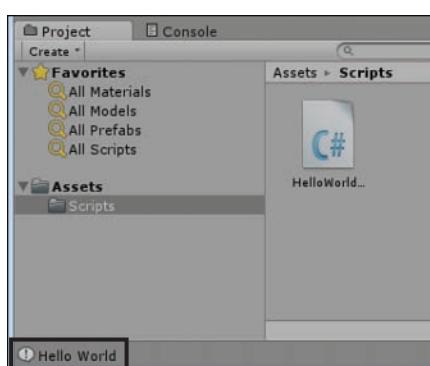
Attaching a Script

To attach a script to a game object, just click the script in the Project view and drag it onto the object (see Figure 8.3). You can drag the script onto the object in the Hierarchy view, Scene view, or the Inspector view (assuming the object is selected). Once attached to an object, the script will become a component of that object and will be visible in the Inspector view.

**FIGURE 8.3**

Click and drag the script onto the desired object.

To see this in action, attach the `HelloWorldScript` you created earlier to the Main Camera. You should now see a component named `Hello World Script (Script)` in the Inspector view. If you run the scene, you see `Hello World` appear at the bottom of the screen in the Console window (see Figure 8.4).

**FIGURE 8.4**

The words `Hello World` output when running the scene.

Anatomy of a Basic Script

In the preceding section, you modified a script to output some text to the screen, but the contents of the script were not explained. In this section, you look at the default template that is applied to every new C# script. Note that scripts written in JavaScript will have the same components even if they look a little different. Listing 8.1 contains the full code that is generated for you by Unity when you make a new script. Listing 8.1 assumes that the script file created was named HelloWorldScript.

LISTING 8.1 Default Script Code

```
using UnityEngine;
using System.Collections;

public class HelloWorldScript : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

This code can be broken down into three parts.

The Using Section

The first part lists the libraries that this script will be using. It looks like this:

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

Generally speaking, you won't be changing this section and should just leave it alone for the time being.

The Class Declaration Section

The next part is called a class declaration. Every script contains a class that is named after the script. It looks like the following:

```
public class HelloWorldScript : MonoBehaviour { }
```

All the code in between the opening bracket { and closing bracket } will be a part of this class and therefore a part of the script. All of your code should go between these brackets. Once again, as above, you rarely change this and should just leave it alone for now.

The Class Contents

The section in between the opening and closing brackets of the class is considered to be “in” the class. All of your code will go here. By default, a script contains two methods inside the class, **Start** and **Update**:

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

}

// Update is called once per frame
void Update () {

}
```

Methods are covered in greater detail in the next hour. For now, just know that any code put inside the **Start** method will run when a scene first starts. Any code put inside the **Update** method will run as fast as possible, even hundreds of times a second.

TIP

Comments

Programming languages have a way for the author of the code to leave messages for those who read the code later. These messages are called comments. Any words that follow two forward slashes (//) will be “commented out.” This means that the computer will skip over them and not attempt to read them as code. You can see an example of commenting in the “Creating a Script” Try It Yourself.

NOTE

The Console

There is another window in the Unity editor that has not been mentioned until now: the Console. Basically, the Console is a window that contains text output from your game. Often, when there is an error or output from a script, messages will get written to the Console. Figure 8.5 shows you the Console and how to access it. If the Console window isn’t visible, you can also access it by clicking **Window > Console**.

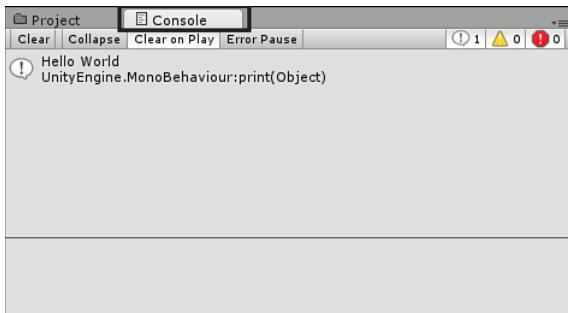


FIGURE 8.5
The Console window.

TRY IT YOURSELF ▼

Using the Built-In Methods

Let's try out the built-in methods **Start** and **Update** and see how they work. The completed **ImportantFunctions** script is available in the book assets for Hour 8. Try to complete the exercise that follows on your own, but if you get stuck, refer to the book assets:

1. Create a new project or scene. Add a script to the scene named **ImportantFunctions**. Double-click the script to open MonoDevelop.

2. Inside the script, add the following line of code to the **Start** method:

```
print ("Start runs before an object Updates");
```

3. Save the script, and in Unity attach it to the Main Camera. Run the scene and notice the message that appears in the Console window.

4. Back in MonoDevelop, add the following line of code to the **Update** method:

```
print ("This is called once a frame");
```

5. Save the script and quickly start and stop the scene in Unity. Notice how, in the Console, there is a single line of text from the **Start** method and a bunch of lines from the **Update** method.

Variables

Sometimes you want to use the same bit of data more than once in a script. What you need is a placeholder for data that can be reused. These placeholders are called *variables*. Unlike traditional math, variables in programming can contain more than just numbers. They can hold words, complex objects, or other scripts.

Creating Variables

Every variable has a name and a type. These are given to the variable when it is created. You create a variable with the following syntax:

```
<variable type> <name>;
```

So, to create an integer named num1, you type the following:

```
int num1;
```

Table 8.1 contains all the primitive (or basic) variable types and the types of data they can hold.

NOTE

Syntax

The term *syntax* refers to the rules of a programming language. The syntax dictates how things are structured and written so that the computer knows how to read them. You may have noticed that every statement, or command, in our scripts ends with a semicolon. This is also a part of the C# syntax. Forgetting the semicolon will cause your script to not work. If you want to know more about the syntax of C#, check out the C# wiki at http://en.wikipedia.org/wiki/C_Sharp_syntax.

TABLE 8.1 C# Variable Types

Type	Description
int	Short for integer, the int stores positive or negative whole numbers.
float	The float stores floating point data (such as 3.4) and is the default number type in Unity.
double	The double also stores floating point numbers; however it is not the default number type in Unity. It can generally hold bigger numbers than floats.
bool	Short for Boolean, the bool stores true or false (actually written in code as true or false).
char	Short for character, the char stores a single letter, space, or special character (such as a, 5, or !). Char values are written out with single quotes ('A').
string	The string type holds entire words or sentences. String values are written out with double quotes ("Hello World").

Variable Scope

The variable scope refers to where a variable is able to be used. As you have seen in scripts, classes and methods use open and close brackets to denote what belongs to them. The area between the two brackets can often be referred to as a *block*. The reason that this is important is that variables are only able to be used in the blocks in which they are created. So if a variable is created inside the `Start` method of a script, it will not be available in the `Update` method. Attempting to use a variable where it is not available will result in an error. They are two different blocks. If a variable is created in the class, but outside of a method, it will be available to both methods because both methods are in the same block as the variable (the class block). Listing 8.2 demonstrates this.

LISTING 8.2 Demonstration of Class and Local Block Level

```
//This is in the "class block" and will
//be available everywhere in this class
private int num1;

void Start () {
    //this is in a "local block" and will
    //only be available in the Start method
    int num2;
}
```

Public and Private

If you look in Listing 8.2, you see the keyword `private` appear before `num1`. This is called an access modifier, and it is needed only for variables declared at the class level. There are two access modifiers you need to use: `private` and `public`. A lot that can be said about the two access modifiers, but what you need to know is how they affect variables at this level. Basically, `private` variables (variables with the word `private` before them) are only usable inside the file they are created in. Other scripts and the editor cannot see them or modify them in any way. They are intended for internal use only. `Public` variables, in contrast, are visible to other scripts and even the Unity editor. This makes it easy for you to change the values of your variables on-the-fly within Unity.

TRY IT YOURSELF ▼

Modifying Public Variables in Unity

Let's see how public variables are visible in the Unity editor:

1. Create a new C# script and in MonoDevelop add the following line in the class but above the `Start` method:

```
public int runSpeed;
```

- ▼**
2. Save the script and then in Unity attach it to the Main Camera.
 3. Select the Main Camera and look in the Inspector view. Notice the script you just attached as a component. Now notice that the component has a new property: Run Speed. You can modify that property in the Inspector view and the change will be reflected in the script at runtime. See Figure 8.6 to see the component with the new property. This figure assumes that the script created was named ImportantFunctions.

**FIGURE 8.6**

The new Run Speed property of the script component.

Operators

All the data in variables is worthless if you have no way of accessing or modifying it. Operators are special symbols that enable you to perform modifications on data. They generally fall into one of four categories: arithmetic operators, assignment operators, equality operators, and logical operators.

Arithmetic Operators

Arithmetic operators perform some standard mathematical operation on variables. They are generally used only on number variables, although a few exceptions exist. Table 8.2 describes the arithmetic operators.

TABLE 8.2 Arithmetic Operators

Operator	Description
+	Addition. Adds two numbers together. In the case of strings, the + sign concatenates, or combines, them together. "Hello" + "World"; //produces "HelloWorld"
-	Subtraction. Reduces the number on the left by the number on the right.
*	Multiplication. Multiplies two numbers together.
/	Division. Divides the number on the left by the number on the right.
%	Modulus. Divides the number on the left by the number on the right but does not return the result. Instead, the modulus returns the remainder of the division. 10 % 2; //returns 0 6 % 5; //returns 1 24 % 7; //returns 3

Arithmetic operators can be cascaded together to produce more complex math strings:

```
x + (5 * (6 - y) / 3);
```

Arithmetic operators work in the standard mathematic order of operations. Math is done left to right, with parentheses done first, multiplication and division done second, and addition and subtraction done third.

Assignment Operators

Assignment operators are just what they sound like. They assign a value to a variable. The most notable assignment operator is the equals sign, but there are more that combine multiple operations together. All assignment in C# is right to left. That means that whatever is on the right side gets moved to the left:

```
x = 5; //This works. It sets the variable x to 5.  
5 = x; //This does not work. You cannot assign a variable to a value (5).
```

Table 8.3 describes the assignment operators.

TABLE 8.3 Assignment Operators

Operator	Description
=	Assigns the value on the right to the variable on the left.
+ =, - =, *=, /=	Shorthand assignment operator that performs some arithmetic operation based on the symbol used and then assigns the result to whatever is on the left. <code>x = x + 5; //Adds 5 to x and then assigns it to x</code> <code>x += 5; //Does the same as above, only shorthand</code>
++, -	Another shorthand operator. These are called the increment and decrement operators. They increase or decrease a number by 1. <code>x = x + 1; //Adds 1 to x and then assigns it to x</code> <code>x++; //Does the same as above, only shorthand</code>

Equality Operators

Equality operators compare two values. The result of an equality operator will always be either true or false. Therefore, the only variable type that can hold the result of an equality operator is a Boolean. (Remember that Booleans can only contain true or false.) Table 8.4 describes the equality operators.

TABLE 8.4 Equality Operators

Operator	Description
<code>==</code>	Not to be confused with the assignment operator (<code>=</code>), this returns true only if the two values are equal. Otherwise, it returns false. <code>5 == 6; //Returns false</code> <code>9 == 9; //Returns true</code>
<code>>, <</code>	These are the “greater than” and “less than” operators. <code>5 > 3; //Returns true</code> <code>5 < 3; //Returns false</code>
<code>>=, <=</code>	These are similar to the “greater than” and “less than” except that they are the “greater than or equal to” and “less than or equal to” operators. <code>3 >= 3; //Returns true</code> <code>5 <= 9; //Returns true</code>
<code>!=</code>	This is the “not equal” operator and returns true if the two values are not the same. Otherwise, it returns false. <code>5 != 6; //Returns true</code> <code>9 != 9; //Returns false</code>

TIP**Additional Practice**

In the book assets for Hour 8, there is a script called `EqualityAndOperations.cs`. Be sure to look through it for some additional practice with the various operators.

Logical Operators

Logical operators enable you to combine two or more Boolean values (true or false) into a single Boolean value. They are useful for determining complex conditions. Table 8.5 describes the logical operators.

TABLE 8.5 Logical Operators

Operator	Description
<code>&&</code>	Known as the AND operator, this compares two Boolean values and determines whether they are both true. If either, or both, of the values is false, this returns false: <code>true && false; //Returns false</code> <code>false && true; //Returns false</code> <code>false && false; //Returns false</code> <code>true && true; //Returns true</code>

Operator	Description
	Known as the OR operator, this compares two Boolean values and determines whether either of them are true. If either, or both, of the values is true, this returns true: <pre>true false; //Returns true false true; //Returns true false false; //Returns false true true; //Returns true</pre>
!	Known as the NOT operator, this returns the opposite of a Boolean value: <pre>!true; //Returns false !false; //Returns true</pre>

Conditionals

Much of the power of a computer lies within its ability to make rudimentary decisions. At the root of this power lies the Boolean true and false. You can use these Boolean values to build conditionals and steer a program down a unique course. As you are building your flow of logic through code, just remember that a machine can only make a single, simple decision at a time. Put enough of those decisions together, though, and you can build complex interactions.

The if Statement

The basis of conditionals is the `if` statement. And it is structured like this:

```
if ( <some Boolean condition> ) {
    //do something
}
```

The `if` structure can be read as “if this is true, do this.” So, if you want to output “Hello World” to the Console if the value of `x` is greater than 5, you could write the following:

```
if (x > 5) {
    print("Hello World");
}
```

Remember that the contents of the `if` statement condition must evaluate to either a true or a false. Putting numbers, words, or anything else in there will not work:

```
if ("Hello" == "Hello") //Correct

if (x + y) //Incorrect
```

Finally, any code that you want to run if the condition evaluates to true must go inside the opening and closing brackets that follow the `if` statement.

TIP**Odd Behavior**

Conditional statements use a specific syntax and can give you strange behaviors if you don't follow it exactly. You may have an `if` statement in your code and notice that something isn't quite right. Maybe the condition code runs all the time even when it shouldn't. You may also notice that it never runs, even if it should. You want to be aware of two common causes for this. First, the `if` condition does not have a semicolon after it. If you write an `if` statement with a semicolon, the code following it will always run. Second, be sure that you are using the equality operator (`==`) and not the assignment operator (`=`) inside the `if` statement. Doing otherwise will lead to bizarre behavior:

```
if (x > 5) //Incorrect
if (x = 5) //Incorrect
```

The `if / else` Statement

The `if` statement is nice for conditional code, but what if you want to diverge your program down two different paths? The `if / else` statement will enable you to do that. The `if / else` is the same basic premise of the `if` statement, except it can be read more like "if this is true do this, else do this other thing." The `if / else` statement is written like this:

```
if (<some Boolean condition>) {
    //Do something
} else {
    //Do something else
}
```

For example, if you want to print "X is greater than Y" to the Console if the variable `x` is larger than the variable `y`, or you want to print "Y is greater than X" if `x` isn't bigger than `y`, you could write the following:

```
if (x > y) {
    print("X is greater than Y");
} else {
    print("Y is greater than X");
}
```

The `if / else if` Statement

Sometimes you want your code to diverge down one of many paths. You might want the user to be able to pick from a selection of options (such as a menu for example). The `if /else if` is structured in much the same way as the previous two structures, except that it has multiple conditions:

```

if( <some Boolean condition> ) {
    //Do something
} else if ( <some other Boolean condition> ) {
    //Do something else
} else { //The else is optional in the IF / ELSE IF statement
    //Do something else
}

```

For example, if you want to output a person's letter grade to the console based on his percentage, you could write the following:

```

if (grade >= 90) {
    print ("You got an A");
} else if (grade >= 80) {
    print ("You got a B");
} else if(grade >= 70) {
    print ("You got a C");
} else if (grade >= 60) {
    print ("You got a D");
} else {
    print ("You got an F");
}

```

TIP

Single-Line if Statements

Strictly speaking, if your `if` statement code is only a single line, you do not need to have the curly braces. Therefore, your code, which may look like this

```

if (x > y) {
    print("X is greater than Y");
}

```

could also be written as follows:

```

if (x > y)
    print("X is greater than Y");

```

However, we recommend you always include the curly braces. This can save a lot of confusion later as your code gets more complex. Code inside curly braces is referred to as a code-block and is all executed together.

Iteration

You have so far seen how to work with variables and make decisions. This is certainly useful if you want to do something like add two numbers together. But what if you want to add all the numbers between 1 and 100 together? What about between 1 and 1000? You definitely would not want to type all of that redundant code out. Instead, you can use something called *iteration* (commonly referred to as *looping*). There are two primary types of loops for you to work with: the `while` loop and the `for` loop.

The `while` Loop

The `while` loop is the most basic form of iteration. It follows a similar structure to an `if` statement:

```
While ( <some Boolean condition> ) {  
    //do something  
}
```

The only difference is that an `if` statement only runs its contained code once, whereas a loop will run the contained code over and over until the condition becomes false. Therefore, if you want to add together all the numbers between 1 and 100 and then output them to the console, you could write something like this:

```
int sum = 0;  
int count = 1;  
  
while (count <= 100) {  
    sum += count;  
    count++;  
}  
  
print(sum);
```

As you can see, the value of `count` will start at 1 and increase by 1 every iteration, or execution of the loop, until it equals 101. When `count` equals 101, it will no longer be less than or equal to 100, and the loop will exit. Omitting the `count++` line will result in the loop running infinitely (so be sure it's there). During each iteration of the loop, the value of `count` is added to the variable `sum`. Once the loop exits, the `sum` is written to the console.

In summation, a `while` loop will run the code it contains over and over as long as its condition is true. Once its condition becomes false, it stops looping.

The `for` Loop

The `for` loop follows the same idea as the `while` loop, except it is structured a bit differently. As you saw in the previous code for the `while` loop, you had to create a count variable, you had to test the variable (as the condition), and you had to increase the variable all on three separate lines. The `for` loop condenses that syntax down to a single line. It looks like this:

```
For (<create a counter>; <Boolean conditional>; <increment the counter >) {  
    //Do something  
}
```

The `for` loop has three special *compartments* for controlling the loop. Notice the semicolons, not commas, in between each section in the `for` loop header. The first compartment creates a variable to be used as a counter (a common name for the counter is `i`, short for iterator). The second compartment is the conditional statement of the loop. The third compartment handles increasing or decreasing the counter. The previous `while` loop example can be rewritten using a `for` loop. It would look like this:

```
int sum = 0;  
  
for (int count = 1; count <= 100; count++) {  
    sum += count;  
}  
  
print(sum);
```

As you can see, the different parts of the loop get condensed down and take up less space. You can see that the `for` loop is really good at things like counting.

Summary

In this hour, you took your first steps into video game programming. You started by looking at the basics of scripting in Unity. You learned how to make and attach scripts. You also looked at the basic anatomy of a script. From there, you studied the basic logical components of a program. You worked with variables, operators, conditionals, and loops.

Q&A

Q. How much programming is required to make a game?

A. Most games use some form of programming to define complex behaviors. The more complex the behaviors need to be, the more complex the programming needs to be. If you want to make games, you should definitely become comfortable with the concepts of programming. This is true even if you don't intend to be the primary developer for a game. With that in mind, know that this book will show you everything you need to know to make your first few simple games.

Q. Is this all there is to scripting?

- A. Yes and no. Presented in this text are the fundamental blocks of programming. They never really change; they just get applied in new and unique ways. That said, a lot of what is presented here is simplified because of the complex nature of programming in general. If you want to learn more about programming, you should read books or articles specifically on the subject.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What three languages do Unity allow you to program with?
2. True or False: The code in the `Start` method runs at the start of every frame.
3. Which variable type is the default floating point number type in Unity?
4. Which operator returns the remainder of division?
5. What is a conditional statement?
6. Which loop type is best suited for counting?

Answers

1. C# and JavaScript.
2. False. The `Start` method runs at the beginning of the scene. The `Update` method runs every frame.
3. `float`.
4. The modulus.
5. A code structure that allows the computer to choose a code path based on a simple decision.
6. The `for` loop.

Exercise

It can often be helpful to view coding structures as building blocks. Alone, each piece is simple. Put together, however, they can build complex entities. In what follows you find multiple programming challenges. Use the knowledge you have gained this hour to build a solution to the problems. Put

each solution in its own script and attach the scripts to the Main Camera of a scene to ensure that they work. You can find the solution to this exercise in the book assets for Hour 8.

1. Write a script that adds together all the even numbers from 2 to 499. Output the result to the console.
2. Write a script that outputs all of the numbers 1–100 to the console. Don't output multiples of 3 or 5, though. Instead, output "Programming is Awesome!" Hint: You can tell whether a number is a multiple of another number if the result of a modulus operation is 0, for example `12 % 3 == 0` because 12 is a multiple of 3.
3. In the Fibonacci sequence, you determine a number by adding the two previous numbers together. The sequence starts with 0, 1, 1, 2, 3, 5 . . . Write a script that determines the first 20 places of the Fibonacci sequence and outputs them to the console.

This page intentionally left blank

HOUR 9

Scripting—Part 2

What You'll Learn in This Hour:

- ▶ How to write methods
- ▶ How to capture user input
- ▶ How to work with local components
- ▶ How to work game objects

In the preceding hour, you learned about the basics of scripting in Unity. In this hour, you take what you have learned and use it to complete more meaningful tasks. You begin by examining methods. You learn what they are, how they work, and how to write them. Then you get hands on with user input. After that, you examine how to access components from scripts. You wrap up the hour by learning how to access other game objects and their components with code.

TIP

Sample Scripts

Several of the scripts and coding structures mentioned in this hour are available in the book assets for Hour 9. Be sure to check them out for additional learning.

Methods

Methods, often called *functions*, are modules of code that can be called and used independently of each other. Each method generally represents a single task or purpose, and often many methods can work together to achieve complex goals. Consider the two methods you have seen so far: Start and Update. Each represents a single and concise purpose. The Start method contains all the code that is run for an object when the scene first begins. The Update method contains the code that is run every frame of the scene.

NOTE**Method Shorthand**

You have seen so far that whenever the `Start` method is mentioned, the word `method` has followed it. It can become cumbersome to always have to specify that a word used is a method. You can't write just `Start`, though, because people wouldn't know if you meant the word or a method. A shorter way of handling this is to use parentheses with the word. So, the method `Start` can be rewritten as just `Start()`. If you ever see something written like `SomeWords()`, you can know instantly that the writer is talking about a method named `SomeWords`.

Anatomy of a Method

Before working with methods, you should look at the different parts that compose them. What follows is the general format of a method:

```
<return type><name>(<parameters>) {  
    <Inside the method's block>  
}
```

Method Name

Every method must have a unique name. Though the rules that govern proper names are determined by the language used, good general guidelines for method names include the following:

- ▶ Make a method name descriptive. It should be an action and preferably a verb.
- ▶ Avoid spaces in method names. Spaces are not allowed.
- ▶ Avoid special characters (!@*%\$, etc.) in method names. Different languages allow different characters. By not using any, you don't run the risk of there being a problem.

Method names are important because that is both how you identify them and also how you use them.

Return Type

Every method has the ability to return a variable back to whatever code called it. The type of this variable is called the *return type*. If a method returns an integer (a whole number), the return type is an `int`. Likewise, if the method returned a true or false, the return type is `bool`. If a method doesn't return any value, it still has a return type. In that instance, the return type is `void` (meaning nothing). Any method that returns a value will do so with the keyword `return`.

Parameter List

Just as methods can pass a variable back to whatever code called it, the calling code can pass variables in. These variables are called *parameters*. The variables sent into the method are

identified in the parameter list section of the method. An example of a method named `Attack` that takes an integer called `enemyID` would look like this:

```
void Attack(int enemyID)
{}
```

As you can see, when specifying a parameter, you must provide both the variable type and the name. Multiple parameters are separated with a comma.

Method Block

This is where the code of the method actually goes. Every time a method is used, the code inside the method block will run in its entirety.

TRY IT YOURSELF ▼

Identifying Method Parts

Take a moment to review the different parts of a method. Given the following method:

```
int TakeDamage(int damageAmount) {
    int health = 100;
    return health - damageAmount;
}
```

Can you identify the following pieces?

1. What is the method's name?
2. What variable type does the method return?
3. What are the method's parameters? How many are there?
4. What code is in the method's block?

TIP

Methods as Factories

The concept of methods can be confusing for someone who is new to programming. Often, mistakes will be made regarding the parameters and return of methods. A good way to keep it straight is to think of a method as a factory. Factories receive raw materials and use that to make products. Methods work the same way. The parameters are the materials you are passing in to the “factory,” and the return is the final product of that factory. Just think of methods that don’t take parameters as factories that don’t require raw goods. Likewise, think of methods that don’t return anything as factories that don’t produce final products. By imagining method as little factories, you can work to keep the flow of logic straight in your head.

Writing Methods

Now that you understand the components of a method, writing them is easy. Before you begin writing your methods, take a moment and ask yourself three main questions:

1. What specific task will the method achieve?
2. Does the method need any outside data to achieve it?
3. Does the method need to give any data back?

Answering these questions will help you determine the method's name, parameters, and return data.

Consider this example: A player has been hit with a fireball. You need to write a method to simulate this by removing 5 health points. You know what the specific task of this method is. You also know that the task doesn't need any data (because you know it takes 5 points) and should probably give the new health value back. You could write the method like this:

```
int TakeDamageFromFireball() {
    int playerHealth = 100;
    return playerHealth - 5;
}
```

As you can see in this method, the player's health is 100 and 5 is taken away from it. The result (which is 95) is passed back. Obviously, this can be improved. For starters, it is said above that the fireball does 5 points of damage, but what if you want it to do more? You would then need to know exactly how much damage a fireball was supposed to do at any given time. You would need a variable, or in this case a parameter. Your new method could be written as follows:

```
int TakeDamageFromFireball(int damage) {
    int playerHealth = 100;
    return playerHealth - damage;
}
```

Now you can see that the damage is read in from the method and applied to the health. Another place where this can be improved is with the health itself. Currently, players can never lose because their health will always refresh back to 100 before having damage taken out. It would be better to store the player's health elsewhere so that its value was persistent. You could then read it in and remove the damage appropriately. Your method would then look like:

```
int TakeDamageFromFireball(int damage, int playerHealth) {
    return playerHealth - damage;
}
```

By examining your needs, you can build better, more robust methods for your game.

NOTE**Simplification**

In the preceding example, the resulting method simply performs basic subtraction. This is oversimplified for instruction's sake. In a more realistic environment, there are many ways to handle this task. A player's health could be stored in a variable belonging to a script. Doing so would mean that it did not need to be read in. Another possibility is a complex algorithm in the `TakeDamageFromFireball` method where the incoming damage is reduced by some armor value, a player's dodging ability, or a magical shield. If the examples here seem silly, just bear in mind that they are that way to demonstrate various elements of the topic.

Using Methods

Once a method is written, all that is left is to use it. Using a method is often referred to as *calling* or *invoking* the method. To call a method, you just need to write the method's name followed by parentheses and any parameters. So, if you were trying to use a method that was named `SomeMethod`, you would write the following:

```
SomeMethod();
```

If `SomeMethod()` requires an integer parameter, you call it like this:

```
//Method call with a value of 5
SomeMethod(5);

//Method call passing in a variable
int x = 5;
SomeMethod(x); //do not write "int x" here.
```

Note that when you call a method, you do not need to supply the variable type with the variable you are passing in. If `SomeMethod()` returns a value, you want to *catch* it in a variable. The code could look something like this (with a Boolean return type assumed; in reality, it could be anything):

```
bool result = SomeMethod();
```

Using this basic syntax is all there is to writing methods.

TRY IT YOURSELF ▼

Calling Methods

Let's work further with the `TakeDamageFromFireball` method described in the previous section. In this exercise, you call the various forms of the method. You can find the solution for this exercise as `FireBallScript` in the book assets for Hour 9:

- TIP**
1. Create a new project or scene. Locate the FireBallScript in the book assets for Hour 9 and import it into your project. Alternatively, create a C# script called **FireBallScript** and enter in the three `TakeDamageFromFireball` methods described earlier. These should go inside the class definition, at the same level of indent as the `Start()` and `Update()` methods, but outside those two methods.

2. In the `Start` method, call the first `TakeDamageFromFireball()` by typing the following:

```
int x = TakeDamageFromFireball();
print ("Player health: " + x);
```

3. Attach the script to the Main Camera and run the scene. Notice the output in the Console.

Now call the second `TakeDamageFromFireball()` in `Start()` by typing the following (placing it below the first bit of code you typed; no need to remove it):

```
int y = TakeDamageFromFireball(25);
print ("Player health: " + y);
```

4. Again, run the scene and note the output in the console. Finally, call the last `TakeDamageFromFireball()` in `Start()` by typing the following:

```
int z = TakeDamageFromFireball(30, 50);
print ("Player health: " + z);
```

5. Run the scene and note the final output. See how all three methods behave a little differently. Notice how you called each one specifically, and the correct version of the `TakeDamageFromFireball()` method was used based on the parameters you passed in.

TIP

Help Finding Errors

If you are getting errors when you try and run your script, pay attention to the reported line number and character number in the console, at the end of the error message. Furthermore, you can “build” your code inside Monodevelop by using Ctrl or Cmd + B. This will check your code, and point out any errors in context, showing you exactly where the troublesome line is. Try it.

Input

Without player input, *video games* would just be *video*. Player input can come in many different varieties. Inputs can be physical like gamepads, joysticks, keyboards, and mice. There are capacitive controllers such as the relatively new touch screens that are found in modern mobile devices. There are also motion devices like the Wii Remote, the PlayStation Move, and the Microsoft Kinect. Rarer is the audio input that uses microphones and a player’s voice to control a game. In this section, you learn all about writing code to allow the player to interact with your game with physical devices.

Input Basics

With Unity (like most game engines), you can detect specific key presses in code to make it interactive. It is a good idea, however, to avoid doing that. Doing so makes it difficult to allow players to remap the controls to their preference. Thankfully, Unity has a simple system for generically mapping controls. With Unity, you look for a specific *axis* to know whether a player intends a certain action. Then, when the player runs the game, he can choose to make different controls mean different axes.

You can view, edit, and add different axes using the Input Manager. To access the Input Manager, click **Edit > Project Settings > Input**. In the Input Manager, you can see the various axes associated with different input actions. By default, there are 18 input axes, but you can add your own if you want. Figure 9.1 shows the default Input Manager with the horizontal axis expanded.

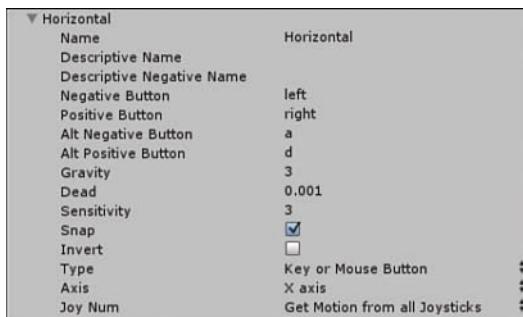


FIGURE 9.1
The Input Manager.

While the horizontal axis doesn't directly control anything (we will write scripts to do that later), it represents that player going sideways. Table 9.1 describes the properties of an axis.

TABLE 9.1 Axis Properties

Property	Description
Name	The name of the axis. This is how you reference it in code.
Descriptive Name/ Descriptive Negative Name	A verbose name for the axis that will appear to the player in the game configuration. The negative is the opposite name. For example: "Go left" and "Go right" would be a name and negative name pair.
Negative Button/ Positive Button	The buttons that pass negative and positive values to the axis. For the horizontal axis, these are the left arrow and the right arrow.
Alt Negative Button/ Alt Positive Button	Alternate buttons to pass values to the axis. For the horizontal axis, these are the A and D keys.

Property	Description
Gravity	How fast the axis will return to 0 once the key is no longer pressed.
Dead	Any input smaller than this value will be ignored. This helps prevent jittering with joystick devices.
Sensitivity	How quickly the axis responds to input.
Snap	When checked, this will cause the axis to immediately go to 0 when the opposite direction is pressed.
Invert	This will invert the controls when checked.
Type	The type of input. The types are keyboard/mouse buttons, mouse movement, and joystick movement.
Axis	The corresponding axis from an input device. This doesn't apply to buttons.
Joy Num	Which joystick to get input from. By default, this gets input from all joysticks.

Input Scripting

Once your axes are set up in the Input Manager, working with them in code is simple. To access any of the player's input, you will be using the `Input` object. More specifically, you will be using the `GetAxis` method of the input object. `GetAxis()` reads the name of the axis in as a string and returns back the value of that axis. So, if you want to get the value of the horizontal axis, you type the following:

```
float hVal = Input.GetAxis("Horizontal");
```

In the case of the horizontal axis, if the player is pressing the left arrow (or the A key), `GetAxis()` will return a negative number. If the player is pressing the right arrow (or the D key), the method will return a positive value.

TRY IT YOURSELF

Reading in User Input

Let's work with the vertical and horizontal axes to get a better idea of how to use player input:

1. Create a new project or scene. Add a script to the project named `PlayerInput`. Attach the script to the Main Camera.

- 2.** Add the following code to the `Update` method in the `PlayerInput` script. This must be in `Update` so that it continuously reads the input.

```
float hVal = Input.GetAxis("Horizontal");
float vVal = Input.GetAxis("Vertical");

if(hVal != 0) {
    print("Horizontal movement selected: " + hVal);
}
if(vVal != 0) {
    print("Vertical movement selected: " + vVal);
}
```

- 3.** Save the script and run the scene. Notice what happens on the Console when you press the arrow keys. Now try out the W, A, S, and D keys. If you don't see anything, click into the Game window with the mouse and try again.

Specific Key Input

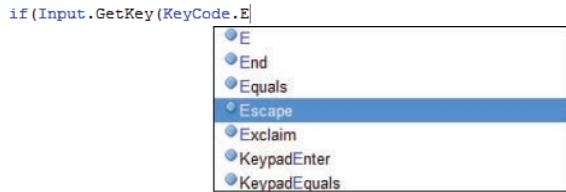
Although you generally want to deal with the generic axes for input, sometimes you do want to determine whether a specific key has been pressed. To do so, you will again be using this `input` object. This time, however, you use the `GetKey` method. This method reads in a special code that corresponds to a specific key. It then returns true if the key is currently down and false if the key is not currently down. To determine whether the K key is currently pressed, you type the following:

```
bool isKeyDown = Input.GetKey(KeyCode.K);
```

TIP

Finding Key Codes

Each key has a specific key code. You can determine the key code of the specific key you want by reading the Unity documentation. Alternatively, you can use the built-in tools of MonoDevelop to find it. Whenever you are working on a script in MonoDevelop, you can always type in the name of an object followed by a period. Doing so will result in a menu popping up with all of the possible options. Likewise, if you type an open parenthesis after typing a method name, the same menu will pop up showing you the various options. Figure 9.2 illustrates using the auto menu to find the key code for the Esc key.

**FIGURE 9.2**

The automatic pop-up in MonoDevelop.

▼ TRY IT YOURSELF

Reading in Specific Key Presses

Let's write a script that will determine whether a specific key is pressed:

1. Create a new project or scene. Add a script to the project named **PlayerInput** and attach it to the Main Camera.
2. Add the following code to the `Update` method in the **PlayerInput** script:

```
if(Input.GetKeyDown(KeyCode.M)) {
    print("The 'M' key is pressed down");
}
```

3. Save the script and run the scene. Notice what happens when you press the M key. In particular note that the key press may be registered several times a second.

Mouse Input

Besides key presses, you want to capture mouse input from the user. There are two components to mouse input: mouse buttons and mouse movement. Determining whether mouse buttons are pressed is much like key presses covered earlier. Again you will be using the `Input` object. This time you use the `GetMouseButton` method. This method takes an integer between 1 and 3 to dictate which mouse button you are asking about. The method returns a Boolean value indicating if the button is pressed. The code to get the mouse button presses looks like this:

```
bool isButtonDown;
isButtonDown = Input.GetMouseButton(0); //left mouse button
isButtonDown = Input.GetMouseButton(1); //right mouse button
isButtonDown = Input.GetMouseButton(3); //center mouse button
```

Mouse movement is only along two axis: x and y. To get the mouse movement, you use the `GetAxis` method of the `input` object. You can use the names `Mouse X` and `Mouse Y` to get the movement along the x axis and the y axis, respectively. The code to read in the mouse would look like this:

```

float value;
value = Input.GetAxis("Mouse X"); //x axis movement
value = Input.GetAxis("Mouse Y"); //y axis movement

```

Unlike button presses, the mouse movement is measured by the amount the mouse has moved since the last frame only. Basically, holding a key will cause a value to increase until it maxes out at -1 or 1 (depending on whether it is positive or negative). The mouse movement, however, will generally have smaller numbers because it is measured and reset every frame.

TRY IT YOURSELF ▼

Reading Mouse Movement

In this exercise, you read in mouse movement and output the results to the Console:

1. Create a new project or scene. Add a script to the project named **PlayerInput** and attach it to the Main Camera.
2. Add the following code to the `Update` method in the **PlayerInput** script:

```

float mxVal = Input.GetAxis("Mouse X");
float myVal = Input.GetAxis("Mouse Y");
if(mxVal != 0) {
    print("Mouse X movement selected: " + mxVal);
}
if(myVal != 0) {
    print("Mouse Y movement selected: " + myVal);
}

```

3. Save the script and run the scene. Read through the console to see the output when you move the mouse around.

Accessing Local Components

As you have seen numerous times in the Inspector view, objects are composed of various components. There is always a transform component, and optionally any number of other components such as a Renderer, Light, and Camera. Scripts are also components, and together these components give a game object its behavior.

Using GetComponent

You can interact with components at runtime through scripts. The first thing you must do is “Get” the component you want to work. If the component is available from the start, you should do this in the `Start()` method so that you don’t waste time with repeating this relatively slow operation.

The `GetComponent<Type>()` method has a slightly new syntax, using chevrons to specify the Type you are looking for (e.g. Light, Camera, and another script name). There are other versions of this method where you specify the type inside the brackets like a normal parameter, but the version used here is more commonly used.

`GetComponent` returns the first component of the specified type that is attached to the same game object as your script. You should then assign this component to a local variable so you can access it later as follows:

```
private Light lightComponent; // A variable to store the light component.

Start () {
    lightComponent = GetComponent<Light> ();
    lightComponent.type = LightType.Directional;
}
```

When you want to access a property of the component, for example, the type of a light, use the variable name followed by a dot. In the example above, we change the `type` property of the light component to be `directional`. Notice how the light component and `type` property are capitalized in the Inspector but lowercased in code. Just remember that when you are attempting to access a specific thing (“this light,” for example), you use lowercase letters.

Accessing The Transform

The most common component you work with is the transform component. By editing this, you can make objects move around the screen. Remember that an object’s transform is made up of its translation (or position), its rotation, and its scale. Although you can modify those directly, it is easier to use some built-in options called the `Translate` method, the `Rotate` method, and the `localScale` variable:

```
//Moves the object along the positive x axis.
//The '0f' means 0 as a float (floating point number). It is the way Unity reads
floats
transform.Translate(0.05f, 0f, 0f);

//Rotates the object along the z axis
transform.Rotate(0f, 0f, 1f);

//Scales the object to double its size in all directions
transform.localScale = new Vector3(1.5f, 1.5f, 1.5f);
```

NOTE

Finding the Transform

As every game object has a transform, there’s no need to do an explicit find operation; we can access the transform directly as above. This is the only component that works this way; the rest must be accessed using a find method, which we cover next. This is a major change since Unity 4, where you had direct access to many components like physics components, etc.

Because `Translate()` and `Rotate()` are methods, if the preceding code were to be put in `Update()`, the object would continually move along the positive x axis while being rotated along the y axis.

TRY IT YOURSELF ▼

Transforming an Object

Take a list or two of your own and try to find the best way to present the information so that it can be easily understood.

Let's see the previous code in action by applying it to an object in a scene:

1. Create a new project or scene. Add a cube to the scene and position it at (0, -1, 0).
2. Create a new script and name it **CubeScript**. Place the script on the cube. In MonoDevelop, enter the following code to the `Update` method:

```
transform.Translate(.05f, 0f, 0f);
transform.Rotate(0f, 0f, 1f);
transform.localScale = new Vector3(1.5f, 1.5f, 1.5f);
```

3. Save the script and run the scene. You may need to move to Scene view to see the full motion. Notice how the effects of the `Translate` and `Rotate` methods are cumulative and the variable `localScale` is not; it does not keep growing.

Accessing Other Objects

Many times, you want a script to be able to find and manipulate other objects and their components. Doing so is simply a matter of finding the object you want and calling on the appropriate component. There are a few basic ways to find objects that aren't local to the script or to the object the script is attached to.

Finding Other Objects

The first and easiest way to find other objects to work with is to use the editor. By creating a public variable on the class level of type `GameObject`, you can simply drag the object you want onto the script component in the Inspector view. The code to set this up looks like this:

```
//This is here for reference
public class SomeClassScript : MonoBehaviour {

    //This is the game object you want to access
    public GameObject objectYouWant;
```

```
//This is here for reference
void Start() {
}
}
```

After you have attached the script to a game object, you see a property in the Inspector called Object You Want (see Figure 9.3). Just drag any game object you want onto this property to have access to it in the script.



FIGURE 9.3

The new Object You Want property in the Inspector.

Another way to find a game object is by using one of the Find methods. As a rule of thumb if you want a designer to be able to connect the object, or it's optional, then connect via the inspector. If disconnecting would break the game, then use a Find method. There are three major ways to find using script: by name, by tag, and by type.

One option is to search by the object's name. The object's name is what it is called inside the Hierarchy view. Assuming that you are looking for an object named Cube, the code would look like this:

```
//This is here for reference
public class SomeClassScript : MonoBehaviour {

    //This is the game object you want to access
    private GameObject target; // Note this doesn't need to be public if using find.

    //This is here for reference
    void Start() {
        target = GameObject.Find("Cube");
    }
}
```

The shortcoming of this method is that it just returns the first item it finds with the given name. If you have multiple Cube objects, you won't know which one you are getting.

TIP**Finding Efficiency**

The code that performs the find has then been placed in the `Start` method. This operation is slow, meaning it takes a large fraction of a second. This is fine in once at the start of the game, but is not something we want to do every frame inside `Update`. Once the found object is in a variable, accessing it is very fast.

Another way to find an object is by its tag. An object's tag is much like its layer (which was covered previously). The only difference is semantics. The layer is used for broad categories of interaction, whereas the tag is used for basic identification. You create tags using the Tag Manager (click **Edit > Project Settings > Tags & Layers**). Figure 9.4 shows how to add a new tag to the Tag Manager.

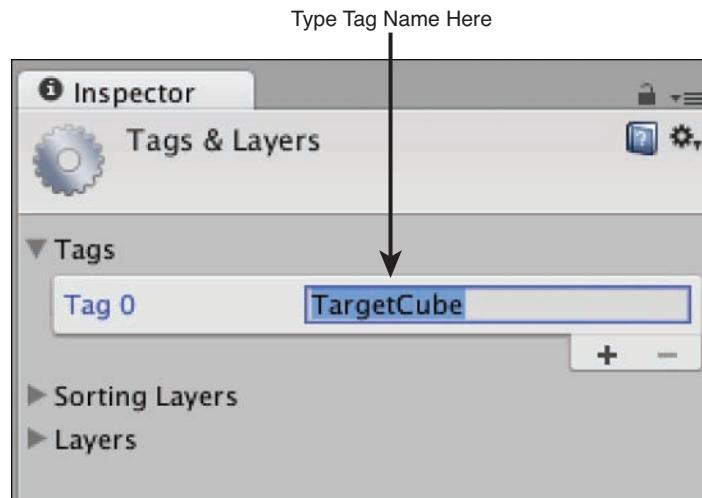


FIGURE 9.4
Adding a new tag.

Once a tag is created, simply apply it to an object using the Tag drop-down list in the Inspector view (see Figure 9.5).

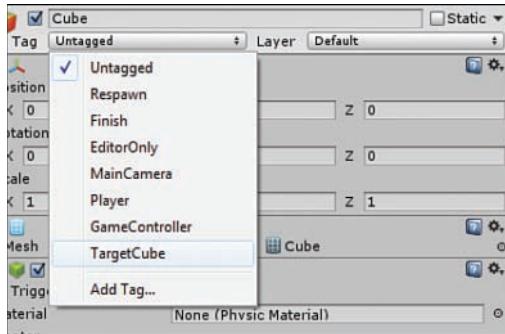


FIGURE 9.5
Selecting a tag.

Now that a tag is added to an object, you can find it using the `FindWithTag` method:

```
//This is here for reference
public class SomeClassScript : MonoBehaviour {

    //This is the game object you want to access
    private GameObject target;

    //This is here for reference
    void Start() {
        target = GameObject.FindGameObjectWithTag("TargetCube");
    }
}
```

Finally you can find an object by its type. An object has a type if it has a script of that name attached. This can be a robust way of finding objects, as changes to script names will automatically update all Find operations. Assuming that you are looking for an object named Cube, the code would look like this:

```
//This is here for reference
public class SomeClassScript : MonoBehaviour {

    //This is the game object you want to access
    private GameObject target;

    //This is here for reference
    void Start () {
        target = GameObject.FindObjectOfType <CubeScript>();
    }
}
```

Modifying Object Components

Once you have a reference to another object, working with the components of that object is almost exactly the same. The only difference is that now instead of simply writing the component name, you need to write the object variable and a period in front of it:

```
//This accesses the local component, not what you want  
transform.Translate(0, 0, 0);
```

```
//This accesses the target object, what you want  
targetObject.transform.Translate(0, 0, 0);
```

TRY IT YOURSELF ▼

Transforming a Target Object

Let's take a moment to modify a target object using scripts:

1. Create a new project or scene. Add a cube to the scene and position it at (0, -1, 0).
2. Create a new script and name it **TargetCubeScript**. Place the script on the Main Camera. In MonoDevelop, enter the following code to the TargetCubeScript:

```
//This is the game object you want to access  
private GameObject target;  
  
//This is here for reference  
void Start() {  
    target = GameObject.Find("Cube");  
}  
  
void Update() {  
    target.transform.Translate(.05f, 0f, 0f);  
    target.transform.Rotate(0f, 0f, 1f);  
    target.transform.localScale = new Vector3(1.5f, 1.5f, 1.5f);  
}
```

3. Save the script and run the scene. Notice how the cube is moving around even though the script was applied to the Main Camera.

Summary

In this hour, you explored more scripting in Unity. You learned all about methods and looked at some ways to write your own. Then, you worked with player inputs from the keyboard and mouse. After that, you learned about modifying object components with code. You finished the hour by learning how to find and interact with other game objects via scripts.

Q&A

Q. How many methods should I write?

- A.** A method should be a single, concise function. You don't want to have too few methods because that would cause each method to do more than one thing. You also don't want to have too many small methods because that defeats the purpose. As long as each process has its own specific method, you have enough.

Q. Why don't we learn more about gamepads?

- A.** The problem with gamepads is that they all differ. In addition, different operating systems treat them differently. The reason they weren't covered in detail this hour is because they are too varied and wouldn't allow for a consistent reader experience (plus not everyone has gamepads).

Q. Is every component editable by script?

- A.** Yes, at least all of the built-in ones.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. True or False: Methods can also be referred to as functions.
2. True or False: Not every method has a return type.
3. Why is it a bad thing to map player interactions to specific buttons?
4. In the Try It Yourself exercises in the sections on local and target components, the cube was translated along the positive x axis and rotated along the z axis. This caused the cube to move around in a big circle. Why?

Answers

1. True.
2. False. Every method has a return type. If the method returns nothing, the type is `void`.
3. The players will have a much harder time remapping the controls to meet their preferences. By mapping your controls to generic axes, the player can change which buttons map to those axes easily.
4. Transformations happen on the local coordinate system (remember Hour 2, “Game Objects”). Therefore, the cube did move along the positive x axis. The direction that axis was facing relative to the camera, however, kept changing.

Exercise

It is a good idea to combine each hour’s lessons together to see them interact in a more realistic way. In this exercise, you write scripts to allow the player directional control over a game object. You can find the solution to this exercise in the book assets for Hour 9 if needed:

1. Create a new project or scene. Add a cube to the scene and position it at (0, 0, -5).
2. Create a new folder called **Scripts** and create a new script called **CubeControlScript**. Attach the script to the cube.

Try to add the following functionality to the script. If you get lost, check the book assets for Hour 9 for help:

1. Whenever the player presses the left or right arrow, move the cube along the x axis negatively or positively, respectively. Whenever the player presses the down or up arrow, move the cube along the y axis negatively or positively, respectively.
2. When the player moves the mouse along the y axis, rotate the cube about the x axis. When the player moves the mouse along the x axis, rotate the cube about the y axis.
3. When the player presses the M key, scale the cube up. When the player presses the N key, scale the cube down.

This page intentionally left blank

HOUR 10

Collision

What You'll Learn in This Hour:

- ▶ The basics of rigidbodies
- ▶ How to use colliders
- ▶ How to script with triggers
- ▶ How to raycast

In this hour, you learn to work with the most prevalent physics concept in video games: collision. *Collision*, simply put, is knowing when the border of one object has come into contact with another object. You begin by learning what rigidbodies are and what they can do for you. After that, you experiment with Unity's powerful built-in physics engines—namely Box2D and PhysX. From there, you learn the more subtle uses of collision with triggers. You end the hour by learning to use a raycast to detect collisions.

Rigidbodies

For objects to take advantage of Unity's built-in physics engine, they must include a component called a *rigidbody*. Adding a rigidbody component makes an object behave like a real-world solid entity. To add a rigidbody component, simply select the object that you want and click **Component > Physics > Rigidbody** (make sure you choose the version without 2D on the end). You will notice the new rigidbody component added to the object in the Inspector (see Figure 10.1).

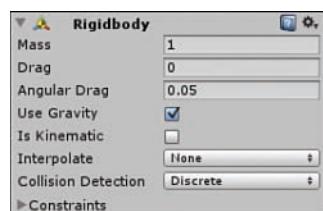


FIGURE 10.1

The rigidbody component.

The rigidbody component has several new properties that you have not seen yet. Table 10.1 describes these properties.

TABLE 10.1 Rigidbody Properties

Property	Description
Mass	The mass of the object in arbitrary units. Use 1 unit = 1 kg unless you have a good reason to deviate. A heavier object will have a higher mass.
Drag	How much air resistance is applied to the object when moving. Higher drag will make an object require more force to move and will stop a moving object more quickly. A drag of 0 will apply no air resistance.
Angular Drag	Much like drag, this is air resistance applied when spinning.
Use Gravity	Determines whether Unity's gravity calculations are applied to this object. Gravity will affect an object more or less depending on its drag.
Is Kinematic	If an object is kinematic, it will not be affected by Unity's physics. This is for times where you want a rigidbody for collisions, but you want control the movement yourself.
Interpolate	Determines how and if motion for an object is smoothed. By default, this is set to Smooth. Interpolate bases the smoothing on the previous frame, whereas Extrapolate is based on the next assumed frame. It is recommended to turn this on for the player object and turn it off for everything else. This will give you the best performance and quality.
Collision Detection	Determines how collision is calculated. Discrete is the default and is how all objects test against each other. The Continuous setting can help if you are having trouble detecting collisions with very fast objects. Be aware, though, that Continuous can have a large impact on performance. The Continuous Dynamic setting will use discrete detection against other discrete objects and continuous detection against other continuous objects.
Constraints	Constraints are movement limitations that a rigidbody enforces on an object. By default, these are turned off. Freezing a position axis will prevent the object from moving along that axis, and freezing a rotation axis will prevent an object from rotating about that axis.

▼ TRY IT YOURSELF

Using Rigidbody

Let's take a moment to see a rigidbody in action:

1. Create a new project or scene. Add a cube to the scene and place it at (0, 1, -5).
2. Run the scene. Notice how the cube floats in front of the camera.
3. Add a rigidbody to the object (click **Components > Physics > Rigidbody**).

4. Run the scene. Notice how the object now falls due to gravity.
5. Continue experimenting with the drag and constraints properties.

Collision

Now that you have your objects moving around, it is time to start getting them to crash into each other. For objects to detect collision, they need a component called a collider. A *collider* is a perimeter that is projected around your object that can detect when other objects enter it.

Colliders

Geometric objects like spheres, capsules, and cubes already have collider components on them when created. You can add a collider to an object without one by clicking **Component > Physics** and then choosing the collider shape you want from the menu. Figure 10.2 illustrates the different collider shapes you can choose from.

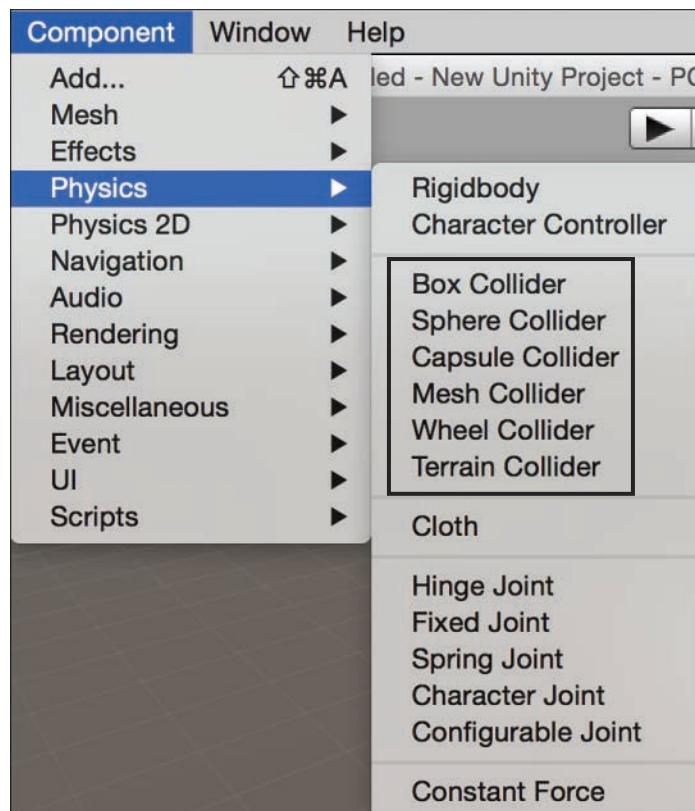


FIGURE 10.2

The different colliders.

Once a collider is added to an object, the collider object appears in the Inspector. Table 10.2 describes the collider properties.

TABLE 10.2 Collider Properties

Property	Description
Edit Collider	This button allows you to graphically adjust the collider size (and in some cases shape) in the Scene view.
Is Trigger	Determines whether the collider is a physical collider or a trigger collider. Triggers are covered in greater detail later this hour.
Material	Colliders enable you to apply physics materials to objects to change the way in which they behave. You can make an object behave like wood, metal, or rubber, for instance. Physics materials are covered later this hour.
Center	The center point of the collider relative to the containing object.
Size	The size of the collider.
Geometric properties	If the collider is a sphere or capsule, you might have an additional property, such as Radius. These behave exactly as you would expect them to.

TIP

Mix and Match Colliders

Using different-shaped colliders on objects can have some interesting effects. For instance, making the collider on a cube much bigger than the cube makes the cube look like it is floating above a surface. Likewise, a smaller collider will allow an object to sink into a surface. Furthermore, putting a sphere collider on a cube will allow the cube to roll around like a ball. Have fun experimenting with the various ways to make colliders for objects.

TRY IT YOURSELF

Experimenting with Colliders

It's time to try out some of the different colliders and see how they interact. Be sure to save this exercise; you use it again later in the hour:

1. Create a new project or scene. Add two cubes to the scene. Place one cube at **(0, 1, -5)** and put a rigidbody on it. Place the other cube at **(0, -1, -5)** and scale it to **(4, .1, 4)** with a rotation of **(0, 0, 15)**. Put a rigidbody on the second cube, as well, but uncheck the **Use Gravity** property.
2. Run the scene and notice how the top cube falls onto the other cube. They then fall away from the screen. Now, on the bottom cube, under the constraints of the rigidbody component, freeze all three axes for both position and rotation.

3. Run the scene and notice how the top cube now falls and stops on the bottom cube. Remove the box collider from the top cube (right-click the **Box Collider** component and select **Remove Component**). Add a sphere collider to the top cube (click **Component > Physics > Sphere Collider**). Give the bottom cube a rotation of (0, 0, 350).
4. Run the scene. Notice how the box rolls off of the ramp like a sphere even though it is a cube.
5. Continue experimenting with the different colliders. Another fun experiment is to change the constraints on the bottom cube. Try only freezing the y axis position and unfreezing everything else. Try out the different ways to make the boxes collide.

TIP**Complex Colliders**

You may have noticed a collider called the Mesh Collider. This collider is specifically left out of the text because it is more a practice in modeling than anything. Basically, a *mesh collider* is a collider that has the exact shape of a 3D model. This sounds useful, but in practice can greatly reduce the performance of your game. Furthermore, Unity puts a severe limit to the number of polygons allowed in a mesh collider. A better habit to get into is to compose your complex object with several basic colliders. If you have a humanoid model, try spheres for the head and hands and capsules for the torso, arms, and legs. You will save on performance and still have some very sharp collision detection.

Physics Materials

Physics materials can be applied to colliders to give objects varied physical properties. For instance, you can use the rubber material to make an object bouncy or an ice material to make it slippery. You can even make your own to emulate a specific material of your choosing.

There are some premade physics materials in the Characters standard assets pack. To import these, click **Assets > Import Package > Characters**. In the Import screen, click the **None** button to deselect all, then scroll down. Put a check against **Physics Materials** near the bottom, and click **Import**. This will bring an **Assets > Standard Assets > PhysicsMaterials** folder into your project that contains physics materials for Bouncy, Ice, Metal, Rubber, and Wood. To create a new physics material, right-click the **Assets** folder in the Project view and select **Create > Physics Material**.

A physics material has a set of properties that determine how it behaves on a physical level (see Figure 10.3). Table 10.3 describes the physics material's properties. You can apply a physics material to an object by dragging it from the Project view onto an object.

**FIGURE 10.3**

The properties of physics materials.

TABLE 10.3 Physics Material Properties

Property	Description
Dynamic Friction	The friction applied when an object is already moving. Lower numbers make an object more slippery.
Static Friction	The friction applied when an object is stationary. Lower numbers make an object more slippery.
Bounciness	The amount of energy retained from a collision. A value of 1 causes the object to bounce without any loss of energy; it will bounce forever. A value of 0 prevents the object from bouncing.
Friction Combine	Determines how friction of two colliding objects is calculated. The friction can be averaged, the smallest or largest can be used, or they can be multiplied.
Bounce Combine	Determines how the bounce of two colliding objects is calculated. The bounce can be averaged, the smallest or largest can be used, or they can be multiplied.
Friction Direction 2	Specify this property if you want the object to have different friction in a specific direction. (Think of an ice skate.)
Dynamic Friction 2	Like Dynamic Friction, only applied to the specified direction. Only used if a Friction Direction 2 is supplied.
Static Friction 2	Like Static Friction, only applied to the specified direction. Only used if a Friction Direction 2 is supplied.

The effects of the physics material can be as subtle or as distinct as you like. Try it out for yourself and see what kind of interesting behaviors you can create.

Triggers

So far, you have seen physical colliders, colliders that react in a positional and rotational fashion using Unity's built-in physics engine. If you think back to Hour 7, "Game 1: Amazing Racer," however, you probably can remember using another type of collider. Remember how the game detected when the player entered the water hazards and finish zone? That was the trigger collider at work. A trigger detects collision just like normal colliders do, but it doesn't do anything specific about it. Instead, triggers call three specific methods that allow you, the programmer, to determine what the collision means:

```
void OnTriggerEnter(Collider other)      //is called when an object enters the trigger
void OnTriggerStay(Collider other)      //is called while an object stays in the trigger
void OnTriggerExit(Collider other)      //is called when an object exits the trigger
```

Using these methods, you can define what happens whenever an object enters, stays in, or leaves the collider. For example, if you want to write a message to the console whenever an object enters the perimeter of a cube, you could add a trigger to the cube. Then attach a script to the cube with the following code:

```
void OnTriggerEnter(Collider other)
{
    print("Object has entered collider");
}
```

You might notice the one parameter to the trigger methods: the variable `other` of type `collider`. This is a reference to the object that entered the trigger. Using that variable, you can manipulate the object, however, you want. For instance, if you want to modify the preceding code to write the name of the object that enters the trigger to the console, you could write the following:

```
void OnTriggerEnter(Collider other)
{
    print(other.gameObject.name + " has entered the trigger");
}
```

You could even go so far as to destroy the object entering the trigger with some code like this:

```
void OnTriggerEnter(Collider other)
{
    Destroy(other.gameObject);
}
```



TRY IT YOURSELF

Working with Triggers

In this exercise, you get a chance to build an interactive scene with a functioning trigger. You can find the completed project for this exercise, called Hour10_TriggerExercise, in the book assets for Hour 10:

1. Create a new project or scene. Add a cube and sphere to the scene. Place the cube at $(-1, 1, -5)$ and place the sphere at $(1, 1, -5)$.
2. Create two scripts named **TriggerScript** and **MovementScript**. Place the trigger script on the cube and the movement script on the sphere.
3. On the cube's collider, check **Is Trigger**. Add a rigidbody to the sphere and uncheck **Use Gravity**.
4. Add the following code to the Update method of the movement script:

```
float mX = Input.GetAxis("Mouse X") / 10;  
float mY = Input.GetAxis("Mouse Y") / 10;  
transform.Translate(mX, mY, 0);
```

5. Add the following code to the trigger script. Be sure to place the code outside of any methods but inside of the class, that is at the same level of indentation as the `Start()` and `Update()` methods:

```
void OnTriggerEnter (Collider other) {  
    print(other.gameObject.name + " has entered the cube");  
}  
  
void OnTriggerStay (Collider other) {  
    print(other.gameObject.name + " is still in the cube");  
}  
  
void OnTriggerExit (Collider other) {  
    print(other.gameObject.name + " has left the cube");  
}
```

6. Run the scene. Notice how the mouse moves the sphere. Collide the sphere with the cube and pay attention to the console output. Notice how the two objects don't physically react, but they still interact. Can you work out which cell of Figure 10.4 this interaction falls into?

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider	collision			trigger	trigger	trigger
Rigidbody Collider	collision	collision	collision	trigger	trigger	trigger
Kinematic Rigidbody Collider		collision		trigger	trigger	trigger
Static Trigger Collider		trigger	trigger		trigger	trigger
Rigidbody Trigger Collider	trigger	trigger	trigger	trigger	trigger	trigger
Kinematic Rigidbody Trigger Collider	trigger	trigger	trigger	trigger	trigger	trigger

FIGURE 10.4

Collider interaction matrix.

NOTE**Collisions Not Working**

Not all collisions result in methods being called in your code. Refer to Figure 10.4 to lookup whether the interaction between two objects will call `OnTriggerEnter`, `OnCollisionEnter`, or neither.

Static Colliders are simply colliders on any game object. They become Rigidbody Colliders when you add a Rigidbody component, and these become Kinematic Rigidbody Colliders when you check the “Is Kinematic” box. For each of these three you can also tick, or untick “Is Trigger.” This leads to the six types of collider as seen in Figure 10.4.

Raycasting

Raycasting is the act of sending out an imaginary line, a ray, and seeing what it hits. Imagine, for instance, looking through a telescope. Your line of sight is the ray, and whatever you can see at the other end is what your ray hits. Game developers use raycasting all the time for things like aiming, determining line of sight, gauging distance, and more. There are a few Raycast methods in Unity. The two most common uses are laid out here. The first Raycast method looks like this:

```
bool Raycast(Vector3 origin, Vector3 direction, float distance, LayerMask mask);
```

Notice that this method takes quite a few parameters. Also notice that it uses a variable called a `Vector3`. A `Vector3` is a variable type that holds three floats inside of it. It is a great way to specify an x, y, and z coordinate without requiring three separate parameters. The first

parameter, `origin`, is the position the ray starts at. The second, `direction`, is which direction the ray travels. The third parameter, `float`, determines how far out the ray will go, and the final variable, `mask`, determines what layers will be hit. You can omit both the `distance` and `mask` variables. If you do, the ray will travel an infinite distance and hit all object types.

As mentioned earlier, there are many things you can do with rays. For instance, if you want to determine whether something is in front of the camera, you could attach a script with the following code:

```
void Update() {
    //cast the ray from the camera's position in the forward direction
    if (Physics.Raycast(transform.position, transform.forward, 10))
        print("There is something in front of the camera!");
}
```

Another way we can use this method is to find the object that the ray collided with. This version of the method uses a special variable type called a `RaycastHit`. Many versions of the `Raycast` method utilize `distance` (or don't) and `layer mask` (or don't). The most basic way to use this version of the method, though, looks something like this:

```
bool Raycast(Vector3 origin, Vector3 direction, out RaycastHit hit, float distance);
```

There is one new interesting thing about this version of the method. You might have noticed that it uses a new keyword that you have not seen before: `out`. This keyword means that when the method is done running, the variable `hit` will contain whatever object was hit. The method effectively sends the value back `out` when it is done. Be careful, as you may lose track of the value of the `hit` variable.

▼ TRY IT YOURSELF

Casting Some Rays

Let's create an interactive "shooting" program. This program will send a ray from the camera and destroy whatever objects it comes into contact with. You can find the completed project for this exercise, called `Hour10_RaycastExercise`, in the book assets for Hour 10.

1. Create a new project or scene. Add four spheres to the scene and change their names to be `Sphere1` through `Sphere4`. Place the spheres at $(-1, 1, -5)$, $(1, 1.5, -5)$, $(-1, -2, 5)$, and $(1.5, 0, 0)$.
2. Create a new script called `RaycastScript` and attach it to the Main Camera. Inside the `Update` method for the script, add the following:

```
float dirX = Input.GetAxis ("Mouse X");
float dirY = Input.GetAxis ("Mouse Y");
```

```
// opposite because we rotate about those axes  
transform.Rotate (dirY, -dirX, 0);  
  
CheckForRaycastHit (); //this will be added in the next step
```

3. Now, add the method `CheckForRaycastHit()` to your script by adding the following code outside of a method but inside the class:

```
void CheckForRaycastHit() {  
    RaycastHit hit;  
    if (Physics.Raycast (transform.position, transform.forward, out hit)) {  
        print (hit.collider.gameObject.name + " destroyed!");  
        Destroy (hit.collider.gameObject);  
    }  
}
```

4. Run your scene. Notice how moving the mouse moves the camera. Try to center the camera on each sphere. Notice how the sphere is destroyed and the message is written to the console.

Summary

In this hour, you learned about object interactions through collision. You learned about the basics of Unity's physics capabilities with rigidbodies. Then, you worked with various types of colliders and collision. From there, you learned that collision is more than just stuff bouncing around when you got hands on with triggers. Finally, you learned to find objects by raycasting.

Q&A

- Q. Should all my objects have rigidbodies?**

- A.** Rigidbodies are useful components that serve largely physical roles. That said, adding rigidbodies to every object can have strange side effects and may reduce performance. A good rule of thumb is to add components only when they are needed, not preemptively.

- Q. There are several colliders we didn't talk about. Why not?**

- A.** Most colliders either behave the same way as the ones we covered or are beyond the scope of this text. For that reason, they are omitted. Suffice to say that this text still provides what you will need to know to make some very fun games.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. This component is required on an object if you want it to exhibit physical traits like falling.
2. True or False: An object can only have a single collider on it.
3. What sorts of things are raycasts useful for?

Answers

1. Rigidbody.
2. False. An object can have many, and varied, colliders on it.
3. Determining what an object can see and finding objects along line of site as well as finding distances between objects.

Exercise

In this exercise, you create an interactive application that utilizes motion and triggers. The exercise requires you to creatively determine a solution (because one is not presented here). If you get stuck and need help, you can find the solution to this exercise, called Hour10_Exercise, in the book assets for Hour 10.

1. Create a new project or scene. Add a cube to the scene and position it at **(-1.5, 0, -5)**. Scale the cube **(.1, 2, 2)** and rename it **LTrigger**.
2. Duplicate the cube. (Right-click the cube in the Hierarchy view and select **Duplicate**.) Name the new cube **RTrigger** and place it at **(1.5, 0, -5)**.
3. Add a sphere to your scene and place it at **(0, 0, -5)**. Add a rigidbody to the sphere and uncheck **Use Gravity**.
4. Create a script named **TriggerScript** and place it on both the LTrigger and the RTrigger. Create a script called **MotionScript** and place it on the sphere.

Now comes the fun part. You will need to create the following functionality in your application:

- ▶ The player should be able to move the sphere with the arrow keys.
- ▶ When the sphere enters, exits, or stays in either of the triggers, the corresponding message should be written to the console.

The name of the trigger that the sphere enters (**LTrigger** or **RTrigger**) should also be written to the console with the above message.

Good luck!

HOUR 11

Game 2: *Chaos Ball*

What You'll Learn in This Hour:

- ▶ How to design the game Chaos Ball
- ▶ How to build the ChaosBall arena
- ▶ How to build the ChaosBall entities
- ▶ How to build the ChaosBall control objects
- ▶ How to further improve ChaosBall

It is time once again to take what you have learned and make another game. In this hour, you make the game *Chaos Ball*, which is a faster-paced arcade-style game. You start by covering the basic design elements of the game. From there, you build arena and game objects. Each object type will be made unique and given special collision properties. Then, you add interactivity to make the game playable. You finish by playing the game and making any necessary tweaks to improve the experience.

TIP

Completed Project

Be sure to follow along in this hour to build the complete game project. In case you get stuck, you can find a completed copy of the game in the book assets for Hour 11. Take a look at it if you need help or inspiration!

Design

You have already learned what the design elements are in Hour 7, “Game 1: *Amazing Racer*.” This time, you get right into them.

The Concept

This is a game slightly akin to *Pinball* or *Breakout*. The player will be in an arena. Each of the four corners will have a color, and four balls with corresponding colors will be floating around. Amid the four colored balls, there will be several yellow balls, called *chaos balls*. Chaos balls exist solely to get in your way and make the game challenging! They are smaller than the four colored balls, but they also move faster. Players will have a flat surface with which they will attempt to knock the colored balls into the correct corners.

The Rules

The rules for this game will state how to play, but will also allude to some of the properties of the objects. The rules for *Chaos Ball* are as follows:

- ▶ The player wins when all four balls are in the correct corners. There is no loss condition.
- ▶ Hitting the correct corner causes a ball to disappear, and the corner light to go out.
- ▶ All objects in the game are super bouncy (they lose no energy on impact).
- ▶ No ball (or player) can leave the arena.

The Requirements

The requirements for this game are simple. This is not a graphically intense game and instead relies on scripting and interaction for its entertainment. The requirements for *Chaos Ball* are as follows:

- ▶ A walled piece of terrain to act as the arena.
- ▶ Textures for the terrain and game objects. These are provided in the Unity standard assets.
- ▶ Several colored balls and chaos balls. These will be generated in Unity.
- ▶ A character controller. This is provided by the Unity standard assets.
- ▶ A game controller. This will be created in Unity.
- ▶ A bouncy physics material. This will be created in Unity.
- ▶ Colored corner indicators. These will be generated in Unity.
- ▶ Interactive scripts. These will be written in MonoDevelop.

The Arena

The first thing you want to create is an area for the action to take part in. The term *arena* is chosen to give the idea that the terrain is quite small and also walled in. Neither the player nor any balls should be able to leave the arena. Otherwise, the arena is quite simple (see Figure 11.1).

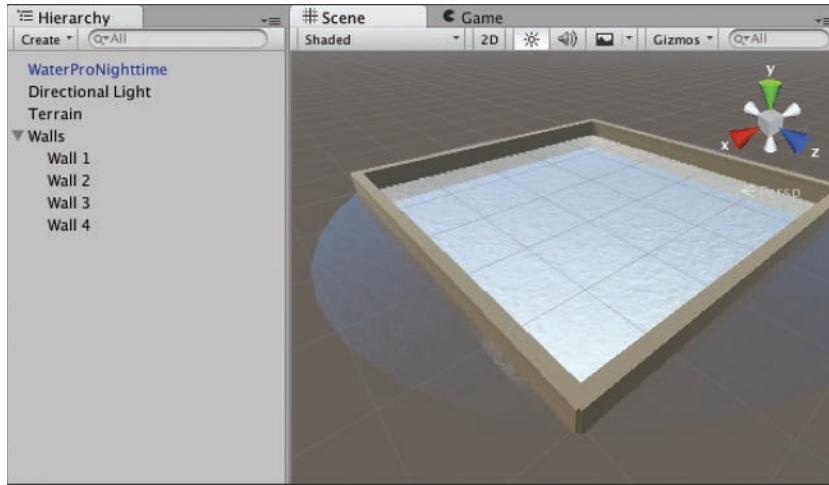


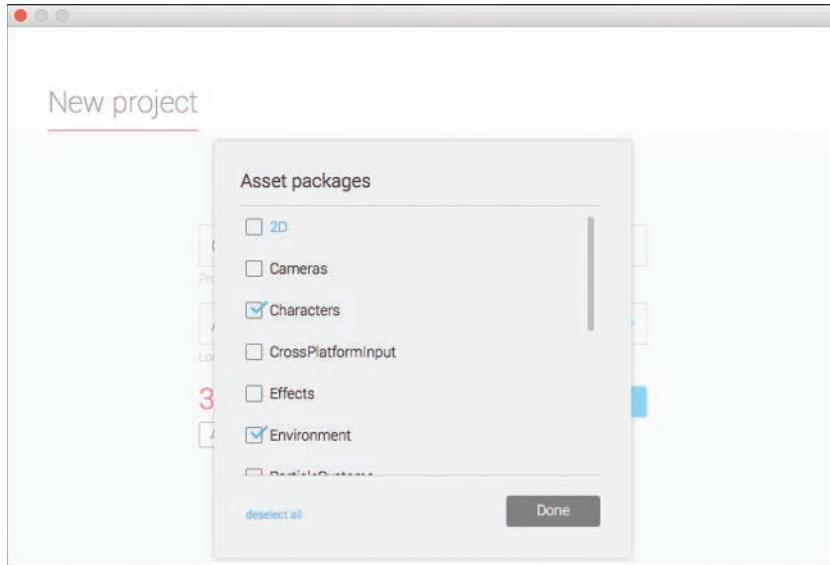
FIGURE 11.1

The arena.

Creating the Arena

As mentioned earlier, this is going to be a simple process because of the simplicity of a basic arena map. To create the arena, follow these steps:

1. Create a new project in a folder called **ChaosBall**. This time at the Create New Project dialog, click Asset Packages . . . and check the boxes next to **Characters** and **Environment** (see Figure 11.2). If you forget, you can import these packages from the Assets menu.

**FIGURE 11.2**

The Create New Project dialog.

2. Add a terrain to the project. Set the terrain width and length to 50 units. Remember this is in the terrain settings, under resolution.
3. Delete the Main Camera.
4. Add a cube to your scene. Place the cube at (0, 1.5, 25) and scale it to (1.5, 3, 51). Notice how it becomes a side wall for the arena. Rename the cube to **Wall 1**.
5. Save the scene as **Main** in a Scenes folder.

TIP**Consolidating Objects**

You might be wondering to yourself why you created only a single wall when the arena will obviously need four. The idea is that you want to do as little redundant, tedious work as possible. Often, if you require several objects that are very similar, you can create one object and then duplicate it multiple times. In this instance, you set up a single wall with its materials and properties and then simply copy it three times. You repeat the same process for the corner nodes, the chaos balls, and the colored balls. Hopefully, you can see how a little planning can save you a fair bit of time.

Texturing

Right about now, the arena is looking pretty pitiful and bland. Everything is white, and there is only a single wall. The next step is to add some textures to liven the place up. You need to texture two objects specifically: the wall and the ground. Feel free to experiment with the texturing as you complete this step. You can make it more interesting if you'd like!

1. Create a new folder called **Materials** under Assets in the Project view. Add a material to the folder (right-click the folder and select **Create > Material**). Name the material simply **Wall**.
2. Set the x axis tiling to **10** (see Figure 11.3).
3. Apply the **Sand Albedo** texture to the wall material in the Inspector view (see Figure 11.3).
4. Click and drag the wall material onto the wall object in the Scene view.

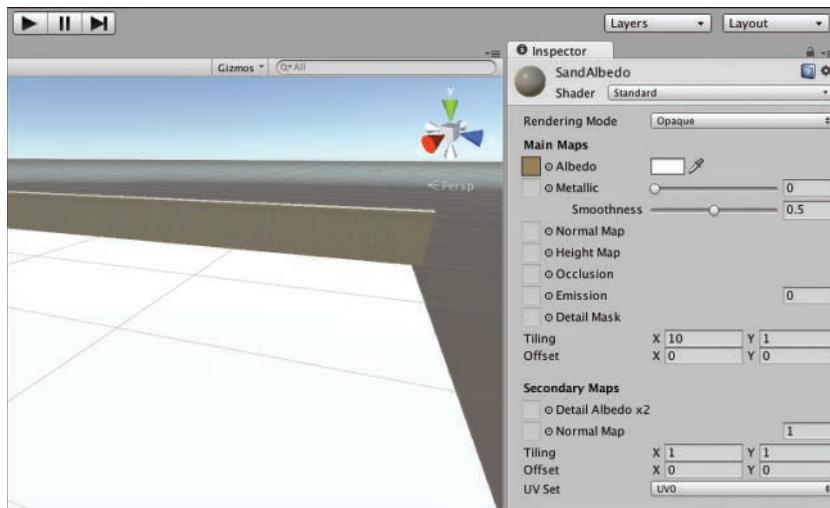


FIGURE 11.3

Adding sand texture to material.

Next, you need to make the ground more interesting. Unity 5 comes with some great water shaders, so let's use them again.

1. Navigate to **Standard Assets > Environment > Water > Water > Prefabs**. Drag **WaterProNighttime** into the scene.
2. Place the water centrally at (25, 0.1, 25) and scale it to (36, 1, 36).

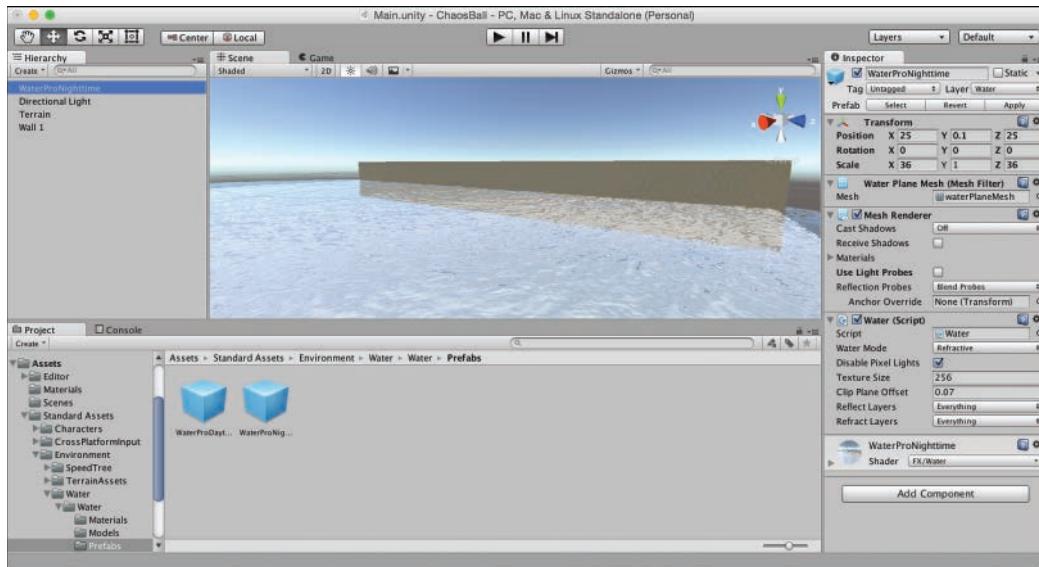


FIGURE 11.4

Adding water to the arena.

Super Bouncy Material

You want objects to bounce off of the walls without losing any energy. What you need is a super bouncy material. If you recall, Unity has a set of physics materials available. The bouncy material they provide, however, is not quite bouncy enough for your needs. Therefore, you need to create a new material, as follows:

1. Right-click the **Materials** folder and select **Create > Physic Material**. Name the material **SuperBouncy**.
2. Set the properties for the super bouncy material as they appear in Figure 11.5. Basically, you want to ensure that the balls are 100% bouncy, so they keep moving at the same speed.

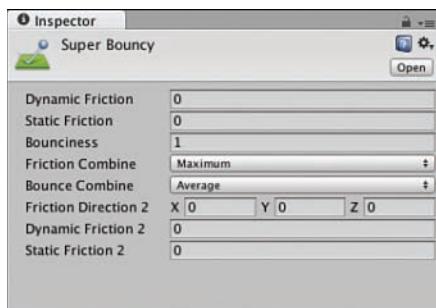


FIGURE 11.5

SuperBouncy material settings.

3. Click and drag the super bouncy material onto the wall object in the scene. It will automatically get applied as the physics material for the collider. You should see the material listed in the Material property of the Box Collider component.

Finish the Arena

Now that the wall and ground is complete, you can finish the arena. The hard work has been done, and now all you need to do is duplicate the walls (right-click in the Hierarchy view and select **Duplicate**). The exact steps are as follows:

1. Duplicate the wall once. Place the new instance at (50, 1.5, 25).
2. Duplicate the wall again. Place it at (25, 1.5, 0) with a rotation of (0, 90, 0).
3. Duplicate the wall created in the previous step (the one that's turned) and place it at (25, 1.5, 50).
4. Create an empty GameObject called **Walls**. Set the position of the new object to (0, 0, 0). Group the four walls you created under this new placeholder object.

Your arena should now have four walls without any gaps or seams (refer to Figure 11.1).

Game Entities

In this section, you create the various game objects required for playing the game. Just like with the arena wall, it will be easier for you to create one instance of each entity and then duplicate it.

The Player

The player in this game will be a modified First Person Character controller. When you created this project, you should have selected to import that character controller's package. Go ahead and click and drag an **FPSController** character controller into the scene. This can be found in the Project tab under **Standard Assets > Characters > FirstPersonCharacter > Prefabs**. Place the controller at (46, 1, 4) with a rotation of (0, 315, 0).

The first thing you want to do is to move the camera up and away from the controller. This will allow the player a greater field of vision while playing the game. To do this, follow these steps:

1. Expand the **FPSController** in the Hierarchy view (click the arrow next to its name) and locate the **FirstPersonCharacter** sub-object, which has a camera on it.
2. After selecting the **FirstPersonCharacter**, position it at (0, 5, -3.5) with a rotation of (43, 0, 0). The camera should now be above, behind, and slightly looking down on the controller.

The next thing to do is to add a bumper to the controller. The bumper will be the flat surface the player will bounce balls off of. To do this, follow these steps:

1. Add a cube to the scene. Rename the cube **Bumper**. Scale the bumper (3.5, 3, 1).
2. Click and drag your super bouncy material onto the bumper.

3. In the Hierarchy view, click and drag the bumper onto the **FPSController**. This will nest the bumper onto the controller. After doing that, change the position of the bumper to (0, 0, 1) with a rotation of (0, 0, 0). The bumper will now be slightly in front of the controller.
4. Give the bumper color by creating a new material (*not* a physics material) called **Bumper Color**. Set the Albedo color to something of your choosing, and drag the material onto the Bumper.

The last thing to do is to tweak the FPSController's default settings to make it more suitable for this game. Carefully set everything as per Figure 11.6, noting the settings that differ from the default are bold.



FIGURE 11.6
The FPSController script settings.

Chaos Balls

The chaos balls will be the fast and wild balls flying around the arena and disrupting the player. In many ways, they are similar to the colored balls, so you will be working to give them universally applicable assets. To create the first chaos ball, follow these steps:

1. Add a sphere to the scene. Rename the sphere **Chaos 1** and position it at (15, 2, 25) with a scale of (.5, .5, .5).
2. Click and drag the super bouncy material onto the sphere.
3. Create a new material (*not* a physics material) for the chaos ball called **ChaosBall** and set the Albedo color to a bright yellow color. Click and drag the material onto the sphere.
4. Add a rigidbody to the sphere. Change the angular drag to **0** and uncheck **Use Gravity**. Change the Collision Detection property to **Continuous**. Under the Constraints property, freeze the y position. We don't want the balls to be able to go up or down.
5. Open the Tag Manager (click **Edit > Project Settings > Tags & Layers**), expand the **Tags** section by clicking the arrow next to **Tags**, and add the tag **Chaos** at Element 0. While you're here, go ahead and add the tags **Green**, **Orange**, **Red**, and **Blue**. These are used later.
6. Select the chaos sphere and change its tag to be **Chaos** in the Inspector view (see Figure 11.7).

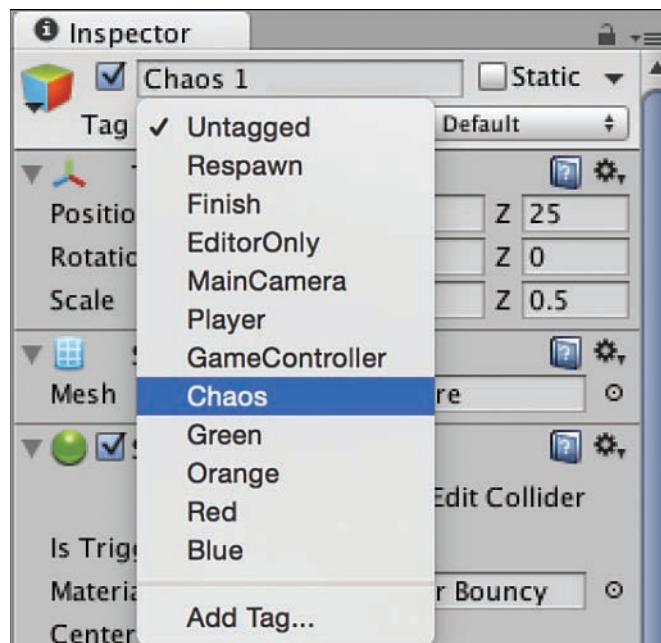


FIGURE 11.7

Choosing the Chaos tag.

The ball is now complete, but it still doesn't do anything. You need to create a script to move the ball all around the arena. You need to create a script called **VelocityScript** and attach it to the chaos ball. Move the script into a Scripts folder. Listing 11.1 contains the full code for the velocity script.

LISTING 11.1 VelocityScript.cs

```
using UnityEngine;
using System.Collections;

public class VelocityScript : MonoBehaviour {
    public float startSpeed = 50;

    // Use this for initialization
    void Start () {
        Rigidbody rigidBody = GetComponent<Rigidbody> ();
        rigidBody.velocity = new Vector3 (startSpeed, 0, startSpeed);
    }
}
```

Run your scene and watch the ball begin to fly around the arena. At this point, the chaos ball is finished. In the Hierarchy view, duplicate the chaos ball four times. Scatter each ball around the arena (be sure to only change the x and z positions) and give each of them a random y axis rotation. Remember that movement along the y axis is locked, so make sure that each ball stays at a y position of 2. Finally, create an empty GameObject called Chaos Balls, position it at (0, 0, 0), and child the balls to it to keep your Hierarchy tidy.

The Colored Balls

While the chaos balls are yellow, and that is a color, the colored balls are the four specific balls needed to win the game. They will be red, orange, blue, and green. As with the chaos balls, you can make a single ball and then duplicate it to make the creation easier.

To create the first ball, follow these steps:

1. Add a sphere to the scene. Rename the sphere **Blue Ball**. Position the sphere somewhere near the middle of the arena, and make sure that the y position is 2.
2. Create a new material called **Blue Ball** and set its color to blue the same way you did for the chaos balls. While you're at it, go ahead and create **Red Ball**, **Green Ball**, and **Orange Ball** materials, and set them to the appropriate color. Click and drag the Blue Ball material onto the sphere.
3. Click and drag the super bouncy material onto the ball.
4. Add a rigidbody to the sphere. Change its angular drag to **0**, uncheck **Use Gravity**, set the collision detection to continuous, and freeze the y position under Constraints.

5. Previously, you created the Blue tag. Now, change the sphere's tag to Blue just like you did for the chaos ball (refer to Figure 11.7).
6. Attach the velocity script to the sphere. In the Inspector, locate Velocity Script (Script) component and change the Start Speed property to 25 (see Figure 11.8). This causes the sphere to move slower than the chaos balls initially.

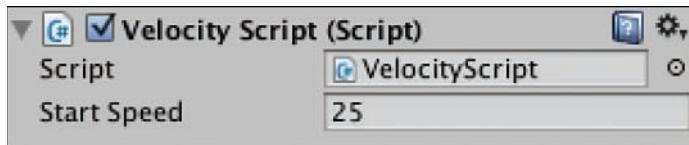


FIGURE 11.8

Changing the Max property.

If you run the scene now, you should see the blue ball moving rapidly around the arena. Now you need to create the other three balls. Each one will be a duplicate of the blue ball. To create the other balls, follow these steps:

1. Duplicate the blue ball. Rename the new ball to its color: **Red Ball**, **Orange Ball**, and **Green Ball**.
2. Give the new ball the tag corresponding to its name. It is important for the name and the tag to be the same thing.
3. Create and drag an appropriate color material onto the new ball. It is important for the ball to be the same color as its name.
4. Give the ball a random location and rotation in the arena, but ensure that its y position is 2.

At this point, the game entities are complete. If you run the scene, you see all of the balls bouncing around the arena.

The Control Objects

Now that you have all the pieces in place, it is time to gamify them. That is, it is time to turn these into a playable game. To do that, you need to create the four corner goals, the goal scripts, and the game controller. Once done, you have yourself a game.

The Goals

Each of the four corners has a specific colored goal that corresponds with a colored ball. The idea behind the goal is that when a ball enters, the goal will check its tag. If the tag matches the color of the goal, there is a match. When a match is found, the ball is destroyed, and the goal is

set to **Solved**. As with the ball objects earlier, you can configure a single goal and then duplicate it to match your needs.

To set up the initial goal, follow these steps:

1. Create an empty game object (click **GameObject > Create Empty**). Rename the game object **BlueGoal** and assign the tag **Blue** to it. Position the game object at (2.3, 2, 2.3).
2. Attach a box collider to the goal and check the **Is Trigger** property. Change the size of the box collider to be (3, 2, 3).
3. Attach a light to the goal (click **Component > Rendering > Light**). Make it a point light and make it the corresponding color of the goal (see Figure 11.9). Change the intensity of the light to 3, and the bounce intensity to 0.

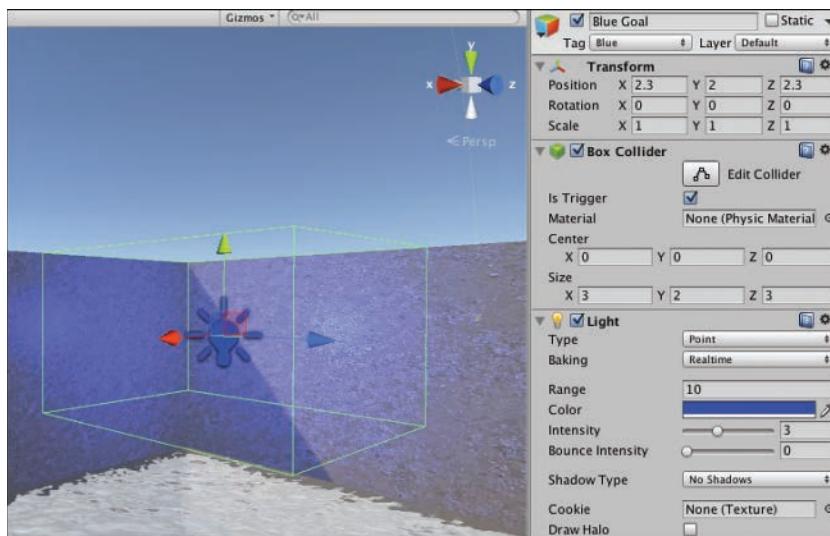


FIGURE 11.9
The blue goal.

Next, you need to create a script called **GoalScript** and attach it to the blue goal. Listing 11.2 shows the contents of the script.

LISTING 11.2 GoalScript.cs

```
using UnityEngine;
using System.Collections;

public class GoalScript : MonoBehaviour {

    void OnTriggerEnter (Collider collider) {
```

```

        GameObject collidedWith = collider.gameObject;

        if (collidedWith.tag == gameObject.tag) {
            GetComponent<Light>().intensity = 0;
            Destroy (collidedWith);
        }
    }
}

```

As you can see in the script, the `OnTriggerEnter()` method will check the tag of every object that contacts it against its own tag. If they match, the object is destroyed, and that goal gets flagged as solved.

When the script is complete and attached to the goal, it is time to duplicate it. To create the other goals, follow these steps:

1. Duplicate the **BlueGoal**. Name the new goal corresponding to its color: **RedGoal**, **GreenGoal**, and **OrangeGoal**.
2. Change the tag of the goal to its corresponding color.
3. Change the color of the point light to the goal's corresponding color.
4. Position the goal. The colors can go in any corner as long as each goal gets its own corner. The three other corner positions are (2.3, 2, 47.3), (47.3, 2, 2.3), and (47.3, 2, 47.3).
5. Organize the goals under a new empty game object called **Goals**.

All the goals should now be set up and operational.

The Game Controller

The last element needed to finish the game is the game controller. This controller will be responsible for checking each goal every frame and determining when all four are solved. For this particular game, the game controller is very simple. To create the game controller, follow these steps:

1. Add an empty game object to the scene. Move it someplace out of the way. Rename it **GameController**.
2. Create a script called **GameControlScript** and add the code from Listing 11.3 to it. Attach the script to the game controller. Put it into the Scripts folder.
3. With the game controller selected, click and drag each ball to its corresponding property on the Game Control Script component (see Figure 11.10).

LISTING 11.3 Game Control Script

```
using UnityEngine;
using System.Collections;

public class GameControlScript : MonoBehaviour {

    // Note you can define several variables of the same type on one line
    public GameObject blueBall, greenBall, redBall, orangeBall;

    private bool isGameOver = false;

    // Update is called once per frame
    void Update () {
        // If all four balls are not (!) existent then game is over
        isGameOver = !blueBall&& !greenBall&& !redBall&& !orangeBall;
    }

    void OnGUI(){
        if(isGameOver){
            Rect rect = new Rect(Screen.width / 2 - 100,Screen.height / 2 - 50, 200, 75);
            GUI.Box(rect, "Game Over");

            Rect rect2 = new Rect (Screen.width / 2 - 30, Screen.height / 2 - 25, 60, 50);
            GUI.Label (rect2, "Good Job!");
        }
    }
}
```

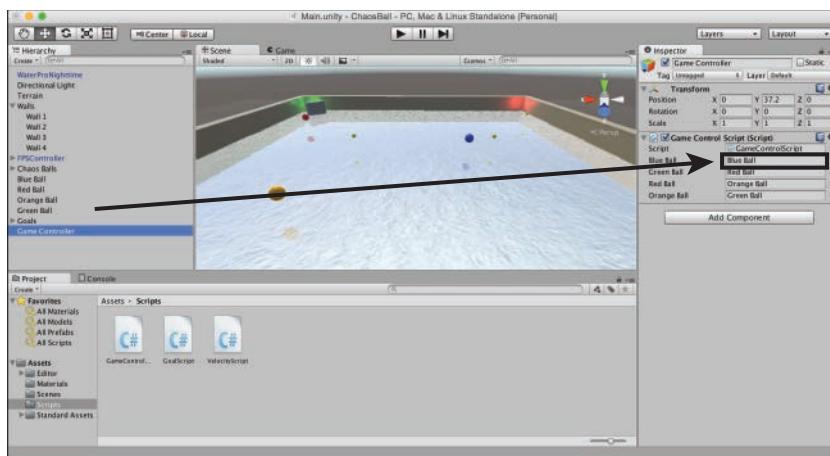


FIGURE 11.10
Adding the goals to the game controller.

As you can see in the preceding script, the game controller has a reference to each of the four balls. Every frame, the controller checks to see if all the balls are not existent. The ! symbol is the symbol for not (as the ball is ‘not’ alive). If they are, the controller sets the variable `is GameOver` to true and displays the game over message on the screen.

Congratulations. *Chaos Ball* is now complete!

Improving the Game

Even though *Chaos Ball* is a complete game, it is hardly as good as it could be. Several features that would greatly improve game play have been omitted. They were left out for brevity, and so that you could experiment with the game and make it better. In a way, you could say that *Chaos Ball* is now a complete prototype. It is a playable example of the game, but it lacks polish. You are encouraged to go back through this chapter and look for ways you can make the game better. Think to yourself as you play it:

- ▶ Is the game too easy or hard?
- ▶ What would make it easier or harder?
- ▶ What would give it that “wow” factor?
- ▶ What parts of the game are fun? What parts of the game are tedious?

In the exercise that follows, you have an opportunity to improve the game and add some of those features. Note that if you get any errors, it means you missed a step. Be sure to go back through and double-check everything to resolve any errors that might arise.

Summary

In this hour, you made the game *Chaos Ball*. You started by designing the game. You determined the concept, the rules, and the requirements. From there, you sculpted the arena and learned that sometimes you can make a single object and duplicate it to save time. From there, you created the player, the chaos balls, the colored balls, the goals, and the game controller. You finished by playing the game and thinking of ways to improve it.

Q&A

- Q.** Why do we use continuous collision detection on the chaos balls? I thought that reduced performance.
- A.** Continuous collision detection can, in fact, reduce performance. In this instance, it is needed, however. The chaos balls are small and fast enough that sometimes they can pass right through the walls.

Q. We created a “chaos” tag, but never used it. How come?

A. This tag is ready for you to make some of the improvements in the exercise below.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

- 1.** How does the player lose the game?
- 2.** What is the positional axis where all of the ball objects frozen on?
- 3.** True or False: The goals utilized the method `OnTriggerEnter()` to determine whether an object was the correct ball.
- 4.** Why were some basic features omitted?

Answers

- 1.** Trick question. The player cannot lose the game.
- 2.** The y axis.
- 3.** True.
- 4.** To give the reader a chance to add them in.

Exercise

The best part about making games is that you can get to make them the way you want. Following a guide can be a good learning experience, but you don’t get the satisfaction of making a custom game. In this exercise, you have an opportunity to modify the game a little to make something more unique. Exactly how you want to change the game is up to you. Here are some suggestions:

- ▶ Try adding a button that allows the player to play again whenever the game is completed. (GUI elements haven’t been covered yet, but this feature existed in the last game; see if you can figure it out.)
- ▶ Try adding a timer so that the player knows how long it took to win.
- ▶ Try adding variances of the chaos balls.
- ▶ Try adding a chaos goal that requires all of the chaos balls to be complete.
- ▶ Try changing the size or shape of the player’s bumper. Try making a complex bumper out of many shapes.
- ▶ Try covering up the circular water with a terrain, plane, or other game objects around the border of the arena.

HOUR 12

Prefabs

What You'll Learn in This Hour:

- ▶ The basics of prefabs
- ▶ How to work with custom prefabs
- ▶ How to instantiate prefabs in code

A prefab is a complex object that has been bundled up so that it can be recreated over and over with little extra work. In this hour, you learn all about prefabs. You start by learning about prefabs and what they do. From there, you learn how to create prefabs in Unity. You learn about the concept of *inheritance*. You finish by learning how to add prefabs to your scene both through the editor and through code.

Prefab Basics

As mentioned earlier, a prefab is a special type of asset that bundles up game objects. Unlike simply nesting objects in the Hierarchy view, a prefab exists in the Project view and can be reused over and over across many scenes. This enables you to build complex objects, like an enemy, and use it to build an army. You can also create prefabs with code. This allows you to generate a nearly infinite number of objects during runtime. The best part is that any game object, or collection of game objects, can be put in a prefab. The possibilities are endless!

NOTE

Thought Exercise

If you are having trouble understanding the importance of prefabs, consider this: In the preceding hour, you made the game *Chaos Ball*. When making that game, you had to make a single chaos ball and duplicate it four more times. What if you want to make changes to all chaos balls at the same time, regardless of where they were in your scene or project? The fact is that doing so can be difficult (sometimes prohibitively so). Prefabs make is incredibly easy though.

What if you have a game that uses an orc enemy type? Again, you could set a single orc up and then duplicate it many times, but what if you want to use the orc again in another scene? You would have to completely remake the orc in the new scene. If the orc were a prefab, though, it would be a part of the project and could be reused again in any number of scenes. Prefabs are an important aspect of Unity game development.

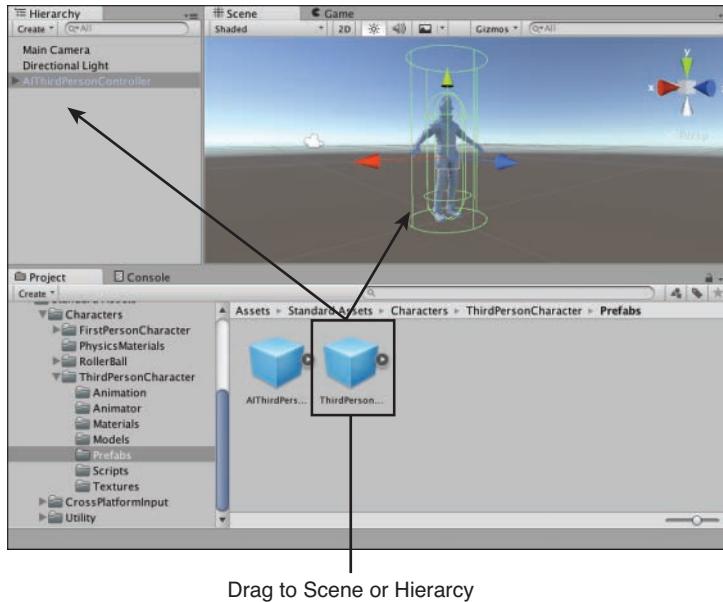
Prefab Terminology

Some terms are important to know when working with prefabs. If you are familiar with the concepts of object-oriented programming practices, you may notice some similarities:

- ▶ **Prefab:** The prefab is the base object. This exists only in the Project view. Think of it as the blueprint.
- ▶ **Instance:** An actual object of the prefab in a scene. If the prefab is a blueprint for a car, an instance is an actual car. If an object in the Scene view is referred to as a prefab, it is really meant that it is a prefab instance. The phrase *instance of a prefab* is synonymous with *object of a prefab* or even *clone* of a prefab.
- ▶ **Instantiate:** The process of creating an instance of a prefab. It is a verb and is used like: “I need to instantiate an instance of this prefab.”
- ▶ **Inheritance:** This does not mean the same thing as standard programming inheritance. In this case, the term *inheritance* refers to the nature by which all instances of a prefab are linked to the prefab itself. This is covered in greater detail later this hour.

Prefab Structure

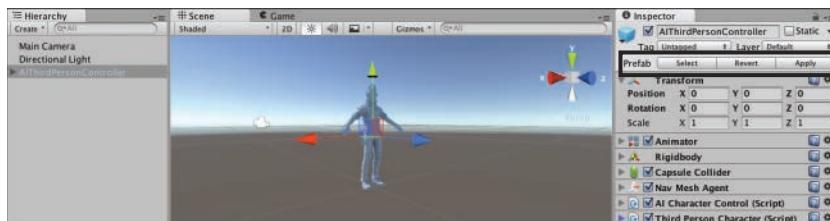
Whether you know it or not, you have already worked with prefabs. Unity’s character controller-sare prefabs. To instantiate an object of a prefab into a scene, you only need to click and drag it into place in the Scene view or Hierarchy view (see Figure 12.1).

**FIGURE 12.1**

Add a prefab instance to a scene.

When looking at the Hierarchy view, you can always tell which objects are instances of prefabs because they will appear blue. This can be a subtle color difference, so note that you can also tell the object is a prefab by looking at the top of the inspector (see Figure 12.2).

Just as with nonprefab complex objects, complex instances of prefabs also have an arrow that allows you to expand them and modify the objects inside.

**FIGURE 12.2**

Prefab instances appearance in Inspector.

Because a prefab is an asset that belongs to a project and not a particular scene, you can edit the prefab in the Project view or the Scene view. If you edit in the Scene view, you must save the changes back to the prefab using the Apply button at the top-right of the inspector (see Figure 12.2).

Just like game objects, prefabs can be complex. Editing the children elements of the prefab is done by clicking the arrow on the right side of the prefab (see Figure 12.3). Clicking this arrow expands the object for editing. Clicking again condenses the prefab again.

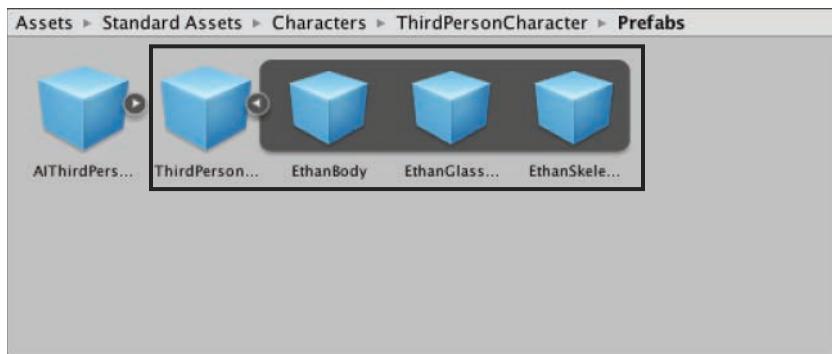


FIGURE 12.3

Expanding the contents of a prefab in the Project view.

Working with Prefabs

Using Unity's built-in prefabs are nice, but often you want to create your own. Creating a prefab is a two-step process. The first step is the creation of the prefab asset. The second step is filling the asset with some content.

Creating a prefab is really easy. Like all other assets, you want to start by creating a folder under Assets in the Project view to contain them. Then, just right-click the newly created folder and select **Create > Prefab** (see Figure 12.4). A new prefab will appear, which you can name whatever you want. Because the prefab is empty, it will appear as an empty white box.

Like all other assets, it is generally a good idea to start by creating a folder under Assets in the Project view to contain your prefabs. Creating a folder to contain your prefabs is not required, but it is a great organization step and should be done to prevent confusion between prefabs and original assets (like meshes or sprites).

The next step is to fill the prefab with something. Any game object can go into a prefab. You simply need to create the object once in the Scene view and then click and drag it onto the prefab asset.

Alternatively, you can shorten this process to a single step by simple dragging any game object from the Hierarchy view down into the project view. Doing so will create and fill the prefab object at the same time.

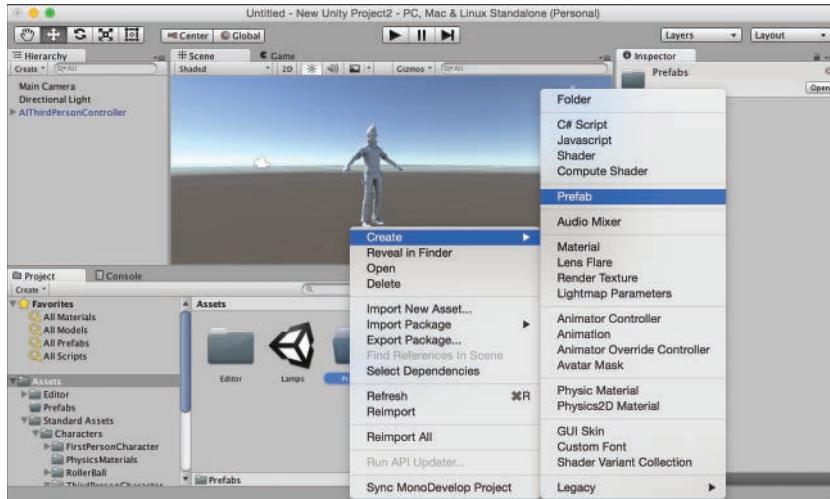


FIGURE 12.4
Creating a new prefab.

TRY IT YOURSELF ▼

Creating A Prefab

Let's create a prefab asset and fill it with a complex game object. The prefab asset created here will be used later in this hour, so don't delete it:

1. Create a new project or scene. Add a cube and a sphere to the scene. Move the directional light to (0, 10, 0).
2. Position the cube at (0, 1, 0) with a scale of (0.5, 2, .5). Add a rigidbody to the cube. Position the sphere at (0, 2.2, 0) with a scale of (1, 1, 1). Put a point light component on the sphere.
3. Click and drag the sphere in the Hierarchy view onto the cube. This will nest the sphere into the cube (see Figure 12.5).

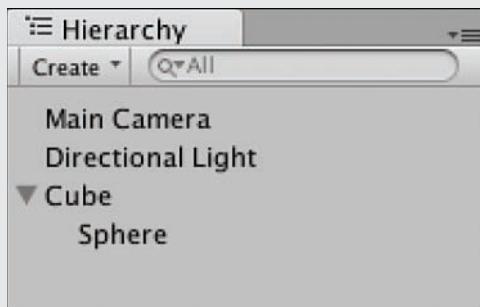


FIGURE 12.5
The sphere nested under the cube.

4. Create a new folder under the Assets folder in the Project view. Name the new folder **Prefabs**. Create a new prefab in the Prefabs folder (right-click and select **Create > Prefab**). Name the new prefab **Lamp**.
5. In the Hierarchy view, click and drag the cube (containing the sphere) onto the lamp prefab in the Project view (see Figure 12.6). You will notice that the prefab now looks like the lamp. You will also notice that the cube and sphere in the Hierarchy view turned blue. At this point, you can delete the cube and sphere from the scene. They are now contained in the prefab.

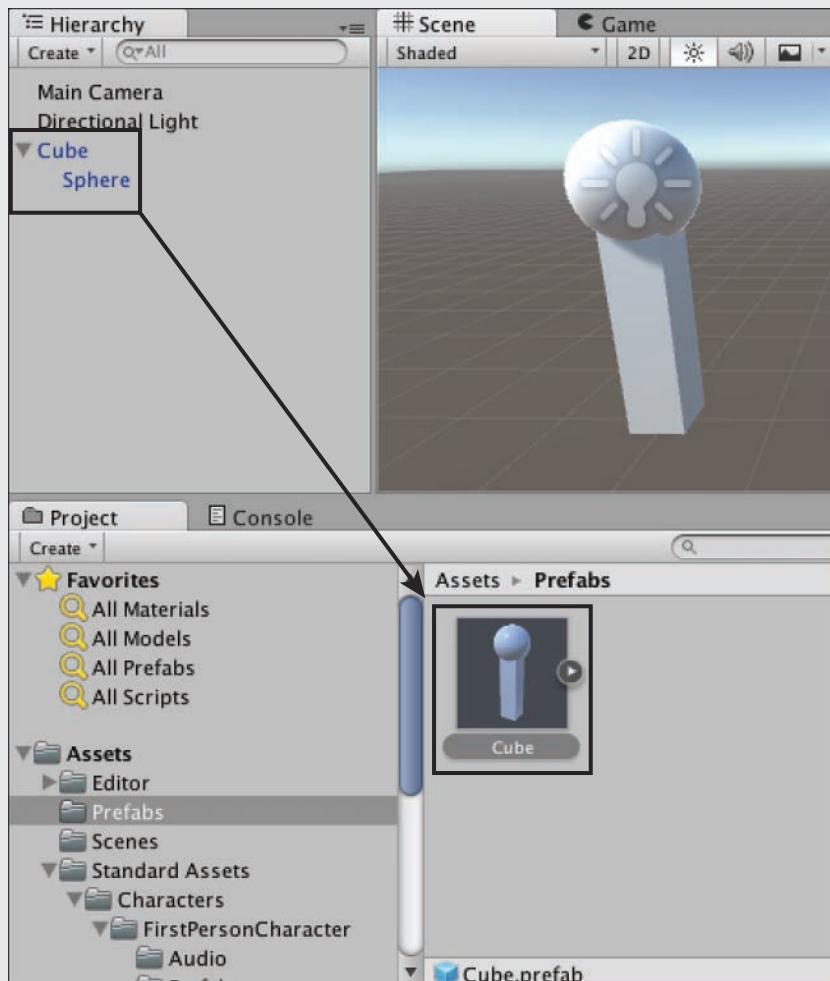


FIGURE 12.6
Adding an object to a prefab.

Adding a Prefab Instance to a Scene

Once a prefab asset is created, it can be added as many times as you want to a scene or any number of scenes in a project. To add a prefab instance to a scene, all you need to do is click and drag the prefab from the Project view into place in the Scene view, or Hierarchy.

If you drag into the Scene, it will be instantiated where you drag. If you drag into a blank part of the Hierarchy, its initial position will be whatever is set in the prefab. If you drag onto another object in the Hierarchy, the prefab will become a child of that object.

TRY IT YOURSELF ▼

Creating Multiple Prefab Instances

In the last exercise, you made a Lamp prefab. This time, you will be using the prefab to create many lamps in a scene. Be sure to save the scene created here; it is used later this hour:

1. Create a new scene in the same project used for the last exercise. Call this scene Lamps.
2. Create an empty game object called Floor; ensure it is at (0, 0, 0).
3. Create a 3D Cube. Position the cube at (0, 0, 0) with a scale of (5, .1, 5).
4. Drag this cube to the Floor object to make it a child. Optionally give the Floor a Gray material so that shadows appear more clearly on it.
5. Drag your Lamp prefab onto the Floor object, and notice how the lamp appears in the middle of the floor.
6. Drag 3 more lamp prefabs onto the floor, and position them near the corners (see Figure 12.7).

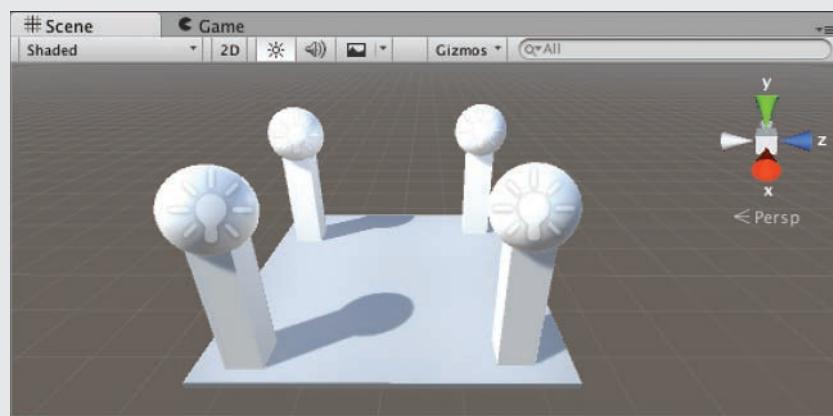


FIGURE 12.7

Placing lamps in the scene.

Inheritance

When the term *inheritance* is used in conjunction with prefabs, it means the link by which the instances of a prefab are connected to the actual prefab asset. That is, if you change the prefab asset, all objects of the prefab are also automatically changed. This is incredibly useful. More often than not, you put a large number of prefab objects into a scene only to realize that they all need a minor change. Without inheritance, you would have to change each one independently.

There are two ways in which you can change a prefab asset. The first is by making changes in the Project view. Just selecting the prefab asset in the Project view will bring up its components and properties in the Inspector view. If you need to modify a child element, you can expand the prefab (described earlier) and change those objects in a similar fashion.

Another way you can modify a prefab asset is to drag an instance into the scene. From there, you can make any major modifications you would like. When finished, simply click **Apply** at the top-right of the Inspector.

TRY IT YOURSELF

Updating Prefabs

So far, you have created a prefab and added several instances to a scene. Now you get a chance to modify the prefab and see how it affects the assets already in the scene. This exercise uses the scene created in the previous exercise. If you have not done that one yet, you need to do so before continuing:

1. Open the scene with the lamps that you created previously.
2. Select the **Lamp** prefab from the Project view and expand it (click the arrow on the right side). Select the **Sphere** child component. In the Inspector, change the color of the light to red (see Figure 12.8). Notice how the prefabs in the scene automatically change.

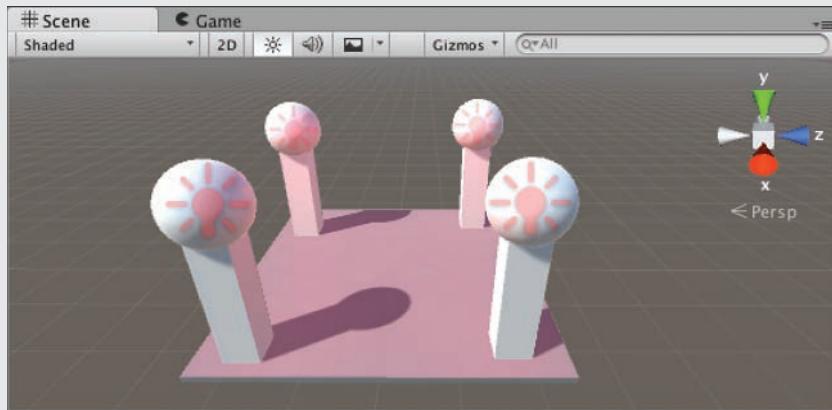


FIGURE 12.8

The modified lamp instances.

3. Select one of the lamp instances in the scene. Expand it by clicking the arrow to the left of its name in the Hierarchy view and select the Sphere child object. Change the sphere's light back to white. Notice how the other prefab objects don't change.
4. With the lamp still selected, click **Apply** to update the prefab modified lamp instance back onto the prefab asset. Notice how all of the instances change back to a white light.

Breaking Prefabs

Sometimes, you need to break a prefab instance's link to the prefab asset. You might want to do this if you need an object of the prefab but you don't want the object to change if the prefab ever changes. Breaking an instance's link to the prefab does not change the instance in any way. It still maintains all of its objects, components, and properties. The only difference is that it is no longer an instance of the prefab and therefore is no longer affected by inheritance.

To break an object's link to the prefab asset, simply select the object in the Hierarchy view. After selecting it, click **GameObject > Break Prefab Instance**. You will notice that the object does not change, but its name turns from blue to black. Once the link is broken, it cannot be reapplied.

Instantiating Prefabs Through Code

Placing prefab objects into a scene is a great way to build a consistent and planned level. Sometimes, however, you want to create instances at runtime. Maybe you want enemies to respawn, or you want them to be randomly placed. It is also possible that you need so many instances that placing them by hand is no longer feasible. Whatever the reason, instantiating prefabs through code is a good solution.

There are two ways to instantiate prefab objects in a scene and they both use the `Instantiate()` method. The first way is to use `Instantiate()` like this:

```
Instantiate(GameObject prefab);
```

As you can see, this method simply reads in a game object variable and makes a new object of it. The location, rotation, and scale of the new object are the same as the prefab in the Project view. The second way to use the `Instantiate()` method is like this:

```
Instantiate(GameObject prefab, Vector3 position, Quaternion rotation);
```

This method requires three parameters. The first is still the object to make a copy of. The second and third parameters are the desired position and rotation of the new object. You might have noticed that the rotation is stored in something called a Quaternion. Just know that this is how Unity stores rotation information. The true application of the Quaternion is beyond the scope of this hour. An example of the two methods of instantiating objects in code can be found in the exercise at the end of this hour.

Summary

In this hour, you learned all about prefabs in Unity. You started by learning the basics of prefabs: the concept, the terminology, and the structure. From there, you learned to make your own prefabs. You explored how to create them, add them to a scene, modify them, and break them. Finally, you learned to instantiate prefab objects through code.

Q&A

- Q. Prefabs seem a lot like classes in object-oriented programming (OOP). Is that accurate?**
- A.** Yes, there are many similarities between classes and prefabs. Both are like blueprints. Objects of both are created through instantiation. Objects of both are linked to the original.
- Q. How many objects of a prefab can exist in a scene?**
- A.** As many as you want. Be aware, though, that after you get above a certain number, the performance of the game will be impacted. Every time you create an instance, it is permanent until destroyed. Therefore, if you create 10,000, there will be 10,000 just sitting in your scene.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What is the term for creating an instance of a prefab asset?
2. What are two ways to modify a prefab asset?
3. What is inheritance?
4. How many ways can you use the `Instantiate()` method?

Answers

1. Instantiation.
2. You can modify a prefab asset through the Project view or by modifying an instance in the Scene view and clicking **Apply** in the Inspector.
3. It is the link that connects the prefab asset to its instances. It basically means that when the asset changes, the objects change as well.
4. Two. You can specify just the prefab or you can also specify the position and rotation.

Exercise

In this exercise, you work once again with the prefab you made earlier this hour. This time, you instantiate objects of the prefab through code and hopefully have some fun with it. You can find the complete project for this exercise as Hour12_Exercise in the book assets for Hour 12:

1. Create a new scene in the same project the Lamp prefab is in. Click the **Lamp** prefab in the Project view and give it a position of $(-1, 0, -5)$.
2. Add an empty game object to your scene. Rename the game object **SpawnPoint** and position it at $(1, 1, -5)$. Add a plane to your scene and position it at $(0, 0, -4)$ with a rotation of $(270, 0, 0)$.
3. Add a script to your project. Name the script **PrefabGenerator** and attach it to the spawn point object. Listing 12.1 has the complete code for the prefab generator script.

LISTING 12.1 PrefabGenerator.cs

```
using UnityEngine;
using System.Collections;

public class PrefabGenerator : MonoBehaviour {
    // We will store a reference to the target prefab from the inspector
    public GameObject prefab;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
        // Whenever we hit the B key, we will generate a prefab at the
        // position of the original prefab
        // Whenever we hit the space key, we will generate a prefab at the
        // position of the spawn object that this script is attached to
        if (Input.GetKeyDown(KeyCode.B)) {
            Instantiate(prefab);
        }

        if (Input.GetKeyDown(KeyCode.Space)) {
            Instantiate(prefab, transform.position, transform.rotation);
        }
    }
}
```

4. With the spawn point selected, drag the Lamp prefab onto the Prefab property of the Prefab Generator component. Now run the scene. Notice how pressing the **B** button creates a lamp at its default prefab position and how pressing the spacebar creates an object at the spawn point.

This page intentionally left blank

HOUR 13

2D Games Tools

What You'll Learn in This Hour:

- ▶ How orthographic cameras work
- ▶ How to position sprites in 3D space
- ▶ How to move and collide sprites

Unity is a powerhouse at creating both 2D and 3D games. A pure 2D game is one where all your assets are simple flat images called sprites. In a 3D game, the assets are 3D models, with 2D textures applied to them. Typically, in a 2D game, the player can only move in two dimensions (e.g., left, right, up, down). This hour is devoted to understanding the basics of creating 2D games in Unity.

The Basics of 2D Games

The principles of design for a 2D game are the same as for a 3D game. You still need to consider the concepts, rules, and requirements of the game. There are pros and cons to making your game 2D. On one hand, a 2D game can be simple and much cheaper to produce. On the other hand, the limitations of 2D can make some types of games impossible. 2D games are built from images called sprites. These are a bit like cardboard cutouts in a kids stage show. You can move them around, place them in front of each other, and make them interact to make a rich environment.

When you create a new Unity project, you'll notice that you have the choice of 2D and 3D (see Figure 13.1). Selecting 2D will default the Scene view to 2D and make the camera orthographic (we'll look at those in a moment). Another thing you will notice about a 2D project is that there's no Directional Light or skybox in the scene. In fact, 2D games are generally unlit because sprites are drawn by a simple type of renderer called a "sprite renderer." Unlike a texture, lighting does not generally affect the way a sprite is drawn.

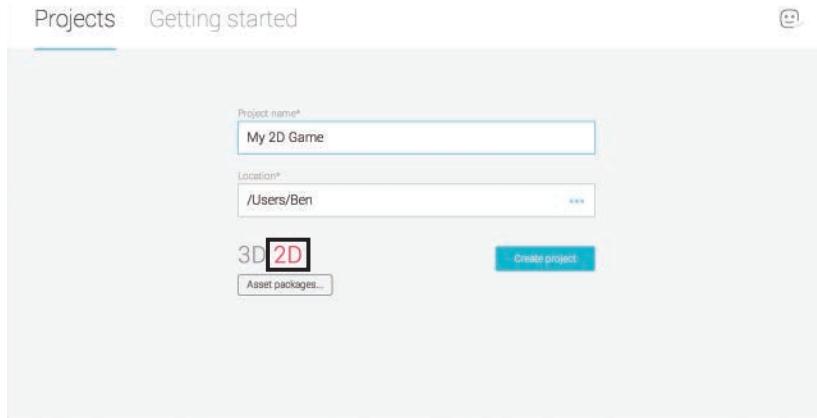


FIGURE 13.1
Setting a project to 2D.

TIP

2D Game Challenges

2D games bring different design challenges. Examples include the following:

- ▶ The lack of depth makes immersion harder to achieve.
- ▶ A lot of game types don't translate well to 2D.
- ▶ 2D games are generally unlit, so the sprites have to be carefully drawn and arranged.

The 2D Scene View

A project with 2D defaults will already be in the 2D scene view. To change between 2D and 3D view, click the 2D button at the top of the scene view (see Figure 13.2). You will notice that the 3D gizmo disappears when you enter 2D mode.

You can move around in 2D by dragging in the scene with the middle or right mouse button held down. You can use the mouse wheel to zoom, or the scroll gesture on your trackpad. The scale of the flat background will keep changing as you zoom. You will need to pay attention to the Transform in the Inspector to keep track of where you are in the scene.

**FIGURE 13.2**

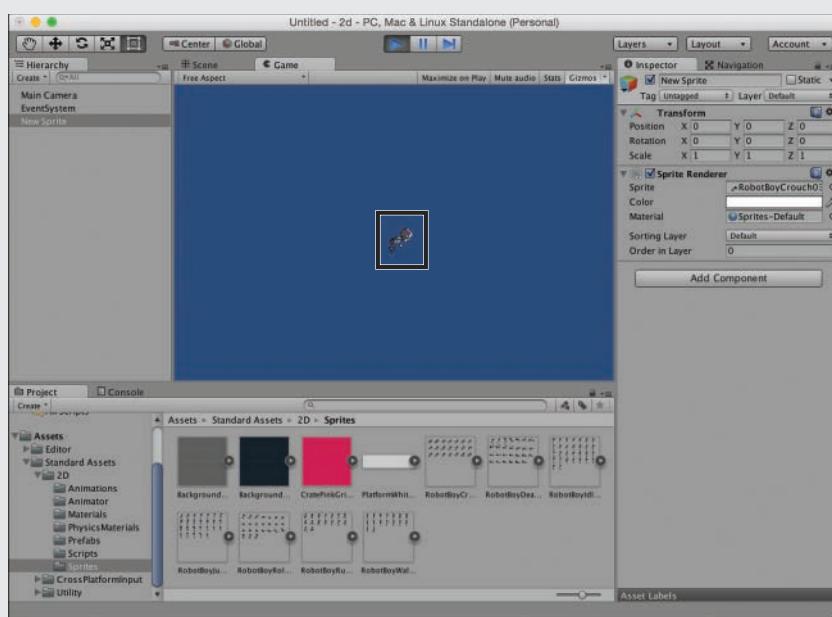
The 2D scene view.

TRY IT YOURSELF ▼

Placing a Sprite

Let's create a simple sprite on the screen.

1. Create a new project, and this time select 2D. This will set up the project defaults for a 2D game. Note there is no directional light in a 2D game.
2. Add a new sprite (click GameObject > 2D Object > Sprite).
3. Import the 2D standard assets package (click Assets > Import Package > 2D).
4. In the inspector under Sprite Renderer, set the Sprite to any of the RobotBoy sprites (see Figure 13.3).
5. Run the scene. You should see the robot sprite against a blue background.

**FIGURE 13.3**

The robot sprite in the scene.

Orthographic Cameras

Because you set up your project as a 2D project, the camera will default to the orthographic type. This means that the camera doesn't show perspective distortion; rather, all objects appear the same size and shape regardless of distance. This type of camera is suitable for 2D games, where we control the apparent depth of sprites by their size and *sorting layer*.

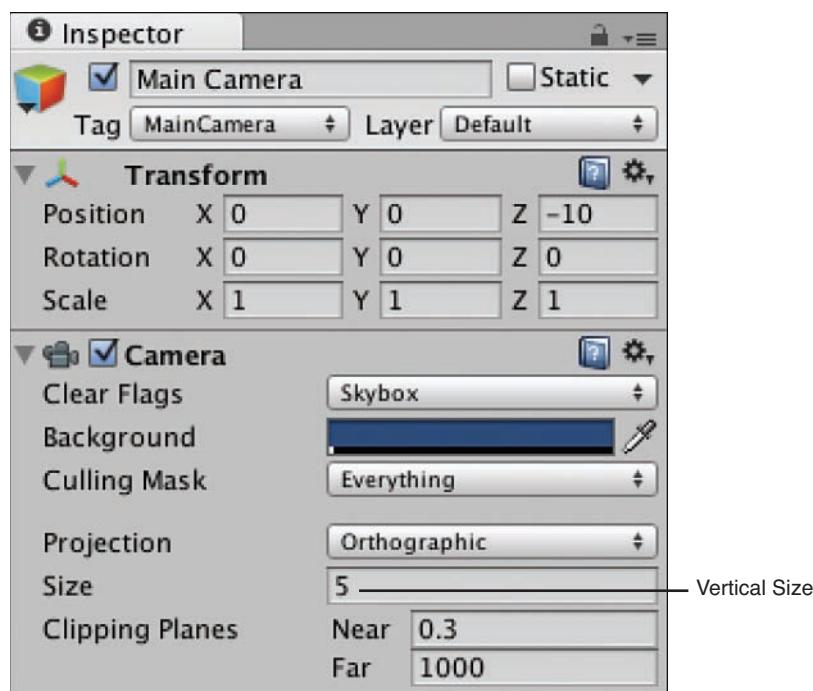
A sorting layer is a way of determining which sprites, and groups of sprites, draw in front of each other. Imagine you are setting up a stage. You would want the background props behind the foreground props, etc. If you sort the sprites from front-to-back properly, and choose appropriate sizes, you can give an impression of depth.

You can see the camera type for yourself by clicking on Main Camera, and noticing the Projection is set to Orthographic in the Inspector. The rest of the camera settings were covered in Hour 6, "Lights and Cameras."

TIP**Size of an Orthographic Camera**

Often you want to change the relative size of sprites in your scene all at once, without resizing them. If the camera doesn't take depth into account, you may be wondering how you do this, as you can't just move them closer to the camera.

The answer is the Size property in the inspector (see Figure 13.4), which only appears for orthographic projection cameras. This is the number of world units from the center to the top of the camera's view, which can be a little confusing, but play around with the numbers a bit and it will make sense in no time.

**FIGURE 13.4**

Setting the size of an orthographic camera.

Adding Sprites

Adding your sprite to a scene is a very easy process. Once a sprite image is imported into your project (again, super simple), you simply need to drag it into your scene.

Importing Sprites

If you wish to use your own image as a sprite, you need to make sure Unity will use it as a sprite. The steps are simple:

1. Locate the `ranger.png` file in the book assets (or use your own).
2. Drag the image into the project view in Unity. If you created a 2D project, this will import as a sprite. If you made a 3D project, you will need to tell Unity to use it as a sprite by setting the Texture Type to Sprite (see Figure 13.5).
3. Simply drag the sprite into the scene view. It's that simple! Note that your scene view has to be in 2D mode in order to drag sprites in.

This will set the image as a sprite, which you can then use by simply dragging into your Scene view.

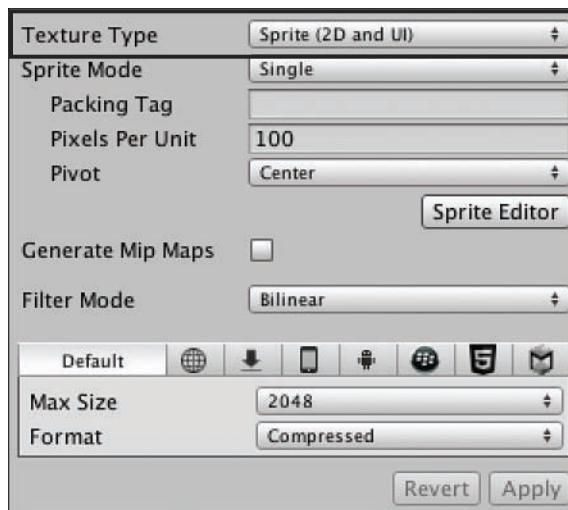


FIGURE 13.5
Making an image texture into a sprite.

Sprite Mode

Single sprites are fine, but the fun really starts when we start animating. Unity provides some powerful tools for using “sprite sheets.” These are images with multiple frames of an animation laid out in a grid.

The Sprite Editor (which we will explore in a moment) helps to automatically extract 10s or 100s of different animation frames from a single image.

TRY IT YOURSELF ▼

Exploring Sprite Mode

In this exercise, we will explore the difference between Single and Multiple sprite mode.

1. Create a new 2D project or a new scene in your existing 2D project.
2. Import `rangerTalking.png` from the book files or source your own sprite sheet.
3. Check that the Sprite Mode is Single, and expand the tray in the Assets to see that Unity is treating this image as a single sprite (see Figure 13.6).

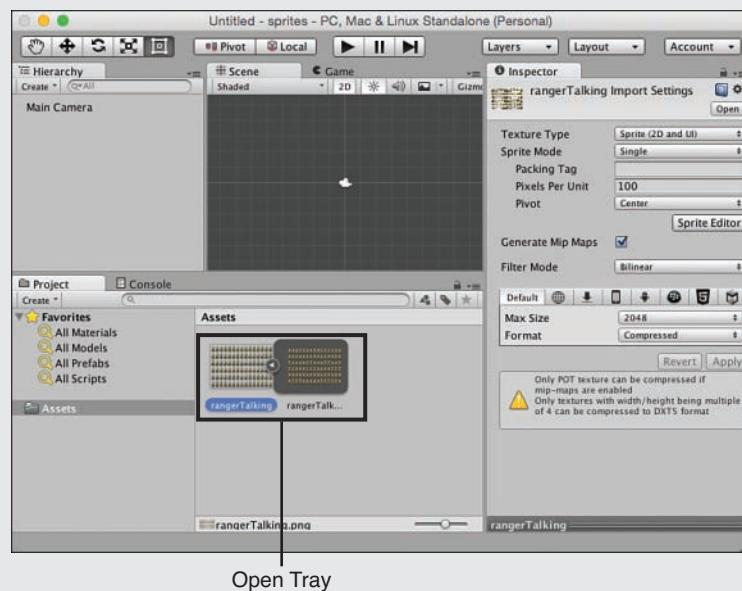
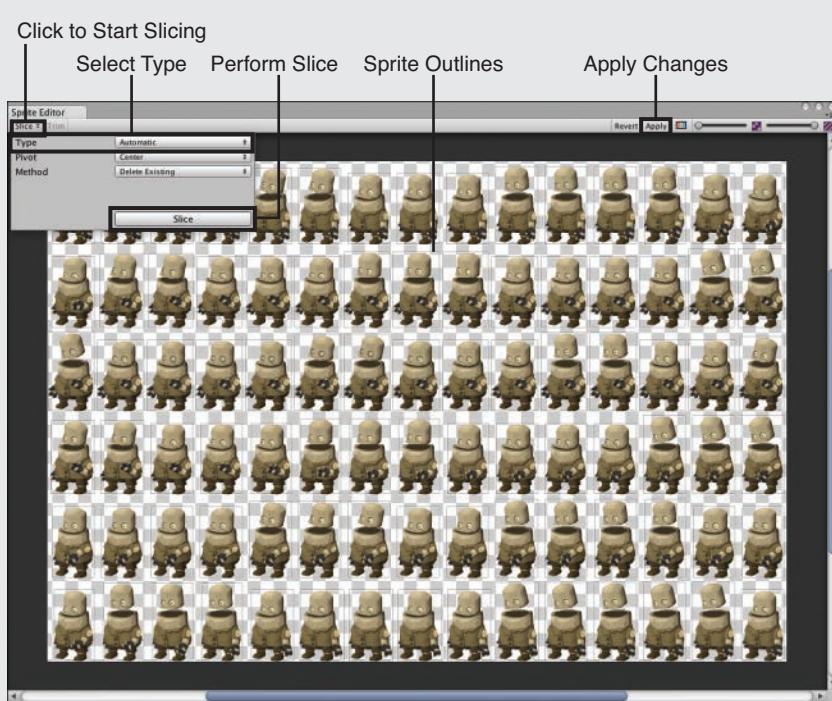


FIGURE 13.6

Importing a sprite in Single mode.

4. Now change the Sprite Mode to Multiple, and click Apply at the bottom of the Inspector. Note that there is temporarily no tray to expand.
5. Click the Sprite Editor button in the Inspector; a window will pop up. Set Type to Automatic and click Slice (see Figure 13.7). Notice how the outlines are automatically detected, but different for each frame.
6. Now set Type to Grid. Adjust the grid to fit your sprite, for the supplied image $x = 62$ and $y = 105$. Leave the other settings and click Slice. Notice the borders are now more even.
7. Look at your sprite in your Assets, and note that the tray now contains all the individual frames ready for animation.

**FIGURE 13.7**

The Sprite Editor Window.

TIP**The Rect Tool**

As explained in Hour 1, the Rect Transform Tool is idea for manipulating rect(angular) sprites. This can be access at the top-left of Unity, as the right-hand most of the five transform tool buttons, or by pressing the Y key. Have a play with it while you have a sprite on screen.

Imported Sprite Sizes

If you need to scale your sprite images so that their sizes match, you could use the scale tool in the scene. Doing that, however, adds inefficiencies and potentially “odd” scaling behavior in the future.

If you need to consistently scale your sprite, say SpriteA always needs to be half as big, consider instead using the Pixels Per Unit import setting. This setting determines how many world units a sprite of a given resolution occupies. For example, a 640×480 image imported with a Pixels Per Unit setting of 100 would occupy 6.4×4.8 world units.

The final block of settings relate to platform-specific overrides, which are beyond the scope of this book. You can safely leave these at their Default values.

Draw Order

To determine what sprites draw in front of each other, Unity has a Sorting Layer system. This is controlled by two settings on the Sprite Renderer component, “Sorting Layer,” and “Order In Layer” (see Figure 13.8).



FIGURE 13.8

The default sprite layer properties.

Sorting Layer

Sorting layers are used to group major categories of sprite by depth. When you first create a new project, there is only one sorting layer called Default (see Figure 13.8). Imagine you have a simple 2D platform game with a background consisting of hills, trees, and clouds. You would want to create a Sorting Layer called Background for this.

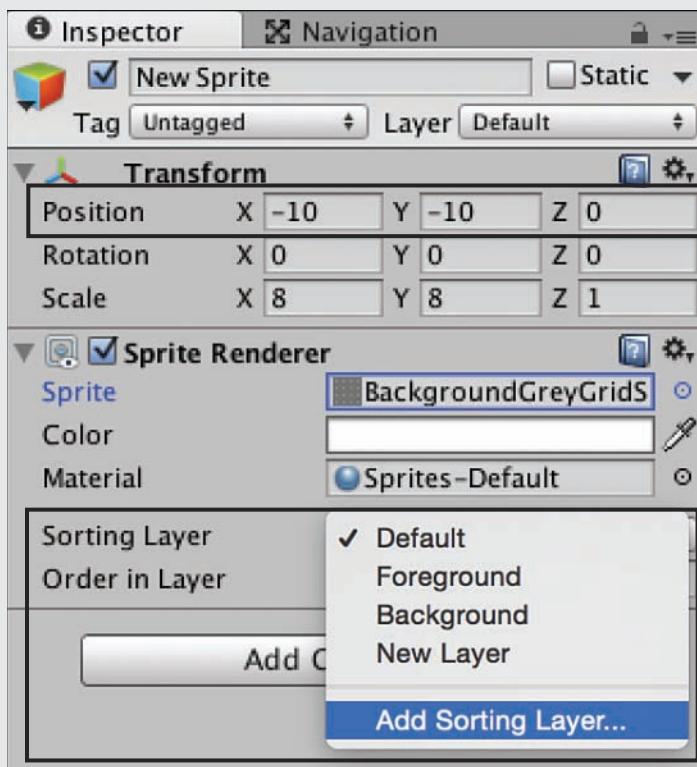
TRY IT YOURSELF ▾

Creating Sorting Layers

Let's create and assign a new sorting layer to a sprite.

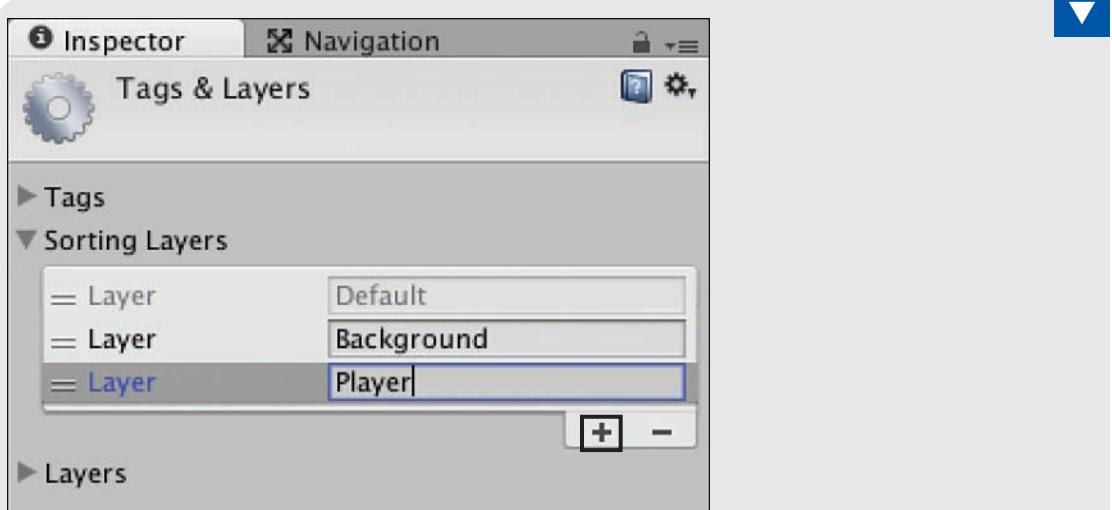
1. Create a new Project, and import the entire 2D Asset Pack (click **Assets > Import Package > 2D**).
2. Add a new sprite (click **GameObject > 2D Object > Sprite**). Set its position to $(-10, -10, 0)$ and scale it to $(8, 8, 1)$. This is another way of adding a sprite, but it's usually easier to just drag it in.

3. Set the Sprite to **BackgroundGreyGridSprite**, and add a new Sorting Layer by clicking Default under Sorting Layer, then clicking Add Sorting Layer (see Figure 13.9).

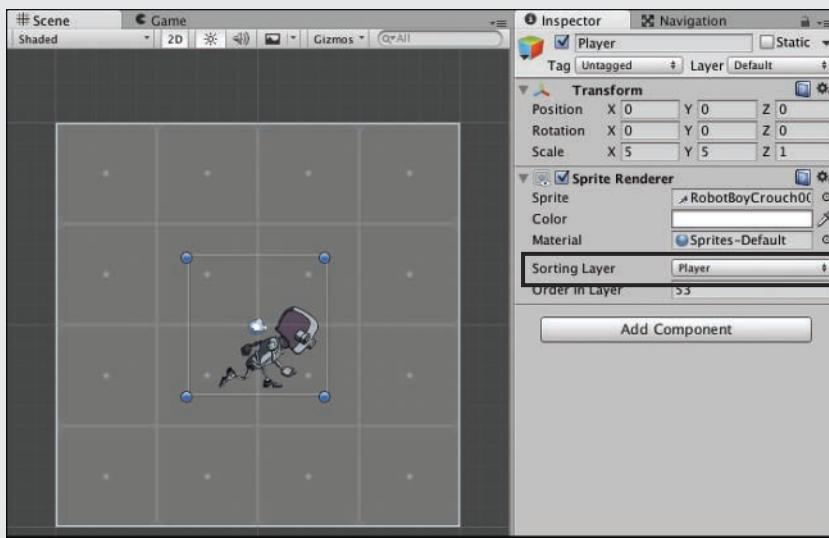
**FIGURE 13.9**

Adding new Sorting Layers.

4. Add two new layers called **Background** and **Player**. Do this by clicking the + button under Sorting Layers (see Figure 13.10). Note the lowest layer in the list, the Player, will be drawn on top. It is a good idea to move the Default layer to the bottom, so that new items are drawn on top.
5. Click back on the New Sprite in the Hierarchy, and set its sorting layer to Background.
6. Add another sprite as in Step 2; this time select **RobotBoyCrouch00** as the Sprite. Position it at (0, 0, 0) and scale it to (5, 5, 0). Set this sprite's Sorting Layer to Player (see Figure 13.11).
7. Note the Player is drawn on top of the background. Now try going back into the Tags and Layers window (**Edit > Project Settings > Tags and Layers**) and rearranging the layers so that the background is drawn on top of the Player.

**FIGURE 13.10**

Managing Sorting Layers.

**FIGURE 13.11**

Setting a sprite's sorting layer.

You would likely want another layer for power-ups, bullets they may spawn, etc. The ground or platform may be on another layer. Finally, if you want to add more sense of depth with subtle foreground sprites, these could go in a Foreground layer.

Order in Layer

Once you have your major layers defined, and assigned to your sprites, you can fine-tune the draw order with this setting. This is a simple priority system, with higher numbers drawing on top of lower numbers.

WARNING

Sprites Going Missing

When you are starting out, it is quite common for sprites to go missing altogether. This is usually either because the sprite is behind a larger element or behind the camera.

You can also find that if you forget to set the Sorting Layer and Order In Layer properties, the z-depth can affect the draw order. This is why it is important to always set the layer for every sprite.

2D Physics

Now that we understand how static sprites work in our game, it's time to add some spice by making them move. Unity has some fantastic tools to help with this. There are two main ways of animating sprites in Unity as detailed in the following. Let's dive in and explore them. Unity has a powerful 2D physics system, integrating a system called Box2D. Just like in 3D games, you have rigidbodies with gravity, various colliders, and physics specific to 2D platform and driving games.

Rigidbody 2D

Unity has a different type of Rigidbody for 2D purposes. This physics component shares many of the same properties with its 3D counterpart you have already met. It is a common mistake to choose the wrong type of Rigidbody or Collider for your game, so be careful you are using items from **Add Component > Physics 2D**.

TIP

Mixing Physics Types

2D and 3D physics can exist in the same scene together, they just won't interact with each other. Also, you can put 2D physics on 3D objects and vice versa!

2D Colliders

Just like 3D colliders, 2D colliders allow game objects to interact with one another when they touch. Unity comes with a range of 2D colliders (see Table 13.1).

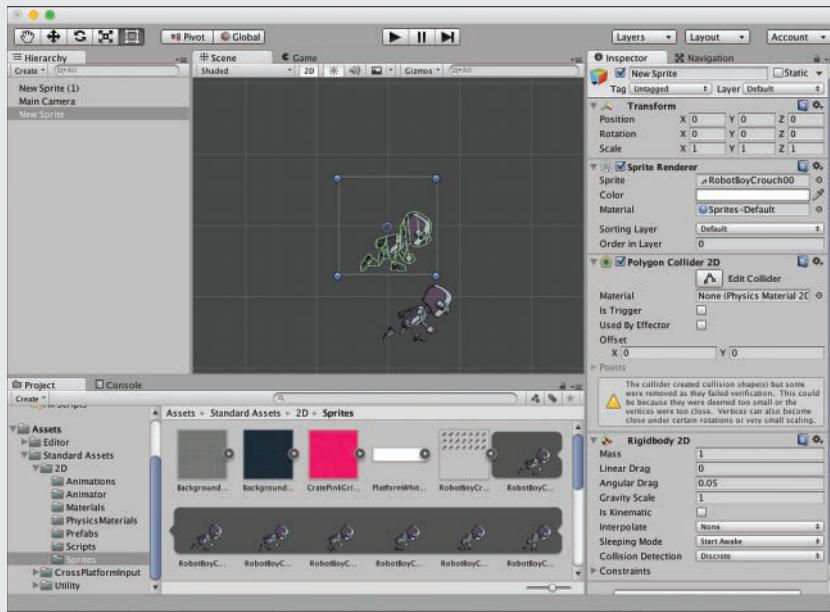
TABLE 13.1 Unity 5's 2D Colliders

Collider	Description
Circle Collider 2D	A circle with fixed radius and an offset.
Box Collider 2D	A rectangle with adjustable width, height, and offset.
Edge Collider 2D	A segmented line that does not need to be closed.
Polygon Collider 2D	Any closed polygon with three or more sides.

TRY IT YOURSELF ▼**Make Two Sprites Collide**

Let's see the effect of 2D colliders by playing with some 2D squares.

1. Create a new 2D Scene, and ensure the 2D package is imported.
2. Find the sprite **RobotBoyCrouch00** in your Assets, and drag into your Hierarchy. Ensure the sprite is at (0, 0, 0).
3. Add a polygon collider (click **Add Component > Physics 2D > Polygon Collider 2D**). You can ignore the warning in the Inspector. Note how this collider roughly fits the outline of the sprite.
4. Duplicate this sprite, and move the duplicate down and to the right to (.3, -1, 0). This will give the top sprite something to fall onto.
5. So that the top sprite responds to gravity, select it, and add a Rigidbody 2D (click **Add Component > Physics 2D > Rigidbody 2D**). See Figure 13.12 for the finished setup.
6. Play the game and notice the behavior.

**FIGURE 13.12**

The finished setup with the top robot selected.

TIP**2D Colliders and Depth**

One thing that you may find a little odd about 2D colliders, if you look at them in a 3D scene view, is that they don't need to be at the same z-depth to collide. They work only on the x and y position of the collider.

Summary

In this hour, you learned about the basics of 2D games in Unity. We started by increasing your understanding of orthographic cameras and how depth works in 2D. You went on to make a simple 2D object move and collide, the basis of many 2D games.

Q&A

Q. Is Unity a good choice for creating 2D games?

A. Yes, Unity has a fantastic set of tools for creating 2D games.

Q. Can Unity deploy 2D games to mobile and other platforms?

A. Absolutely, one of Unity's core strengths is its capability to deploy to many platforms with relative ease. 2D games are no exception to this, and many very successful 2D games have been made with Unity.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What camera projection renders all objects without perspective distortion?
2. What does the size setting of an orthographic camera relate to?
3. Do two 2D sprites need to be at the same z-depth in order to collide?
4. Will sprites render if they are behind the camera?

Answers

1. Orthographic.
2. The size setting specifies half of the vertical height that the camera covers, specified in World Units.
3. No, 2D collisions only take account of x and y position.
4. No, this is a common cause of lost sprites when making 2D games.

Exercise

In this exercise, we are going to use some Unity standard assets, so that you can see what can be achieved when you combine sprites with a little animation, some character control script, and some colliders.

1. Create a new 2D project, or a new scene in your existing project.
2. Import the 2D asset pack (click **Assets > Import Package > 2D**). This time leave everything selected.

3. Find the **CharacterRobotBoy** in **Assets > Standard Assets > 2D > Prefabs**, and drag it into the Hierarchy. Set the position to (3, 1.8, .0). Note how this prefab comes with many components in the inspector.
4. Find the **PlatformWhiteSprite** in **Assets > Standard Assets > 2D > Sprites** and drag this into the Hierarchy. Set the position to (0, 0, 0), and scale to (3, 1, 1). Add a Box Collider 2D, so that your player doesn't fall through the floor!
5. Duplicate this platform game object. Position the duplicate at (7.5, 0, 0), rotate it to (0, 0, 30) to make a ramp, and scale it to (3, 1, 1).
6. Move the **Main Camera** to (11, 4, -10), and adjust its size to 7. See Figure 13.13 for the final setup.
7. Click **Play**. Use the cursor keys to move and the spacebar to jump.

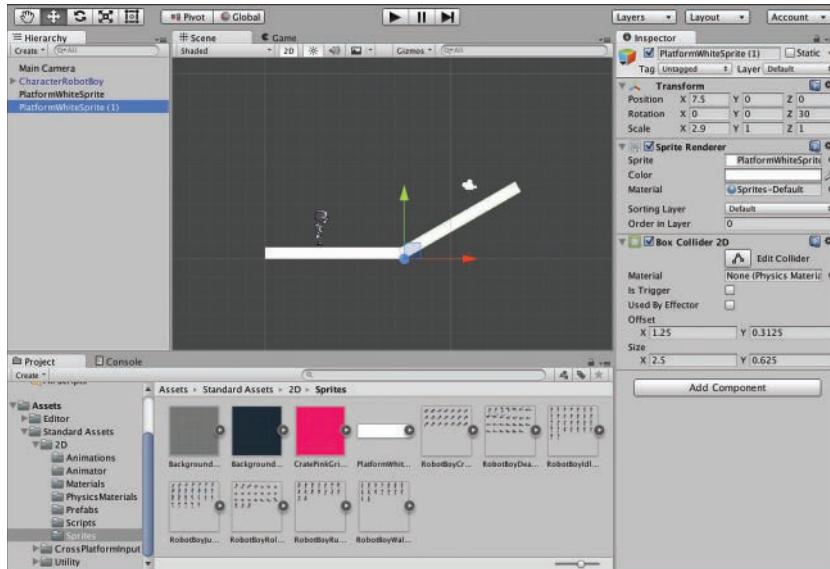


FIGURE 13.13
The final exercise setup.

HOUR 14

User Interfaces

What You'll Learn in This Hour:

- ▶ The elements of a User Interface (UI)
- ▶ An overview of all the UI elements
- ▶ About the different UI render modes
- ▶ How to build a simple menu system

A user interface (UI) is a special set of components responsible for sending information to, and reading information from, the user. In this hour, you learn all about using Unity's built-in UI system. You start by examining the UI basics. From there, you get to try out the various UI elements such as text, images, buttons, and more. You finish by creating a simple but complete menu system for your games.

Basic UI Principles

As mentioned previously, user interfaces (commonly referred to as UIs) are special layers that exist to give information to the user and to accept simple inputs from the user. This information and input could take the form of an HUD (Heads Up Display) drawn overtop of your game or some object actually located within your 3D worlds.

In Unity, the UI is based on a **Canvas** onto which all the UI elements are painted. This canvas needs to be the parent of all UI objects for them to work and is the main object driving your entire UI.

TIP

UI Design

As a general rule, you want to sketch your UI ahead of time. A fair bit of thought needs to go into what to display on the screen, where it will be displayed, and how. Too much information will cause the screen to feel cluttered. Too little information will leave the players confused or unsure. Always look for ways to condense information or make information more meaningful. Your players will thank you.

TIP**New UI**

As of Unity 4.6 Unity has had a new UI implemented. No longer do you need to create your UIs with many lines of confusing code. That being said, the old UI system is still there. If you are familiar with the legacy system, you may be tempted to use it. Please don't. The legacy system is only still there for debugging, backwards compatibility with old projects, and editor extensions. It is not nearly as efficient or powerful as the new system, so it would be best for you to use it!

The Canvas

A canvas is the basic building block for a UI, under which all the UI elements are contained. All the UI elements you add to the scene will be child objects of the Canvas in the Hierarchy, and must stay as children (otherwise they'll disappear)!

Adding a canvas to a scene is very easy. You can add one simply by clicking **GameObject > UI > Canvas**. Once added to a scene, you are now ready to begin building the rest of a UI.

▼ TRY IT YOURSELF

Adding a Canvas

Let's adds a canvas to a scene and explore its unique features.

1. Create a new project (can be either 2D or 3D).
2. Add a UI Canvas to the scene (Click **GameObject > UI > Canvas**).
3. Zoom out to see the whole canvas (double click it in Hierarchy). Notice how big it is!
4. Note in the inspector the odd transform component the canvas has. This is a "rect" transform and we will discuss it shortly.

NOTE**EventSystem**

You may have noticed that when you added a canvas to your scene, you also got an **EventSystem** game object. Don't worry, this object is always added when you add a canvas. The Event System is what allows our users to interact with the UI be pressing buttons or dragging elements. Without the events, our UI would never know if it was being used (so don't delete it)!

The Rect Transform

You will notice that a canvas (and all UI elements) have this **Rect Transform** rather than the normal 3D Transform you are familiar with. Rect, short for rectangle, transforms give you

fantastic control over the positioning rescaling of UI elements while remaining very flexible. This allows you to create one user interface, and ensure that it works well on a wide range of devices.

In the instance of the canvas you created earlier, you will notice that the Rect Transform is entirely grayed out (see Figure 14.1). This is because, in its current form, the canvas derives its values entirely from the Game View (and by extension the resolution and aspect ratio of any devices your game runs on). This means that the canvas will always take up the entire screen. A good workflow is to make sure the first thing you do whenever building a UI is first selecting a target aspect ratio to work with. This can be done from the Aspect Ratio drop down in the Game View (see Figure 14.2).

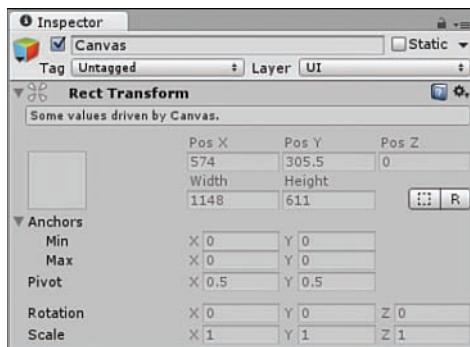


FIGURE 14.1

The Rect Transform of a canvas.

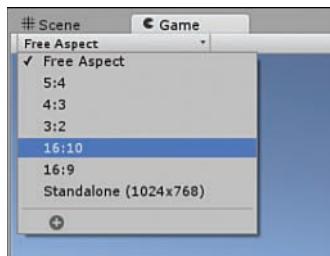


FIGURE 14.2

Setting the game's aspect ratio.

Rect transforms themselves work a little different from a traditional transform. With normal 2D and 3D objects, the transform is concerned with determining how far (and with what alignment) an object is from the world origin. The UI, however, is unconcerned with world origin and instead needs to know how it's aligned in relation to its anchor point. More will be explained about rect transforms and anchors later when we have a UI element that can actually use it.

Anchors

A key concept in making UI elements work is that of anchors. Every UI element comes with an anchor and uses that anchor to find its place in the world in relation to the rect transform of its parent. These anchors determine how the element is resized and repositioned when the Game window changes size and shape. Additionally, an anchor has two “modes”: together and split. When an anchor is together as a single point, the object knows where it is by determining how far (in pixels) its pivot is from the anchor. When an anchor is split, however, the UI element bases its bounding box on how far (again in pixels) each of its corners is from each corner of the split anchor. Confusing? Let’s try it out!

TRY IT YOURSELF

Using A Rect Transform

Rects and anchors can seem confusing, so let’s use one in an example.

1. Create a new 2D project.
2. Add a UI Image (click **GameObject > UI > Image**). Note that adding an image to your scene without a canvas will automatically put a canvas in your scene and put the image on it.
3. Zoom out so you can see the whole Image and Canvas. Note that it is much easier working with the UI when your scene is in 2D mode (button at top of scene view labeled “2D”) and you are using the “Rect” tool (hotkey “T”).
4. Try dragging the image around the canvas. Also, try dragging the anchor around the canvas. Notice how the lines show you how far the image’s pivot is from the anchor. Also notice the properties of the Rect transform in the inspector and how they change (see Figure 14.3).

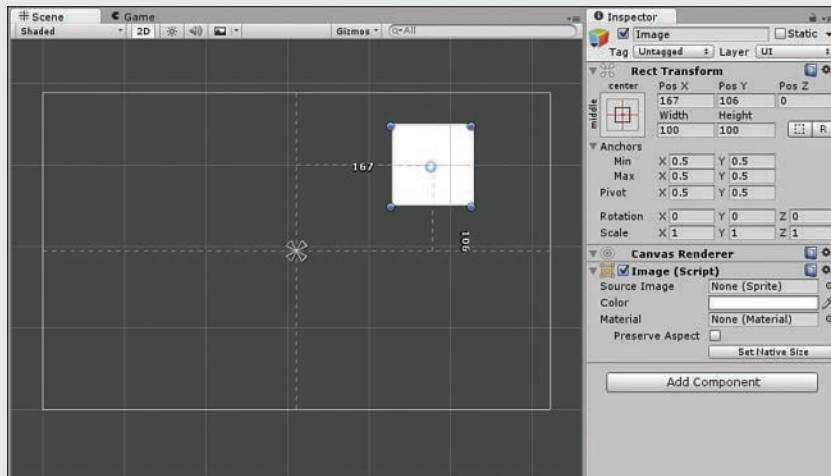


FIGURE 14.3

The anchor at a single point.

5. Now try splitting your anchor. You can do this by dragging any of the corners of the anchor away from the rest. With your anchor split, move your image around again. Notice how the properties of the rect transform changes (see Figure 14.4). Hint: where did Pos X, Pos Y, Width, and Height go?

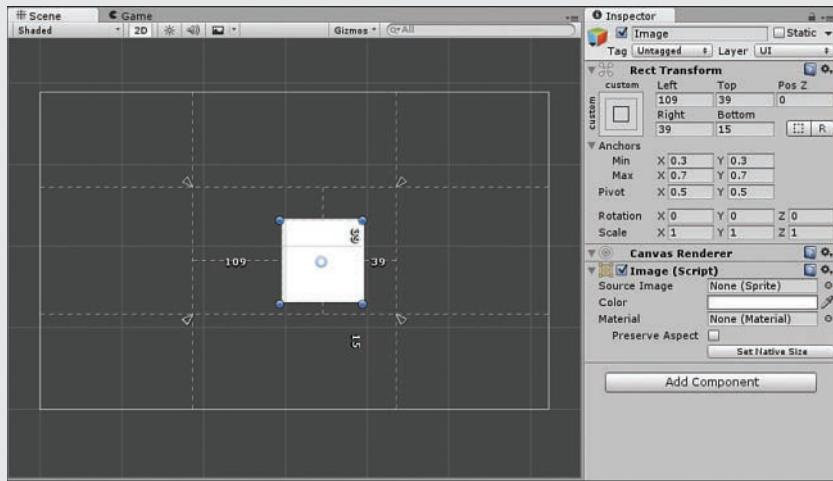
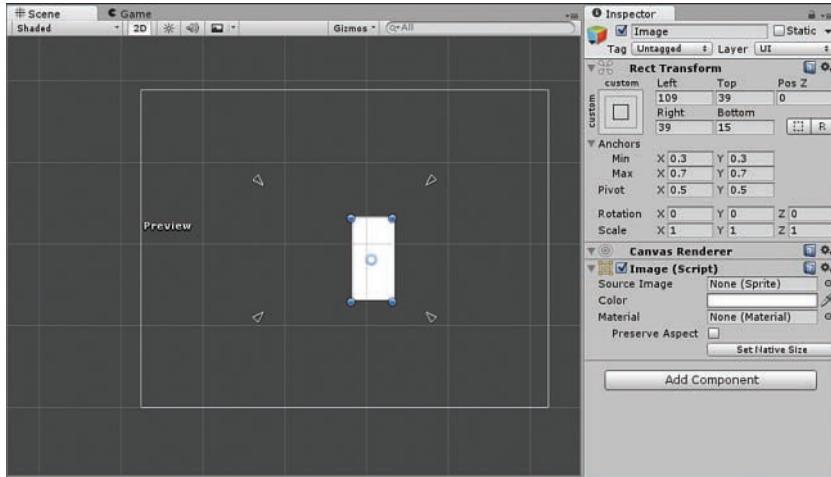


FIGURE 14.4

The anchor after being split.

So what exactly does splitting an anchor (or keeping it together) do? In simple terms, an anchor that is a single point will fix your UI element in place relative to that spot. So if the canvas changes size, the element won't. Splitting the anchor causes the element to fix its corners relative to the corners of the anchor. If the canvas resizes, so will the element. You can easily preview this behavior within the Unity editor. Using the above example, if you select the image and then click and drag the border of the canvas (or any other parent if you have more elements), the word "preview" will appear and you can see what will happen when using different resolutions (see Figure 14.5). Try it out with both a single and a split anchor. See how differently they behave.

**FIGURE 14.5**

Previewing a canvas change.

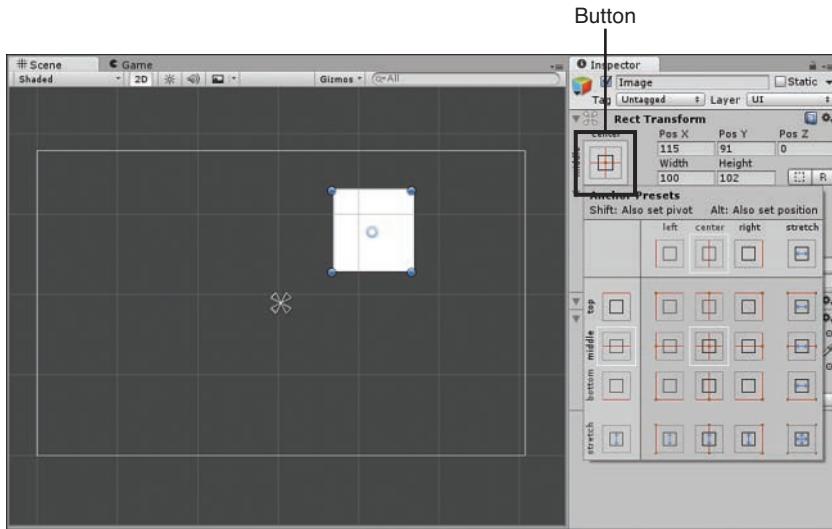
TIP**Getting Anchors Right**

Anchors may seem a little odd at first, but understanding them is the key to understanding the UI. Get the anchors and everything else just falls into place. A great thing to get into the habit of when working with UI elements is to always place the anchor, and then place the object (objects will snap to their anchors, but not vice versa).

Once you get into that habit (place anchor, place object, place anchor, place object, etc.), everything becomes a lot easier! Invest some time in playing with Anchors until you understand them.

TIP**The Anchor Button**

You don't always have to manually drag the anchors around your scene to place them. You can also type their values into the **Anchors** property in the inspector (a value of 1 is 100%, .5 is 50%, and so on). If even that is too much work for you, there is a convenient anchor button that enables you to place the anchor (and additionally the pivot and position) in one of 24 preset locations (see Figure 14.6). Sometimes it pays to be lazy!

**FIGURE 14.6**

The anchor button.

Additional Canvas Components

So far, we've talked about the canvas a bit, but still haven't even mentioned the actual canvas component! Truth be told, there isn't much with the component itself that we need to concern ourselves with. The important element is the **Render Mode**, and we will look at that in detail later.

Additionally, depending on what version of Unity you are using, you may have a couple additional components. Again, these are very simple to use and won't really be covered in detail here (there's just too much good stuff to get to). The **Canvas Scaler** allows you to specify how, if at all, you would like your UI elements resized as the screen of your target device changes (for instance, seeing the same UI on a web page versus a high DPI Retina iPad device). The **Graphical Raycaster** component works with the EventSystem and allows your UI to receive button clicks and screen touches. It exists to allow us raycast abilities without needing to drag in the entire physics engine to do it. You can safely leave this alone.

UI Elements

At this point, you're probably pretty tired of the canvas, so let's get working with some UI elements (also called UI controls). Unity has several built in controls available to you to get started with. Don't worry if you don't see a control that you want though. Unity's new UI library is open source, and plenty of custom controls are being created by members of the community all the time. As a matter of fact, if you're up for the challenge, you can even create your own controls and share them with everyone else!

Even though Unity has many controls that can be added to your scene, you will notice that most of them are simple combinations and variations of two basic elements: images and text. This makes sense if you think about it: a panel is just a full sized image, a button is just an image with some text, and a slider is really three images stacked together. As a matter of fact, the whole UI is built to be a system of basic building blocks that are stackable to get the functionality that you want.

Images

Images are the fundamental building piece of a UI. They can range from background images, to buttons, to logos, to health bars, to everything in between. If you completed the exercises above, then you already have a slight familiarity with images, but let's look a little closer. As we saw already, we can add an image to a canvas by clicking **GameObject > UI > Image**. Table 14.1 lists the properties of an image (which is just a game object with an **Image** component).

TABLE 14.1 Properties of the Image Component

Property	Description
Source Image	The image to be displayed. This must be a sprite (we looked at sprites in Hour 13).
Color	Any color tinting and opacity changes to be applied to the image.
Material	A material (if any) to be applied to the image.
Preserve Aspect	Whether or not the image will maintain its original aspect ratio regardless of scaling.
Set Native Size	A button that sets the size of the image object to the size of the image file being used.

Besides the basic properties, there isn't much more to using an image. It is as basic as they come!

TRY IT YOURSELF

Using an Image

Let's try creating a background image. This exercise uses the **BackgroundSpace.png** file in the book assets for Hour 14.

1. Create a new project.
2. Import **BackgroundSpace.png** into your project, ensuring that it is imported as a sprite (see Hour 13 if you don't remember how to do this).
3. Add an image to your scene (**GameObject > UI > Image**).

4. Set the `BackgroundSpace` sprite as the `Source Image` property of the `Image` object.
5. Resize the image to fill the entire canvas. Switch over to the game view and see what happens if you change the aspect ratio. Notice how the image may get cut off or fail to fill the screen.
6. Split the anchor of the image so that the four corners of the anchor reach the four corners of the canvas (see Figure 14.7). Now switch back to the game view and see what happens this time when you change the aspect ratio. Notice how the image always fills the screen and is never cut off, but it may skew.

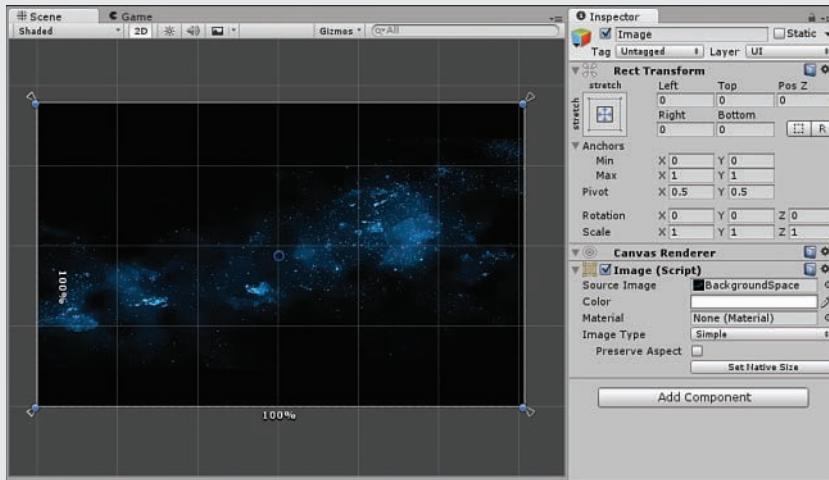


FIGURE 14.7
Expanding the image and anchors.

NOTE

UI Materials

We mentioned a material property for our image component above. It's worth noting that the material is completely optional and not required for our UI. Furthermore, in our current canvas mode (explained more later) the material property doesn't do a whole lot. In other modes, however, the material can allow us to apply lights and shader effects to UI elements.

Text

Text objects (which are really just text components) are the elements we use to display text to the user. If you've ever used text formatting controls before (think: blogging software, word processors like Word or WordPad, or anywhere you would use and style text), then the text component

will be very familiar. You can add a Text element to a canvas by clicking **GameObject > UI > Text**. Table 14.2 lists the properties of a Text element. Since most text properties are self-explanatory, only the new or unique properties are listed.

TABLE 14.2 Properties of the Text Component

Property	Description
Text	The text you'd like displayed.
Rich Text	Whether or not you'd like to support rich text tags within your text.
Horizontal and vertical Overflow	How to handle a situation where the text does not fit within the bounding box of the UI element that contains it. A value of Wrap means that text will wrap down to the next line. Truncate means that the text that doesn't fit will be removed. Overflow means that the text can spill out of the box. If text does not fit within the box and Overflow is not set, it may disappear.
Best Fit	An alternative to Overflow (and will not work if overflow is used). Best fit will resize your text to fit within the bounding box of the object. If selected, you will be able to choose a minimum and maximum size. The font will then expand or shrink between those two values to always fit within the text box.

It is worth taking a moment to try out the different settings for overflow and best fit. Otherwise, you may be surprised to find your text mysteriously disappears (has been truncated) and it may take time to figure out why!

Buttons

Buttons are elements that allow click input from the user. They may seem complex at first look, but remember (as mentioned above) that a button is really just an image with a text object child and a little more functionality. We can add text elements to our scene by clicking **GameObject > UI > Button**.

Where a button is different from either of the controls we've seen so far is that it is intractable. Because of that, it has some interesting properties and features. For instance, buttons can have transitions, be navigated, and have **On Click** event handlers! Table 14.3 lists the different Button component properties.

TABLE 14.3 Properties of the Button Component

Property	Description
Interactable	Can the user click the button?

Property	Description
Transition	How do you want the button to respond to user interaction (see Figure 14.8)? The available events to respond to are Normal (nothing happening), Highlighted (mouse over), Pressed, and Disabled. By default the button simply changes color (Color Tint). You could also remove any transitions (None), or cause the button to change images (Sprite Swap). Finally, you can choose Animation to use full animations to make your buttons look very impressive!
Navigation	This property allows you to specify how users will navigate between buttons if they are using devices like controllers or joysticks (no mouse or touch screen). Clicking Visualize allows you to see buttons will be navigated (requires more than one button on a canvas to work).
On Click()	What happens when you click the button (covered in more detail below).

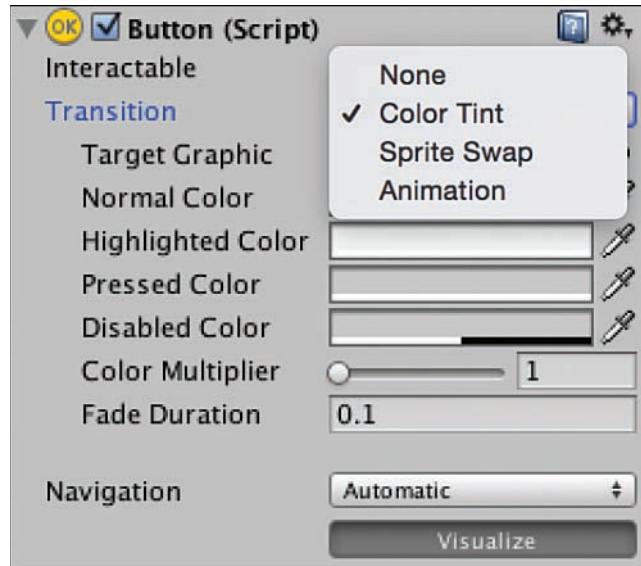


FIGURE 14.8

The transition type selector.

On Click ()

Once the user has enjoyed marveling at your various button transition effects, they may eventually click it! You can use the **On Click ()** property at the bottom of the inspector to call a function from a script, and to access many other components. You can pass parameters to any method you call, allowing the designer to control the behavior without entering code.

Some advanced uses of this functionality might be to call methods on a game object, make a camera look at a target directly, and much more.

TRY IT YOURSELF

Using A Button

Let's put all this to practice by creating a button, giving some color transitions, and making it change its text when clicked.

1. Create a new project.
2. Add a Button to the scene (click **GameObject > UI > Button**).
3. Under the Button (Script) component in the Inspector, set the Highlighted Color to red, and the Pressed Color to green (see Figure 14.9 for the finished settings).

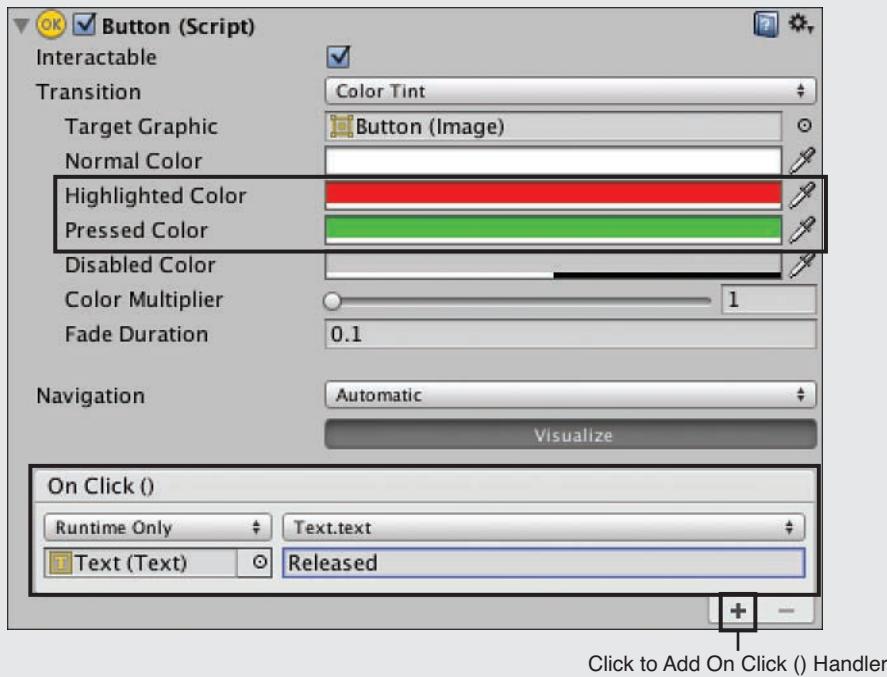


FIGURE 14.9

The finished button settings.

4. Add a new On Click () handler by clicking the little + sign at the bottom of the inspector (see Figure 14.9).
5. Now the handler is looking for the object to manipulate, it will say "None (Object)." Expand the button game object in the Hierarchy so you can see the Text child object. Drag the text onto the Object property of the event handler.

6. The function drop down (it says “No Function”) is where you specify what you want the button to do to the chosen object. Choose **Text.text** (1) by clicking the drop down and selecting **Text (2) > string text** (3) (see Figure 14.10).

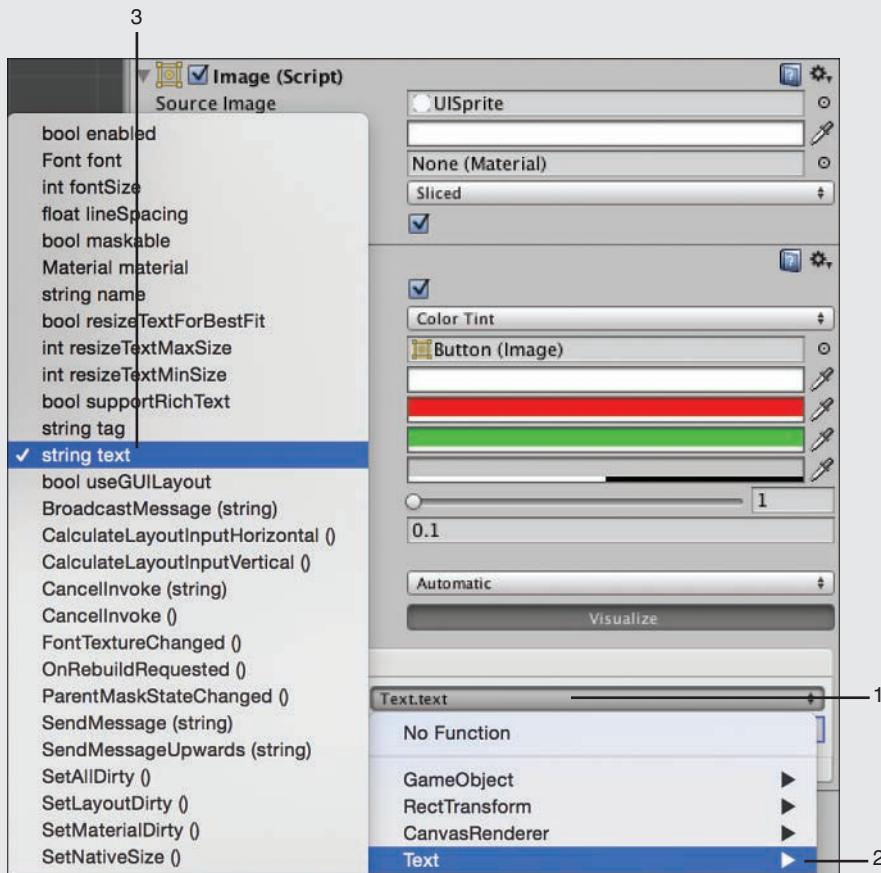


FIGURE 14.10

Setting the click even to change the button text.

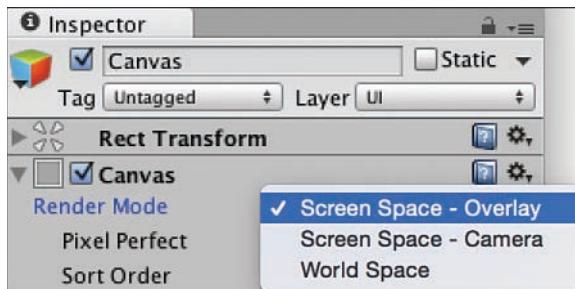
7. A value box will appear. Type “Released” into it.
8. Play the game and try hovering over the button, pressing and holding, then releasing the button. Notice the color changes and the text change when you click.

TIP**Sorting Elements**

Now that we are familiar with various elements, it is a good time to mention how they are drawn. We may have noticed that the canvas component we looked at above had a **Sorting Layer** property (like we saw with 2D images). This property is only used to sort between multiple canvases within the same scene. To sort UI elements on the same canvas we use the order of the objects in the Hierarchy. Therefore, if you want an object drawn on top of another object, you would move it lower in the Hierarchy so it is drawn later.

Canvas Render Modes

Unity 5 offers three powerful options for the way your UI is rendered to the screen. You can choose the mode in the inspector by selecting the Canvas, and choosing **Render Mode** (see Figure 14.11). The use of each type of canvas is very complex, so we aren't trying to master them right now. Instead, the aim here is to outline them so that you can choose what is right for your game(s).

**FIGURE 14.11**

The three different canvas render modes.

Screen-Space Overlay

This is the easiest to use and also least powerful version of the canvas modes (it is also the default). A screen-space overlay UI draws over the top of everything on the screen, regardless of the camera settings, or the position of the camera in the world. In fact, where this UI appears in the Scene view bears no relation to the objects in the world because it isn't actually rendered by a camera.

The UI will show in the scene view at a fixed position, with the bottom-left at (0, 0, 0) in the world. The scale of the UI will be different to the world scale, and what you see on the canvas will be at a scale of 1 world unity for every pixel in your game view. If you are using this type of UI in your game, and find its place in the world to be inconvenient while working you can hide

it to get it out of the way. This can be done by clicking the **Layers** drop down in the editor and hiding the “eye” icon next to the UI layer (see Figure 14.12). This hides the UI in the scene view only (it will still be there when you run your game). Just be sure to not forget to turn it back on or else you might get confused as to why your UI won’t show up!



FIGURE 14.12
Hiding your UI.

Screen-Space Camera

This mode is similar to screen-space overlay, but the UI is rendered by a camera of your choice. You can rotate and scale UI elements to create much more dynamic, 3D interfaces.

Unlike the screen-space overlay mode, this mode uses the camera to render the UI. This means effects such as lighting affect the UI, and objects can even pass between the camera and UI. This can take some extra work, but the payoff is your interface can feel much more part of the world.

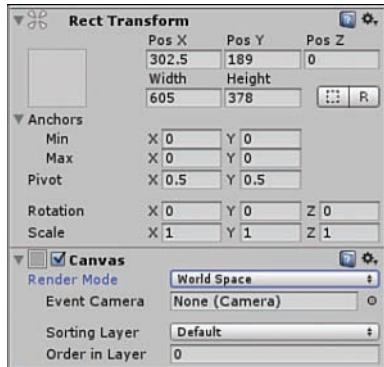
Note that in this mode the UI stays in a fixed position relative to the camera you chose to render it. Moving the camera also moves the canvas. It can be a good idea to use a second camera solely for rendering your canvas (so it isn’t in the way of the rest of your scene).

World Space

The final type of UI to consider is world-space UI. Imagine a virtual museum where every object you look at had detailed information about the object appear in the world right next to it.

Furthermore, this pop-up information could include buttons allowing you to read more or go to other sections of the museum. If you can imagine that, then you are only scratching the surface of what you can do with a world space canvas.

You will notice that the Rect Transform of the canvas in world space mode is no longer grayed-out, and “the canvas itself” can be edited and resized (see Figure 14.13). Since the canvas is actually a game object in the world (in this mode), it no longer is drawn over the rest of your game like an HUD. Instead, it fixed in the world and can be a part of, or blend it with, the rest of your scene objects!

**FIGURE 14.13**

The Rect Transform is available with World Space UI.

TRY IT YOURSELF

Explore the Render Modes

We will briefly look at the three different UI render modes to see how this setting works.

1. Create a new 3D project.
2. Add a UI Canvas to the scene (Click **GameObject > UI > Canvas**).
3. Note the Rect Transform is disabled. Also try zooming in the Scene view to see where the canvas sits.
4. Switch the render mode to Screen-Space Camera. Under “Render Camera” choose the Main Camera. Notice the canvas change size and position when you do this.
5. Notice what happens when you move the camera. Also notice what happens when you change the **Projection** of the camera from **Perspective** to **Orthographic** and back.
6. Switch to World Space UI. Note that you can now change the Rect Transform for the canvas, move it, rotate it, and scale it.

Summary

You started this chapter by understanding the building blocks of any UI, the Canvas and Event System. We went on to look at the Rect Transform, and how Anchor points help you make versatile UIs that can work on many devices. From there, we explored various UI elements available to use. We then briefly looked at the fundamentally different UI types of screen-space overlay, screen-space camera and world space UI.

Q&A

Q. Does every game need a user interface?

A. Usually, a game benefits from having a well-thought-out UI. It is rare for a game to have no UI whatsoever. That said, it is always a good idea to be minimalist with a UI, giving the player just the information they need—when they need it.

Q. Can I mix canvas render modes in a scene?

A. Yes, you can; you may well want to have more than one canvas in your scene with different render modes.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What does UI stand for?
2. What two game objects always come along with a UI in Unity 5?
3. What type of UI would you use to put a question mark above a player's head in a 3D game?
4. What render mode is most likely to be best for a simple Heads Up Display (HUD)?

Answers

1. User Interface (an easy warm-up question!).
2. A Canvas and an EventSystem.
3. World-space UI, because the interface element is positioned within the world space, not relative to the player's eyes.
4. Screen-Space Overlay.

Exercise

In this exercise, we will build a simple but complete menu system that you can adapt for use in all your games. We will make use of a splash-screen, a fade-in, some background music, and more.

1. Create a new 2D project. Add a UI Panel (click **GameObject > UI > Panel**). Note how this also adds the required Canvas and Event System to the Hierarchy for you.

2. Import the **Hour 14 Package** provided in the book files (double click the Hour 14 Package.unitypackage in your file browser). Click **clouds.jpg** in your Assets, ensure the Texture Type is set to **Sprite (2D and UI)** as per the previous hour.
3. Set this image as the Source Image in the Panel's inspector. Note how the image is a little transparent by default, letting the Main Camera's background color show through. Feel free to adjust the transparency by bringing up the color dialog, then adjusting the A slider, where A stands for Alpha.
4. Add a title and a subtitle (click **GameObject > UI > Text**). Move these so they sit nicely on the panel (see Figure 14.14).

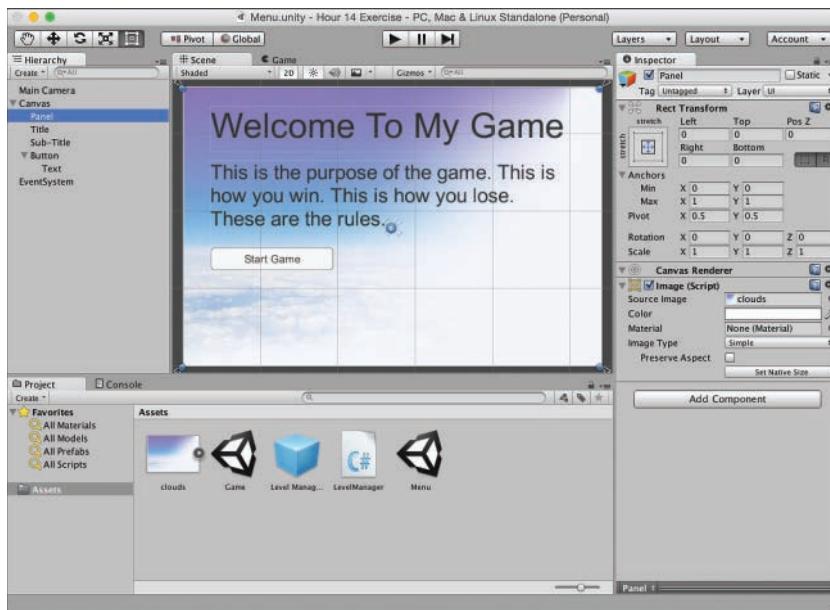


FIGURE 14.14
The complete UI.

5. Add a button (click **GameObject > UI > Button**). Name the button Start, and set its Text child-object to say “Start Game.” Position the button where you wish, remembering to select the button (not the Text child) before you drag.
6. Save your scene as “Menu” (click **File > Save Scene**). Now create a new scene and save it as “Game” to act as a placeholder for your game. Finally add both scenes to the Build Order by opening the Build Settings (click **File > Build Settings**) and dragging both scenes into the **Scene In Build** section. Ensure the Menu scene is on top.
7. Switch back to your menu scene. Drag the LevelManager prefab that you imported from your Assets into the Hierarchy.

8. Find the Start button, and set its On Click property to the Level Manager's LoadGame() method (see Figure 14.15).

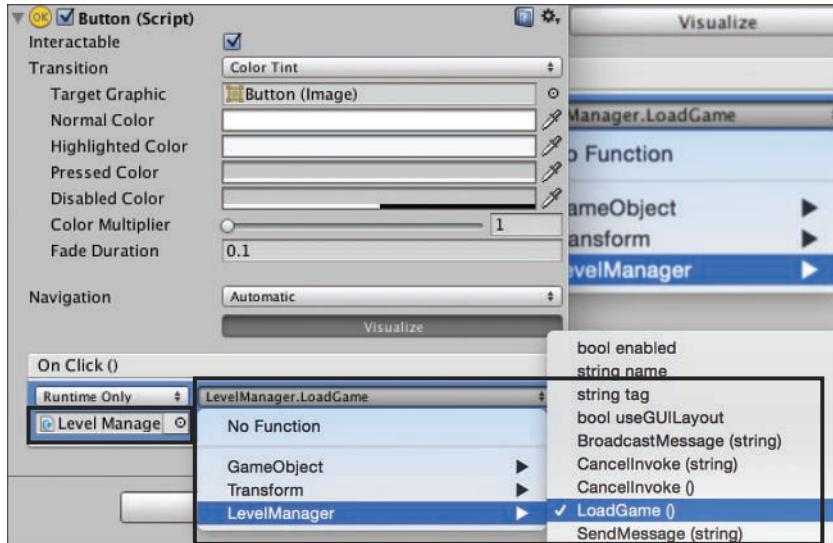


FIGURE 14.15

Setting the Start Game button's On Click() handler.

9. Play your scene. Click the Start Game button and the game should switch to the empty Game scene. Congratulations, you have a menu system to use in future games!

This page intentionally left blank

HOUR 15

Game 3: *Captain Blaster*

What You'll Learn in This Hour:

- ▶ How to design the game *Captain Blaster*
- ▶ How to build the *Captain Blaster* world
- ▶ How to build the *Captain Blaster* entities
- ▶ How to build the *Captain Blaster* controls
- ▶ How to further improve *Captain Blaster*

Let's make a game! In this hour, you make a 2D scrolling shooter game titled *Captain Blaster*. You start by designing the various elements of the game. From there, you begin building the scrolling background. Once the idea of motion is established, you begin building the various game entities. After the entities are done, you construct the controls and gamify the project. You finish the chapter by analyzing the game and identifying places for improvement.

TIP

Completed Project

Be sure to follow along in this hour to build the complete game project. In case you get stuck, you can find a completed copy of the game in the book assets for Hour 15. Take a look at it if you need help or inspiration!

Design

You have already learned what the design elements are in Hour 7, "Game 1: *Amazing Racer*." This time, you get right into them.

The Concept

As mentioned earlier, *Captain Blaster* is a 2D scrolling shooter style game. The premise is that the player will be flying around a level, destroying meteors and trying to stay alive. The neat

thing about 2D scrolling games is that the players themselves don't actually have to move at all. The scrolling background simulates the idea that the player is going forward. This reduces the required player skill and allows you to create more challenges in the form of enemies.

The Rules

The rules of this game state how to play, but also allude to some of the properties of the objects. The rules for *Captain Blaster* are as follows:

- ▶ Players play until they are hit by a meteor. There is no win condition.
- ▶ The player can fire bullets to destroy the meteors. The player earns 1 point per meteor destroyed.
- ▶ Players can fire two bullets per second.
- ▶ The player is bounded by the sides of the screen.
- ▶ Meteors will come continuously faster until the player loses.

The Requirements

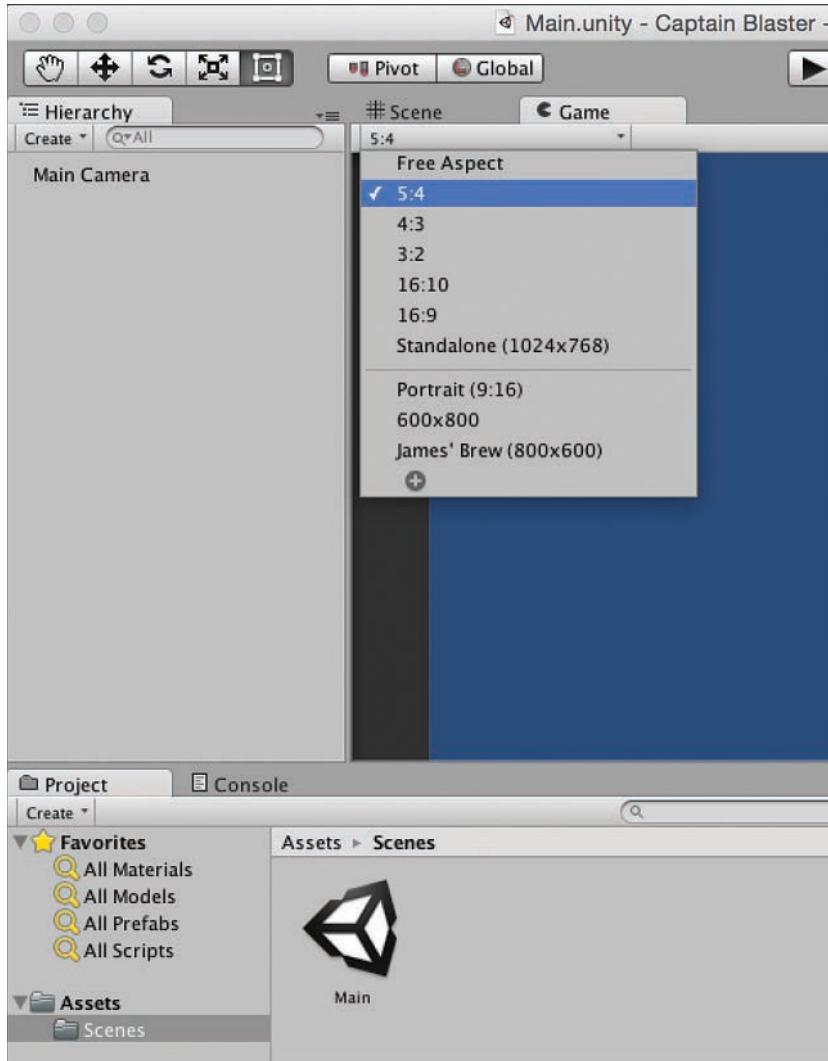
The requirements for this game are simple and are as follows:

- ▶ A background texture to be outer space.
- ▶ A ship model and texture.
- ▶ A meteor model and texture.
- ▶ A game controller. This will be created in Unity.
- ▶ Interactive scripts. These will be written in MonoDevelop as usual.

The World

Because this game takes place in space, the world will be fairly simple to implement. The idea is that the game will be 2D and the background will move vertically behind the player to make it seem like the player is moving forward. In actuality, the player will be stationary. Before you get the scrolling in place, though, you need to set your project up. Start with these steps:

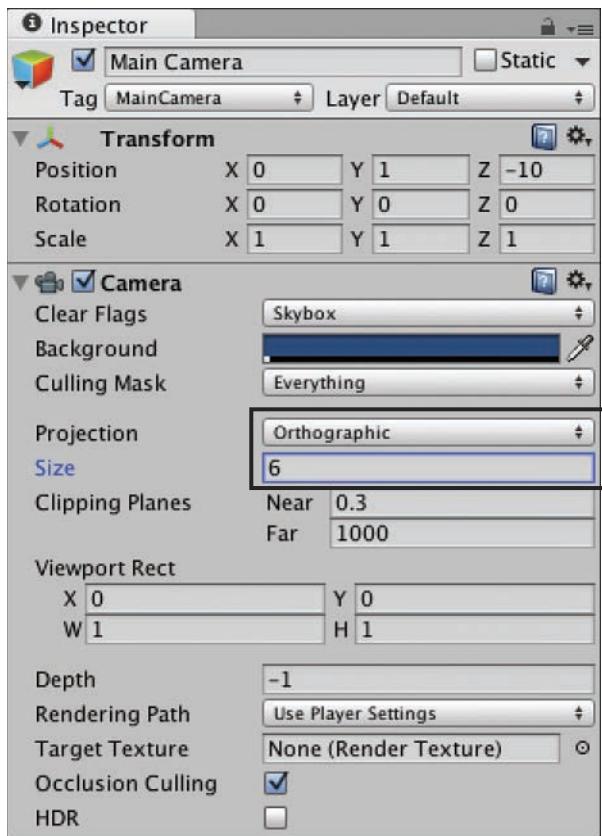
1. Create a new 2D project in a folder named **Captain Blaster**.
2. Create a **Scenes** folder and save your scene as **Main**.
3. In the Game view, change the aspect ratio to **5:4** (see Figure 15.1).

**FIGURE 15.1**

Setting the game aspect ratio.

The Camera

Now that the scene is set up properly, it is time to work on the camera. As you chose a 2D project, you have an orthographic camera. This camera lacks depth perspective and is great for making 2D games. To set up the Main Camera simply set the Size property to 6. (See Figure 15.2 for a list of the camera's properties.)

**FIGURE 15.2**

The Main Camera properties.

The Background

The scrolling background can be a little tricky to get set up correctly. Basically, you have two background objects moving down the screen. As soon as the bottom object goes off screen, you place it above the screen. You keep flipping back and forth between them and the player never knows. To create the scrolling background, follow these steps:

1. Create a new folder named **Background**. Locate the **Star_Sky.png** image from the book files and import it into Unity by dragging it into the **Background** folder you just created. Remember that since you made a 2D project, it automatically imports as a sprite.
2. Select the newly imported sprite in the Project view and change its **Pixels Per Unit** property in the Inspector view to 50. Drag your **Star_Sky** sprite into the scene and ensure that it is positioned as (0, 0, 0).

3. Create a new script in your **Background** folder named **ScrollBackground** and drag it onto the background sprite in the scene. Put the following code in the script:

```
public float speed = -2f;

// Update is called once per frame
void Update () {
    transform.Translate(0f, speed * Time.deltaTime, 0f);

    if(transform.position.y <= -20){
        transform.Translate(0f, 40f, 0f);
    }
}
```

4. Duplicate the background sprite and place it at (0, 20, 0). Run the scene. You should notice the background seamlessly stream by.

NOTE

Alternate Organization

Up until now, we have used a fairly straightforward system for organization. Assets went into corresponding folders (sprites in a sprites folder, scripts in a scripts folder, etc.). In this chapter, however, we are going to do something new. This time, we are going to group assets based on their “entity” (all background files together, ship assets together, etc.). This system works well if you find yourself wanting to find all related assets quickly. Don’t worry if you like the old system though. You can still search and sort asset names and types using the search bar and filter properties located at the top of the project view. As told to me recently by Ben Tristem (Howdy, Ben!), “There’s more than one way to do things!”

NOTE

Seamless Scrolling

You might notice a small line in the previous scrolling background. This is due to the fact that the image used for the background wasn’t made specifically to tile together. Generally, this isn’t very noticeable, and the actions of the game will more than cover up for it. If you want a more seamless background in the future, however, you want to use an image made to tile together.

Game Entities

In this game, you need to make three primary entities: the player, the meteor, and the bullet. The interaction between these items is also very simple. The player fires bullets. Bullets destroy meteors. Meteors destroy the player. Because the player can technically fire a large number of bullets, and because a large number of meteors can spawn, you need a way to clean them up. Therefore, you also need to make triggers that destroy bullets and meteors that enter them.

The Player

Your player will be a spaceship. The sprites for both the spaceship and the meteors can be found in the book assets for Hour 15 thanks to Krasi Wasilev (<http://freegameassets.blogspot.com>). To create the player, follow these steps:

1. Create a new folder in your Assets called Spaceship and import **spaceship.png** from the book files into this folder. Note it is currently facing downwards. This is ok.
2. Select the spaceship sprite and in the Inspector set Sprite Mode to Multiple and click Apply. Then click Sprite Editor to start slicing your sprite sheet (if you forgot about slicing, look back at Hour 13).
3. Click Slice in the upper left of the Sprite Editor Window, and set Type to Grid. Set x = 116 and y = 140 (see Figure 15.3). Click Slice, and notice the outlines around the ships. Click Apply, and close the Sprite Editor window.

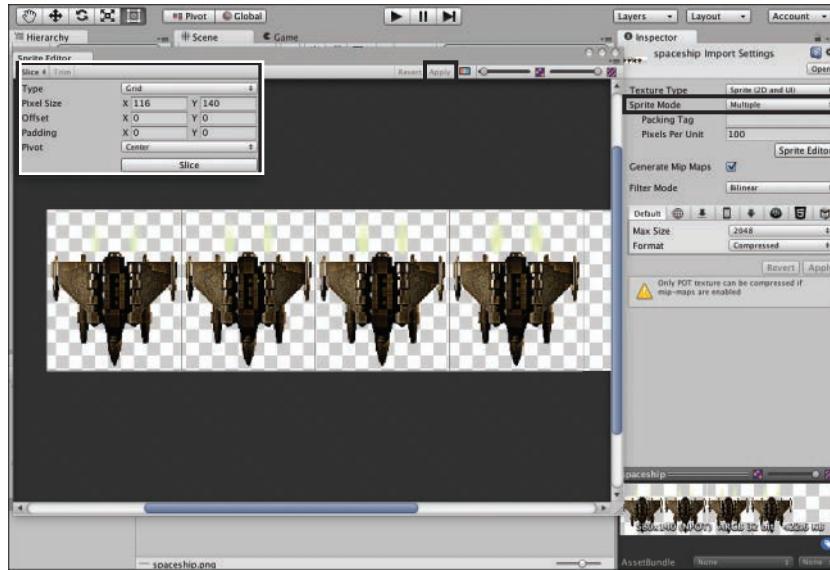
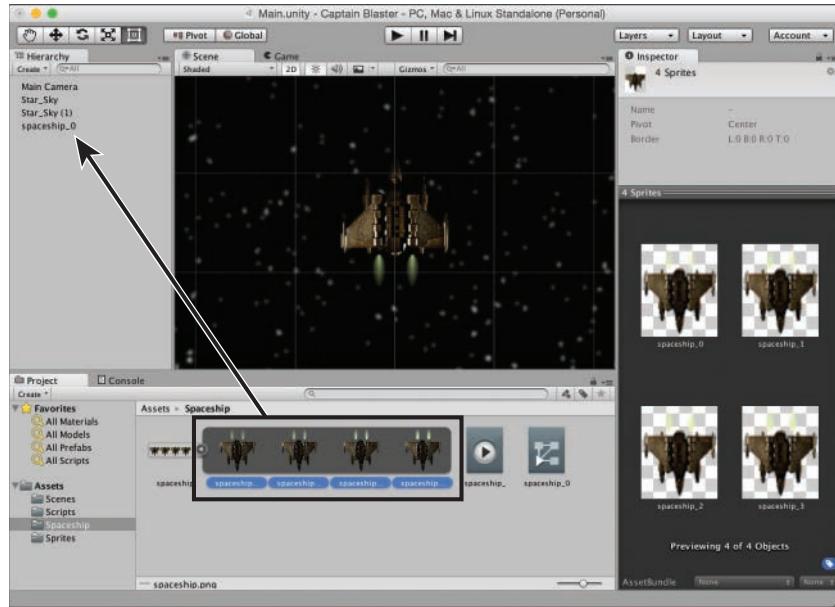


FIGURE 15.3

Slicing the spaceship sprite sheet.

4. Now open the tray of the spaceship and select all the frames. You can do this by clicking the first frame, holding down Shift, and then clicking the last frame (see Figure 15.4).

**FIGURE 15.4**

The finished spaceship sprite.

5. Drag the sprite frames to the Hierarchy. This will create an animated sprite automatically creates and controller, and animation clip in the same folder. We will learn more about animation in Hour 18, “Animation.”
6. Set the ship’s position to $(0, -5, 0)$, and scale it to $(1, -1, 1)$. Note that scaling to -1 in the y axis turns the ship to face upward.
7. Play the game, and notice the subtle animation of the ship’s engines.
8. Finish by adding a Polygon Collider to the ship (**Add Component > Physics2D > Polygon Collider**). This collider will automatically surround your ship for very decent collision detection accuracy. Be sure to check the **Is Trigger** property in the Inspector to ensure that it is a trigger collider.
9. Play the game, and notice the subtle animation of the ship’s engines.

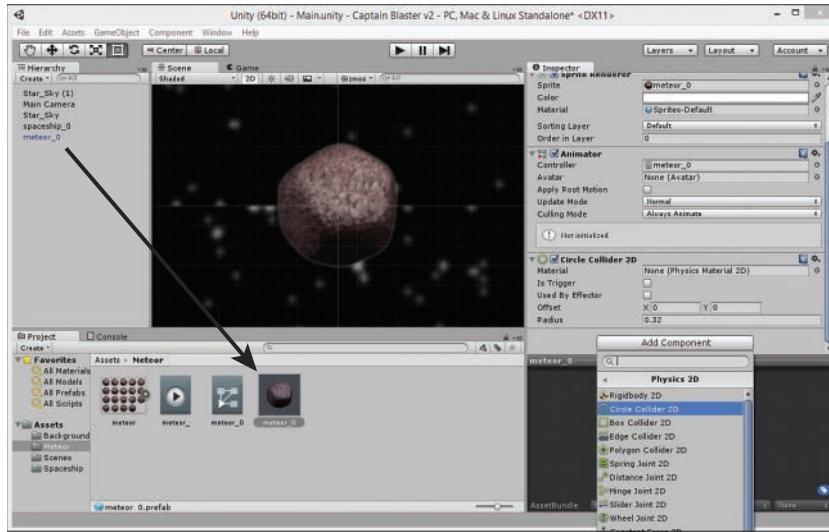
You should now have a nice, animated upward-facing spaceship ready to destroy some meteors!

The Meteors

The steps for the meteors are similar to those of the spaceship. The only difference is that the meteors will end up in a prefab for later use:

1. Create a new folder called Meteor, and import **meteor.png** into it. This is a sprite sheet containing 19 frames of animation.
2. Set the Sprite Mode to **Multiple**, and then enter the **Sprite Editor** as before.
3. This time set the Slice Type to Automatic, leaving the rest of the settings default (center pivot, delete existing). Note the boxes around each frame. Apply your changes, and close the Sprite Editor window.
4. Select all 19 frames of the meteor animation (note they are numbered starting at 0). Drag these frames into the Hierarchy. Unity will create another animated sprite, with the necessary animation components for you . . . handy hey! Remember, you may be prompted to save the assets. Be sure to do so.
5. Double-click **meteor_0** in the Hierarchy to focus your Scene view on it. Add a Circle Collider 2D component from the Inspector (**Add Component > Physics2D > Circle Collider 2D**). Note the green outline roughly fits the outline of the sprite. This is good enough to work in our game.
6. Add a **Rigidbody2D** to the meteor (**Add Component > Physics2D > Rigidbody2D**). Set the **Gravity Scale** property to 0.
7. Drag the meteor from your Hierarchy, into your **Meteor** folder in the Project view (see Figure 15.5). This will create a prefab of the meteor for later use.
8. Now that you have captured the meteor setup in a prefab, delete the instance in your Hierarchy.

You now have a reusable meteor just waiting to cause havoc.

**FIGURE 15.5**

Creating the meteor prefab.

The Bullets

Bullets will be simple in this game. Because they will be moving very quickly, they won't need any detail. To create the bullet, follow these steps:

1. Create a folder named **Bullet** and import **bullet.png** into it. With the new bullet sprite selected in the Project view, set the **Pixels Per Unit** property in the Inspector to 400.
2. Drag a bullet sprite into your scene. Using the **Color** property of the **Sprite Renderer** component, give the bullet a strong green color.
3. Add a **Circle Collider 2D** component from the Inspector (**Add Component > Physics2D > Circle Collider 2D**). Additionally, add a **Rigidbody2D** to the bullet (**Add Component > Physics2D > Rigidbody2D**) and set the **Gravity Scale** property to 0.
4. Drag the bullet from the Hierarchy view into your **Bullet** folder to turn it into a prefab. Delete the bullet object from your scene.

That's the last of the primary entities. The only thing left to make is the triggers that will prevent the bullets and meteor from traveling forever.

The Triggers

The triggers (which we will call “Shredders”) are simply two colliders that will sit above and below the screen. Their job is to catch any errant bullets and meteors, and “shred” them.

1. Add an empty game object to the scene (**GameObject > Create Empty**) to the scene and name it **Shredder**. Position it at (0, -10, 0).
2. Add a Box Collider 2D to the shredder object (**Add Component > Physics2D > Box Collider 2D**). In the Inspector view, be sure to put a check in the **Is Trigger** property of the Box Collider 2D component and set its size to (16, 1).
3. Duplicate the shredder and place the new one at (0, 10, 0). Remember you can drag this new shredder to the bottom of the hierarchy to keep things tidy.

Later these triggers will be used to destroy any objects that hit them, such as stray meteors or bullets.

The UI

Finally we will add a simple User Interface to display the player’s current score and to say “Game Over” when the player dies.

1. Add a UI Text element to the scene (**GameObject > UI > Text**) and rename it **Score**.
2. Position the score text’s anchor in the upper left corner of the canvas and set its position to (100, -30, 0) (see Figure 15.6).

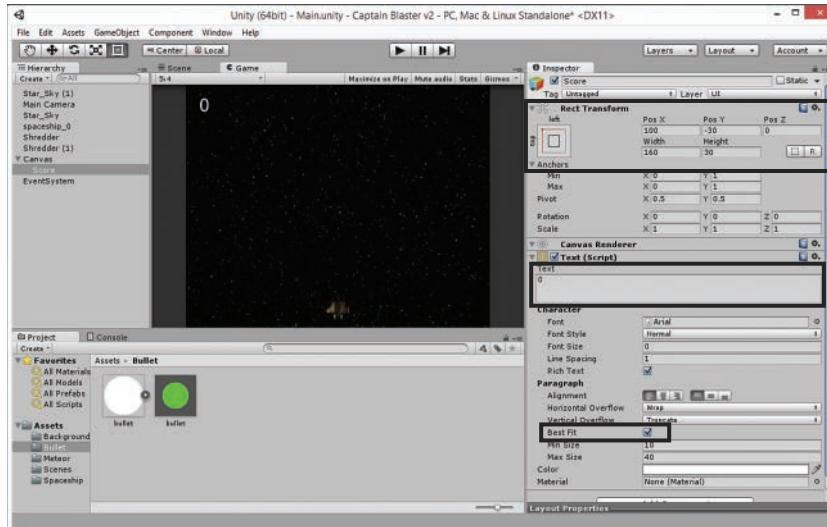


FIGURE 15.6
The player score settings.

- Set the Text property of the Score object to 0 (which is the initial score), check **Best Fit**, and set the **Color** to white.

Later we will connect this score display to our GameControl script so that it can be updated. Now all of your entities are in place and it is time to begin turning this scene into a game.

Now we will add the Game Over text.

- Add another UI Text element to the scene and rename it **Game Over**. The text should appear in the center of the canvas. If it doesn't, go ahead and place it in the center.
- Set the width to 200, and the height of the Game Over text to 100.
- Change the Text to "Game Over!", check the **Best Fit** property, set the paragraph alignment to centered, and change the **Color** to red.
- Finally, uncheck the box next to **Text(Script)** to disable the text until it is needed (see Figure 15.7). Note that Figure 15.7 illustrates the text before being disabled so you can see what it looks like.

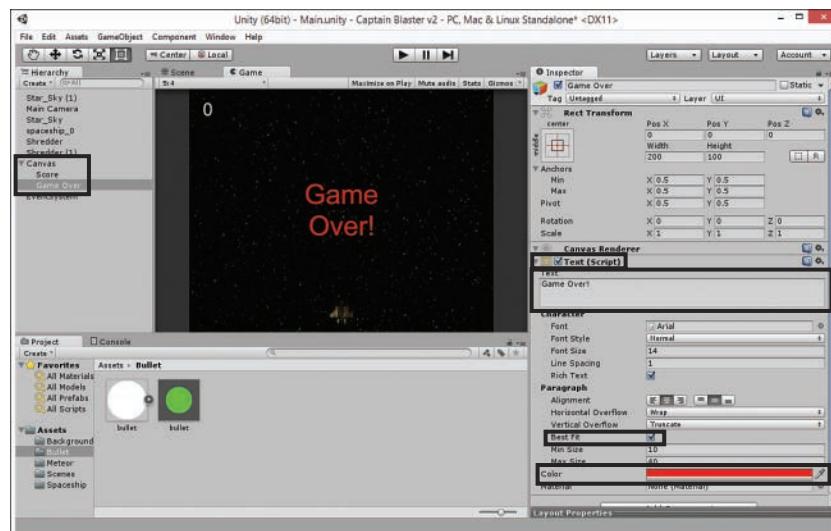


FIGURE 15.7

The game over sign settings.

Controls

Various script components need to be assembled to make this game work. The player needs to be able to move the ship and shoot bullets. The bullets and meteors need to be able to move automatically. A meteor spawn object will keep the meteors flowing. The shredders will need to be able to clean up objects, and a control will need to keep track of all the action.

The Game Control

The game control is basic in this game, so you add that first. Create an empty game object and name it **GameControl**. For tidiness set its position to (0, 0, 0). Since the only asset we will have for this object will be a script, create a new folder called **Scripts** so that we have a place for any simple scripts we create. Create a new script in our scripts folder called **GameControl** and attach it to the game control object. Overwrite the contents of the script with the following code:

```
using UnityEngine;
using UnityEngine.UI; // Note this new line is needed for UI
using System.Collections;

public class GameControl : MonoBehaviour {

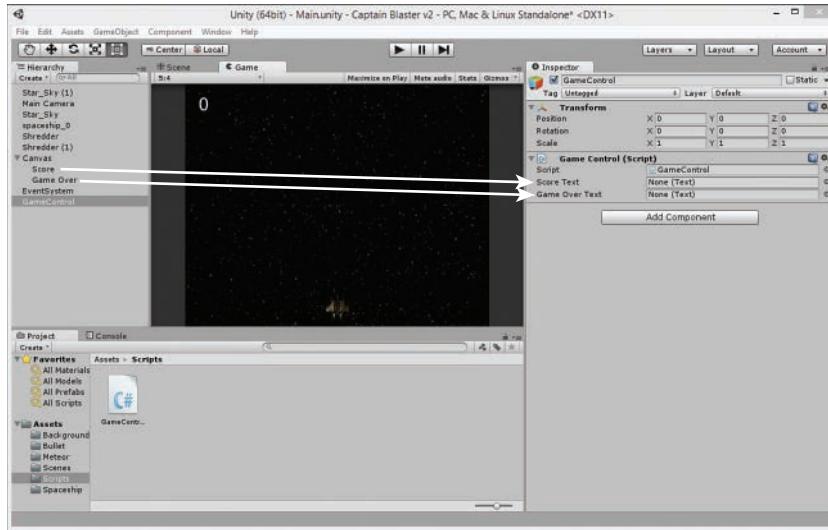
    public Text scoreText, gameOverText; // Note we declare two text elements here
    int playerScore = 0;

    public void AddScore () {
        playerScore++;
        scoreText.text = playerScore.ToString();
    }

    public void PlayerDied () {
        gameOverText.enabled = true; // Display the Game Over! Text
        Time.timeScale = 0; // This freezes the game
    }
}
```

In this code, you can see that the control is responsible keeping the score, and knowing when the game is running. The control has two public functions: `PlayerDied()` and `AddScore()`. `PlayerDied()` is called by the player when a meteor hits it. `AddScore()` is called by a bullet when it kills a meteor.

Remember to drag the Score and Game Over elements onto the **GameControl** script (see Figure 15.8).

**FIGURE 15.8**

Attaching the text elements to GameControl.

The Meteor Script

Meteors are basically going to fall from the top of the screen and get in the player's way. Create a new script in your Meteor folder and call it **MeteorMover**.

Select your **Meteor** prefab. In the Inspector view, locate the Add Component button (see Figure 15.9). Click **Add Component > Scripts > MeteorMover**.

Overwrite the code in the MeteorMover script with the following:

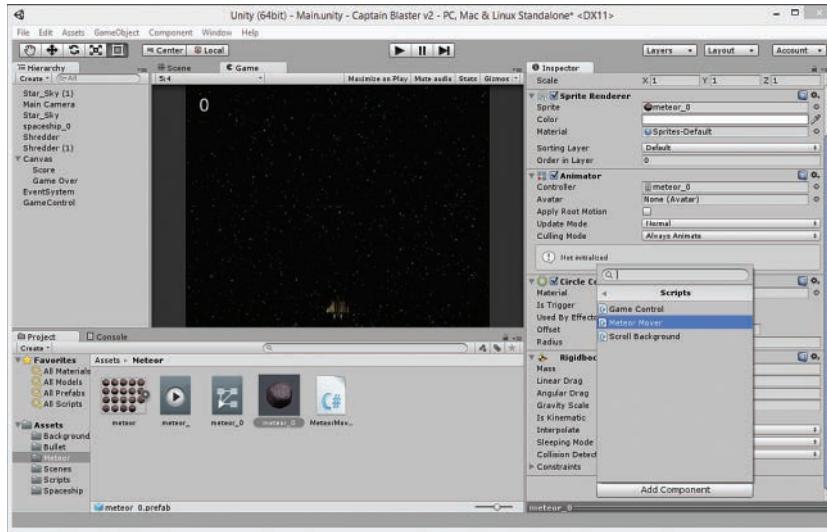
```
using UnityEngine;
using System.Collections;

public class MeteorMover : MonoBehaviour {

    public float speed = -2f;

    private Rigidbody2D rigidBody;

    // Use this for initialization
    void Start () {
        rigidBody = GetComponent<Rigidbody2D> ();
        //Give meteor an initial downward velocity
        rigidBody.velocity = new Vector2 (0, speed);
    }
}
```

**FIGURE 15.9**

Adding the Meteor script to the Meteor prefab.

The meteor is very basic and only contains a single public variable to represent its downward speed. In the `Start()` method, we get a reference to the `Rigidbody2D` component on the meteor. We then use that `Rigidbody` to set the velocity of the meteor (it moves downward because speed has a negative value). Notice that the meteor is not responsible for determining collision.

Feel free to drag the Meteor prefab into the Hierarchy and click Play so you can see the effect of the code you just wrote. The meteor should slide down, and spin slightly. Delete the Meteor instance from the Hierarchy when you're done.

The Meteor Spawn

So far, the meteors are just prefabs with no way of getting into the scene. What you need is an object responsible for spawning the meteors at an interval. Create a new empty game object. Rename the game object **MeteorSpawn** and place it at (0, 8, 0). Create a new script named **MeteorSpawn** in your Meteor folder and place it on the meteor spawn object. Overwrite the code in the script with the following:

```
using UnityEngine;
using System.Collections;

public class MeteorSpawn : MonoBehaviour {

    public float minSpawnDelay = 1f;
    public float maxSpawnDelay = 3f;
    public GameObject meteorPrefab;
```

```

void Start () {
    Spawn ();
}

void Spawn () {
    // Create a meteor at a random x position
    Vector3 spawnPos = transform.position + new Vector3(Random.Range(-6, 6), 0, 0);
    Instantiate (meteorPrefab, spawnPos, Quaternion.identity);

    Invoke ("Spawn", Random.Range (minSpawnDelay, maxSpawnDelay));
}
}

```

This script is doing a few interesting things. The first thing is that it is creating two variables to manage the meteor timing. It also declares a `GameObject` variable, which will be the meteor prefab. In the `Start()` method we call the function `Spawn()`. This function is responsible for creating and placing the meteors.

You can see that the meteor is spawned at the same y and z coordinate as the spawn point, but the x coordinate is offset by a number between -6 and 6. This is to allow the meteors to spawn across the screen and not always in the same spot. Once the position for the new meteor is determined, the `Spawn()` function instantiates (creates) a meteor at that position with no rotation (`Quaternion.identity`). The last line “invokes” a call to the `spawn` function again. This method, `Invoke()`, will call the named function (in this case `Spawn()`) after a random amount of time. That random amount is controlled by our two timing variables.

In the Unity editor, click and drag your meteor prefab from the Project view onto the **Meteor Prefab** property of the Meteor Spawn Script component of the meteor spawn object (try saying that fast!). Run the scene and you should notice meteors spawning across the screen (Attack, my minions!).

The DestroyOnTrigger Script

Now that you have meteors spawning everywhere, it is a good idea to begin cleaning them up. Create a new script called **DestroyOnTrigger** in your Scripts folder (since it is a single unrelated asset) and attach it to both the upper and lower shredder objects you created previously. Add the following code to the script. Ensure that the code is outside of a method but inside of the class:

```

void OnTriggerEnter2D(Collider2D other) {
    Destroy(other.gameObject);
}

```

This basic script simply destroys any object that enters it. Because the player cannot move vertically, you don't need to worry about them getting destroyed. Only bullets and meteors can enter the triggers.

The ShipControl Script

Right now, meteors are falling down and the player can't get out of the way. You need to create a script to control the player next. Create a new script called **ShipControl** in your SpaceShip folder and attach it to the spaceship object in your scene. Replace the code in the script with the following:

```
using UnityEngine;
using System.Collections;

public class ShipControl : MonoBehaviour {

    public float playerSpeed = 10f;
    public GameController gameController;
    public GameObject bulletPrefab;
    public float reloadTime = 0.5f; // Player can fire a bullet every half second

    private float elapsedTime = 0;

    void Update () {
        elapsedTime += Time.deltaTime; // Keeping track of time for bullet firing

        // Move the player left and right
        float xMovement = Input.GetAxis ("Horizontal") * playerSpeed * Time.deltaTime;
        float xPosition = Mathf.Clamp (xMovement, -7f, 7f); // Keep ship on screen
        transform.Translate (xPosition, 0f, 0f);

        // Spacebar fires. The default InputManager settings call this "Jump"
        // Only happens if enough time has elapsed since last firing.
        if (Input.GetButtonDown("Jump") && elapsedTime > reloadTime) {

            // Instantiate the bullet 1.2 units in front of the player
            // and in the foreground at z=-5
            Vector3 spawnPos = transform.position;
            spawnPos += new Vector3(0, 1.2f, 0);
            Instantiate(bulletPrefab, spawnPos , Quaternion.identity);

            elapsedTime = 0f; // Reset bullet firing timer
        }
    }

    // If a meteor hits the player
    void OnTriggerEnter2D (Collider2D other)
    {
        gameController.PlayerDied();
    }
}
```

A lot of work is done in this script. It starts by creating variables for the speed, the bullet prefabs, the GameControl script, and bullet timing.

In the `Update()` method, the script starts by getting the current time. This is used to determine whether enough time has passed to fire a bullet. If you remember the rules, the player can only fire a bullet every half second. The player is then moved along the x axis based on input. After that, the script determines if the player is pressing the spacebar. Normally in Unity, the spacebar is considered a jump action. This could be renamed in the Input Manager, but it was left as it is to avoid any confusion. If it is determined that the player is pressing the spacebar, the script checks the elapsed time against the `reloadTime` (currently half a second). If the time is greater, the script creates a bullet. Notice that the script creates the bullet just a little above the ship. This is to prevent the bullet from colliding with the ship. Finally, the elapsed time is reset to 0 so the count for the next bullet firing can start.

The last part of the script contains the `OnTriggerEnter2D()` method. This gets called whenever a meteor hits the player. When that happens, the control script is informed that the player died.

Back in the Unity editor, click and drag the bullet prefab onto the Bullet property of the player script. Likewise, click and drag the game control object onto the player script to give it access to the control script (see Figure 15.10). Run the scene and notice how you can now move the player. The player should be able to fire bullets (although they don't move). Also notice that the player can now die and end the game.

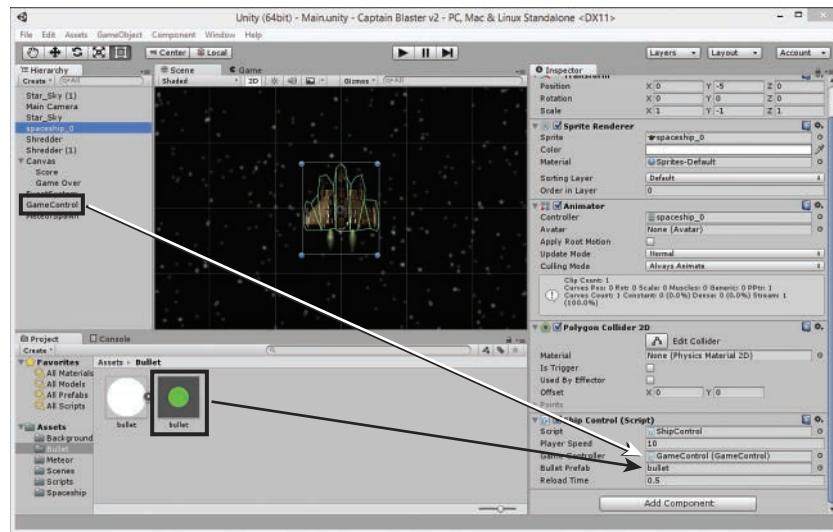


FIGURE 15.10

Connecting the `ShipControl` script.

The Bullet Script

The last bit of interactivity you need is to make the bullets move and collide. Create a new script called **Bullet** in your Bullet folder and add it to the bullet prefab. Replace the code in the script with the following:

```
using UnityEngine;
using System.Collections;

public class Bullet : MonoBehaviour {

    public float speed = 10f;
    private GameController gameController; // Note this is private this time

    // Use this for initialization
    void Start () {
        // Because the bullet doesn't exist until the game is running
        // we must find the Game Controller a different way.
        gameController = GameObject.FindObjectOfType<GameController> ();
    }

    // Update is called once per frame
    void Update () {
        transform.Translate (0f, speed * Time.deltaTime, 0f); // Move up
    }

    void OnCollisionEnter2D (Collision2D other) {
        Destroy (other.gameObject); // Destroy the meteor
        gameController.AddScore (); // Increment the score
        Destroy (gameObject); // Destroy the bullet
    }
}
```

The major difference between this script and the meteor is that this script needs to account for collision and the player scoring. The script declares a variable to hold the control script, just like the player. Because the bullet isn't actually in the Scene view, however, it needs to get access to the control script a little differently. In the `Start()` method, the script searches for the `GameControl` object by type using the `GameObject.FindObjectOfType<Type>()` method. The control script is then stored in the variable `control`. It is worth noting that using Unity's `Find()` methods (such as the one used here) is very slow and should be used sparingly.

Because neither the bullet nor the meteor has a trigger collider on it, the use of the `OnTriggerEnter2D()` method will not work. Instead, the script uses the method `OnCollisionEnter2D()`. This method does not read in a `Collider2D` variable. Instead, it reads in a `Collision2D` variable. The differences between these two methods are irrelevant in this case. The only work being done is destroying both objects and telling the control script that the player scored.

Go ahead and run the game. You notice that the game is now fully playable. Although you cannot win (that is intentional), you certainly can lose. Keep playing and see how high of a score you can get!

As a fun challenge, consider trying to make the values of `minSpawnDelay` and `maxSpawnDelay` get smaller over time, causing the meteors to spawn more quickly as the game is played.

Improvements

It is time to improve the game. Like the previous games, there are several places left intentionally basic. Be sure to play through the game several times and see what you notice. What things are fun? What things are not fun? Are there any obvious ways to break the game? Note that a very easy cheat has been left in the game to allow players to get a high score. Can you find it?

Here are some things you could consider changing:

- ▶ Try modifying the bullet speeds, firing delay, or bullet flight path.
- ▶ Try allowing the player to fire two bullets side by side.
- ▶ Try adding a different type of meteor.
- ▶ Give the player extra health; maybe even a shield.
- ▶ Allow the player to move vertically as well as horizontally.

This is a common genre, and there are many ways you can make it unique. Try to see just how custom you can make the game. It is also worth noting that as you learn about particle systems later in this book, this game is a prime candidate for trying them out.

Summary

In this hour, you made the game *Captain Blaster*. You started by designing the game elements. From there, you built the game world. You constructed and animated a vertically scrolling background. From there, you built the various game entities. You added interactivity through scripting and controls. Finally, you examined the game and looked for improvements.

Q&A

Q. Did Captain Blaster really achieve the military rank of captain or is it just a name?

A. It's hard to say, as it is all mostly speculation. One thing is for certain, they don't just give spaceships to lieutenants!

- Q. Why delay bullet firing by half a second?**
- A.** Mostly it is a balance issue. If the player can fire too fast, the game has no challenge.
- Q. Why use a polygon collider on the ship?**
- A.** Since the ship has an odd size, a standard shaped collider wouldn't be very accurate. Luckily, we can use the polygon collider to map closely to the geometry of the ship.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

- 1.** What is the win condition for the game?
- 2.** How does the scrolling background work?
- 3.** Which objects have rigidbodies? Which objects have colliders?
- 4.** True or False: The meteor is responsible for detecting collision with the player.

Answers

- 1.** Trick question. The player cannot win the game. The highest score, however, allows the player to "win" outside of the game.
- 2.** Two identical sprites are stacked on top of each other. They then "leap frog" across the camera to seem endless.
- 3.** The bullets and meteors have rigidbodies. The bullets, meteors, player, and triggers have colliders.
- 4.** False.

Exercise

This exercise will be a little strange compared to the ones you have done so far. A common part of the game refinement process is to have the game play tested by people who aren't involved with the development process. This allows people who are completely unfamiliar with the game to give honest, first-experience feedback. This is incredibly useful. The exercise is to have other people play the game. Try to get a good diverse group of people. Try to get some avid gamers and some people who don't play games. Try to get some people who are fans of this genre and some people who aren't. Compile their feedback into groupings of good features, bad features, and things that can be improved. In addition, try to see whether there are any commonly requested features that currently aren't in the game. As a last part, see if you can implement or improve your game based on the feedback received.

HOUR 16

Particle Systems

What You'll Learn in This Hour:

- ▶ The basics of particle systems
- ▶ How to work with modules
- ▶ How to use the curve editor

In this hour, you learn how to use Unity's particle system. You start by learning all about particle systems in general and how they work. You experiment with the many different particle system modules. You wrap the hour up by experimenting with the Unity curve editor.

Particle Systems

A particle system is basically an object or component that emits other objects, commonly referred to as *particles*. These particles can be fast, slow, flat, shaped, small, or large. The definition is very generic because these systems can achieve a great variety of effects with the proper settings. They can make jets of fire, plumes of billowing smoke, fireflies, rain, fog, or anything else you can think of. These effects are commonly referred to as *particle effects*.

Particles

A particle is a single entity that is emitted by a particle system. Because many particles are generally emitted quickly, it is important for particles to be as efficient as possible. This is the reason that most particles are 2D billboards. Remember that a billboard is a flat image that always faces the camera. This gives the illusion that they are 3D, while still giving great performance.

Unity Particle Systems

To create a particle system in a scene, you can either add a particle system object or add a particle system component to an existing object. To add a particle system object, click **GameObject > Particle System**. To add a particle system component to an existing object, select the object and click **Add Component > Effects > Particle System**.

TRY IT YOURSELF

Creating a Particle System

In this exercise, you create a particle system object in your scene:

1. Create a new project or scene.
2. Add a particle system by clicking **GameObject > Particle System**.
3. Notice how the particle system is emitting white particles in the Scene view (see Figure 16.1). This is the basic particle system. Try rotating and scaling the particle system to see how it reacts.

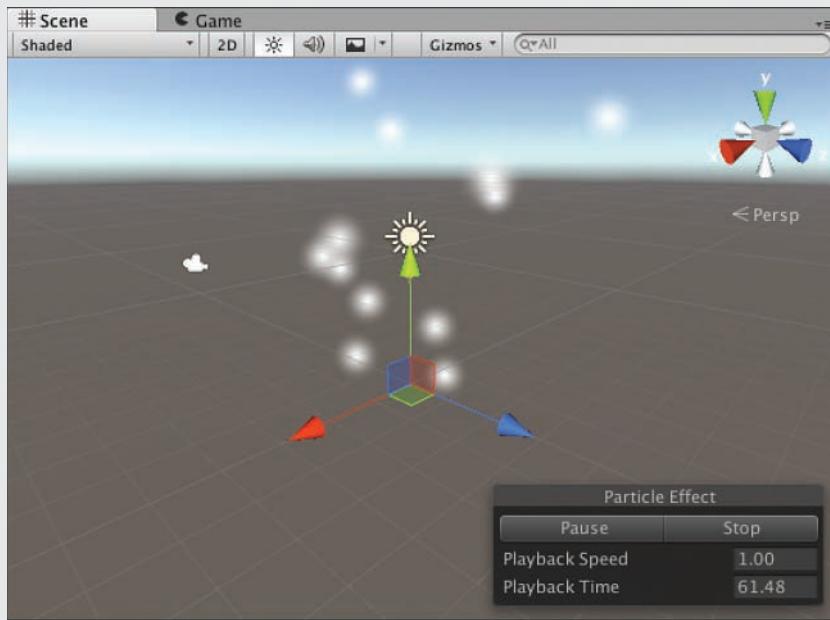


FIGURE 16.1
The basic particle system.

NOTE

Custom Particles

By default, the particles in Unity are small white spheres that fade into transparency. This is a really useful generic particle, but it can only take you so far. Sometimes, you want something more specific (to make fire, for example). If you want, you can make your own particles out of any 2D image to make effects to exactly suit your needs.

Particle System Controls

You might have noticed that when you added a particle system to your scene it began emitting particles in the Scene view. You may have also noticed the particle effect controls that appeared (see Figure 16.2). These controls allow you to pause, stop, and restart the particle animation in a scene. This can be very helpful when tweaking the behavioral components of a particle system.

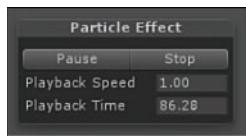


FIGURE 16.2

The particle effects control.

The control also allows you to speed up the play back and also tells you how long the effect has been playing. This can prove very useful when testing duration effects. Note that the control only shows the Playback Speed and Playback Time when the game is stopped.

NOTE

Particle Effects

To create complex and visually appealing effects, you want several particle systems to work together (a smoke and a fire system, for example). When multiple particle systems are working together, it is called a *particle effect*. In Unity, creating a particle effect is achieved by nesting particle systems together. One particle system can be the child of another, or they can both be children of a different object. The result of a particle effect in Unity is that they are treated as one system and that the particle effect controls will control the entire effect as one unit.

Particle System Modules

At its root, a particle system is just a point in space that emits particle objects. How the particles look, behave, and the effects they cause are all determined by modules. Modules are various properties that define some form of behavior. In Unity's particle system, modules are an integrated and essential component. This section is going to list each module and explain briefly what it does.

Note that with the exception of the default module (covered first) all modules can be turned on and off. To turn modules on or off, put a check mark by the module's name. To hide or show modules, click the plus sign (+) next to the Particle System modules (see Figure 16.3). You can also click the name of the particle system in the list to toggle visibility. By default, all modules are visible and only the Emission, Shape, and Renderer modules are enabled. To expand a module, simply click its title.

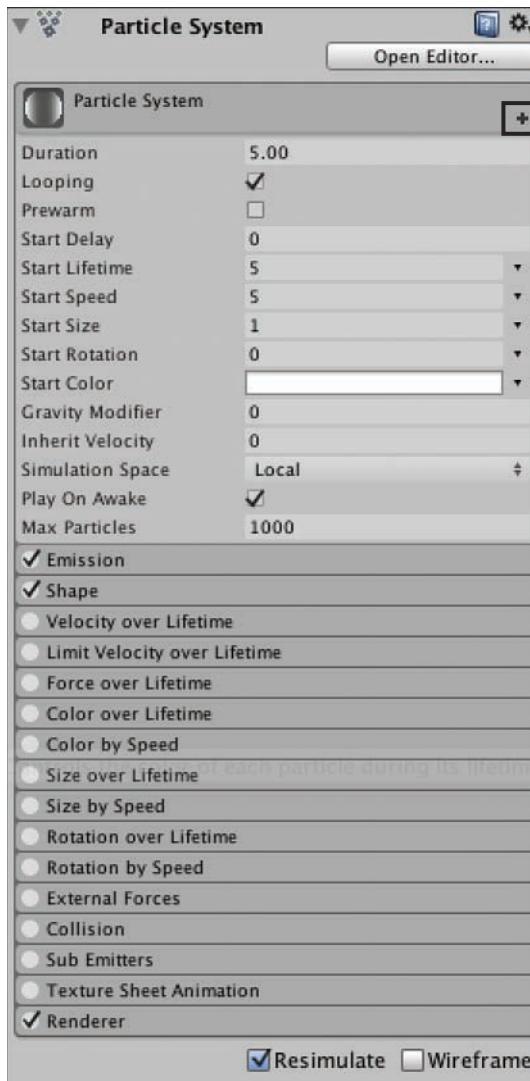


FIGURE 16.3
Showing all modules.

NOTE**Brief Overview**

Several modules have properties that are either self-explanatory (like the length and width property of a rectangle) or have been covered previously. For the sake of simplicity (and to prevent this hour from being 30 pages) these will be omitted. If you see more properties on your screen than are covered in this text, don't worry. That is intentional.

NOTE**Constant, Curve, Random**

A value curve allows you to change the value of a property over the lifetime of a particle system, or individual particle. You know which properties can use curves by the downward-facing arrow next to the value. The options you are given are Constant, Curve, Random Between Two Constants, and Random Between Two Curves. For the sake of this section, all values are treated as constant. Later in this hour, you get a chance to explore the curve editor in detail.

Default Module

The default module is simply labeled Particle System. This module contains all the specific information that every particle system requires. Table 16.1 describes the properties of the default module.

At the bottom of the Particle System component you will notice two further settings, *resimulate* and *wireframe*. *Resimulate* checkbox determines whether or not property changes should be applied immediately to particles already generated by the system (the alternative is that existing particles are left as they are and only the new particles have the changed properties). The *wireframe* checkbox draws the particles as just outlines, improving performance in Scene view.

TABLE 16.1 Default Module Properties

Property	Description
Duration	How long, in seconds, the particle system runs.
Looping	Determines if the particle system starts over once the duration has been reached.
Prewarm	If this is selected, the particle system starts as if it had already emitted particles from a previous cycle.
Start Delay	How long, in seconds, the system will wait before emitting particles.
Start Lifetime	How long, in seconds, each particle will live.
Start Speed	The initial speed of particles.

Property	Description
Start Rotation	The initial rotation of particles.
Start Color	The color of emitted particles.
Gravity Modifier	How much of the world's gravity is applied to the particles.
Inherit Velocity	How much of the system's velocity (if any) is imparted on the particles.
Simulation Space	Determines if the particles are simulated in local or world space.
Play On Wake	Determines whether the particle system begins emitting particles immediately when created.
Max Particles	The total number of particles that can exist for a system at a time. If this number is reached, the system ceases emitting until some particles die.

Emission Module

The Emission module is used to determine the rate in which particles are emitted. Using this module, you can dictate whether particles stream at a constant rate, in bursts, or somewhere in between. Table 16.2 describes the Emission module properties.

TABLE 16.2 Emission Module Properties

Property	Description
Rate	Number of particles emitted over time or distance.
Bursts	If the time option for rate is chosen, this is used to dictate the number of bursts. Create a burst by clicking the plus sign (+) and remove a burst by clicking the minus sign (-) (see Figure 16.4).

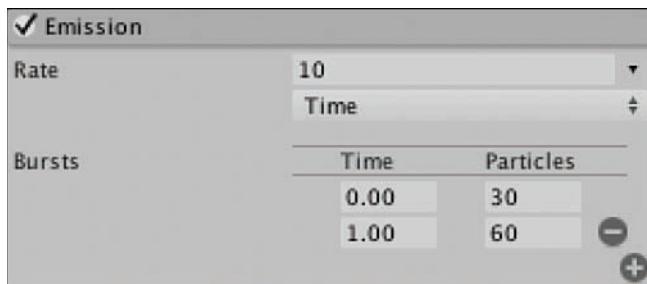


FIGURE 16.4
The Emission module.

Shape Module

Just as its name would imply, the Shape module determines the shape formed by the emitted particles. The shape options are Sphere, Hemisphere, Cone, Box, and Mesh. In addition, each shape has a set of properties used to define it. These properties are things like radius for cones and spheres. There are fairly self-explanatory and are not covered here.

Velocity over Lifetime Module

The Velocity over Lifetime module directly animates each particle by applying an x-, y-, and z-axis velocity to it. Note that this is a velocity change of each particle over the lifetime of the particle, not over the lifetime of the particle system. Table 16.3 describes the properties of the Velocity over Lifetime module.

TABLE 16.3 Velocity over Lifetime Module Properties

Property	Description
XYZ	The velocity applied to each particle. This can be a constant, curve, or random number between a constant or curve.
Space	Dictates whether the velocity is added based on local or world space.

Limit Velocity over Lifetime Module

This long-named module can be used to dampen or clamp the velocity of a particle. Basically, it prevents, or slows down, particles that exceed a threshold speed on one or all of the axes. Table 16.4 describes the properties for the Limit Velocity over Lifetime module.

TABLE 16.4 Limit Velocity over Lifetime Module Properties

Property	Description
Separate Axis	If unchecked, this property uses the same value for each axis. If checked, speed properties for each axis appear as well as a property for local or world space.
Speed	The threshold speed for each or all axes.
Dampen	The value, between 0 and 1, that a particle will be slowed by if it exceeds the threshold is determined by the speed property. A value of 0 will not slow a particle at all, but a value of 1 will slow the particle 100%.

Force over Lifetime Module

The Force over Lifetime module is similar to the Velocity over Lifetime module. The difference is that this module applies a force, not a velocity, to each particle. This means that the particle will continue to accelerate in the specified direction. This module also allows you to randomize the force each frame, as opposed to all upfront.

Color over Lifetime Module

The Color over Lifetime module allows you to change the color of the particle as time passes. This is useful for creating effects like sparks, which start out bright orange and end a dark red before disappearing. To use this module you must specify a gradient of color. You can also specify two gradients and have Unity randomly pick a color between them. Gradients can be edited using Unity's gradient editor (see Figure 16.5).

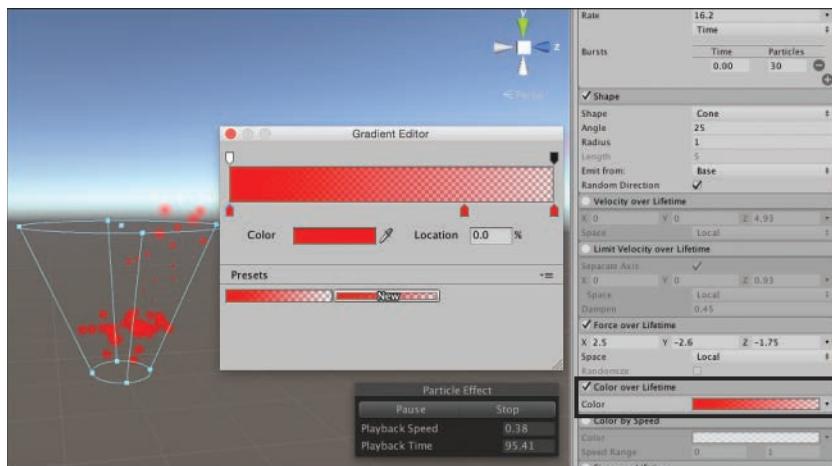


FIGURE 16.5
The gradient editor.

Note that the color of the gradient will be multiplied by the Start Color property of the default module. This means that if your start color is black, this module will have no effect.

Color by Speed Module

The Color by Speed module allows you to change the color of a particle based on its speed. Table 16.5 describes the properties of the Color by Speed module.

TABLE 16.5 Color by Speed Module Properties

Property	Description
Color	A gradient (or two gradients for random colors) that is used to dictate the color of the particle.
Speed Range	The minimum and maximum speed values that are mapped to the color gradient. Particles going the min speed are mapped to the left side of the gradient, and colors at the max speed (or beyond) are mapped to the right side of the gradient.

Size over Lifetime Module

The Size over Lifetime module allows you to specify a change in the size of a particle. The size value must be a curve and will dictate whether the particle grows or shrinks as time elapses.

Size by Speed Module

Much like the Color by Speed module, the Size by Speed module changes the size of a particle based on its speed between a minimum and maximum value.

Rotation over Lifetime Module

The Rotation over Lifetime module allows you to specify a rotation over the life of a particle. Note that the rotation is of the particle itself, and not a curve in the world coordinate system. What this means is that if your particle is a plain circle, you will not be able to see the rotation. If the particle has some detail, however, you will notice it spin. The values for the rotation can be given as a constant, curve, or random number.

Rotation by Speed Module

The Rotation by Speed module is the same as the Rotation over Lifetime module except that it changes values based on the speed of the particle. Rotation will change based on a min and max speed value.

External Forces Module

The External Forces module allows you to apply a multiplier to any forces that exist outside of the particle system. A good example of this is any wind forces that may exist in a scene. The Multiplier property scales the forces either up or down depending on its value.

Collision Module

The Collision module allows you to set up collisions for particles. This is useful for all sorts of collision effects, like fire rolling off a wall or rain hitting the ground. You can set the collision to work with predetermined planes (Plane mode: most efficient), or with objects in the scene (World mode: slows performance). The Collision module has some common properties and some unique properties depending on the collision type chosen. Table 16.6 describes the common properties of the Collision module. Tables 16.7 and 16.8 describe the properties that belong to Planes mode and World mode, respectively.

TABLE 16.6 Common Collision Module Properties

Property	Description
Planes/World	Dictates the type of collision used. Planes mode will collide off of predetermined planes. World mode will collide off of any object in a scene.
Dampen	Determines the amount a particle is slowed when it collides. Values range from 0 to 1.
Bounce	Determines what fraction of the component of velocity is kept. Unlike dampen, this only affects the axes the particle bounces on. Values range between 0 and 1.
Lifetime Loss	Determines how much life of a particle is lost on collision. Values range from 0 to 1.
Min Kill Speed	The minimum speed of a particle before it is killed by collision.
Send Collision Messages	Determines whether collision messages are sent to objects that collide with particles.

TABLE 16.7 Plane Mode Properties

Property	Description
Planes	A collection of transforms used to determine where the particles can collide. The y axis of the transforms provided determines the rotation of the plane.
Visualization	Used to determine how the planes are drawn in the Scene view. They can either be solid or grid.
Scale Plane	Resizes the visualization of the planes.
Particle Radius	Can be used to make the particles seem bigger or smaller for collision purposes.

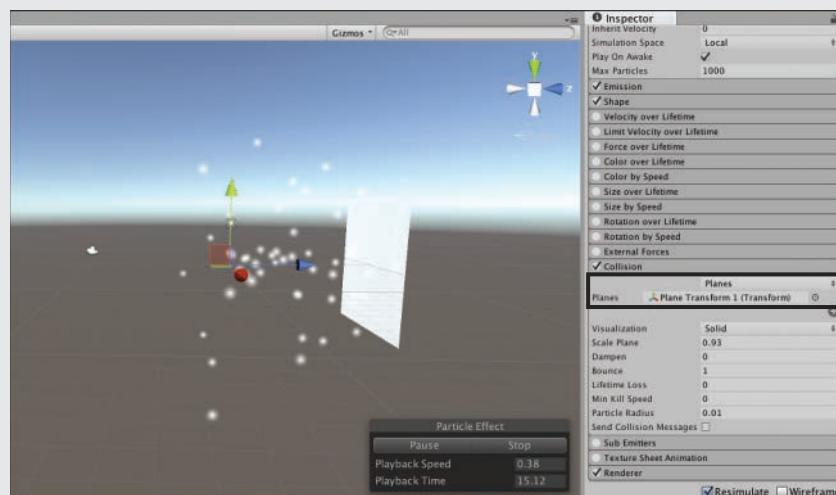
TABLE 16.8 World Mode Properties

Property	Description
Collides With	Determines which layers the particles collide with. This is set to Everything by default.
Collision Quality	The quality of the world collision. The values are High, Medium, and Low. Obviously, High is the most CPU intensive and most accurate, and Low is the least.
Voxel Size	This property is more advanced and is used to determine the density of the Voxels used in medium- and low-quality settings. Basically, leave this value as is unless you know what you are doing.

TRY IT YOURSELF ▼**Making Particles Collide**

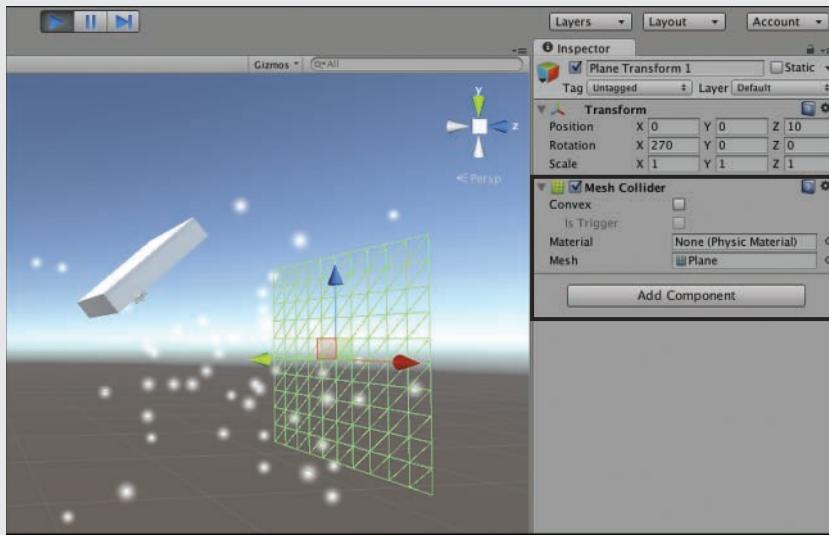
In this exercise, you set up collision with a particle system. This exercise uses both Planes and World collision modes:

1. Create a new project or scene. Add a particle system to the scene and place it at (0, 0, 0).
2. In the Inspector, enable the Collision module by clicking the circle next to its name. Click the small plus sign (+) next to the Planes property and a plane should appear (see Figure 16.6). Notice how the particles are already bouncing off of the plane. Move and rotate the plane around and see how it affects the particles.

**FIGURE 16.6**

Adding a plane transform.

- ▼
3. Add a cube to the scene. Position the cube at (0, 4, 0), rotate it to (320, 0, 0), and give it a scale of (5, 1, 5). Notice how the particles move right through the cube because we are in planes collision mode.
 4. Set the Collision module to World mode by selecting the Particle System in the Hierarchy, and changing Planes to World where highlighted in Figure 16.6. Notice how the particles now begin to bounce off of the cube.
 5. Finally, add a Mesh Collider to the **Plane Transform 1** (a child of the Particle System), and set its Mesh to Plane (see Figure 16.7). Notice you can now deselect the Particle System in the hierarchy, and particles will still bounce off of the (invisible) plane.

**FIGURE 16.7**

Adding a plane collider to the plane, so that collisions happen in world mode.

TIP**Emitter versus Particle Settings**

Some modules modify the emitter, while others modify the particles. You may wonder why there is a color property, and also a color over lifetime property. One controls the color over the life of the emitter, while the other makes particles change color over their life.

Sub Emitter Module

The Sub Emitter module is an incredibly powerful module that enables you to spawn a new particle system at certain events for each particle of the current system. You can create a new particle system every time a particle is created, dies, or collides. This can be used to generate complex and intricate effects (like fireworks). This module has three properties: Birth, Death, and Collision. Each of these properties holds zero or more particle systems to be created on the respective events.

Texture Sheet Module

The Texture Sheet module allows you to change the texture coordinates used for a particle over the life of the particle. In essence, this means that you can put several textures for a particle in a single image and then switch between them during the life of a particle. Table 16.9 describes the properties of the Texture Sheet module.

To make this work you need to have a sprite-sheet imported as a texture, attach it as the Albedo of a material, and assign the material to the particle system.

TABLE 16.9 **Texture Module Properties**

Property	Description
Tiles	Determines the tiling of the texture.
Animation	Determines if the whole image contains textures for the particle or if only a single row does.
Cycles	Specifies the speed of the animation.

Renderer Module

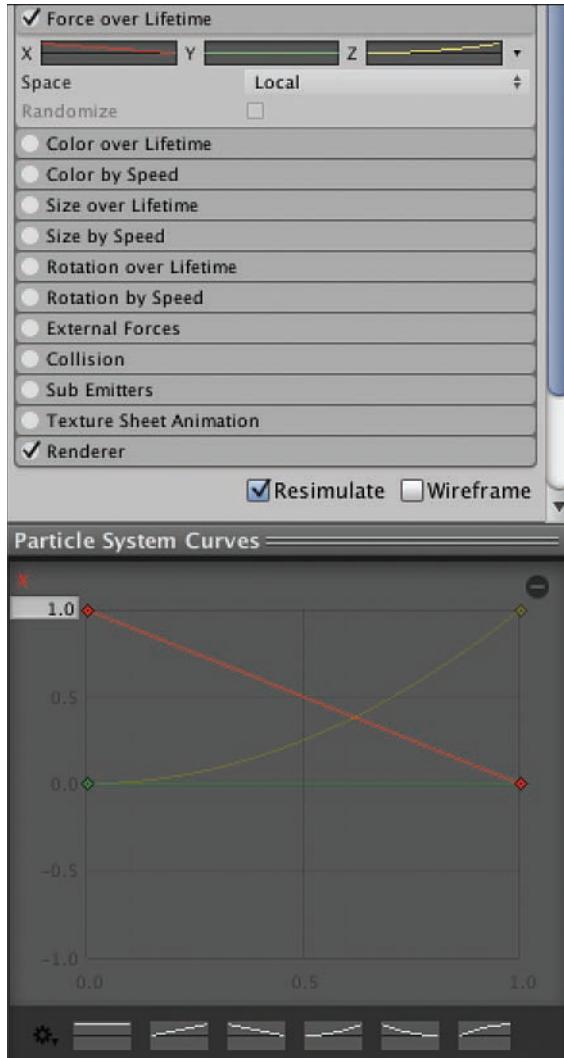
The Renderer module dictates how the particles are actually drawn. It is here that you can specify the texture used for the particles and their other drawing properties. Table 16.10 describes some of the properties of the Renderer module.

TABLE 16.10 Renderer Module Properties

Property	Description
Render Mode	Determines how the particles are actually drawn. The modes are Billboard, Stretched Billboard, Horizontal Billboard, Vertical Billboard, or Mesh. All the billboard modes cause the particles to align with either the camera or two out of three axes. The mesh mode causes the particles to be drawn in 3D as determined by a mesh.
Normal Direction	Dictates how much the particle faces the camera. A value of 1 causes the particles to look directly at the camera.
Material	The material used to draw the particle.
Sort Order	The order that particles are drawn. Can be None, By Distance, Youngest First, or Oldest First.
Sorting Fudge	Determines the order that the particle system is drawn. The lower the value, the more likely the system is to be drawn on top of other particles.
Cast Shadows	Determines if particles cast shadows.
Receive Shadows	Determines if particles receive shadows.
Max Particle Size	Set max relative size. Valid values range between 0 and 1.

The Curve Editor

Several values in the various modules listed previously had the option to be set as Constant or Curve. The Constant option is fairly self-explanatory. You give it a value, and it is that value. What if you want that value to change over a period of time, though? That is where the curve system comes in very handy. Using this feature, you have a very fine level of control over how a value behaves. You can see the curve editor at the bottom of the Inspector view (see Figure 16.8). You may need to drag it up by the horizontal handle.

**FIGURE 16.8**

The curve editor in the Inspector.

The title of the curve is whatever value you are determining. In Figure 16.8, the value is for the force applied along the x axis in the Force over Lifetime module. The range dictates the minimum and maximum values available. This can be changed to allow for a greater (or lesser) range. The curve is the values themselves over a given course of time and the presets are generic shapes that you can give to the curve.

The curve is moveable at any of the key points. These key points are shown as visible points along the curve. By default, there are only two key points: one at the beginning and one at the

end. You can add a new key point anywhere on the curve by right-clicking it and choosing **Add Key Point**.

You can get an even larger curve editor by clicking the “Open Editor . . .” button at the top right of the Particle System component.

▼ TRY IT YOURSELF

Using the Curve Editor

Let's get familiar with the curve editor. In this exercise, you change the size of the particles emitted over the duration of one cycle of the particle system:

1. Create a new project or scene. Add a particle system and position it at (0, 0, 0).
2. Click the drop-down arrow next to the Start Size property and chose **Curve**.
3. Change the range of the curve from 1.0 to 2.0 by clicking on the y axis at the top left of the curve editor.
4. Right-click the curve at about the midpoint and add a key. Do the same for the end of the curve. Now drag the start and end points of the curve down to 0 (see Figure 16.9). Notice how the particles emitted change in size over the 5-second cycle of the particle system.

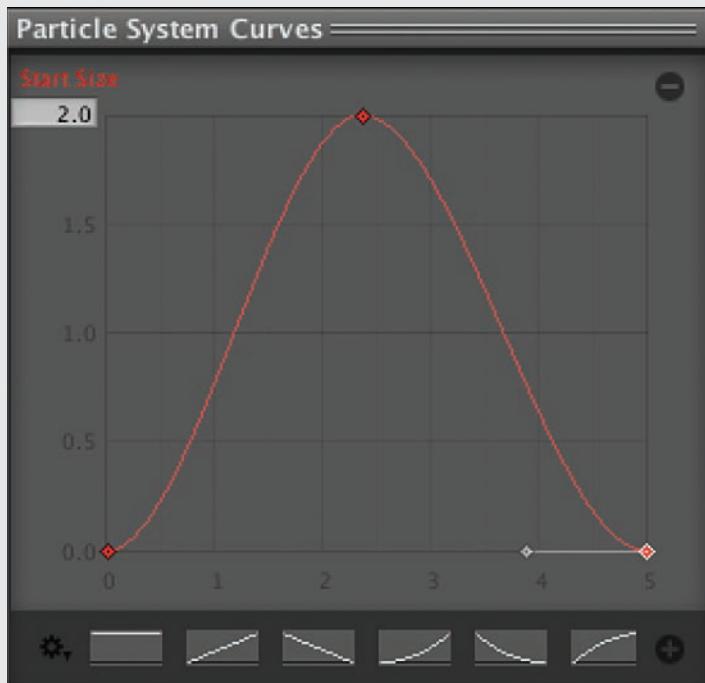


FIGURE 16.9
Start Size curve settings.

Summary

In this hour, you were introduced to particle systems in Unity. You learned the basics of particles and particle systems. You then went on a lengthy review of the many modules that make up the Unity particle system. You wrapped the hour up by looking at the functionality of the curve editor.

Q&A

Q. Are particle systems inefficient?

- A. They can be. It depends on the settings you give them. A good rule of thumb is to only use a particle system if it provides some value to you. They can be great visually, but don't overdo it.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What is the term for a 2D image that always faces the camera?
2. How do you open a larger Particle Effect editor window?
3. Which module controls how a particle is drawn?
4. True or False: The curve editor is used for creating curves that change values over time.

Answers

1. A billboard.
2. Click the “Open Editor . . .” button at the top of the Particle System component in the Inspector.
3. The Renderer module.
4. True.

Exercise

In this exercise, you experiment with some exciting particle effects provided as standard packages with Unity 5. This exercise is both a chance to play around with existing effects and to create your own. There is no correct “solution” for you to look at. Just follow the steps here and use your imagination:

1. Import the particle effects package by clicking **Assets > Import Package > ParticleSystems**. Be sure to leave all assets checked and click **Import**.
2. Navigate to **Assets > Standard Assets > ParticleSystems > Prefabs**. Click and drag the FireComplex and Smoke prefabs into the Hierarchy. Experiment with positioning and settings of these effects. Click Play to see the effect.
3. Continue experimenting with the rest of the provided particle effects (be sure to check out the Explosion and Fireworks effects at least).
4. Now that you have seen what is possible, see what you can create yourself. Try out the various modules and try to come up with your own custom effects.

HOUR 17

Animations

What You'll Learn in This Hour:

- ▶ The requirements for animation
- ▶ The different types of animations
- ▶ How to create animations in Unity

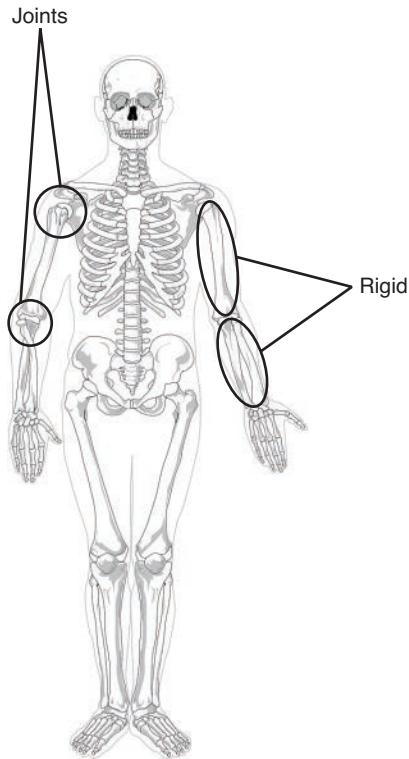
In this hour, you learn about animations in Unity. You start by learning exactly what animations are and what is required for them to work. After that, you look at the different types of animations. From there, you learn how to create your own custom animations with Unity's animation tools.

Animation Basics

Animations are premade sets of visual motions. In a 2D game, this involves having several sequential images that can be flipped through very quickly. The result is that the object appears to be moving. This effect is similar to an old-fashioned flip book. Animation in a 3D world is much different. In 3D games, you use models to represent your game entities. You cannot simply switch between models to give the illusion of motion. Instead, you have to actually move the parts of the model. Doing so requires both a rig and an animation. Furthermore, animations can also be thought of as "automations"; meaning we can use animations to automate object changes such as the size of colliders, value of script variables, or even the color of materials!

The Rig

Achieving complex animated actions, such as walking, without a rig is impossible (or impossibly difficult). The reason is that without a rig, the computer has no way of knowing which parts of a model are supposed to move and how they are supposed to move. So, what exactly is a rig? Much like a human skeleton (see Figure 17.1), the rig dictates the parts of a model that are rigid, which are often called *bones*. It also dictates which parts can bend. These bendable parts are called *joints*.

**FIGURE 17.1**

The skeleton as a rig.

The bones and joints work together to define a physical structure for a model. It is this structure that will be used to actually animate the model. It is worth noting that 2D animations, simple animations, and animations on simple objects do not require any particular or complex rig.

The Animation

Once a model has a rig (or not, in the case of simple animations), it can be given an animation. On the technical level, an animation is just a series of instructions for a property or a rig.

These instructions can be played just like a movie. They can even be paused, stepped through, or reversed. Furthermore, with a proper rig, changing the action of a model is as simple as changing the animation. The best part of all is that if you have two completely different models that have the same rigging (or different rigging, as you will discover in Hour 18), you can apply the same animations to each of them identically. Thus, an orc, a human, a giant, and a werewolf can all perform the exact same dance.

NOTE**3D Artists Wanted**

The truth about animation is that most of the work is done outside of programs like Unity. Generally speaking, the modeling, texturing, rigging, and animations are all created by professionals, known as 3D artists, in programs such as Blender, Maya, 3D Studio Max, etc. Creating these assets requires a significant amount of skill and practice. Therefore, their creation is not covered in this text. Instead, this book shows you how to take already made assets and put them together in Unity to build interactive experiences. Remember that there is more to making a game than simply putting pieces together. You may make a game work, but artists make it look good!

Animation Types

So far you've read about things like animations, rigs, and "automations." These terms may seem a bit nonsensical at this point and you may be wondering how they relate and what exactly is needed for you to use animations in a game. In this section, we will look more at each type of animation and get some understanding of how they work so that we can begin making them.

2D Animations

In a sense, 2D animations are the simplest type of animations. As explained above, a 2D animation functions very much like a flip book or animated cartoon (or even film-based movies). The idea behind a 2D animation is that images are presented in sequential order at a very fast pace. The result is that the eye is tricked into seeing motion.

These are very easy to set up within Unity, but they are more difficult to modify. The reason is that 2D animations require art assets (the images being shown) in order to work. Any changes to the animation would require you (or an artist) to make changes to the images themselves in other software, such as Photoshop or Gimp. Changes to the images themselves within Unity isn't possible.

TRY IT YOURSELF

Slicing a Sprite sheet for Animation

Let's prepare a sprite sheet for animation. The project created in this exercise will be used later, so be sure to save it:

1. Create a new 2D project.
2. Import the **RobotBoyRunSprite.png** image from the 2D assets package. You can do this by clicking **Assets > Import Package > 2D** and importing *only* the **RobotBoyRunSprite.png** asset (See Figure 17.2).

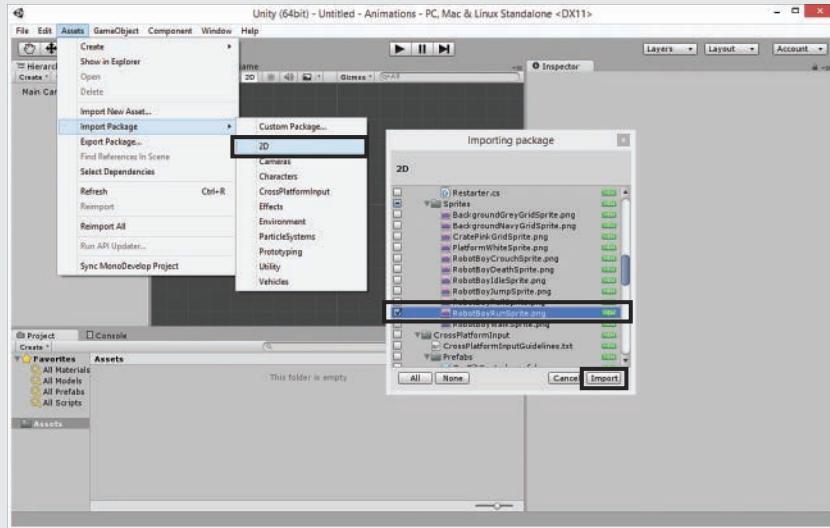
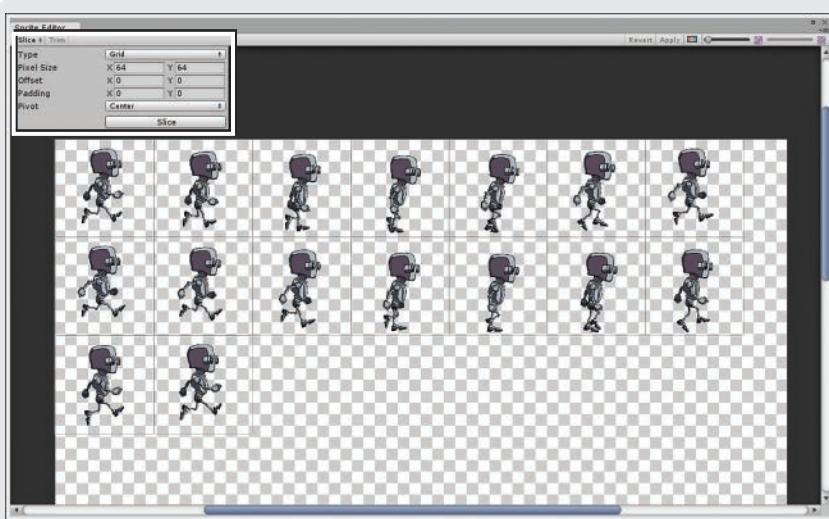


FIGURE 17.2
Importing the sprite sheet.

3. While this image has already been prepared for animation (it is an animated character in the 2D assets package), we are still going to examine it as a review. Select the newly imported **RobotBoyRunSprite.png** file in the Project view.
4. In the Inspector view ensure that the **Sprite Mode** is set to Multiple and then click **Sprite Editor**. In the upper left corner, click **Slice** and then choose **Grid** as the type. Notice the grid sizes and the resulting sprite slices (See Figure 17.3).

**FIGURE 17.3**

Slicing the sprite sheet.

5. Close the Sprite Editor window.

Now that we have a series of sprites we can turn them into an animation.

NOTE

Reinventing the Wheel

We are using assets from Unity's 2D asset package. You may have noticed that these assets are already animated. This means that we are doing work that has already been done. This is done on purpose to illustrate the work that went into animated these characters originally. Better yet, if you want to explore the completed assets in the 2D asset package you can see how they are put together to see how more complex examples are achieved.

Creating the Animation

You've prepared your assets and so you are now ready to turn them into an animation. As with anything, there is a simple way and a complicated way to accomplish this. The complicated way involves creating an animation asset, specifying sprite renderer properties, adding key frames, and providing values. Since you haven't learned how to do that yet (which you will in the next section), let's go with the simple route. Unity has a strong automated workflow and we will tap into that to create our animation.

TRY IT YOURSELF

Creating an Animation

Let's use the project we created in the last exercise to make an animation:

1. Open the project created in the previous exercise.
2. Locate the **RobotBoyRunSprite** asset in the Project view. Expand the sprite drawer to see all of the subsprites (click the small arrow on the right side of the sprite).
3. Select all of the sprites from that sprite sheet (select the first sprite and then select the last sprite while holding the shift key). Once selected, drag all of the frames into the Scene view (or Hierarchy view, either works), and let go (See Figure 17.4).

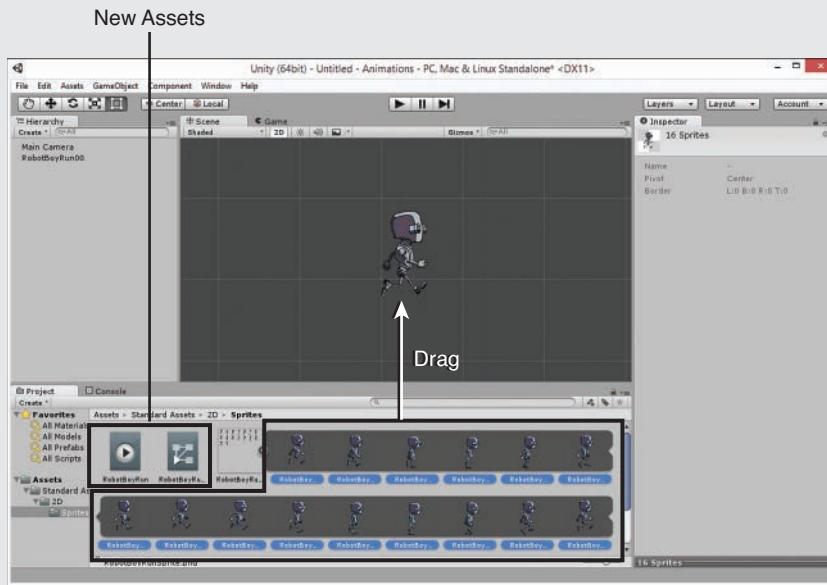


FIGURE 17.4
Creating the animation.

4. Depending on your version of Unity, you may or may not be prompted with a “Save As” dialog. If you aren’t prompted, two new assets will be created in the same folder as the sprite sheet. If you are prompted, choose a name and locate for your new animation. Either way, Unity has now automated the process of creating an animated sprite character in your scene. The two assets created (the animation asset and another asset called an **animator controller**) will be covered in greater detail in Hour 18.
5. Run your scene and you will see the animated robot boy playing its running sequence.

That's it! 2D animations really are very easy to create.

NOTE

More Animations

You now know how to make a single 2D animation, but what if you wanted a series of animations (such as walk, run, idle, jump, etc.) that all work together? Luckily, the concepts you've just learned continue to work in more complex scenarios. Creating animations continue to work the same way. Getting the animations to work together, however, requires a larger understanding of Unity's Mecanim system. You will learn that system in great detail in Hour 18. During Hour 18 you will be using imported 3D animations. Just remember that anywhere you can use a 3D animation, you can also use any other type of animation. Thus, the concepts you learn in Hour 18 are equally applicable to 2D and custom animations!

Animation Tools

Unity has a set of animation tools built in that enable us to create and modify animations without leaving the editor. We have already used them unknowingly when we made a 2D animation and now it is time to dig in and see just what we can do.

Animation Window

To begin using Unity's animation tools, you will need to open the Animation window. You can do this by clicking **Window > Animation** (not Animator). This will open a new view that you can resize and dock like any other Unity window. Generally, it is a good idea to dock this window so you can use it and other parts of the editor without them overlapping. Figure 17.5 illustrates the Animation window and its various elements.

Note that for the purpose of this figure, the 2D animated sprite from the previous exercise is selected. See if you can identify how Unity used the sprites you dragged into the scene to make the animation you saw when you ran your project. Table 17.1 runs through some of the more important parts of the Animation window.

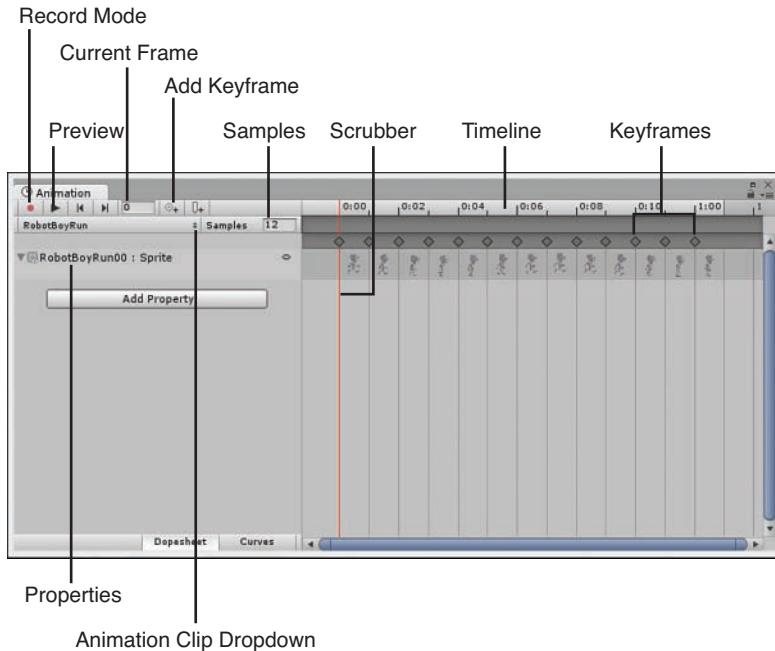


FIGURE 17.5
The Animation window.

TABLE 17.1 Important Parts of the Animation Window

Component	Description
Properties	The listing of the component properties modified by this animation. Expanding a property allows you to see the value of that property based on the value of the scrubber on the timeline.
Samples	This is the number of animation frames available in one second of animation.
Timeline	The visual representation of changes to properties over time. The timeline allows you to specify when values change.
Key frames	Key frames represent special points on the timeline. A key frame allows you to specify the desired property value at that point of time. Regular frames (not shown) also contain values but you cannot directly control them.
Add Key frame	This button allows you to add a key frame to the timeline for the selected property at the position of the scrubber.
Scrubber	The red line, known as the Scrubber , allows you to choose the point on the timeline that you would like to edit.
Current Frame	An indicator of what frame the scrubber is currently on.

Component	Description
Preview	Causes the animation to run in the Scene view.
Record Mode	This toggle puts you in and out of record mode. In record mode, any changes you make to the selected object will be recorded into the animation. Use with caution!
Animation Clip Dropdown	This dropdown menu allows you to change between the various animations associated with the selected object as well as create new animation clips.

Hopefully, looking at this window can give more insight into how your 2D animation works. An animation called RobotBoyRun was created. This animation had a sample rate of 12 frames per second. Each frame of the animation contains a key frame which sets the **Sprite** property of the objects **Sprite Renderer** to a different image, thus causing your character to appear to change. All this (and more) was done automatically!

Creating a New Animation

Now that you're familiar with the tools, let's put them to use. Creating animations is a task of placing key frames and then choosing values for them. The value of the frames in between all of the key frame will be added for you to smoothly transition between them. An example of this might be making an object move up and down. You could easily achieve this by choosing to modify the **Position** of the object's **Transform** and adding three key frames. The first would have a "low" y-axis value, the second would be higher, and the third would be the same as the first. The result is that the object would bounce up and down. This is all fairly conceptual, so let's try an example.

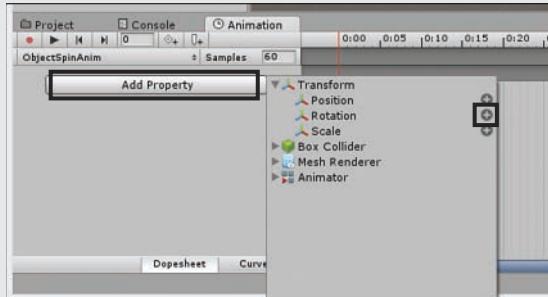
TRY IT YOURSELF ▼

Make an Object Spin

In this exercise we are going to make an object spin. We will be doing this to a simple cube, but in reality we could apply this animation to any object we wanted and the result would be the same. Be sure to save the project you create here for later use:

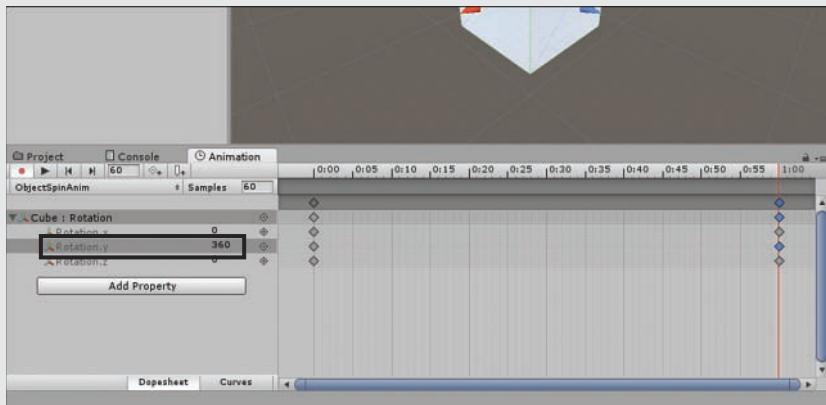
1. Create a new 3D project.
2. Add a cube to the scene and ensure that it is positioned at (0, 0, 0).
3. Open the Animation window (**Window > Animation**) and dock it in your editor.
4. With the cube selected, you should see a **Create** button right in the middle of the Animation view. Click it with the might of Thor! You will be prompted to save your animation, so go ahead and save it with the name **ObjectSpinAnim**. Note: If you don't see a create button in the middle of the Animation view (which didn't exist in older versions), look up at Figure 17.5 and locate the Animation Clip dropdown. Clicking there will give you the option to create a new animation clip. Feel free to also click that with the might of Thor.

5. Click the Add Property button in the Animation view and then click the (+) “plus” icon next to **Transform > Rotation** (see Figure 17.6).

**FIGURE 17.6**

Adding the Rotation property.

6. You should now have two key frames added to your animation: one at frame 0 and another at frame 60 (or “1 second,” see the Tip below for more info about the timing of the timeline). If you expand the Rotation property, you will be able to see the properties of the individual axes. Furthermore, selecting a key frame will allow you to see and adjust the values of that key frame.
7. Select the end key frame and then set the value of the **Rotation.y** property to 360 (see Figure 17.7). Even though the starting value of 0 and end value of 360 are technically the same, doing this will cause the cube to rotate. Play your scene and see the animation!

**FIGURE 17.7**

Adding the value.

8. Be sure to save your scene as we will be using it later.

TIP**Animation Timing**

The values on the timeline might seem a bit odd at first glance. In fact, reading the timeline has a lot to do with knowing the sample rate of the given animation clip. Based on a default sample rate of 60 frames per second, the timeline would count frames 0 to 59 and then instead of frame 60 it has “1:00” (for 1 second). Therefore, a time of 1:30 would mean one second and 30 frames. This is easy when we have a 60-frame per second animation, but what about a 12-frame per second animation? Then the counting would be “1, 2, 3 . . . 11, 12, one second.” To put it another way, you could see “0:10, 0:11, 1:00, 1:01, 1:02 . . . 1:10, 1:11, 2:00 . . . ” and so on. The most important thing to take away from this is that the number preceding the colon is the seconds and the number following it is the frames.

TIP**Moving the Timeline**

If you would like to zoom out or pan in the timeline to see more frames or look at different frames, you can do so easily. The window uses the same navigation style as a Scene view in 2D mode. That is, scrolling the mouse wheel zooms the timeline in and out while holding alt (option on a Mac) and dragging pans around.

Record Mode

While the tools you have used to far have been very easy, there are even easier ways to work with animations. An important element in the animations tool is **Record Mode** (see Figure 17.5 above for the location of it). In record mode any changes you make to the object will be recorded into the animation. This can be very powerful for quick and accurate additions to an animation. It can also be very dangerous. Consider what would happen if you forgot you were in record mode and made a bunch of changes to an object. Those changes would be recorded into the animation and repeated over and over whenever it was played! It is because of this that it is generally advisable to always ensure that you are not in record mode when not using it. Also, Unity (in an attempt to make your life easier) will put you into record mode when you move the scrubber, create a new clip, or modify a clip. Therefore, it is also a good idea to hide the Animation window when not in use.

This is a lot of scary warning for a tool we haven’t even used yet, but don’t worry. It really isn’t that bad (and it is super powerful to boot). With a little discipline it is a fantastic tool. Worst case scenario, if Unity records something for you, you can just delete it from the Animation yourself.

TRY IT YOURSELF

Use the Record Mode

In this exercise, we will be making the cube change color while it spins. We will be using the scene created in the previous exercise:

1. Open the scene with your spinning cube. Create a new material called **CubeColor** and apply it to the cube (this is so we can change the color).
2. Ensure the cube is selected and then open the Animation window. You should see your rotation animation that was previously created (if you don't, make sure the cube is selected).
3. Enter record mode by clicking the **Record Mode** button in the Animation window. You will see the Rotation property in the Inspector view turn red. This signifies that the rotation values are driven by the animation. Now, click and drag along the timeline to move the scrubber, and position the scrubber at frame 0:00 (see Figure 17.8).

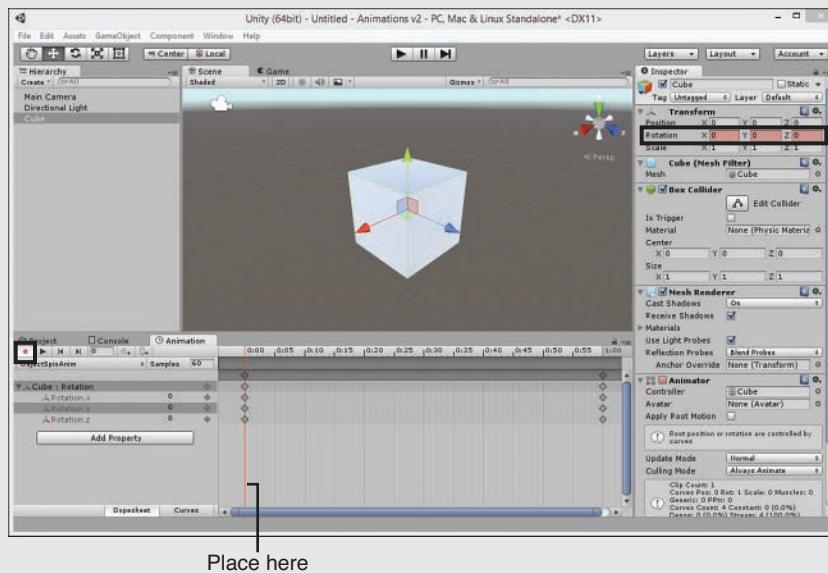


FIGURE 17.8
Getting ready to record.

4. In the Inspector view, locate the **CubeColor** material on the cube and change its **Albedo** color to red. Notice that a new property and key frame has been added to the Animation window. If you expand the new property you will see the r, g, b, and a values of the color at that key frame (1, 0, 0, 1).

5. Move the scrubber further down the timeline and change the color in the Inspector again. Repeat this step as many times as you'd like. If you'd like the animation to loop smoothly, be sure that the last key frame (at position 1:00 so it synchronizes with the rotation as well) is the same color as the first key frame (see Figure 17.9).

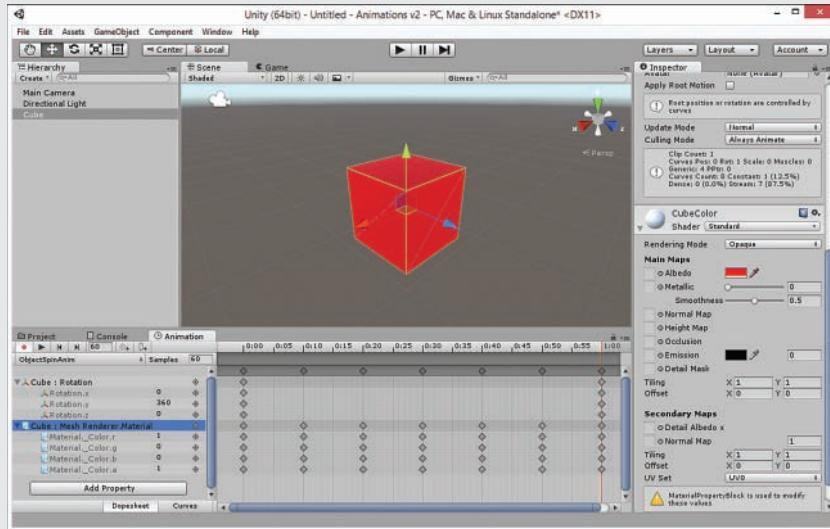


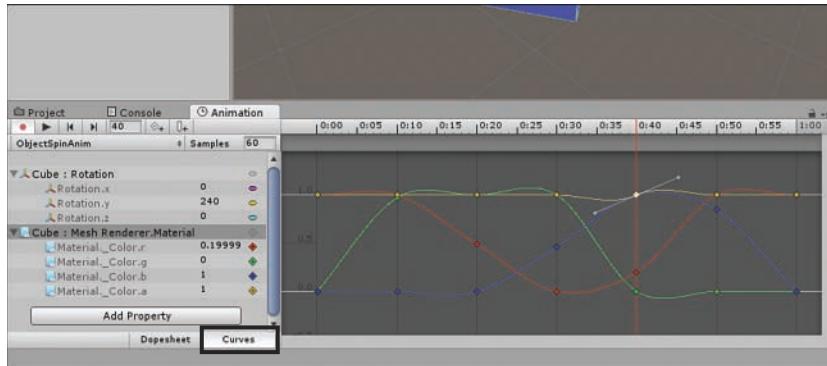
FIGURE 17.9

Recording the color values.

6. Play your scene to see your animation run in the Game view. Note that playing a scene will bring you out of record mode (which is handy since we are done using it). You should now have a spinning multicolored cube in your scene!

The Curves Editor

The last tool we are going to look at is the Curves editor. So far, we have been in what's called "dopesheet" view. This is the view with the key frames listed out and itemized in a flat fashion. You may have noticed that while we drive the values of the key frames, we don't control the values of the normal frames. If you wondered how those values are determined, wonder no more. Unity blends the values (often called "interpolation") between key frames for you to create smooth transitions. In the Curves editor you can see what that looks like. To enter the Curves editor, simply click the button labeled **Curves** at the bottom of the Animation view (see Figure 17.10).

**FIGURE 17.10**

The Curves editor.

In this mode, you can toggle which values you want to see by clicking their properties on the left. Here, you can see exactly how values are being transitioned between the key frames. You can drag a key frame to change its value, or even double click on the curve to create a new key frame at that point. If you don't like how Unity is generating the value in between the key frames, you can right click on a key frame and choose **Free Smooth**. This will create two handles which you can then move to change how values are smoothed. Feel free to play around and see what sort of craziness you can create with the curves.

Summary

In this hour you were introduced to animations in Unity. You started by looking at the basics of animations. You learned about animations and rigging. From there, you learned about the various types of animations in Unity. After that you began creating animations. You started with 2D animations and then created custom animations both manually and with the Record Mode tool.

Q&A

Q. Can animations be blended together?

A. Yes, they can. This is covered in the next hour with the new Unity Mecanim system.

Q. Can any animation be applied to any model?

A. Only if they are rigged exactly the same or if the animation is a simple one that doesn't require a rig. Otherwise, the animations may behave very strangely or just do not work at all.

Q. Can a model be rerigged in Unity?

A. Yes, and you see how in the next hour.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. The “skeleton” of a model is known as what?
2. Which animation type flips through images quickly?
3. What do we call frames of animation that have an explicit value?

Answers

1. The rig or rigging.
2. 2D animation.
3. Key frames.

Exercise

This exercise is more of a “sandbox” exercise. Feel free to take some time and get used to creating animations. This is a very powerful toolset and it certainly pays to be familiar with it. Try completing the following:

- ▶ Make an object flying around a scene in a large arc.
- ▶ Make an object flicker by cycling its renderer on and off.
- ▶ Change the properties of an object’s scale and material to make it appear to “morph.”

This page intentionally left blank

HOUR 18

Animators

What You'll Learn in This Hour:

- ▶ The basics of animators
- ▶ How to use the animator's state machine
- ▶ Controlling animations from script via parameters
- ▶ An introduction to blend trees

In this hour, you take what you learned about animations previously and put it to use with Unity's Mecanim animation system and animators. You start by learning about Animators and how they work. From there, you look at how to rig, or change the rigging of, models in Unity. After that, you create an animator and configure it. Finally, we will see how animations are blended together to produce amazingly realistic results.

NOTE

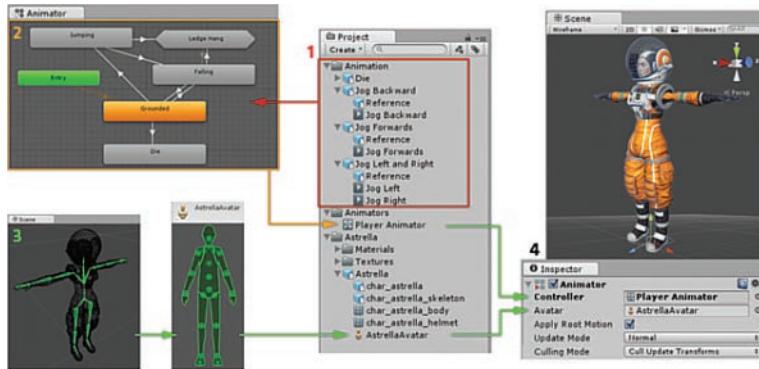
Warning: Get Your Hands Dirty!

This section is like one big Try It Yourself. Make sure you save your project in-between the practical exercises, as each builds on the one before. You will certainly want to be in front of your computer while you read this section: the topic is best learnt by doing!

Animator Basics

All animation in Unity starts with an Animator component. In Hour 17, while you were creating and learning about animations, you were still using animators without really knowing it. At its heart, Unity's animation system (Mecanim) comprises three pieces: the animation clip, an animator controller, and an animator component. These three pieces all exist to make our characters come to life.

Figure 18.1 is taken from Unity's online documentation about the Animator (<http://docs.unity3d.com/Manual/class-Animator.html>), and shows how these parts relate to one another.

**FIGURE 18.1**

How the parts of a humanoid animation relate.

The animation clips (1) are the various motions that you either import or create in Unity. The Animator Controller (2) contains our animations and determines which clips are playing at any given moment. Complex models have something called an Avatar (3). This Avatar acts as the “translator” between the animator controller and the rigging of the model. We generally can ignore the avatar because it is set up and used automatically. Finally, both the animator controller (which we can just call the “controller”) and the avatar are put together on the model using an Animator Component (4). Seem like a lot to remember? Don’t worry. Most of this stuff is either intuitive or automatic.

One of the best things about Unity’s animation system is that you can “re-target” an animation onto other game objects. If you animate a cube, you can apply this to a sphere. If you animate a character, you can apply the animation to another character with the same rigging (or different rigging, as we will see soon). This means you can have an orc and a man doing the same happy dance together!

NOTE

Specific Case

In order to get the most out of this hour, we are going to pursue a very specific use case. That is, we will be using 3D animations on a humanoid model (a very common use case to be sure). We are doing this because it will allow us to learn about 3D animations, importing models and animations, working with rigs, and using Unity’s awesome humanoid retargeting system. Just remember, with the exception of humanoid retargeting, everything covered in this hour applies completely to any other type of animation. So, if you are building a multipart 2D animation system, all the knowledge learned here still matters.

Rigging Revisited

In order to begin building a complex animation system, we first need to ensure our model's rigging is prepared. If you recall from the preceding hour, models and animations have to be rigged exactly the same way to function. This means that getting animations made for one model to work on a different model is very difficult. Therefore, animations and models are generally made specifically to work together.

If you are using a humanoid model though (two arms, two legs, head, and torso), you have the ability to tap into Mecanim's animation retargeting tools. With the Mecanim system, humanoids can have their rigging remapped in the editor without using any 3D modeling tools. The result is that any animation made for a humanoid model can work with any other humanoid you have. Now animators (the people, not the Unity asset) can produce large libraries of animations that can be applied to a large range of models using many different rigs.

Importing a Model

For this hour, we will use Ethan, a model from the Characters standard asset pack. This model comes with a lot of different items, and you will go through each piece to ensure that it is configured properly. To import the model, click **Assets > Import Package > Characters**. Leave everything checked, and click **Import**.

Now go ahead and find Ethan in your Project tab under **Assets > Standard Assets > Characters > ThirdPersonCharacter > Models** (see Figure 18.2).

Notice this model is a .fbx file; this is a common 3D export format. Unity also supports .dae (Collada), .3DS, .dxf, and .obj files. These export formats can be compact, and allow you to import from a wide range of 3D packages. However, if you edit the file, the changes won't apply immediately in Unity, which can slow your prototyping down.

Unity also directly supports the files of a few of the most common 3D applications. This includes Max, Maya, Blender, Cinema3D, Modo, Lightwave, and Cheetah3D. This allows for rapid prototyping, because if you save your file in the third party software, Unity immediately updates. The disadvantage, however, is that you require a licensed copy of the software for the import to work (which may cost money). Also, the files may be larger than needed, as they contain everything the parent program creates, not just what is required to import into Unity.

If you click the little arrow, you can expand out the model to see all the constituent parts (see Figure 18.3). How these are structured will depend on how the model was exported from whatever 3D application it was made in.

These components are, from left to right: Ethan's body with texture, the textured glasses, the definition of the skeleton, the raw EthanBody mesh, the raw EthanGlass mesh, and finally a definition of Ethan's "avatar" which is used for rigging.

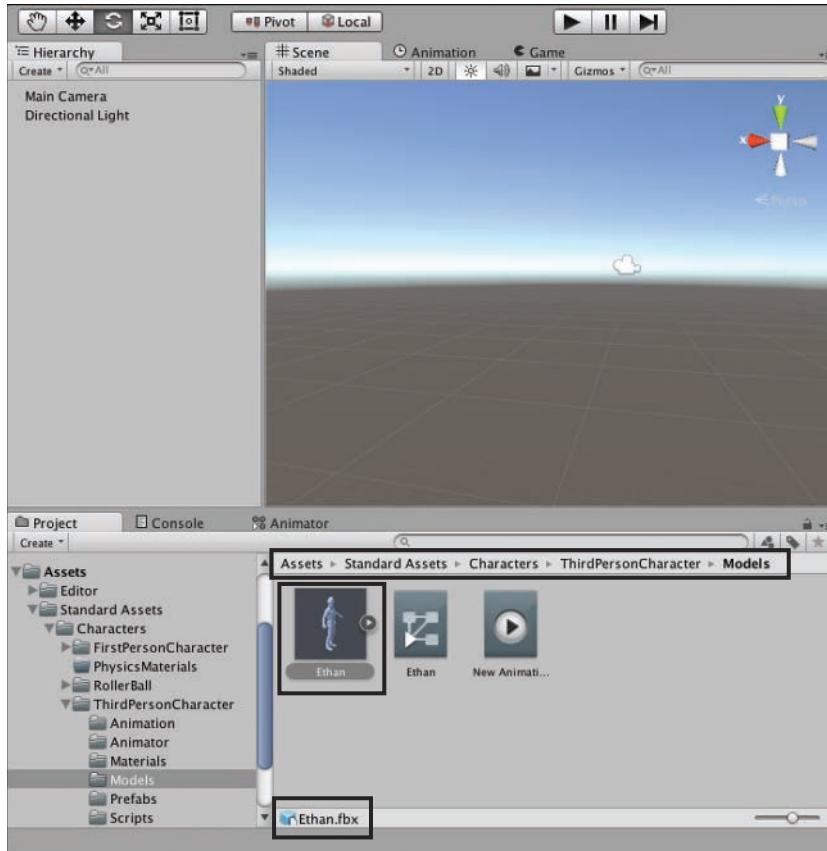


FIGURE 18.2
Finding the Ethan model.



FIGURE 18.3
The components of the imported 3D model.

NOTE

Previewing Meshes

If you click on either of the Ethan or Glasses models in the tray, you should notice a small preview window at the bottom of the inspector (if not drag it up to show it). Here you can rotate that sub-model around to take a look at it from all angles (see the bottom of Figure 18.4).

Now click on the main asset, the Ethan to the left of the little tray arrow. Cast your eye over to the inspector where you will see the model import settings (see Figure 18.4).

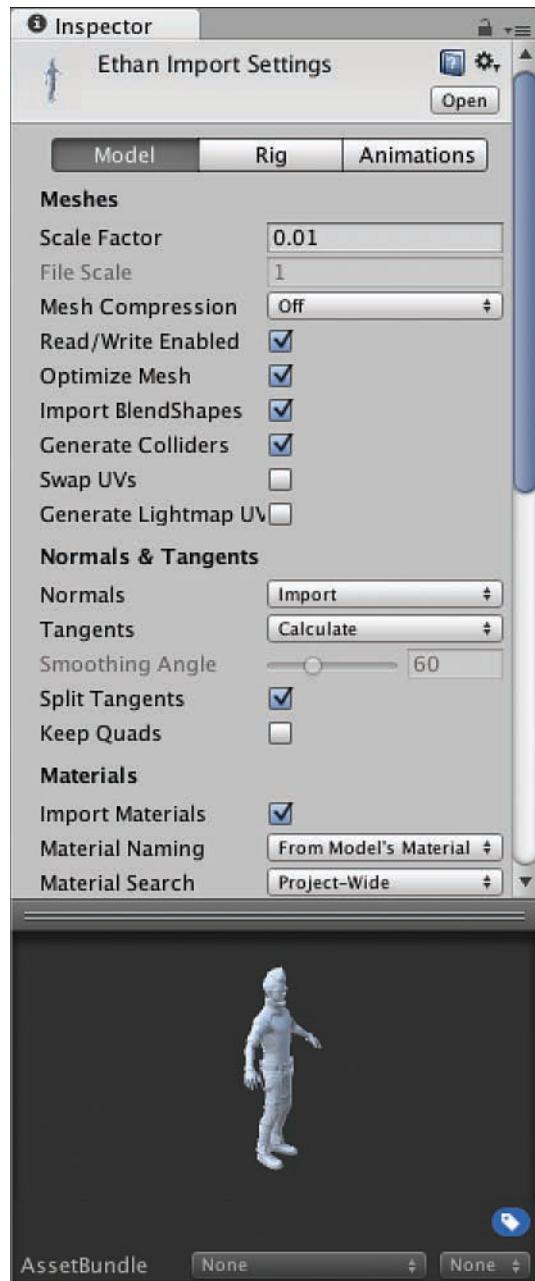


FIGURE 18.4

The model inspector view.

The Model tab is responsible for all of the settings that dictate how the model itself is imported into Unity. These items can be safely ignored for the purposes of this hour. The tab you are concerned with for now is the Rig tab. The Animations tab will be covered a little later.

Configuring Your Assets

Now that we have a model and animations imported (they came with the rest of the assets), we need to configure them. The process for configuring our models and animations to work together is fairly identical.

Rig Preparation

We configure a model's rig in the import settings, under the **Rig** tab in the Inspector view. The property we are most concerned with here is the **Animation Type**. There are currently four types available to us in the drop down: None, Legacy, Generic, and Humanoid. A value of "None" means that Unity will ignore this model's rig. Legacy is for Unity's old animation system and shouldn't be used. Generic is for all nonhumanoid models (simple models, vehicles, buildings, animals, etc.), and all models import into Unity default to this Animation Type. Finally, Humanoid (which is the one we will be using) is for all humanoid characters. This setting is what allows Unity to retarget animations for us.

As we can see, Ethan is already set up properly as a humanoid. When you set a model as humanoid, Unity will automatically go through the process of mapping the rig for you. If you'd like to see how easy this is, you can simply change the Animation Type to Generic, press apply, and then change it back (which is exactly how this model was set up for you originally, no extra work was hidden). If you'd like to see the work that Unity did for you, you can enter the rigging tool by clicking the Configure button (see Figure 18.5).

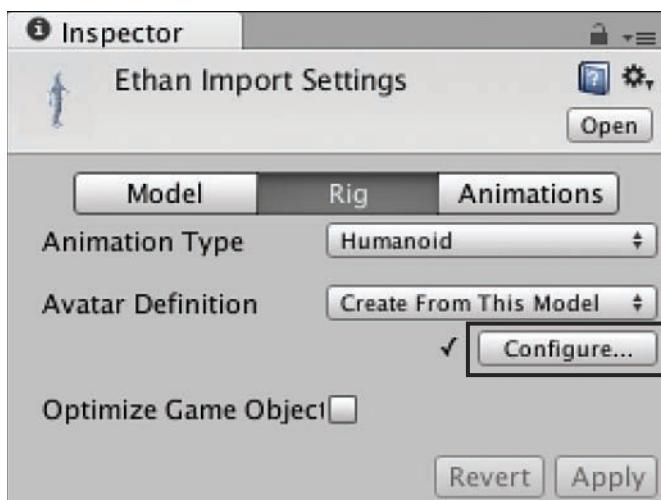


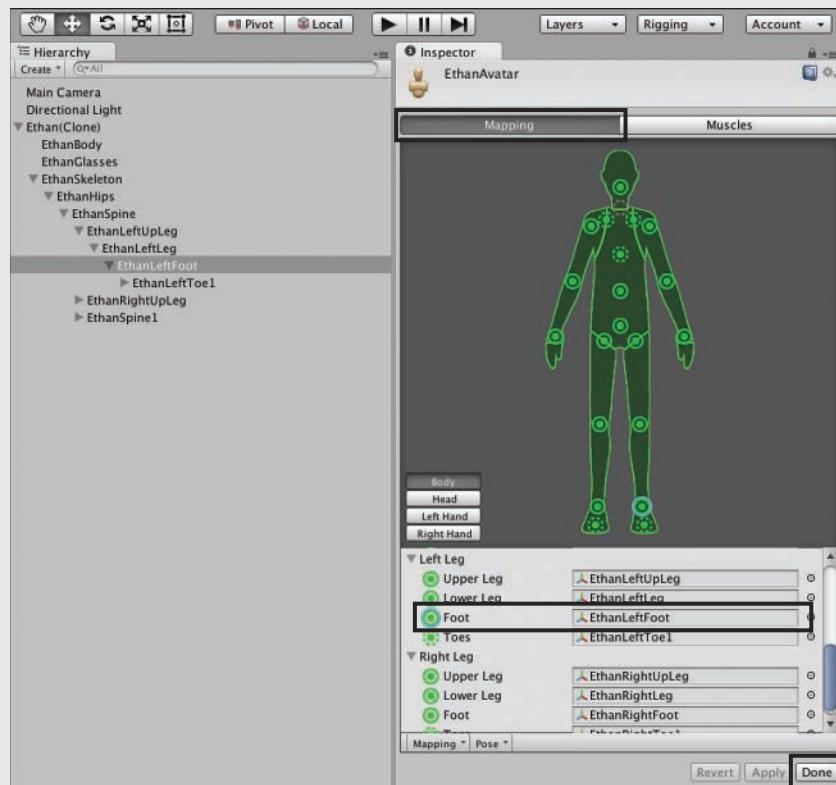
FIGURE 18.5
The rig settings.

TRY IT YOURSELF ▼

Explore How Ethan Is Rigged

Let's take a look at how Ethan is "rigged." This will give you a much more practical idea of how a rigged model is assembled.

1. Click **Configure** on the Rig tab of the **Ethan.fbx** Inspector, as described above. Doing so will launch you into a new scene, so save your old one if prompted.
2. Rearrange your interface so that you can see mainly the Hierarchy and Inspector. Hour 1 covered how to close and move tabs. You may want to save this layout. You can always go back to Default later.
3. With the Mapping tab selected, click on various green circles (see Figure 18.6).
4. Notice how this highlights the corresponding child of EthanSkeleton in the Hierarchy, and puts a blue circle around the corresponding skeleton point below the outline. Notice all of

**FIGURE 18.6**

The rigging view with the left foot selected.



the extra points of the rig in the Hierarchy view. Those pieces aren't important for humanoids and thus won't be retargeted. Don't worry though, they still play a part in ensuring the model looks correct when moving.

5. Continue to explore other body parts by clicking Body, Head, Left Hand, etc. These are all joints, which can be completely retargeted for any humanoid.
6. Click **Done** when you are finished. Notice the temporary Ethan(Clone) disappears from the Hierarchy.

At this point, you have previewed Ethan, and seen how his skeleton is arranged. He is now ready to go!

Animation Preparation

For this hour, we could use the animations that came with Ethan, but that would be boring and wouldn't illustrate the flexibility of the Mecanim system. Instead, we are going to use some other animations provided in the book files for Hour 18. Each animation has a series of options that control how it behaves that must be specifically configured the way you want it. For example, you need to ensure that the walking animation loops appropriately so that transitions don't have any obvious seams. In this section, you go through each animation and prepare it.

Start by dragging the Animations folder from the book assets into the Unity editor. There are three animations that you will be working with: Idles, WalkForward, and WalkForwardTurns. Also, each of these three animations needs to be set up uniquely. If you look in the animations folder, you will see that the animations are actually .fbx files. This is because the animations themselves are located inside their default models. Don't worry, though; we will be able to modify and extract them inside Unity.

Idle Animation

To set up the idle animation, follow these steps (see Table 18.1 for an explanation of the settings):

1. Select the **Idles.fbx** file in the Animations folder. In the Inspector, select the **Rig** tab. Change the animation type to **Humanoid** and click apply. This tells Unity that the animation is also for a humanoid.
2. Once the rig is configured, click the **Animations** tab in the Inspector. Set the Start frame to 128 and check the boxes next to **Loop Time** and **Loop Pose**. Additionally, check the box **Bake Into Pose** for all of the **Root Transform** properties. Ensure that your settings match the ones in Figure 18.7.
3. Click **Apply** at the bottom of the Inspector. The animation itself should now be properly configured. You can find it by expanding the Idles.fbx file (see Figure 18.8). Be sure to remember how to access that animation. The model itself is irrelevant. It's the animation you want.

TABLE 18.1 Important Animation Settings

Setting	Description
Loop Time	Whether or not the animation loops.
Root Transform	The Root Transform settings have three parts: Rotation, Position (Y), and Position (XZ). These control whether or not the animation is allowed to change an object's rotation, vertical position (y axis), and horizontal position (x/z plane), respectively.
Bake Into Pose	Whether or not the animation is allowed to actually move an object. Checking this box means that the animation will not actually change the object and instead will just appear to.
Offset	The offset allows us to specify some amount to modify the original position of the animation. For instance, modifying the offset under Root Transform Rotation will cause us to make small rotation changes to the model along the y axis. This is useful for correcting any motion error that is in the animation.

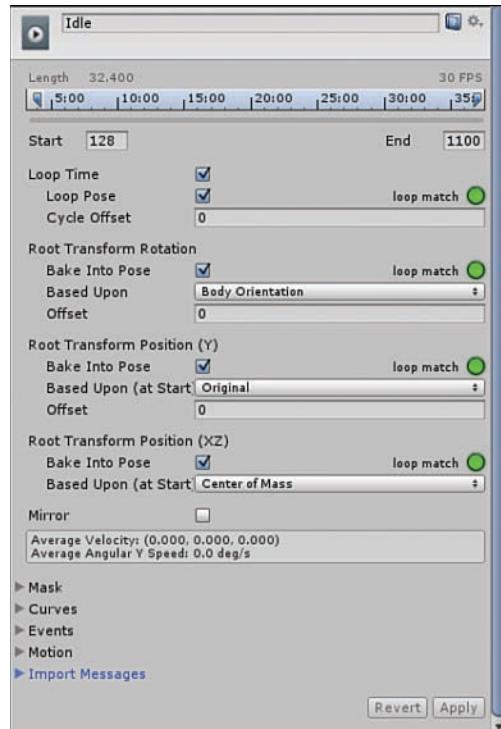


FIGURE 18.7
The Idle animation.



FIGURE 18.8

NOTE

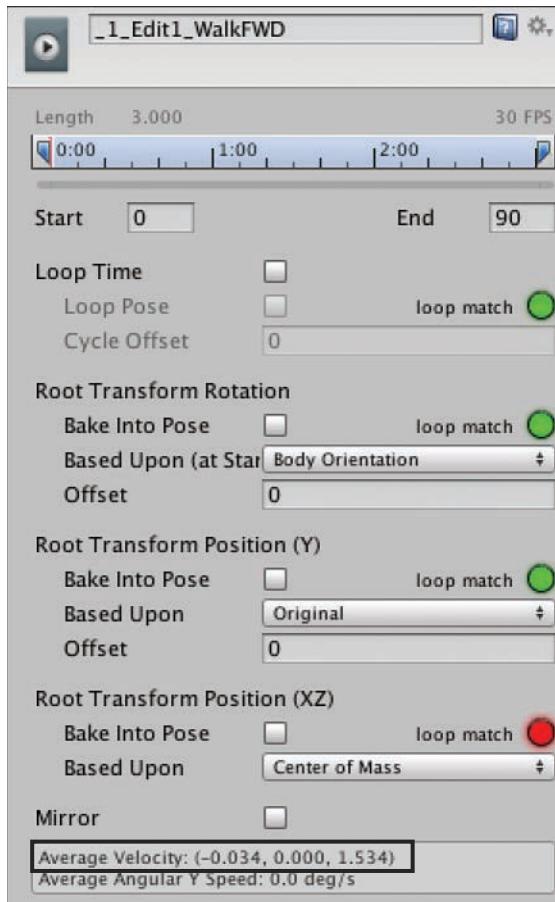
Red Light, Green Light

You might have noticed the green circles present in the animation settings (as in Figure 18.7). Those are nifty little tools that are used to designate whether your animations are lined up. The fact that the circles are green means that they will loop seamlessly. If any circles had been yellow, it would have indicated that the animation came close to looping seamlessly but there was a minor difference. A red circle indicates that the beginning and end of the animation don't line up at all and a seam would be very apparent. If you have an animation that doesn't line up, you can change the Start and End properties to find a segment of the animation that does.

Walk Animation

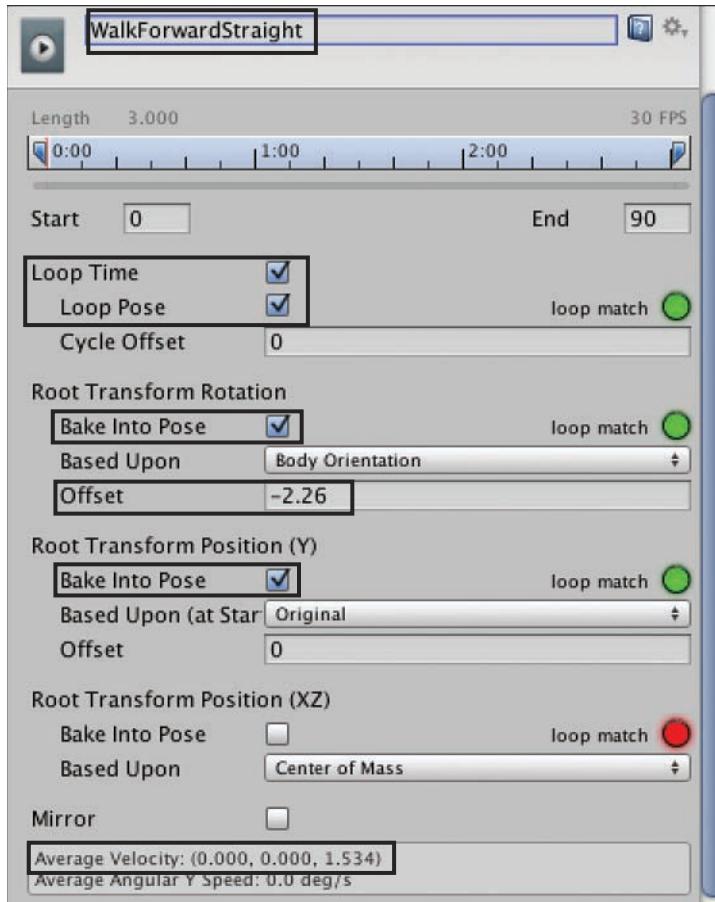
To set up the walking animation, follow these steps:

1. Select the **WalkForward.fbx** file in the Animations folder and complete the rigging the same way you did for the idle animation.
 2. Under the Animations tab, you should have the settings demonstrated in Figure 18.9. You should note two things. First, the Root Transform Position (XZ) has a red circle next to it. This is good. What this means is that at the end of the animation the model is in a different x and z axis position. Because this is a walking animation, that is the behavior that you want. The other thing you should notice is the Average Velocity indicator. You should notice an x-axis velocity of -0.034 and a z-axis velocity of 1.534. The z-axis velocity is good because you want the model moving forward, but the x-axis velocity is a problem because it will cause the model to drift sideways while walking. You need to adjust this setting.

**FIGURE 18.9**

The walk animation settings.

3. The clip currently has a very ugly name, change this to **WalkForwardStraight** by clicking the name, and changing it.
4. To adjust the x-axis velocity, you need to check the **Bake into Pose** properties for both the Root Transform Rotation and Root Transform Position (Y) properties. You also need to set the Root Transform Rotation offset to **-2.26** so that the animation loops properly.
5. Finally, you want to check the **Loop Time** and **Loop Pose** properties. Figure 18.10 contains the fixed settings. When done, click the **Apply** button.

**FIGURE 18.10**

The fixed walk animation settings.

Walk Turn Animation

The walk turn animation allows the model to smoothly change direction while walking forward. This one differs a little from the other two because you need to make two animations out of a single animation recording. This sounds trickier than it really is. The steps to complete this are as follows:

1. Select the **WalkForwardTurns.fbx** file in the Animations folder and complete the rigging the same way you did for the idle animation.

2. By default, there will be a long animation with the name `_7_a_U1_M_P_WalkForwardTurnRight`. You could modify that, but it will be easier to just delete it and start over. Type **WalkForwardTurnRight** into the Clip Name text field, and then click the plus sign (+) to create a new clip (see Figure 18.11). You may need to click on the Clips list to get the name to change.

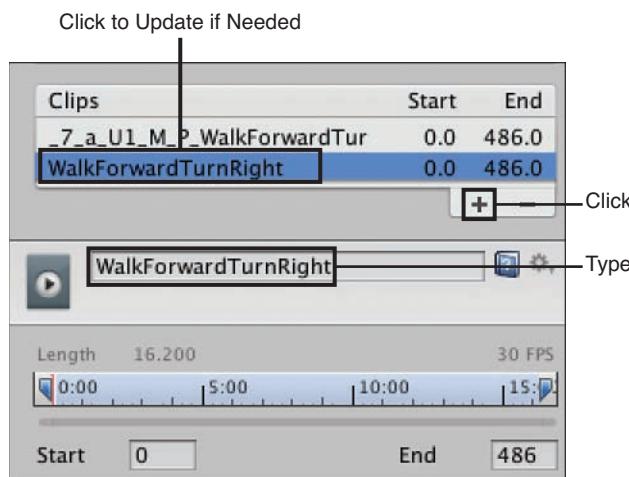
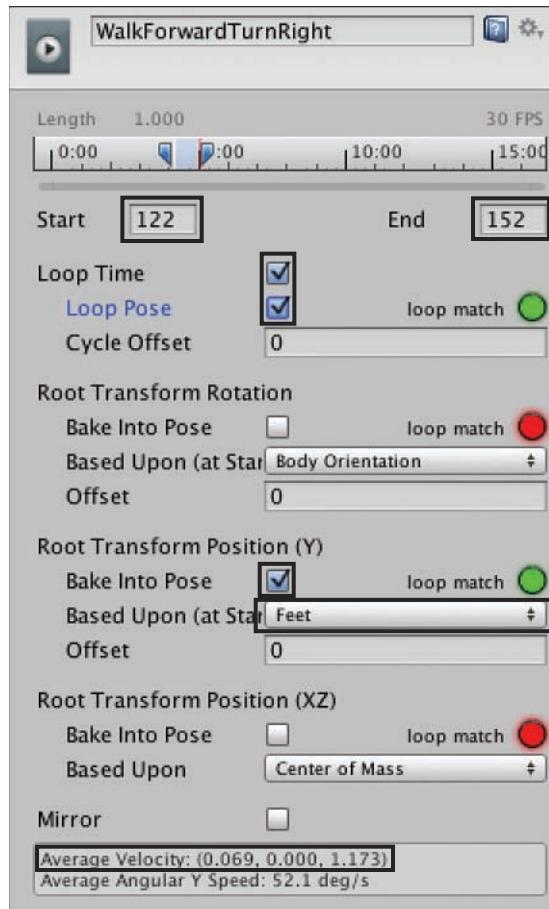


FIGURE 18.11

Adding an animation clip.

3. Now you can remove the old animation clip. Select `_7_a_U1_M_P_WalkForwardTurnRight` and click the minus sign (-) to remove it.
4. With the `WalkForwardTurnRight` clip selected, set the properties to match Figure 18.12 (the Start is 122, and the end is 152). This will cut the clip down and ensure that it only contains the model moving in a rightward circle. (Be sure to preview it to see what it looks like.) After you have done this, click **Apply**.
5. Create a `WalkForwardTurnLeft` animation clip the same way you made the right turning clip in step 2. The properties for the `WalkForwardTurnLeft` clip will be exactly the same as the `WalkForwardTurnRight` clip except that you need to put a check in the **Mirror** property (see Figure 18.13). Remember to click **Apply**.

At this point, all the animations are set up and ready to go. Now all that's left to do is build the animator.

**FIGURE 18.12**

The right turn settings.

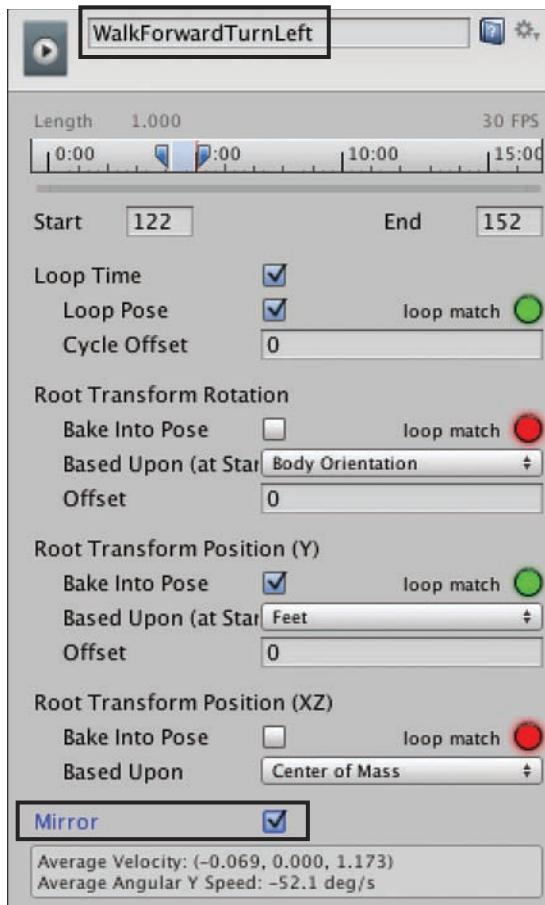


FIGURE 18.13

Mirroring the animation.

Creating an Animator

Animators in Unity are assets. This means that they are a part of a project and exist outside of any one scene. This is nice because it allows easy reuse over and over again. To add an animator to your project, in Project view right-click a folder and select **Create > Animator Controller** (don't do that just yet).

TRY IT YOURSELF

Setting Up the Scene

In this exercise, you set up a scene and prepare for the rest of the materials for the hour. Be sure to save the scene created here, because you'll need it later:

1. If you have not done so already, create a new project and complete the model and animation preparation steps in the previous section.
2. Drag the Ethan model into your scene (from **Assets > Standard Assets > Characters > ThirdPersonCharacter > Models**) and give it a position of (0, 0, -5).
3. Nest the Main Camera under the Ethan model (in the Hierarchy view, drag the Main Camera onto the model) and position the camera at (0, 1.5, -1.5) with a rotation of (20, 0, 0).
4. In your Project view, create a new folder named **Animators**. Right-click the new folder and select **Create > Animator Controller**. Name the animator **PlayerAnimator**. With Ethan selected in the scene, drag the animator onto the **Controller** property of the Animator component in the Inspector (see Figure 18.14).

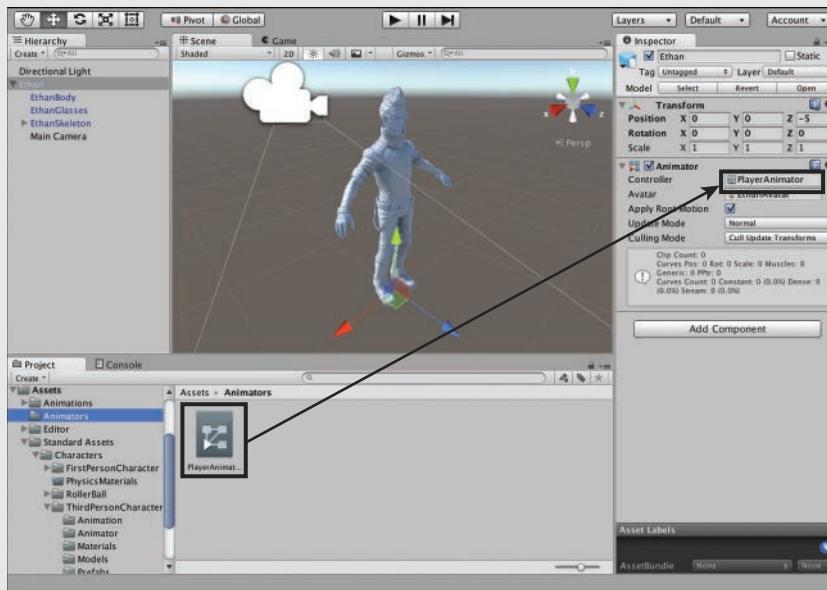
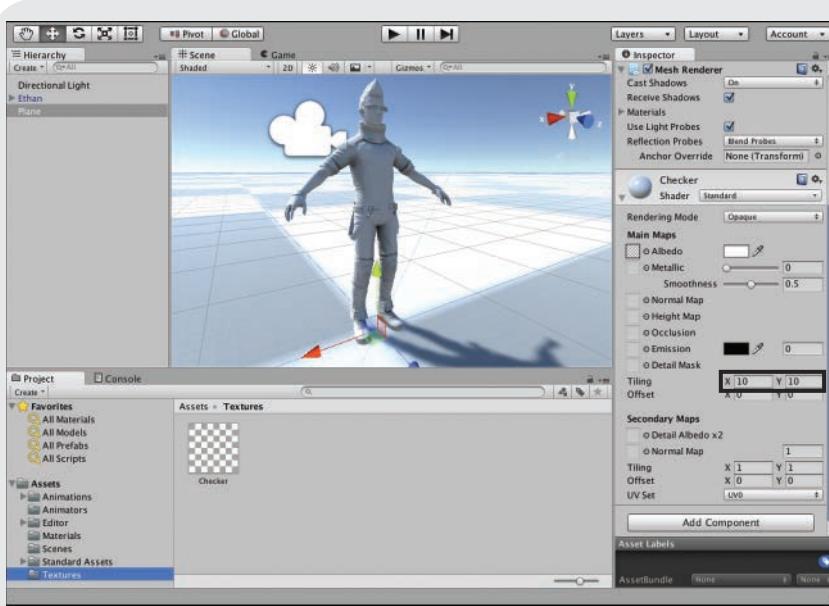


FIGURE 18.14

Adding the animator to the model.

5. Add a plane to your scene. Position the plane at (0, 0, -5) with a scale of (50, 1, 50). Locate the file **Checker.tga** in the book assets for Hour 18 and import it into your project, in a new folder called **Textures**. Apply the texture to the plane, and set the X and Y tiling to 10 (see Figure 18.15).

**FIGURE 18.15**

Setting the tiling of the checker texture.

6. Run the scene and make sure that everything looks correct. Note that at this point nothing is animated. This is a good time to save your scene as **Main**.

The Animator View

Double-clicking an animator brings up the Animator view. This view functions like a flow graph, allowing you to visually create animation paths and blending. This is the real power of the Mecanim system.

Figure 18.16 shows the basic Animator view. You can move around the Animator view by dragging with the middle mouse button held down. For a new animator, this is very plain. There is only a base layer, no parameters, and an Any State. These will soon be discussed more fully.

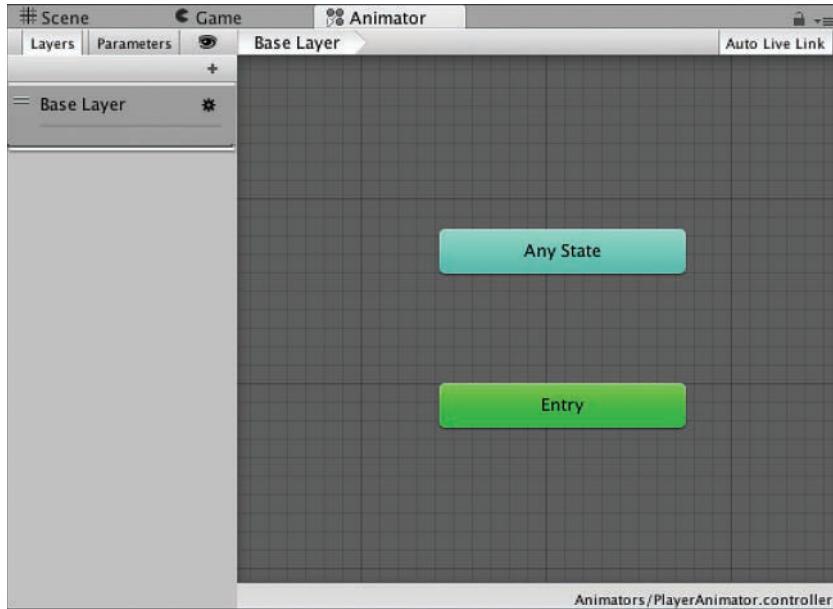
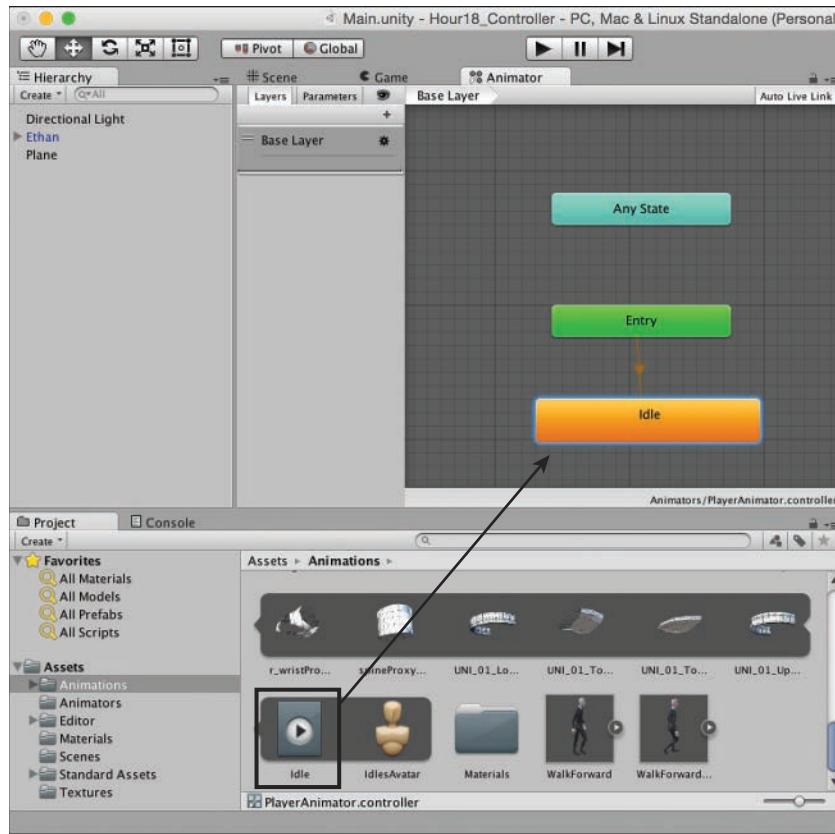


FIGURE 18.16
The Animator view.

The Idle Animation

The first animation you want to apply to Ethan is the idle animation. Now that the entire long set up process is complete, adding this animation is simple. You need to locate the Idle animation clip, which is stored inside the Idles.fbx file (see Figure 18.8 earlier in the hour), and drag it onto the animator in the Animator view (see Figure 18.17).

**FIGURE 18.17**

Applying the idle animation.

You should now be able to run your scene and see the Ethan model looping through the idle animation.

Parameters

Parameters are like variables for the animator. You set them up in the animator view and then manipulate them with scripts. These parameters control when animations are transitioned and blended. To create a parameter, simply click the plus sign (+) in the Parameters tab in the Animator view.

TRY IT YOURSELF

Adding Parameters

In this exercise, you add two parameters. This exercise builds off of the project and scene you have been working on thus far this hour:

1. Make sure that you've completed all of the steps up to this point.
2. In the Animator view, click the plus sign (+) to create a new parameter. Choose a **Float** parameter and name it **Speed** (see Figure 18.18).

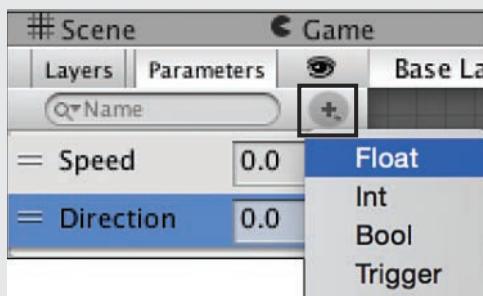


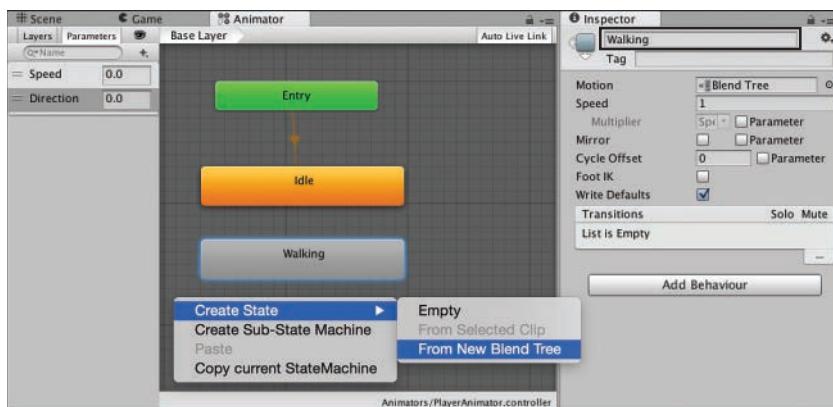
FIGURE 18.18
Adding parameters.

3. Repeat step 2 to create a parameter named **Direction**.

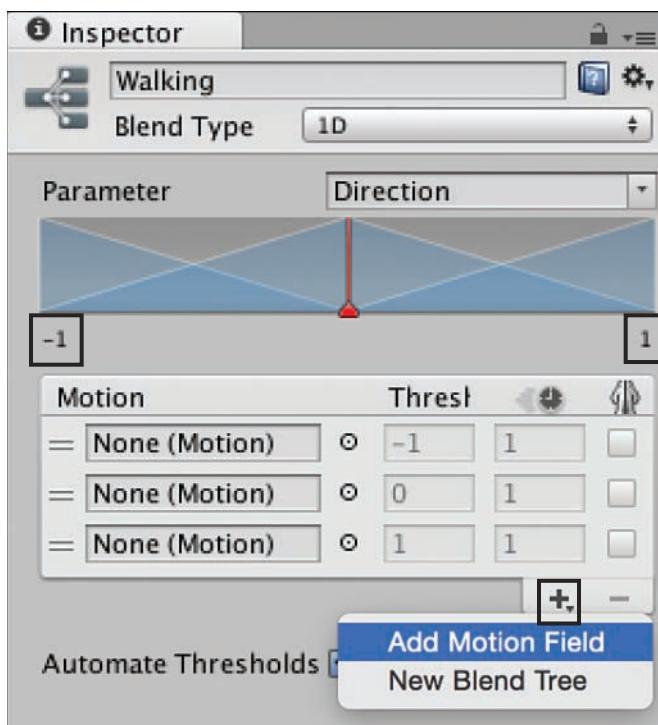
States and Blend Trees

Your next step is to create a new state. States are essentially statuses that the model is currently in that define what animation is playing. The model Ethan will have two states: Idle and Walking. Idle is already in place. Because the walking state can be any of three animations, you want to create a state that uses a blend tree. A blend tree will seamlessly blend one or more animations together based on some parameter. To create a new state, follow these steps:

1. Right-click a blank spot in the Animator view and select **Create State > from New Blend Tree**. In the Inspector view, name the new state **Walking** (see Figure 18.19).
2. Double-click the new state to expand it. In the Inspector, change the Parameter property to **Direction** and add three motions by clicking the plus sign (+) under motions and selecting **Add Motion Field**. Under the graph, set the minimum value to -1 and the maximum value to +1 (see Figure 18.20).

**FIGURE 18.19**

Creating and naming a new state.

**FIGURE 18.20**

Adding motion fields.

3. Now drag each of the three walking animations into the three motion fields. Remember that the turning animation clips are located under WalkForwardTurns.fbx, and the straight walking animation is under WalkForward.fbx. Make sure that they are in the order: Turn Left, Straight, Turn Right (see Figure 18.21).

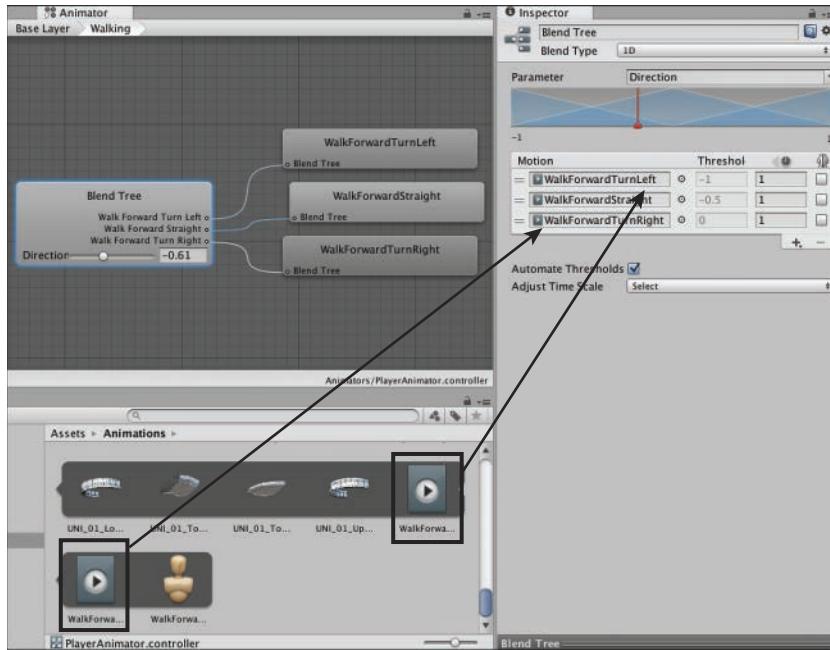


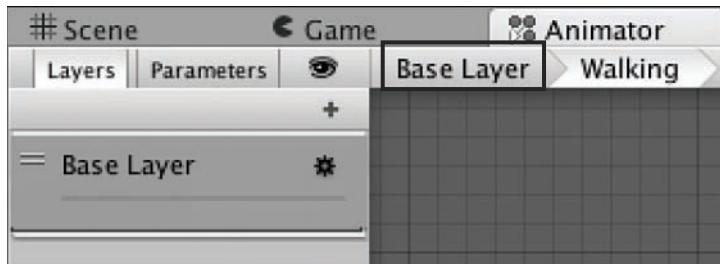
FIGURE 18.21
Changing minimum values and adding animations to a blend tree.

Your walking animation is now ready to blend based on the direction parameter. You can get out of the expanded view by clicking the Base Layer breadcrumb at the top of the animator view (see Figure 18.22).

Transitions

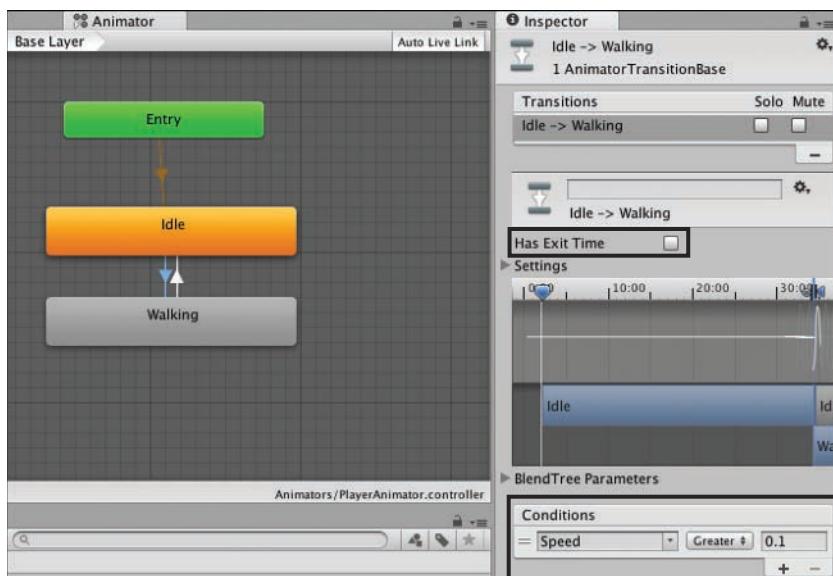
The last thing you need to do to ensure that your animator is finished is to tell the animator how to transition between the idle and walking animations. You need to set up two transitions. One transitions the animator from idle to walking, and the other transitions back. To create a transition, follow these steps:

1. Right-click the **Idle** state and select **Make Transition**. This will create a white line that follows your mouse. Click the **Walking** state to connect the two.

**FIGURE 18.22**

Navigating the Animator view.

2. Repeat step 1, except this time connecting the Walking state to the Idle state.
3. Edit the Idle to Walking transition by clicking the white arrow on it. Set the Conditions to be **Speed Greater** than the value **.1** (see Figure 18.23). Do the same for the Walking to Idle transition, except set the condition to **Speed Less Than** the value **.1**.
4. Uncheck the Has Exit Time box, which will allow the Idle animation to be interrupted when the walk key is pressed.

**FIGURE 18.23**

Modifying transitions.

The animator is finished. You might notice that when you run the scene there aren't any working movement animations. This is because the speed and direction parameters are never changed. In the next section, you learn how to change these through scripting.

Scripting Animators

Now that everything has been set up with the model, the rigging, the animations, the animator, the transitions, and the blend tree, it is finally time to make the whole thing interactive. Luckily, the actual scripting components are simple. Most of the hard work was already done in the editor. At this point, all you need to do is manipulate the parameters you created in the animator to get Ethan up and running. Because the parameters you set up were of type `float`, you need to call the animator method:

```
SetFloat (<name> , <value>);
```

TRY IT YOURSELF

The Final Touches

This exercise takes the project you have been working on during this hour and adds the scripted component to make it all work:

1. Create a new folder called **Scripts** and add a new script to it. Name the script **AnimationControl**. Attach the script to the Ethan model in the scene—this step is important!
2. Add the following code to the script:

```
private Animator anim;

void Start () {
    //Get a reference to the animator
    anim = GetComponent<Animator> ();
}

void Update () {
    anim.SetFloat ("Speed", Input.GetAxis ("Vertical"));
    anim.SetFloat ("Direction", Input.GetAxis( "Horizontal"));
}
```

3. Run the scene and notice that the animations are controlled with the vertical and horizontal axes.

That's it! If you run your scene after adding this script, you might notice something strange. Not only does Ethan animate through idle, walking, and turning, but the model also moves. This is due to two factors.

The first is that the animations chosen have a built-in movement to them. This was done by the animators outside of Unity. If this hadn't been done, you would have to program the movement yourself.

The second factor is that by default the animator allows the animation to move the model. This can be changed in the Apply Root Motion property of the Animator component (see Figure 18.24), but in this case it will cause some strange effects!

We have provided our final project file in the book files if you want to compare notes.

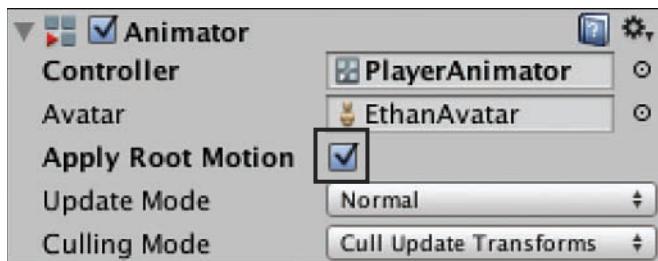


FIGURE 18.24
Root motion animator property.

Summary

You started this chapter by building a very simple animation from scratch. We started with a cube, and added an Animator component. We then created an Animator Controller, and linked it to the Animator. We then went onto create two animation states and corresponding motion clips. Finally, we showed you how to make the states blend together.

Q&A

Q. Can you do keyframe animations on humanoids in Unity?

A. Unity doesn't allow keyframe animation on humanoids, and whereas you may find workarounds on the Internet, you are better off creating humanoid animations in a dedicated 3D package and importing the result to Unity.

Q. Can an object be moved by both the Animator and the physics engine?

A. Whereas this is possible with some care, generally you want to avoid trying to mix the two. At least at any one time, you want to be clear on whether the animator or the physics engine is controlling a game object.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. To what must an Animator component have a reference to work?
2. What color is the default animation state in the Animator tab?
3. How many motion clips (motions) can an animation state have?
4. What do you use to trigger animation transitions from script?

Answers

1. An Animator Controller must be created and connected to the Animator component. In the case of humanoid characters, an Avatar is also required.
2. Orange.
3. This depends; an animation state can be single clip, a blend tree, or another state machine.
4. Animator parameters.

Exercise

There is a lot of information required to produce a robust and high-quality animation system. In this hour, you got to see one way and one group of settings to achieve this. Plenty of other assets are available, however, and learning is paramount to success.

Your exercise for this hour is to continue studying the Mecanim system. Be sure to start by browsing Unity's documentation on the system. You can find this on Unity's website at <http://docs.unity3d.com/Manual/AnimationSection.html>.

You can also explore the fully animated Ethan prefab, found at **Assets > Standard Assets > Characters > ThirdPersonCharacter > Prefabs > ThirdPersonController.prefab**. This prefab has a much more complex Animator, with three blend-trees for Airborne, Grounded, and Crouching states.

HOUR 19

Game 4: *Gauntlet Runner*

What You'll Learn in This Hour:

- ▶ How to design the game *Gauntlet Runner*
- ▶ How to build the *Gauntlet Runner* world
- ▶ How to build the *Gauntlet Runner* entities
- ▶ How to build the *Gauntlet Runner* controls
- ▶ How to further improve *Gauntlet Runner*

Let's make a game! In this hour, you make a 3D gauntlet running game appropriately titled *Gauntlet Runner*. You start the hour off with the design of the game. From there, you focus on building the world. Once done, you build the entities and controls. You wrap the hour up by playing the game and seeing where improvements can be found.

TIP

Completed Project

Be sure to follow along in this hour to build the complete game project. If you get stuck, you can find a complete copy of the game in the book assets for Hour 19. Take a look at it if you need help or inspiration.

Design

You have already learned what the design elements are in Hour 7, "Game 1: *Amazing Racer*." This time you get right into them.

The Concept

In this game, you will be playing as a robot running as far as possible through a gauntlet tunnel, attempting to grab power ups to extend your game time. You need to avoid obstacles that will slow you down. The game ends when you run out of time.

The Rules

The rules of this game state how to play, but also allude to some of the properties of the objects. The rules for *Gauntlet Runner* are as follows:

- ▶ Players can move left or right and jump. They cannot move in any other manner.
- ▶ If players hit an obstacle, they will be slowed by 50% for 1 second.
- ▶ If players grab a power up, their time is extended by 1.5 seconds.
- ▶ Players are bounded by the sides of the tunnel.
- ▶ The loss condition for the game is running out of time.
- ▶ There is no win condition, but players aim to travel as far as possible.

The Requirements

The requirements for this game are simple, as follows:

- ▶ A gauntlet texture.
- ▶ A player model.
- ▶ A power up and obstacle. These will be created in Unity.
- ▶ A game controller. This will be created in Unity.
- ▶ A power up particle effect. This will be created in Unity.
- ▶ Interactive scripts. These will be written in MonoDevelop.

The World

The world for this game will simply be three cubes configured to look like a gauntlet. The entire setup is fairly basic; it's the other components of the game that add challenge and fun.

The Scene

Before setting up the ground with its functionality, get your scene set up and ready to go. To prepare the scene, do the following:

1. Create a new 3D project called **Gauntlet Runner**. Create a new folder called **Scenes** and save your scene as **Main** in that folder.
2. Position the Main Camera at (0, 3, -10.7) with a rotation of (33, 0, 0). Save your scene.

The camera for this game will be in a fixed position hovering over the gameplay. The rest of the world will pass underneath it.

The Ground

The ground in this game will be scrolling in nature; however, unlike the scrolling background used in *Captain Blaster*, you will not actually be scrolling anything. This is explained more in the next section, but for now just understand that you need to create only one ground object to make the scrolling work. The ground itself will consist of three basic cubes and a simple texture. To create the ground, follow these steps:

1. Add a cube to the scene. Name it **Ground** and position it at (0, 0, 15.5) with a scale of (10, .5, 50). Add another cube to the scene named **Wall** and position it at (-5.5, .7, 15.5) with a scale of (1, 1, 50). Duplicate the wall piece and position the new wall items at (5.5, .7, 15.5).
2. Create two new folders: **Textures** and **Materials**. In the book assets for Hour 19, locate the Checker.tga file and drag it into the Textures folder. In the Materials folder, create a new material named **Ground**. Create another called **Wall**.
3. Set the Albedo of **Ground** to be the Checker file that you just imported. Modify the Albedo Color property of the material to give it a color of your choice, we will use green. Apply the material to the ground. You can tweak the material color while viewing it in the Game tab.
4. Finally, set the Albedo of the **Wall** material to a color of your choice, we're using blue. Apply this material both of the wall gameobjects.

That's it! The ground is fairly basic.

Scrolling the Ground

You have seen before that you can scroll a background by creating two instances of that background and moving them in a “leap frog” manner. In this game, you are going to use a more clever solution. Each material has a set of texture offsets. These can be seen in the Inspector when a material is selected. What you want to do is modify those offsets at runtime via a script. If the texture is set to repeat (which it is by default), the texture will loop around seamlessly. The result, if done correctly, is a seemingly scrolling object without any actual movement. To create this effect, follow these steps:

1. Create a new folder named **Scripts**. Create a new script called **Ground**. Attach the script to both the ground and the walls.
2. Add the following code to the script (replacing the `Update()` method that is already there):

```
public float speed = .5f;
private Renderer groundRenderer;
```

```

void Start () {
    groundRenderer = GetComponent<Renderer>();
}

void Update () {
    float offset = Time.time * speed % 1; // The % 1 keeps offset
between0 and 1
    groundRenderer.material.mainTextureOffset = new Vector2(0, -offset);
}

public void SlowDown() {
    speed = speed / 2;
}

public void SpeedUp() {
    speed = speed * 2;
}

```

- Run the scene and notice your gauntlet scrolling. This is an easy and efficient way to create a scrolling 3D object.

You might have noticed the two additional methods in the previous scrip: `SlowDown()` and `SpeedUp()`. These aren't used now, but they will be necessary later when the player hits an obstacle. Figure 19.1 illustrates the running scene set up as described previously.

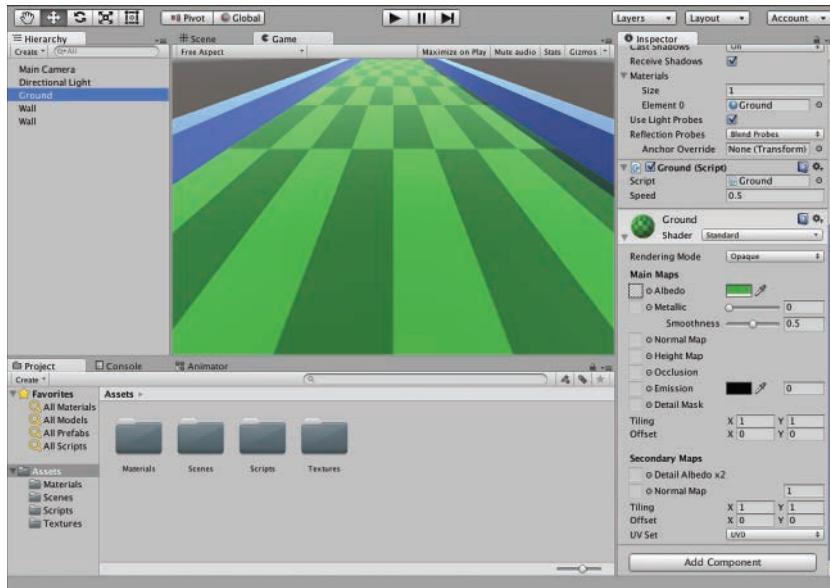


FIGURE 19.1
The running gauntlet.

The Entities

Now that you have a scrolling world, it is time to set up the entities, namely the player, the power ups, the obstacles, and a trigger zone. The trigger zone will be used to clean up any items that make it past the player. You do not need to create a spawn point for this game. Instead, you are going to explore a different way of handling it, by letting the game control create the power ups and obstacles.

The Power Ups

The power ups in this game are going to be simple spheres with some effects added to it. You will be creating the sphere, positioning it, and then making a prefab out of it. To create the power up, follow these steps:

1. Add a sphere to the scene. Position the sphere initially (0, 1, -7.5) so that you can see it while you build it. Add a rigidbody to the sphere and uncheck **Use Gravity**.
2. Create a new material named **Powerup** and give it a yellow color. Set the Metallic to 0, and the Smoothness to 1. Apply the material to the sphere.
3. Add a point light to the sphere (click **Add Component > Rendering > Light**). Give the light a yellow color. Set the Bounce Intensity to 0.
4. Add a particle system to the sphere (click **Component > Effects > Particle System**). Give the particles a start color of yellow and a start lifetime of 2.5.
5. Now that you have set it up in context, move the sphere to (0, 1, 42), which is where the prefab will spawn later.
6. Create a new folder called **Prefabs**. Click and drag the sphere from the Hierarchy view into the prefabs folder. This is another way of creating a prefab from a game object. Delete the sphere from the scene.

Note that by setting the position of the object before putting it into the prefab you can simply instantiate the prefab and it will appear at that spot. The result is that you will not need a spawn point. Figure 19.2 illustrates the finished power up.

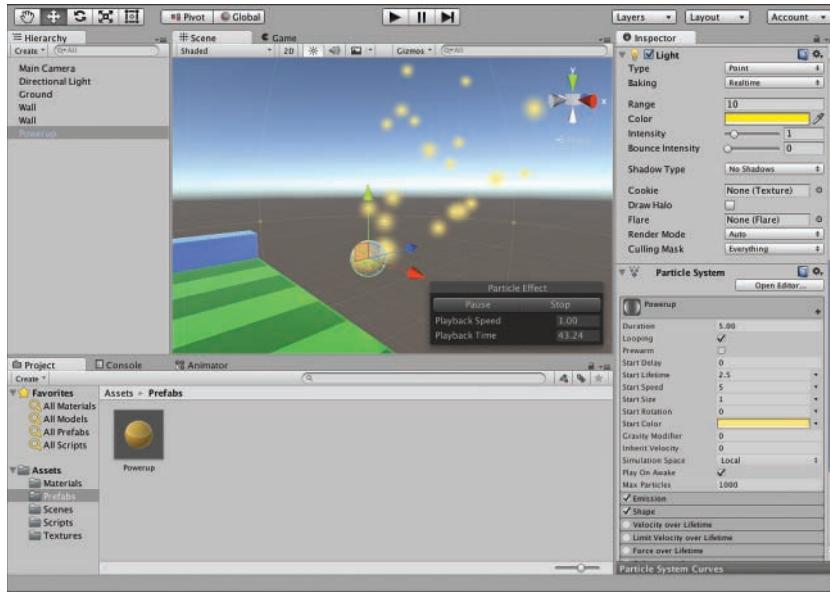


FIGURE 19.2
The finished power up.

The Obstacles

For this game, the obstacles are represented by small dark brown cubes. The player has the option of either avoiding them or jumping over them. To create the obstacles, follow these steps:

1. Add a cube to the scene. Position it at (0, .4, 42) with a scale of (1, .2, 1). Add a rigidbody to the cube and uncheck **Use Gravity**.
2. Create a new material called **Obstacle**. Make the color of the material black and apply it to the cube.
3. Create a new prefab named **Obstacle**. Drag the cube from the hierarchy onto the prefab and then delete the cube.

The Trigger Zone

Just like in previous games, the trigger zone exists to clean up any game objects that make it past the player. To create the trigger zone, follow these steps:

1. Add a cube to the scene. Rename the cube **TriggerZone** and position it at (0, 1, -20) with a scale of (10, 1, 1).
2. On the Box Collider component of the trigger zone, put a check mark in the **Is Trigger** property.

The Player

The player is where a large portion of the work for this game will go. The player will be using two new animations that you haven't worked with yet: run and jump. You'll start by getting the player ready for Mecanim animations:

1. Locate the folder named Robot Kyle in the book assets for Hour 19. This is a model provided free for use by Unity. To save the time of finding it on the Asset Store, though, it has been provided here. Drag that folder into the Project view in Unity to import it.
2. Locate and select the Robot Kyle.fbx file in the Model folder under the Robot Kyle folder. In the Inspector, select the **Animations** tab and deselect **Import Animation**. Click **Apply**.
3. Under the Rig tab, change the animation type to **Humanoid**. Click **Apply**.

You should now see a check mark next to the **Configure** button (see Figure 19.3). If you don't, you need to click **Configure** and configure the rig. You can find instructions for doing so in Hour 18, "Animators" (although it shouldn't be necessary).

You now need to get the animations ready to be placed in an animator, as follows:

1. Locate the Animations folder in the book assets for Hour 19. Drag the folder into the Project view in Unity to import it.
2. In the newly imported Animations folder, locate the Jump.fbx file and select it. In the Inspector, click the **Rig** tab and change the animation type to **Humanoid**. Click **Apply**.
3. Under the Animations tab of Jump.fbx, change the properties of the jump animation to match Figure 19.4. Note that the Offset property under the Root Transform Rotation

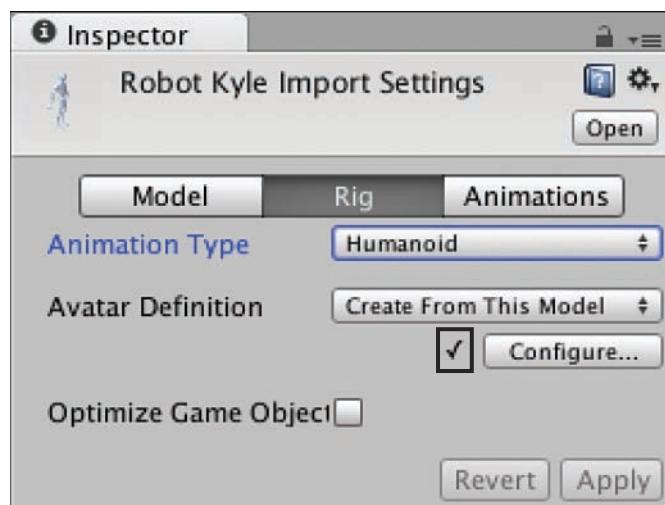
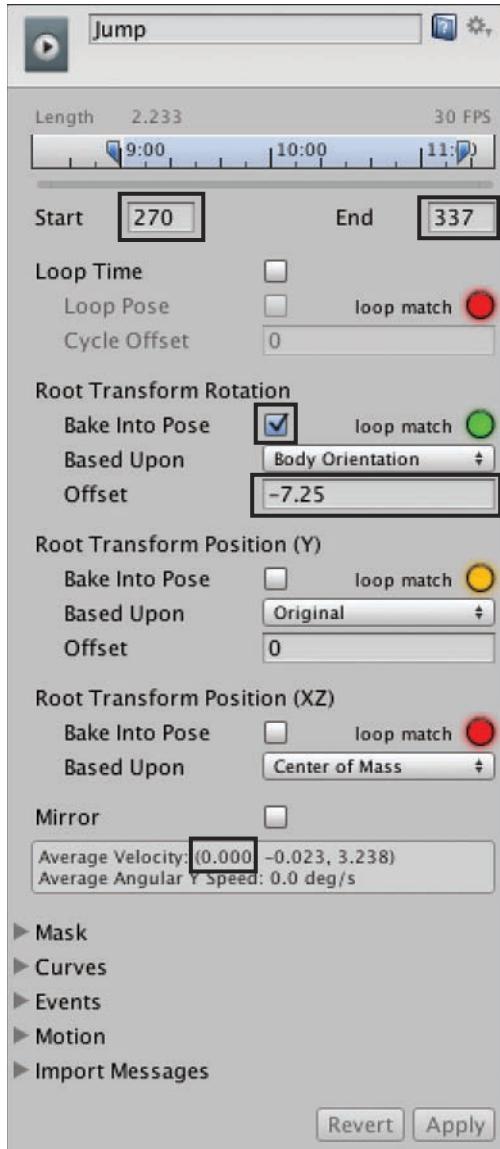


FIGURE 19.3

The rig settings.

**FIGURE 19.4**

Properties for the jump animation.

property might need to be different from the one in the image. What is important is that the Average Velocity property has a value of 0 for the x axis. Click **Apply**.

4. Select the **Runs.fbx** file in the Animations folder. Complete step 2 again to correct the rig for this model. Under the Animations tab, notice that there are three clips: RunRight, Run,

and RunLeft. Select **Run** and ensure that the properties match Figure 19.5. Again, the important part is that the x axis value for the Average Velocity property is 0. Click **Apply**.

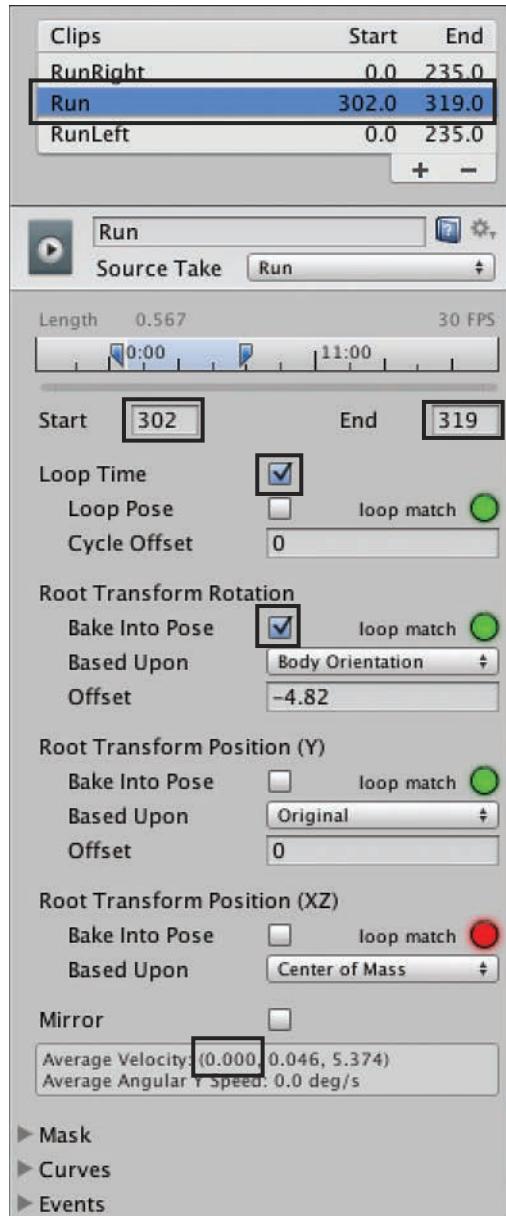


FIGURE 19.5

The run animation properties.

Now that the animations are prepared, you can begin making the animator. This will be a simple two-state animator without the need for any blending trees. To prepare the animator, follow these steps:

1. Create a new folder called **Animators**. Create a new animator in the folder (right-click and select **Create > Animator Controller**). Name it **Player**.
2. Double-click the animator to open the Animator view. Right-click in the Animator and create a new state called **Run**. This will become the default state. Set the Motion in the Inspector of the run state to **Run**, and put a check mark in the **Foot IK** property (see Figure 19.6).

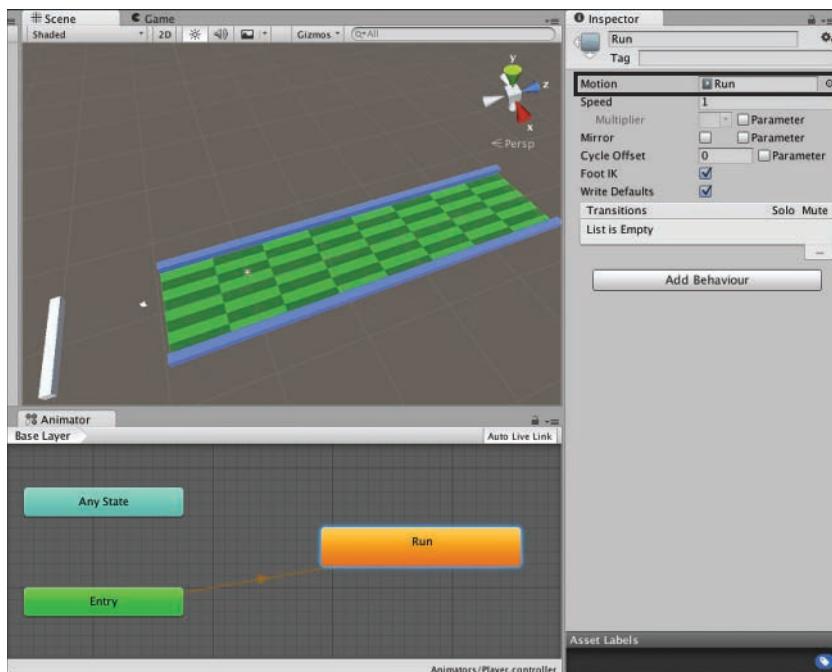


FIGURE 19.6
Adding the Run animation clip.

3. Locate the **Jump.fbx** file in the Animations folder. Expand the file and locate the **Jump** animation clip. Drag the clip onto the Animator view, this is another way of creating an animation state with associated clip. Click the newly created **Jump** state, and in the Inspector put a check mark in the **Foot IK** property and change the **Speed** property to **1.25**.
4. Add a new parameter to the animator by clicking the plus sign (+) in the Parameters box in the Animator view. The parameter should be a Trigger named **Jump** (see Figure 19.7).

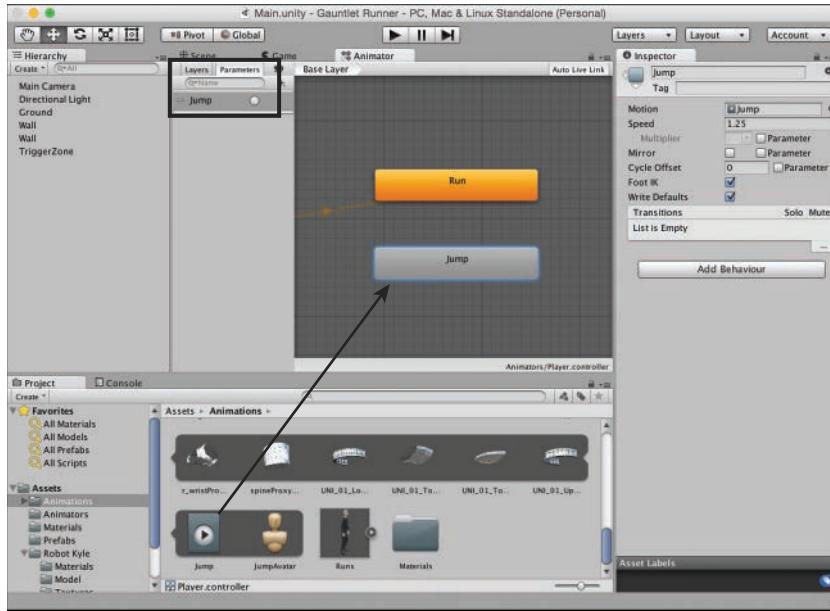
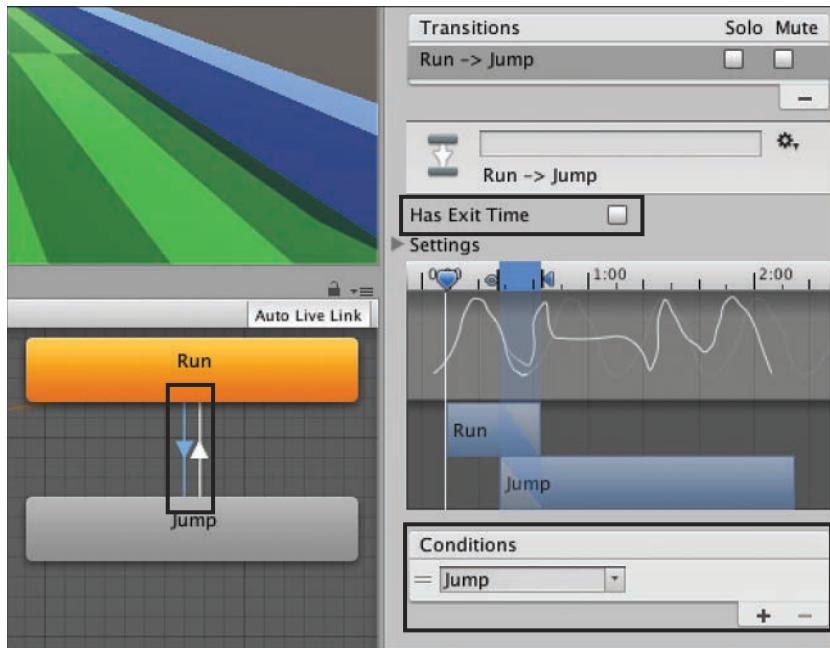


FIGURE 19.7
Adding the Jumping animation state and trigger parameter.

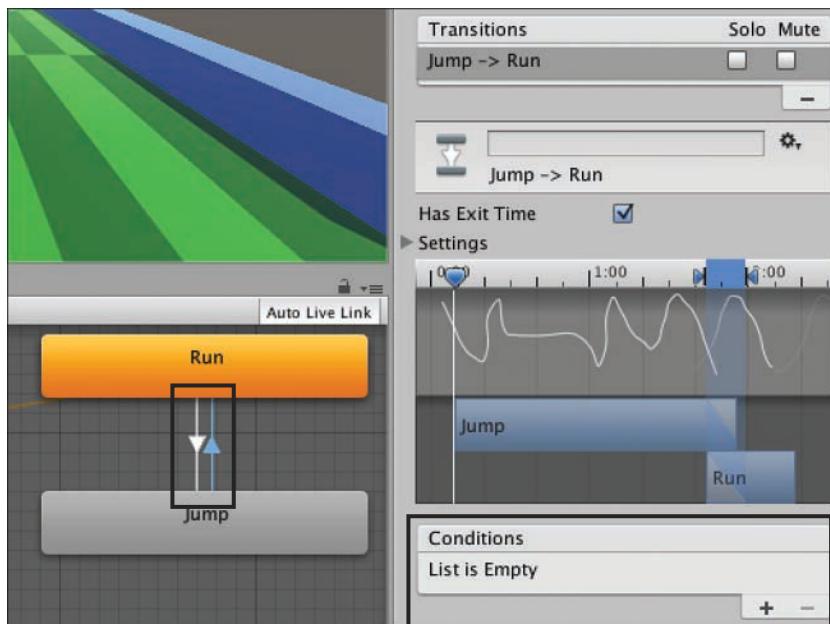
5. Right-click the **Run** state in the animator and select **Make Transition**. Click the **Jump** state to link them together. Right-click the **Jump** state and select **Make Transition**. Link it back to the Run state.
6. Click the white arrow that transitions from Run to Jump. In the Inspector, under Conditions click the + symbol to make the jump trigger the transition criteria (see Figure 19.8).
7. Click the white arrow that transitions from Jump to Run. Ensure that the properties in the Inspector match Figure 19.9.

Now that the player model is ready for animations, you need to place it in the scene, as follows:

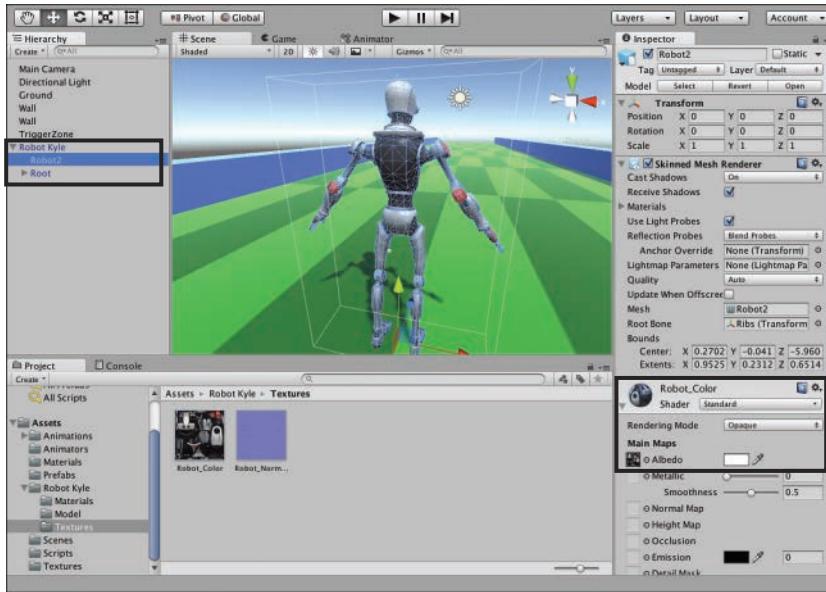
1. Locate the Robot Kyle.fbx file and drag it into your scene. Position the robot at (0, .25, -8.5).
2. Now add some texture to Kyle by expanding RobotKyle in the Inspector, and selecting the Robot2 child. Change the Shader type to Standard, and set the Albedo to **Robot_Color** (see Figure 19.10).
3. Add a capsule collider to the model (click **Add Component > Physics > Capsule Collider**). Set the Y value of the collider to **.95** and the height of the collider to **1.72**.
4. Drag the Player.controller onto the Controller property of the Animator component. Also ensure that the **Apply Root Motion** check box is not checked.

**FIGURE 19.8**

The Run transition properties.

**FIGURE 19.9**

The Jump transition properties.

**FIGURE 19.10**

Setting Robot Kyle's textures.

The player entity should now be set up and ready to go. If you run the scene, you should notice the robot running with the gauntlet moving underneath it. The effect is that the robot looks like it is running forward.

The Controls

It's now time to add the controls and interactivity to get this game going. Because the positions for the power ups and obstacles are in the prefabs already, there is no need to create a spawn point. Therefore, most all of the control will be placed on a game control object.

Trigger Zone Script

The first script you want to make is the one for the trigger zone. Remember that the trigger zone simply destroys any objects that make their way past the player. To create this, simply create a new script named **TriggerZone** and attach it to the trigger zone game object. Place the following code in the script:

```
void OnTriggerEnter(Collider other)
{
    Destroy (other.gameObject);
}
```

The trigger script is very basic and just destroys any object that enters it.

The Game Control Script

This script is where a majority of the work takes place. To start, create an empty game object in the scene and name it **GameControl**. This will simply be a placeholder for your scripts. Create a new script named **GameControl** and attach it to the game control object you just created. Following is the code for the game control script. There is some complexity here, so be sure to read each line carefully to see what it is doing. Add the following code to the script:

```
// Public so we can tune in inspector, and access from other scripts
public float startSpeed = -0.4f;
public float timeExtension = 1.5f;
public Ground ground;

private float timeRemaining = 10;
private float totalTimeElapsed = 0;
private bool isGameOver = false;

void Update () {
    if (isGameOver) {return;}

    totalTimeElapsed += Time.deltaTime;
    timeRemaining -= Time.deltaTime;
    if (timeRemaining <= 0) {
        isGameOver = true;
    }
}

public void SlowWorldDown () {
    CancelInvoke("SpeedWorldUp"); // Cancel any commands to speed world up
    ground.SlowDown ();
    Invoke ("SpeedWorldUp", 1); // Speed the world up again after 1 second
    Time.timeScale = 0.5f;
}

void SpeedWorldUp() {
    Time.timeScale = 1f;
    ground.SpeedUp();
}

public void PowerupCollected() {
    timeRemaining += timeExtension;
}

// Note this is using Unity's legacy GUI system, which still works in Unity 5
void OnGUI() {
    if(!isGameOver) {
        Rect boxRect = new Rect(Screen.width / 2 - 50, Screen.height - 100,
        ➔ 100, 50);
    }
}
```

```

    GUI.Box (boxRect, "Time Remaining");

    Rect labelRect = new Rect(Screen.width / 2 - 10, Screen.height - 80,
    ➔ 20, 40);
    GUI.Label (labelRect, ((int)timeRemaining).ToString());
} else {
    Rect boxRect = new Rect(Screen.width / 2 - 60, Screen.height / 2 - 100,
    ➔ 120, 50);
    GUI.Box (boxRect, "Game Over");

    Rect labelRect = new Rect(Screen.width / 2 - 55, Screen.height / 2 -
    ➔ 80, 90, 40);
    GUI.Label (labelRect, "Total Time: " + (int)totalTimeElapsed);

    Time.timeScale = 0;
}
}
}

```

NOTE

Old UI System

Please note that we have used Unity's old GUI system in this game (as we did in Amazing Racer). This still works fine in Unity 5, but is being replaced in the long-run by the much more powerful UI system as introduced in Hour 14. As a challenge, why not try replacing it yourself?

Remember that one of the premises of this game is that everything slows down when the player hits an obstacle. This is done by changing the `Time.timeScale` for the entire game. The remaining variables maintain the game timing and state.

The `Update()` method keeps track of time. It adds the time since the last frame (`Time.deltaTime`) to the `totalTimeElapsed` variable. It also checks to see whether the game is over, which happens when the time remaining reaches 0. If the game is over, it sets the `isGameOver` flag.

The `SlowWorldDown()` and `SpeedWorldUp()` methods work in conjunction. Whenever a player hits an obstacle, the `SlowWorldDown()` method is called. This method basically slows down time. It then calls the `Invoke()` method. This method basically says, "Call the method written here in *x* seconds," where the method called is the one named in the quotes and the number of seconds is the second value. You might have noticed the call to `CancelInvoke()` at the beginning of the `SlowWorldDown()` method. This basically cancels any `SpeedWorldUp()` methods waiting to be called because the player hit another obstacle. In the previous code, after 1 second, the `SpeedWorldUp()` method is called. This method speeds everything back up so that play can resume like normal.

The `PowerupCollected()` method is called by the player and adds the extension time to the time remaining.

Finally, the `OnGUI` method draws the remaining time to the scene while the game is running and the total time the game lasted once it has ended.

The Player Script

This script has two responsibilities: manage the player movement and collision controls, and manage the animator. Create a new script called **Player** and attach it to the robot model in the scene. Add the following code to the script:

```
public GameControl control;
private Animator anim;

public float strafeSpeed = 4f;
private bool jumping = false;

void Start () {
    anim = GetComponent<Animator>();
}

void Update () {
    float xMove = Input.GetAxis ("Horizontal") * Time.deltaTime * strafeSpeed;
    transform.Translate (xMove, 0f, 0f);

    if (transform.position.x > 3) {
        transform.Translate (3f, 0, 0);
    } else if (transform.position.x < -3) {
        transform.Translate (-3f, 0, 0);
    }

    if (Input.GetButtonDown ("Jump")) {
        anim.SetTrigger ("Jump");
    }

    if (anim.GetCurrentAnimatorStateInfo(0).IsName ("Run")) {
        jumping = false;
    } else {
        jumping = true;
    }
}

void OnTriggerEnter (Collider other) {
    if (other.gameObject.name == "Powerup(Clone)") {
        control.PowerupCollected ();
    } else if(other.gameObject.name == "Obstacle(Clone)" && ! jumping) {
        control.SlowWorldDown ();
    }

    Destroy (other.gameObject);
}
```

The first two variables hold the game control and animator references. The second two variables contain the movement-related information. The value for the `anim` variable is set in the `Start()` method.

The `Update()` method starts by moving the player based on input. It then checks to make sure that the player isn't farther than -3 or 3 on the x axis. If the player is, the player is set back to -3 or 3. This keeps the player in the gauntlet. The `Update()` method then checks to see whether the player is currently in the jumping animation. If he is, the local jumping flag is set to true (so that the player doesn't collide with obstacles), and the animator jumping parameter is set to false (so the jump animation doesn't loop). If the player isn't currently jumping, the animator sets the appropriate flag and checks to see whether the player presses the Jump button (spacebar by default).

In the `OnTriggerEnter()` method, the script checks to see what the player collided with. If a player collides with a power up, the appropriate method is called. To collide with an obstacle, the player must also not be jumping. If this is the case, the `SlowWorldDown()` method is called.

The Move Script

Both the power ups and obstacles need to move towards the player. We will apply the same script to both of them to create this behavior. Create a script named **Move** and add it to both the power up and obstacle prefabs. You can do this by selecting them both in the Inspector and clicking **Add Component > Scripts > Move**. Add the following the `Move.cs` script:

```
private GameController control;

void Start () {
    control = GameObject.FindObjectOfType<GameController> ();
}

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

This script is simple. There is a placeholder for the game control script. Then, at each `Update()` method call, the object is moved by the control's current speed. In this way, the control can change the speed of every object in the scene.

The Spawn Script

The spawn script is responsible for creating the objects in this scene. Because position data is in the prefabs, this script will be placed on the game control object. Create a new script called **Spawn** and attach it to the `GameController` object. Add the following code to the script:

```
public GameObject obstaclePrefab;
public GameObject powerupPrefab;
```

```

public float spawnCycle = 0.5f;
private float timeElapsed = 0;
private bool spawnPowerup = true;

void Update () {
    timeElapsed += Time.deltaTime;
    if (timeElapsed > spawnCycle) {
        GameObject temp;
        if (spawnPowerup) {
            temp = Instantiate (powerupPrefab) as GameObject;
            Vector3 pos = temp.transform.position;
            temp.transform.position = new Vector3 (Random.Range (-3, 4), pos.y, pos.z);
        } else {
            temp = Instantiate (obstaclePrefab) as GameObject;
            Vector3 pos = temp.transform.position;
            temp.transform.position = new Vector3 (Random.Range (-3, 4), pos.y, pos.z);
        }

        timeElapsed -= spawnCycle;
        spawnPowerup = !spawnPowerup;
    }
}

```

The script contains a reference to the power up and obstacle game objects. The next variables control the timing and order of the object spawns. The power ups and obstacles will take turns spawning, and therefore there is a flag to keep track of which one is going.

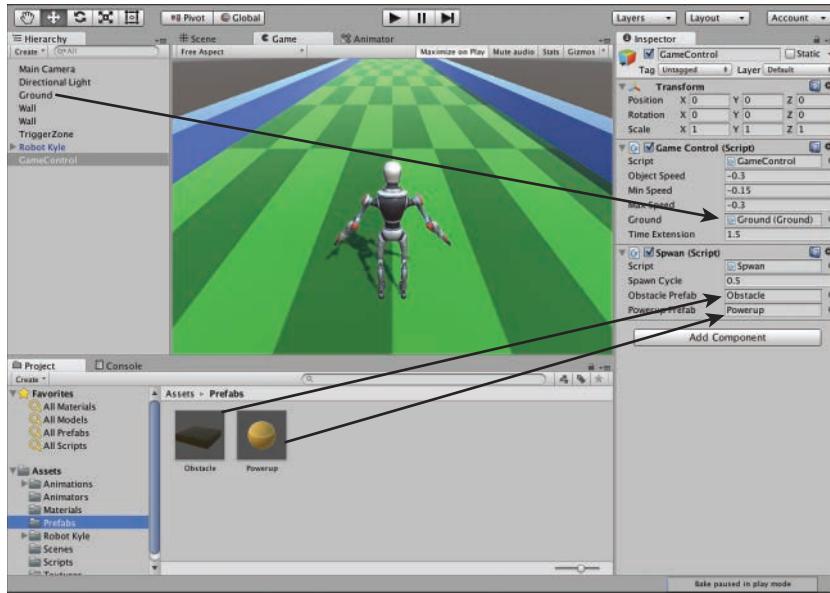
In the `Update()` method, the elapsed time is incremented and then checked to see if it is time to spawn a new object. If it is time, the script then checks to see which object it should spawn. It then spawns either a power up or an obstacle. The created object is then moved left or right randomly. Finally, the `Update()` method decreases the elapsed time and flips the power up flag so that the opposite object will be spawned next time.

Putting It All Together

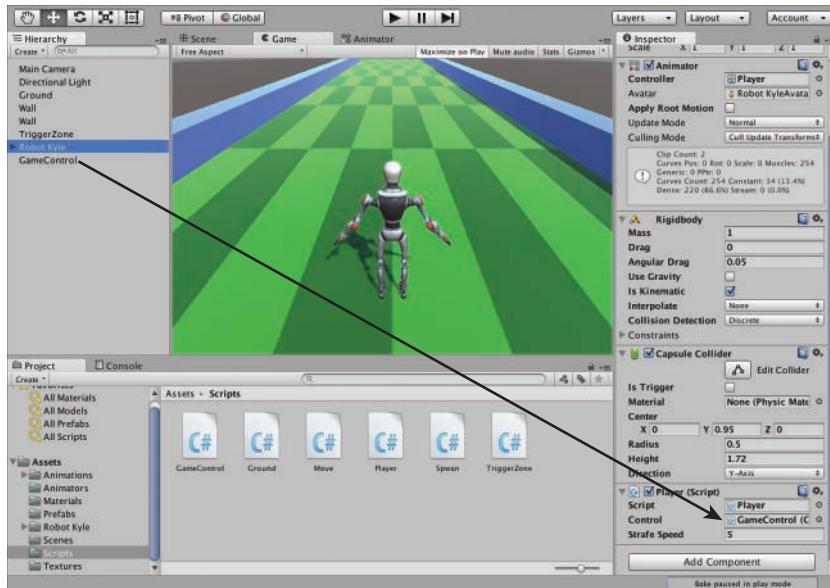
This is the last part of the game. You need to link the scripts and objects together. Start by selecting the `GameControl` object in the Hierarchy view. Drag the `Ground` object to its corresponding property in the `Game Control Script` component (see Figure 19.11). Drag the `Powerup` and `Obstacle` prefabs onto their corresponding properties in the `Spawn Script` component.

Next, select the `Robot Kyle` model in the hierarchy and drag the `GameControl` object onto the `Control` property of the `Player Script` component (see Figure 19.12).

That's it! The game is now complete and playable.

**FIGURE 19.11**

Dragging the objects to their properties.

**FIGURE 19.12**

Adding the game control to the player script.

Room for Improvement

As always, a game is not fully complete until it is tested and adjusted. Now it is time for you to play through the game and see what you like and what you don't like. Remember to keep track of the features that you think really enhance the gameplay experience. Also keep track of the items you feel detract from the experience. Be sure to make notes on any ideas you have for future iterations of the game. Try to have friends play the game as well and record their feedback about the game. All of these things will help you make the game unique and more enjoyable.

Summary

In this hour, you made the game *Gauntlet Runner*. You started by laying out the design elements of the game. From there, you built the gauntlet and got it to scroll using a texture trick. You then built the various entities for your game. After that, you built the various controls and scripts. Last but not the least, you tested the game and recorded some feedback.

Q&A

- Q. The movements of the objects and the ground aren't exactly lined up. Is that normal?**
- A.** In this case, yes. A fine level of testing and tweaking is required to get these to sync perfectly. This is one element you can focus on refining.
- Q. The jumping animation looks a little off. Is that normal?**
- A.** Again, this is normal in this circumstance. The animations used in this hour were provided by Unity for their Mecanim demo. Therefore, they are being used in a manner they weren't exactly designed for. (They were meant to control the movement of the player.) So, it looks a little off. Sometimes, game development is a matter of doing what you can with the tools you are provided.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. How does the player lose the game?
2. How does the scrolling background work?
3. What two states did you create in the animator?
4. How does the game control the speed of all of the objects in the scene?

Answers

1. The game is lost when time runs out.
2. The gauntlet stays stationary. Instead of moving, the texture is scrolled along the object. The result is that the ground appears to move.
3. Run and Jump.
4. We change the timescale of the entire game. However, notice the power ups and obstacles still appear to move fast. Why could this be?

Exercise

It is time for you to attempt to implement some of the changes you noted when play testing this game. You should make an attempt to make the game unique to you. Hopefully, you were able to identify some weaknesses of the game or some strengths that you would like to improve. Here are some things to consider changing:

- ▶ Try adding new/different power ups and obstacles.
- ▶ Try changing the old GUI code for Unity's new UI system.
- ▶ Try to increase or decrease the difficulty by changing how often power ups and obstacles spawn. Also change how much time is added by power ups or how long the world is slowed. You could even try to adjust how much the world is slowed or give different objects different slowed speeds.
- ▶ Give the power ups and obstacles a new look. Play around with textures and particle effects to make them look awesome.
- ▶ Show the total distance traveled to act like a score, and perhaps even make the game speed continually increase to create a lose condition.

This page intentionally left blank

HOUR 20

Audio

What You'll Learn in This Hour:

- ▶ The basics of audio in Unity
- ▶ How to use audio sources
- ▶ How to work with audio via scripts

In this hour, you learn about audio in Unity. You start by learning about the basics of audio. From there, you explore the audio source components and how they work. You also take a look at individual audio clips and their role in the process. Finally, you learn how to manipulate audio in code.

Audio Basics

A large part of any experience involves the sounds of that experience. Consider taking a scary movie and adding a laugh track to it. All of a sudden, what should be a tense experience becomes a funny one. The same is true for video games. Most of the time players don't realize it, but the sound is a very large part of the overall gameplay. Audio cues like chimes mark when a player unlocks a secret. Roaring battle cannons add a touch of realism to a war simulation game. Using Unity, amazing audio effects are easy to implement.

Parts of Audio

For sounds to work in a scene, you need three things: the audio listener, the audio source, and the audio clip. The audio listener is the most basic component of an audio system. The listener is a simple component that's sole responsibility is "hearing" the things that are happening in a scene. An easy way to think of them is like an ear in your world. By default, every scene starts with an audio listener attached to the Main Camera (see Figure 20.1). There are no properties available for the audio listener, and there is nothing you need to do to make it work. It is a common practice to put the audio listener on whatever game object represents the player. Note that if you put an audio listener on any other game object, you need to remove it from the Main Camera. Only a single audio listener is allowed per scene.



FIGURE 20.1
The audio listener.

The audio listener listens for sound, but it is the audio source that actually emits the sound. This source is a component that can be put on any object in a scene (even the object with the audio listener on it). There are many properties and settings involved with the audio source, and these are covered in their own section later this hour.

The last item required for functioning audio is the audio clip. Just as you would assume, the audio clip is the sound file that actually gets played by an audio source. Each clip has some properties that you can set to change the way Unity plays them. Unity supports the following 10 audio formats: .aif, .aiiff, .wav, .mp3, and .ogg, .mod, .it, .s3m, and .xm. Together, these three items give your scene an audio experience.

2D and 3D Audio

One concept to be aware of with audio is the idea of 2D and 3D audio. 2D audio clips are the most basic types of audio. They play at the same volume regardless of the audio listener's proximity to the audio source in a scene. 2D sounds are best used for menus, warnings, soundtracks, or any audio that must always be heard the exact same way. The greatest asset of 2D sounds is also their greatest weakness. Consider if every sound in your game played at the exact same volume regardless of where you were. It would quickly spiral out of control.

3D audio solves the problems of 2D audio. These audio clips feature something called *roll off*, which dictates how sounds get quieter or louder depending on how close the audio listener gets to the audio source. In sophisticated audio systems, like Unity's, 3D sounds can even have a simulated Doppler effect (more on that later). If you are looking for realistic audio in a scene full of different audio sources, 3D audio is the way to go.

The dimensionality of different audio clip is managed in the individual settings sound file settings.

Audio Sources

As mentioned before, the audio sources are the components that actually play audio clips in a scene. It is the distance between these sources and the listeners that determines how 3D audio clips sound. To add an audio source to a game object, select the desired object and click **Add Component > Audio > Audio Source**.

The audio source component has a series of properties that give you a fine level of control over how sound players in a scene. Table 20.1 describes the various properties of the audio source component.

TABLE 20.1 Audio Source Properties

Property	Description
Audio Clip	The actual sound file to play.
Output	Optionally output the sound clip to an Audio Mixer, a new Unity 5 feature.
Mute	Determines whether the sound is muted.
Bypass Effects	Determines whether audio effects are applied to this source. Selecting this property turns off effects.
Bypass Listener Effects	Determines whether audio listener effects are applied to this source. Selecting this property turns off effects.
Bypass Reverb Zones	Determines whether reverb zone effects are applied to this source. Selecting this property turns off effects.

Property	Description
Play On Awake	Determines if the audio source will begin playing the sound as soon as the scene launches.
Loop	Determines if the audio source will restart the audio clip once it has finished playing.
Priority	Importance of the audio source. 0 is the most important, and 255 is the least important. Use 0 for music so it always plays.
Volume	The volume of the audio source where 1 is the equivalent of 100% volume.
Pitch	The pitch of the audio source.
Stereo Pan	Sets the position in the stereo field of the 2D component of the sound.
Spatial Blend	Sets how much the 3D engine has an effect on the audio source.
Reverb Zone Mix	Sets the amount of the output signal that gets routed to the reverb zones.
3D Sound Settings	Settings applied to 3D audio clips. Covered in greater detail later.

NOTE**Audio Priorities**

Every system has a finite number of audio channels. This number is not consistent and depends on many factors such as the system's hardware and operating system. It is for this reason that most audio systems employ a priority system. In a priority system, sounds are played in the order that they are received until the max number of channels are used. Once all the channels are in use, lower-priority sounds are swapped out for higher-priority sounds. Just be sure to remember that in Unity a lower-priority number means a higher actual priority!

Importing Audio Clips

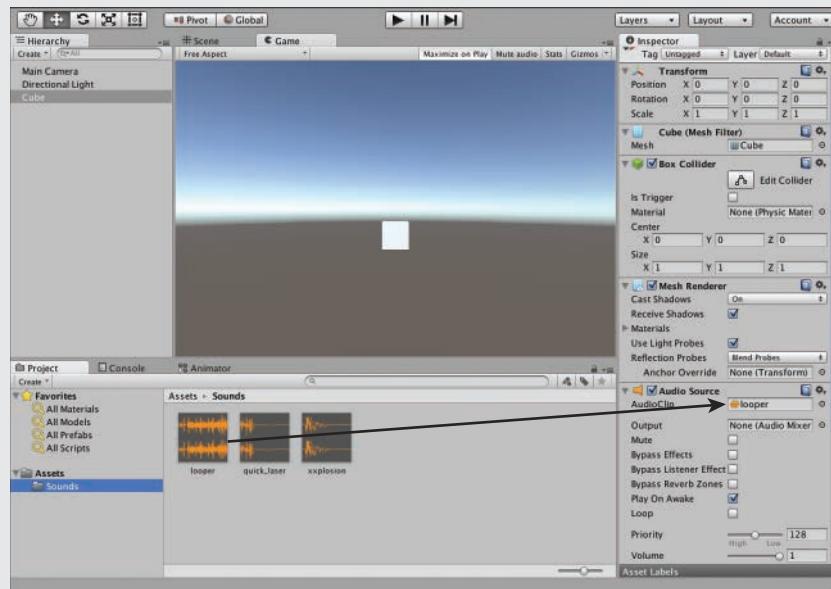
Audio sources don't do anything unless have audio to play. In Unity, importing audio is as easy as importing anything else. You just need to click and drag the files you want into the Project view to add them to your assets. These audio files have been graciously given to you to use by Jeremy Handel (<http://handelabra.com>).

TRY IT YOURSELF ▼

Testing Audio

Let's test out our audio in Unity and make sure that everything works. Be sure to save this scene because it will be used in the next section:

1. Create a new project or scene. Locate the **Sounds** folder in the book assets for Hour 20 and drag it into the Assets folder In Project view in Unity.
2. Create a cube in your scene and position it at (0, 0, 0). Add an audio source to the cube (click **Add Component > Audio > Audio Source**). Locate the file **looper.ogg** in the newly imported Sounds folder and drag it into the **Audio Clip** property of the audio source on the cube (see Figure 20.2).
3. Ensure that the **Play On Awake** property is checked and run your scene. Notice the sound playing. The audio should stop after about 20 seconds (unless you set it to loop).

**FIGURE 20.2**

Adding a clip to a source.

NOTE**Mute Audio Button**

At the top of the Game window, there is a button that is new to Unity 5, called **Mute audio** (between Maximise on Play and Stats). If you don't hear anything when your game is playing, check this button is not pressed in.

Testing Audio in the Scene View

It would get a bit taxing if you needed to run a scene every time you wanted to test out your audio. Not only would you need to start up the scene, you would also need to navigate to the sound in the world. That is not always easy, or even possible. Instead, you can test your audio in the Scene view.

To test audio in the Scene view, you need to turn scene audio on. Do this by clicking the scene audio toggle (see Figure 20.3). When you do this, an imaginary audio listener is used. This listener is positioned on your frame of reference in the Scene view (not on the position of the actual audio listener component).

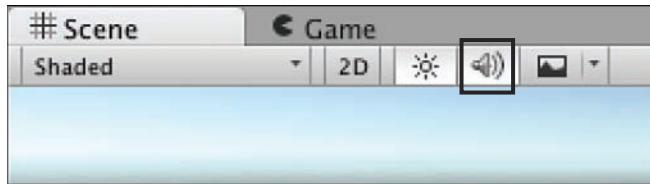


FIGURE 20.3
The audio toggle.

TRY IT YOURSELF

Audio in the Scene View

This exercise shows you how to test your audio in the Scene view. It uses the scene created in the previous exercise:

1. Open or create the scene from the previous exercise.
2. Turn on the scene audio toggle (refer to Figure 20.3).
3. Move around the Scene view. Notice how the sound stays the same volume, regardless of your distance from the cube emitting the sound. By default, all sound sources default to 2D.
4. Drag the Spatial Blend over to 3D (see Figure 20.4). Now try moving around in Scene view again. Note the sound now gets quieter as you get further away.

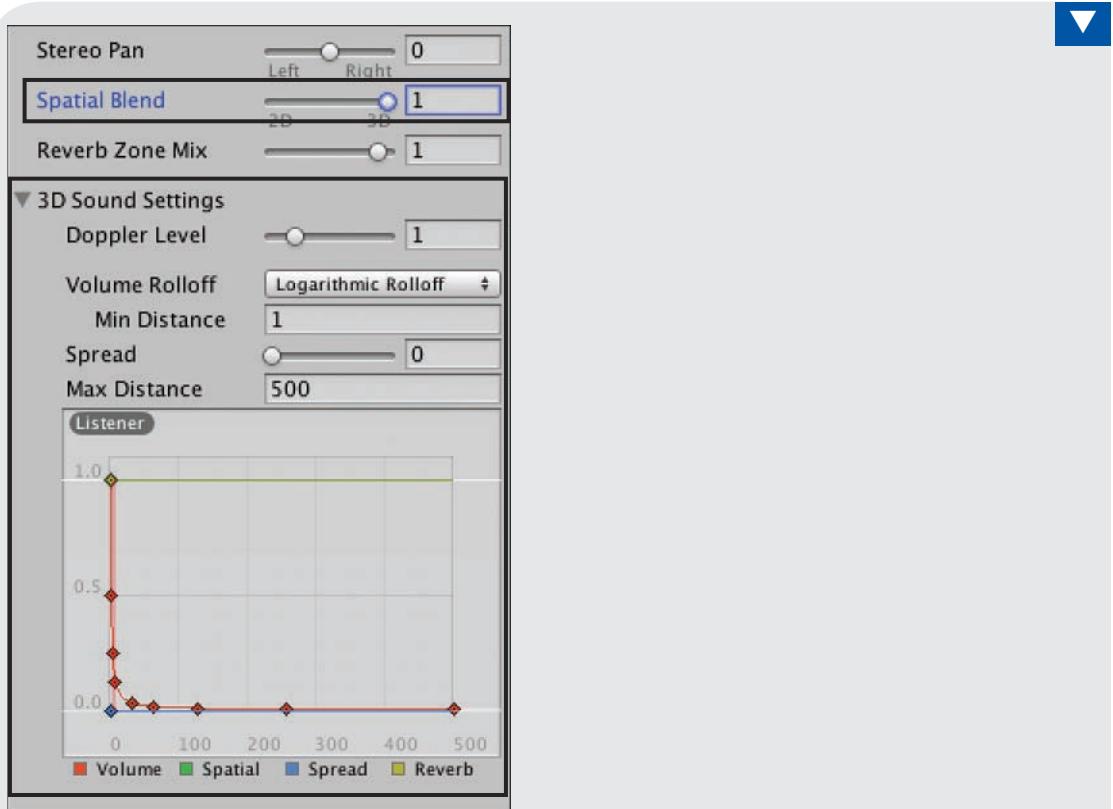


FIGURE 20.4
The 3D audio settings.

3D Audio

As mentioned previously, all audio is set to be fully 2D by default. It is easy to change it to fully 3D by setting the Spatial Blend to 1. This means that all audio will be subject to the 3D audio effects that are distance and movement based. These effects are modified by the 3D properties of the audio component (see Figure 20.4).

Table 20.2 describes the various 3D audio properties.

TIP

Use The Graph

In this case, a picture really does say a thousand words! As you play with the settings, look at the graph. It will tell you how loud the audio will be at various distances, and is a great way of visualizing the effect of the controls.

TABLE 20.2 3D Audio Properties

Property	Description
Doppler Level	Determines how much Doppler effect is applied to the audio. A setting of 0 means no effect will be applied. The Doppler effect is how sound is distorted while you are traveling toward or away from it.
Volume Rolloff	Determines how the change in sound volume over distance is applied. Logarithmic is set by default. You can also choose Linear or set your own curve with a custom roll off.
Min Distance	The distance away from the source you have to be before receiving 100% volume. The higher the number, the farther away you can be and still receive 100% volume.
Spread	How spread out the various speakers of a system are. A setting of 0 means that all speakers are at the same position and that the signal is essentially a mono signal. Leave this alone unless you understand more about audio systems.
Max Distance	The farthest you can be from the source and still hear some volume.

2D Audio

Sometimes, you want some of the audio to play regardless of its position in the scene. The most common example of this is background music. To switch an audio source from 3D to 2D, drag the Spatial Blend over to 2D (see Figure 20.4). Notice you can also have a blend of 2D and 3D—meaning the audio will always be heard regardless of your distance from it.

The settings above the 3D Sound Settings section such as Priority, Volume, Pitch, etc. apply to both the 2D and 3D portion of the sound. The settings in the 3D Sound Settings section obviously only apply to the 3D portion.

Audio Scripting

Playing audio from an audio source when it is created is nice, assuming that's the functionality that you want. If you want to wait and play a sound at a certain time, or play different sounds from the same source, however, you need to use scripting. Luckily, there isn't too much difficulty with managing your audio through code. Most of it works just like any audio player you're used to. Just pick a song and press Play. All audio scripting is done using variables and methods that are a part of the object audio.

Starting and Stopping Audio

When dealing with audio in script, the first thing you need to do is find the audio source component. Use the following code to do this . . .

```
private AudioSource audioSource;

void Start () {
    // Find the audio source component on the cube
    audioSource = GetComponent<AudioSource> ();
}
```

Now that you have an object to reference, `audioSource`, you can start calling methods on it. The most basic functionality you could want is simply starting and stopping an audio clip. These are controlled by two methods simply named `Start()` and `Stop()`. Using these methods looks like this:

```
audioSource.Start(); //Starts a clip
audioSource.Stop(); //Stops a clip
```

This code will play the clip specific by the `Audio Clip` property of the audio source component. You also have the ability to start a clip after a delay. To do that, you use the method `PlayDelayed()`, which takes in a single parameter that is the time in seconds to wait before playing the clip. This method looks like:

```
audioSource.PlayDelayed(<some time in seconds>);
```

You can tell whether a clip is currently playing by checking the `isPlaying` variable, which is a part of the `audio` object, in code. To access this variable, and thus see if the clip is playing, you could type the following:

```
if(audioSource.isPlaying)
{
    //The track is playing
}
```

As the name implies, this variable is true if the audio is currently playing and false if it is not.

TRY IT YOURSELF ▼

Starting and Stopping Audio

In this exercise, you use scripts to start and stop an audio clip:

1. Create a new project or scene. Import the **Sounds** folder from the book assets if you haven't done so already. Place a cube in your scene at position (0, 0, 0) and put an audio source on it.

- ▼
2. Drag the looper.ogg file from the Sounds folder onto the Audio Clip property of the audio source on the cube. Also be sure to uncheck **Play On Wake** and to check the **Loop** properties of the audio source.
 3. Create a new folder named **Scripts** and create a new script in it called **AudioScript**. Attach the script to the cube. Change the entire script code for the following:

```
using UnityEngine;
using System.Collections;

public class AudioScript : MonoBehaviour {

    private AudioSource audioSource;

    // Use this for initialization
    void Start () {
        // Find the audio source component on the cube
        audioSource = GetComponent<AudioSource> ();
    }

    // Update is called once per frame
    void Update () {
        if (Input.GetButtonDown ("Jump")) {
            if (audioSource.isPlaying == true) {
                audioSource.Stop ();
            } else {
                audioSource.Play ();
            }
        }
    }
}
```

4. Play the scene. You can start and stop the audio by pressing the spacebar. Notice how the audio clip starts over every time you play the audio.

TIP

Unmentioned Properties

All the properties of the audio source that are listed in the Inspector are also available via scripting. For instance, the Loop property is accessed in code with the `audioSource.loop` variable. As mentioned before, all of these variables are used in conjunction with the audio object. See how many you can find!

Changing Audio Clips

You can easily control which audio clips to play via scripts. The key is to change the audio clip property in the code before using the `Play()` method to play the clip. Always be sure to stop the current audio clip before switching to a new one; otherwise, the clip won't switch.

To change the audio clip of an audio source, assign a variable of type `AudioClip` to the `clip` variable of the object `audio`. For example, if you had an audio clip called `newClip`, you could assign it to an audio source and playing it using the following code:

```
audioSource.clip = newClip;  
audioSource.Play();
```

You can easily create a collection of audio clips and switch them out in this manner. We will do this in the exercise at the end of this hour.

Summary

In this hour, you learned about using audio in Unity. You started by learning about the basics of audio and the components required to make it work. From there, you explored the audio source component. You learned how to test audio in the Scene view, and how to use 2D and 3D audio clips. You finished the hour by learning to manipulate audio through scripts.

Q&A

Q. How many audio channels does a system have on average?

A. It truly varies for every system; most modern gaming platforms can simultaneously play dozens or hundreds of audio at the same time. The key is to know your target platform, and use the priority system well.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What items are needed for working audio?
2. True or False: 3D sounds play at the same volume regardless of the listener's distance from the source?
3. What method allows you to play an audio clip after a delay?

Answers

1. An audio listener, source, and clip.
2. False. 2D sounds play at the same volume.
3. `PlayDelayed()`.

Exercise

In this exercise, you create a basic sound board. This sound board will allow you to play one of three sounds. You also have the ability to start and stop the sounds and to turn looping on and off. You can find the completed exercise as Hour20_Exercise in the book assets for Hour 20:

1. Create a new project or scene. Add a cube to the scene at position (0, 0, -10) and add an audio source to the cube. Be sure to uncheck the **Play OnAwake** property. Locate the **Sounds** folder in the book assets for Hour20 and drag it into the Assets folder.
2. Create a new folder called **Scripts** and create a new script named **AudioScript** in it. Attach the script to the cube. Change the script to contain the following:

```
using UnityEngine;

using System.Collections;

public class AudioScript : MonoBehaviour {

    public AudioClip clip1;
    public AudioClip clip2;
    public AudioClip clip3;

    private AudioSource audioSource;

    // Use this for initialization
    void Start () {
        // Find the audio source component on the cube
        audioSource = GetComponent<
```

```
    }

    if (Input.GetKeyDown(KeyCode.L)) {
        audioSource.loop = ! audioSource.loop; //toggles looping
    }

    if (Input.GetKeyDown(KeyCode.Alpha1)) {
        audioSource.Stop();
        audioSource.clip = clip1;
        audioSource.Play();
    } else if (Input.GetKeyDown(KeyCode.Alpha2)) {
        audioSource.Stop();
        audioSource.clip = clip2;
        audioSource.Play();
    } else if (Input.GetKeyDown(KeyCode.Alpha3)) {
        audioSource.Stop();
        audioSource.clip = clip3;
        audioSource.Play();
    }
}
```

3. In the Unity editor, select the cube in your scene. Drag each of the **looper.ogg**, **quick_laser.ogg**, and **xplosion.off** audio files from the Sounds folder onto the Clip1, Clip2, and Clip3 properties of the audio script.
4. Run your scene. Notice how you can change your audio clips with the 1–3 number keys. You can also start and stop the audio with the spacebar. Finally, you can toggle looping with the **L** key.

This page intentionally left blank

HOUR 21

Mobile Development

What You'll Learn in This Hour:

- ▶ How to prepare for mobile development
- ▶ How to use a devices accelerometer
- ▶ How to use a devices touch display

Mobile devices such as phone and tablets are becoming common gaming devices. In this hour, you learn about mobile development with Unity for Android and iOS devices. You begin by looking at the requirements for mobile development. From there, you learn how to accept special inputs from a device's accelerometer. Finally, you learn about touch interface input.

NOTE

Requirements

This hour covers the development for mobile devices specifically. So, if you do not have a mobile device (iOS or Android), you will not be able to follow along with any of the hands-on exercises. Don't worry though the reading should still make sense. You will still be able to make games for mobile devices. You just won't be able to play them.

Preparing for Mobile

Unity makes developing games for mobile devices easy. Since Unity version 4.1, the mobile plugins are even free! You will also be happy to know that developing for mobile platforms is almost identical to developing for other platforms. This means that you can build a game once and deploy it everywhere. There is no longer any reason why you can't build your games for every major platform. This level of cross-platform capability is unprecedented. Before you can begin working with mobile devices in Unity, however, you need to get your computer set up and configured to do it.

NOTE**Multitudes of Devices**

There are many different types of mobile devices. At the time of this writing, Apple has two classes of mobile device we can target: iPad and iPhone/iPod. Android has an untold number of phones and tablets. Each of these devices has slightly different hardware and steps to configure them correctly. Therefore, this text simply attempts to guide you through the installation process. It would be impossible to write an exact guide that would work for everyone. In fact, several guides by Unity, Apple, and Android (Google) already exist that explain the process better than this text could. You are referred to them when needed.

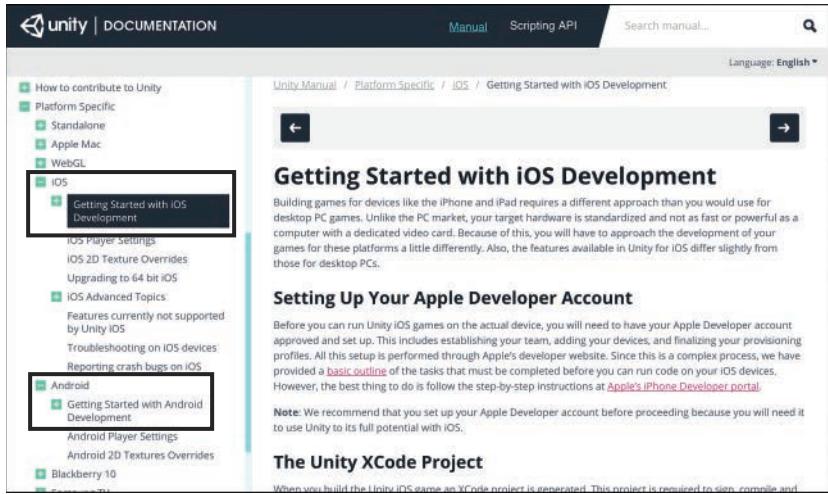
Setting Up Your Environment

Before even opening Unity to make a game, you need to set up your development environment. The specifics of this differ depending on your target device and what you are trying to do, but the general steps are as follows:

- 1.** Install the software development kit (SDK) of the device you are targeting.
- 2.** Ensure that your computer recognizes and can work with your device (only important if you want to test on the device).
- 3.** Tell Unity where to find the SDK (required for Android only).

If these steps seem a bit cryptic to you, don't worry. Plenty of resources are available to assist you with these steps. The best place to start is with Unity's own documentation. You can access Unity's documentation at <http://docs.unity3d.com>. This site contains the living document of everything that is Unity.

As you can see in Figure 21.1, the Unity documentation has guides to assist you in setting up both the iOS and Android environments. These documents are updated as the steps to set the environment changes. After you have completed the steps to configure your development environment for your target environment, or if you're not planning on following along with a device, continue on to the next section.

**FIGURE 21.1**

Platform specific documentation.

The Unity Remote

The most basic way to test your games on a device is to build your projects, put the resulting files on the device, and then run it. This can be a cumbersome system and one you're sure to tire of quickly. Another way to test your games is to build the project and then run it through an iOS or Android emulator. Again, this requires quite a few steps and involves configuring and running an emulator. These systems can be useful if you are doing extensive testing on advanced things, such as performance and rendering. For basic testing, though, there is a much better way: the Unity Remote.

The Unity Remote is an app you can download from your mobile devices application store that enables you to test your applications out on your mobile device while it is running in the Unity editor. In a nutshell, this means that you can experience your game running on a device in real time alongside development and use the device to send device inputs back to your game. You can find more information about the Unity Remote at <http://docs.unity3d.com/Manual/UnityRemote4.html>.

To find the Unity Remote application, search for the term *Unity Remote* in your device's application store. From there, you can download and install it just like any other application.

Once installed, the Unity Remote will act as both a display for your game and a controller. You will be able to send click information, accelerometer information, and multi-touch input back to Unity.

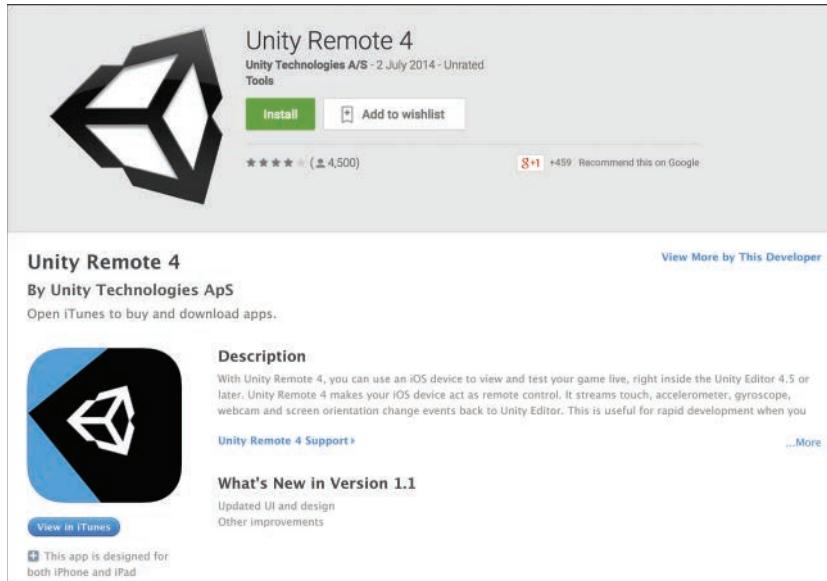


FIGURE 21.2
The different app stores.

TRY IT YOURSELF

Testing Device Setup

Let's take a moment to ensure that your mobile development environment is set up correctly. In this exercise, you use the Unity Remote from your device to interact with a scene in Unity. If you don't have a device set up, you won't be able to perform all of these steps, but you can still get the idea of what's happening by reading along. If these steps don't work, it means that something with your environment is not set up correctly:

1. Create a new 2D project or scene, and add a UI Button at (0, 0, 0).
2. Set the button's Pressed Color to red in the Inspector, under Button (Script).
3. Run the scene and ensure that clicking the button changes its color. Stop the scene.
4. Attach your mobile device to your computer with a cable. Once the computer recognizes your device, open the Unity Remote.
5. Click **Edit > Project Settings > Editor** and chose your device in the Inspector under **Unity Remote > Device**.
6. Run the scene again. After a second, you should see the button appear on your mobile device. You should now be able to tap the button on your device's screen to change its color.

Accelerometers

Most modern mobile devices come with a built-in accelerometer. An accelerometer relays information about the physical orientation of the device. It can tell whether the device is moving, tilted, or flat. It can also detect these things in all three axes. Figure 21.3 shows a mobile device's accelerometer axes and how they are oriented. This is called a *portrait orientation*.

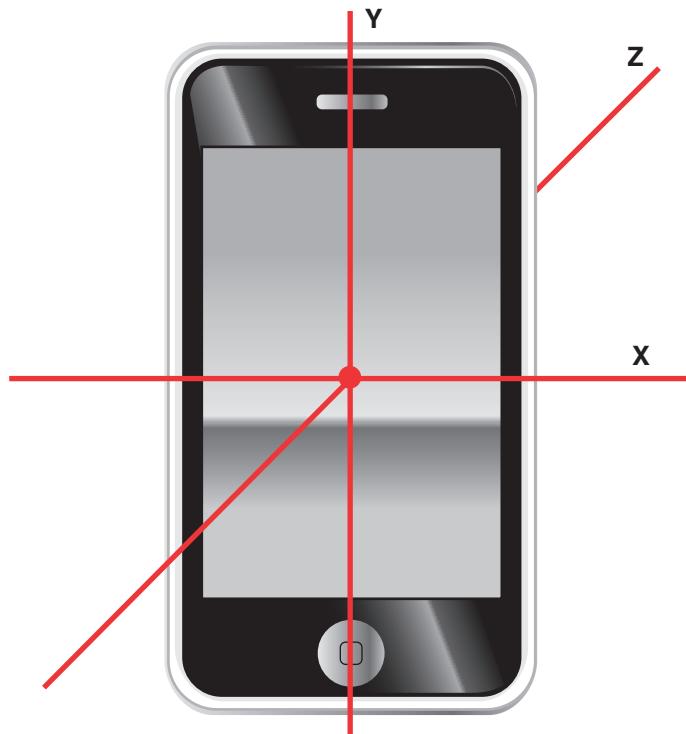


FIGURE 21.3
Accelerometer axes.

As you can see in Figure 21.3, the default axes of a device align with the 3D axes in Unity while the device is being held upright directly in front of you. If you turn the device to use it in a different orientation, you need to convert the accelerometer data to the correct axis.

Designing for the Accelerometer

You need to keep in mind a few things when designing a game to use a mobile device's accelerometer. The first is that you can only ever reliably use two of the accelerometer's axes at any given time. The reason for this is that no matter the orientation of the device, one axis will always be actively engaged by gravity. Consider the orientation of the device in Figure 21.3. You

can see that while the x and z axes can be manipulated by tilting the device, the y axis is currently reading negative values, as gravity is pulling it down. If you were to turn the phone so that it rested flat on a surface, face up, you would only be able to use the x and y axes. In that case, the z axis would be actively engaged.

Another thing to consider when designing for an accelerometer is that the input is not extremely accurate. Mobile devices do not read from their accelerometers at a set interval, and often have to approximate values. The result is the inputs read from an accelerometer can be jerky and uneven. Also accelerometers give input values from -1 to +1 with full 180 degree rotation of the device. No one plays games while fully tilting their devices, so input values are generally less than their keyboard counterparts (e.g., -.5 to .5).

Using the Accelerometer

Reading accelerometer input is done via scripts just like any other form of user input. All you need to do is read from the Vector3 variable named **acceleration**, which is a part of the object **Input**. Therefore, you could access the x, y, and z axis data by writing the following:

```
Input.acceleration.x;  
Input.acceleration.y;  
Input.acceleration.z;
```

Using these values, you can manipulate your game objects accordingly.

NOTE

Axis Mismatch

When using accelerometer information in conjunction with the Unity Remote, you might notice that the axes aren't lining up with the way they were described earlier in the "Accelerometers" section. This is because the Unity Remote bases the game's orientation on the aspect ratio chosen. This means that the Unity Remote will automatically display in landscape orientation (holding your device sideways so that the longer edge is parallel to the ground) and translates the axes for you. Therefore, when you are using the Unity Remote, the x axis runs along the long edge of your device, and the y axis runs along the short edge. It might seem strange, but chances are you were going to use your device like that anyway. This saves you a step.



TRY IT YOURSELF

Moving a Cube with the Power of Your Mind . . . or Your Phone

In this exercise, you use a mobile device's accelerometer to move a cube around a scene. Obviously, to complete this exercise you need a configured and attached mobile device with an accelerometer:

1. Create a new project or scene. Add a cube to the scene and position it at (0, 0, 0).

2. Create a new script called **AccelerometerScript** and attach it to the cube. Put the following code in the `Update()` method of the script:

```
float x = Input.acceleration.x * Time.deltaTime;
float z = -Input.acceleration.z * Time.deltaTime;
transform.Translate(x, 0f, z);
```

3. Ensure that your mobile device is plugged in to your computer. Hold the device in a landscape orientation and run the Unity Remote. Run the scene. Notice how you can move the cube by tilting your phone. Notice which axes of the phone move the cube along the x and z axes.

Multi-Touch Input

Mobile devices tend to be controlled largely by touch-capacity screens. These screens can detect when and where you touch them. They usually can track multiple touches at a time. The exact number of touches varies based on the device.

Touching the screen doesn't just give the device a simple touch location. In fact, there is quite a bit of information stored about each individual touch. In Unity, each screen touch is stored in a **Touch** variable. This means that every time you touch a screen, a **Touch** variable will be generated. That **Touch** variable will exist as long as your finger remains on the screen. If you drag your finger along the screen, the **Touch** variable tracks that. These **Touch** variables are stored together in a collection called **touches**, which is a part of the **Input** object. If there is currently nothing touching the screen, than this collection of touches will be empty. To access this collection, you could enter the following:

```
Input.touches;
```

Using that collection, you could iterate through each touch variable to process its data. Doing so would look something like this:

```
Foreach(Touch touch in Input.touches)
{
    //Do something
}
```

As mentioned before, each touch contains more information than the simple screen data where the touch occurred. Table 21.1 contains all the properties of the **Touch** variable type.

TABLE 21.1 Touch Properties

Property	Description
<code>deltaPosition</code>	The change in touch position since the last update. This is useful for detecting finger drags.
<code>deltaTime</code>	The amount of time that has passed since the last change to the touch.

Property	Description
fingerId	The unique index for the touch. For example, these would range from 0 to 4 on devices that allow five touches at a time.
phase	The current phase of the touch. The phases can be Began, Moved, Stationary, Ended, and Canceled.
position	The 2D position of the touch on the screen.
tapCount	The number of taps the touch has performed on the screen.

Each of these properties is useful for managing complex interactions between the user and game objects.

▼ TRY IT YOURSELF

Tracking Touches

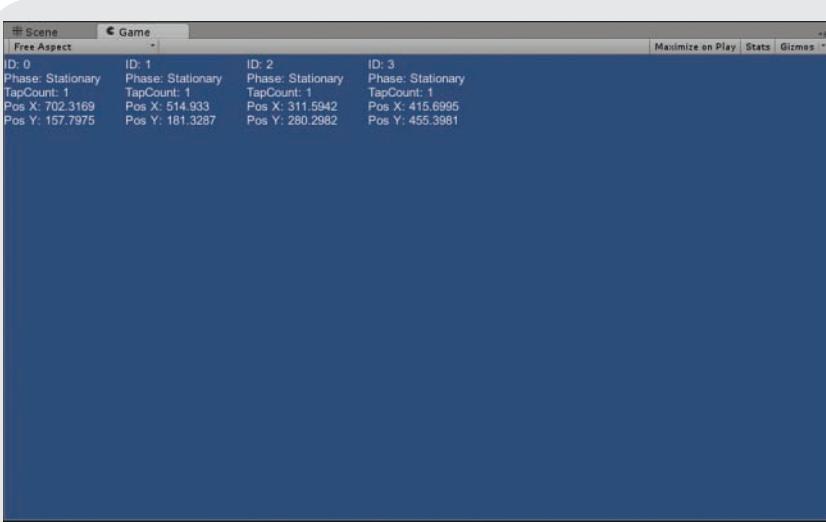
In this exercise, you track finger touches and output their data to the screen. Obviously, to complete this exercise you need a configured and attached mobile device with multi-touch support:

1. Create a new project or scene.
2. Create a new script called **TouchScript** and attach it to the Main Camera. Put the following code in the script:

```
void OnGUI () {
    foreach(Touch touch in Input.touches) {
        string message = "";
        message += "ID: " + touch.fingerId + "\n";
        message += "Phase: " + touch.phase.ToString() + "\n";
        message += "TapCount: " + touch.tapCount + "\n";
        message += "Pos X: " + touch.position.x + "\n";
        message += "Pos Y: " + touch.position.y + "\n";

        intnum = touch.fingerId;
        GUI.Label(new Rect(0 + 130 * num, 0, 120, 100), message);
    }
}
```

3. Ensure that your mobile device is plugged in to your computer. Run the scene. Touch the screen with your finger and notice the information that appears (see Figure 21.4). Move your finger and see how the data changes. Now touch with more fingers simultaneously. Move them about and take them off of the screen randomly. See how it tracks each touch independently. How many touches can you get on your screen at a time?

**FIGURE 21.4**

Touch output on the screen.

CAUTION

Do Because I Say, Not Because I Do!

In the preceding exercise, you created an `OnGUI()` method that collected information about the various touches on the screen. The part of the code where the string `message` is being built with the touch data is a *big* no-no. You should never perform processing in an `OnGUI()` method, because it can greatly reduce efficiencies in your project. This was just the easiest way to build the example without unneeded complexity and for demonstration purposes only. Always keep update code where it belongs: in `Update()`.

Summary

In this hour, you learned about using Unity to develop games with mobile devices in mind. You started by learning how to configure your development environment to work with Android and iOS. From there, you worked hands on with a device's accelerometer. You finished up the hour by experimenting with Unity's touch-tracking system.

Q&A

- Q.** Can I really build a game once and deploy it to all major platforms, mobile included?
- A.** Absolutely! The only thing to consider is that mobile devices generally don't have as much processing power as desktops. Therefore, you might experience some performance issues if your game has a lot of heavy processing or effects. You will need to ensure that your game is running efficiently if you plan to also deploy it on mobile platforms.
- Q.** What are the differences between iOS and Android devices?
- A.** From a Unity point of view, there isn't much difference between these two operating systems. They are both treated as mobile devices. Be aware, though, that there are some hardware differences that can affect your games.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. What tool allows you to send live device input data to Unity while it is running a scene?
2. How many axes on the accelerometer can you realistically use at a time?
3. How many touches can a device have at once?

Answers

1. The Unity Remote.
2. Two axes. The third will always be engaged by gravity depending on how you are holding the device.
3. It depends entirely on the device. If a device doesn't have multi-touch, it can have only a single touch at a time. If it does have multi-touch, it can have many.

Exercise

In this exercise, you move objects about a scene based on touch input from a mobile device. Obviously, to complete this exercise you need a configured and attached mobile device with multi-touch support. If you do not have that, you can still read along to get the basic ideas. The completed exercise can be found as Hour21_Exercise in the book assets for Hour 21:

1. Create a new 3D project or scene.
2. Add three cubes to the scene and name them **Cube1**, **Cube2**, and **Cube3**. Position them at $(-3, 1, 5)$, $(0, 1, -5)$, and $(3, 1, -5)$, respectively.

3. Create a new folder named **Scripts**. Create a new script called **InputScript** in the Scripts folder and attach it to the three cubes.

4. Add the following code to the `Update()` method of the script:

```
foreach (Touch touch in Input.touches) {  
    float xMove = touch.deltaPosition.x * 0.05f;  
    float yMove = touch.deltaPosition.y * 0.05f;  
  
    if (touch.fingerId == 0 && gameObject.name == "Cube1") {  
        transform.Translate (xMove, yMove, 0f);  
    }  
    if (touch.fingerId == 1 && gameObject.name == "Cube2") {  
        transform.Translate (xMove, yMove, 0f);  
    }  
    if (touch.fingerId == 2 && gameObject.name == "Cube3") {  
        transform.Translate (xMove, yMove, 0f);  
    }  
}
```

5. Run the scene and touch the screen with up to three fingers. Notice how you can move the three cubes independently. Also notice how lifting one finger does not cause the other fingers to lose their cubes or their place.

This page intentionally left blank

HOUR 22

Game Revisions

What You'll Learn in This Hour:

- ▶ How to make *Amazing Racer* mobile capable
- ▶ How to make *Chaos Ball* mobile capable
- ▶ How to make *Captain Blaster* mobile capable
- ▶ How to make *Gauntlet Runner* mobile capable

Let's make a lot of games! More specifically, let's revisit the games you made before and make them mobile capable. You will start by adding movement, looking, and jumping capabilities to *Amazing Racer*. From there, you will add turning and moving controls to *Chaos Ball*. Next, you will change orientations to work in Portrait mode with *Captain Blaster*. Finally, you will add input controls to *Gauntlet Runner*.

NOTE

Completed Games

Each of the completed *mobile-friendly* games is available in the book assets for Hour 22, named after their original game project.

Cross-Platform Input

Making a game playable on multiple platforms is known as cross-platform development. Before we dive in and show you how to modify each game for mobile, let's take a moment to understand the process.

Virtual Controls

Unity implements a powerful virtual control system, which provides a layer of abstraction between our code and the physical controls. This means that we can write one set of code and have the controls work on multiple devices.

When we change the target platform from mobile (touch) to desktop (mouse + keyboard), or even a console (gamepad), the controls are sent to a virtual control layer. We can then query that layer in our code in a consistent way to retrieve button presses, or joystick and mouse movements.

The general pattern for converting your game to mobile is a three-step process:

1. Convert the project to your target platform (as described below).
2. Adjust your code to query the virtual controls if necessary.
3. Add your mobile controls, and adjust them as necessary.

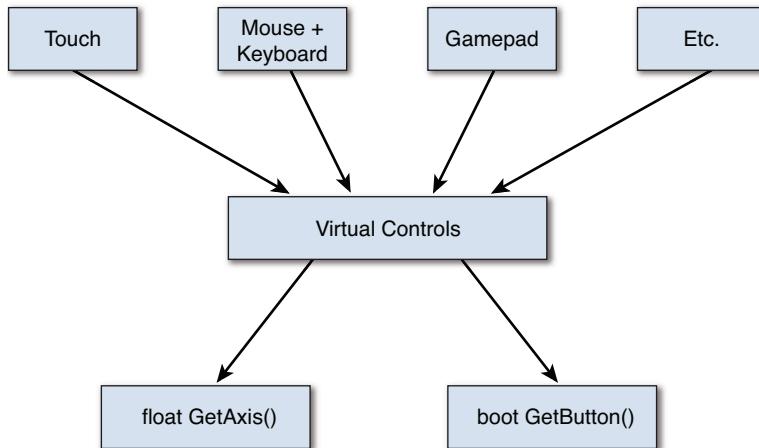


FIGURE 22.1
The virtual control layer.

TIP

Virtual Controls for Replay

Because there is a virtual layer between the controls and your player, you can actually drive your player with code. Simulating control input like this can be helpful in several situations, including when you want to show a replay or a gameplay preview.

Converting Projects to Mobile

There are some common steps you need to take in order to convert all your projects to mobile. They are listed here so that we don't need to repeat them for each project. The screenshots use Amazing Racer as an example, but the process is identical for all four projects.

Set up your project for mobile by following these steps:

1. Change the target platform by clicking **File > Build Settings**, and then select iOS or Android in the platform list (see Figure 22.2). Now click **Switch Platform** and wait for the process to complete. This may take several minutes depending on the speed of your computer. Finally, close the Build Settings window.

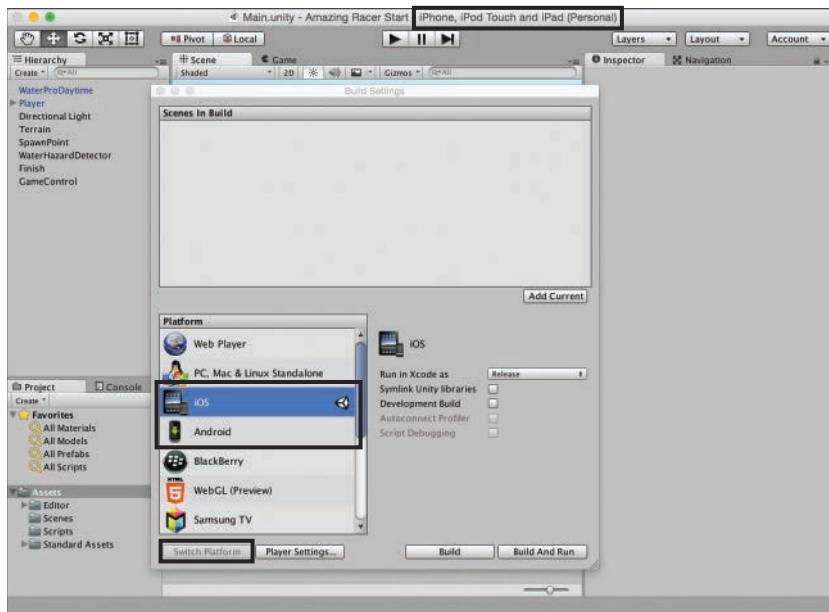


FIGURE 22.2

Switching platforms.

2. Plug in your mobile device with a cable, and enable the Unity Remote by clicking **Edit > Project Settings > Editor** and choosing your device (see Figure 22.3).
3. Check under **Assets > Standard Assets > CrossPlatformInput** in your Project tab to see if the **CrossPlatformInput** standard asset folder has been imported. If not, it can be imported via **Assets > Import Package > CrossPlatformInput**.
4. Ensure there is an **EventSystem** in your Hierarchy, as touch controls will need this. If not, add one by clicking **GameObject > UI > Event System**.
5. Save your project, so that you don't need to wait for conversion again in the future.

TIP

They Won't Get Added Twice

For both the standard asset packages, and the UI Event System, Unity is smart enough not to add them again if they are already there . . . phew!

These are the steps you will need to follow to convert all your projects to mobile. If you have already followed through these steps with Amazing Racer, great. If not, you'll be reminded to in just a moment.

For each project, we will now add either a tilt or a joystick control prefab from the standard assets. We will then ensure the code references the virtual input layer, and tweak the controls for each game.

The great thing about this approach is that if we convert the project back to another platform that we've already configured the controls for, they will continue to work as we left them.

Amazing Racer

Now let's convert Amazing Racer to mobile control. If you have an iOS or Android device, and a cable, you can even try it in the Unity remote. We will be using two control prefabs for this game. Because we used a standard FirstPersonCharacter, the scripts are already set up to call the virtual control layer, so we can skip that step.

Using Tilt Control

For this game, we'll be using two types of control at the same time. The tilt of the device will be used to control the view, and a virtual joystick and button for movement and jumping.

▼ TRY IT YOURSELF

Add Tilt Control

Let's use the CrossPlatformInput package to make device tilt control the player's view:

1. Open the Amazing Racer that you built in Hour 7. If you can't find the file, we have provided it in the book files for this hour.
2. Convert the project to mobile by following the four steps in the previous section (if you haven't done so already). This could take a few minutes, so grab yourself a drink!
3. Navigate to **Assets > Standard Assets > CrossPlatformInput > Prefabs** in your Project tab, and drag the **MobileTiltControlRig** prefab into your hierarchy.
4. Expand the **MobileTiltControlRig** in your Hierarchy, and inspect the **TiltSteerInputV** child object. Change the settings to match Figure 22.3.
5. Now inspect the **TiltSteerInputH** child object. Set the Named Axis to Mouse X, the Tilt Around Axis to Forward Axis, the Full Tilt Angle to 20, and the Centre Angle Offset to 0.
6. Open the Unity Remote app on your device, and run the game in Unity. After a few seconds, you should see the button appear on your mobile device. You should now be able to tilt your device to control the view—cool hey!

7. Save the project, as you will need it in the next exercise.

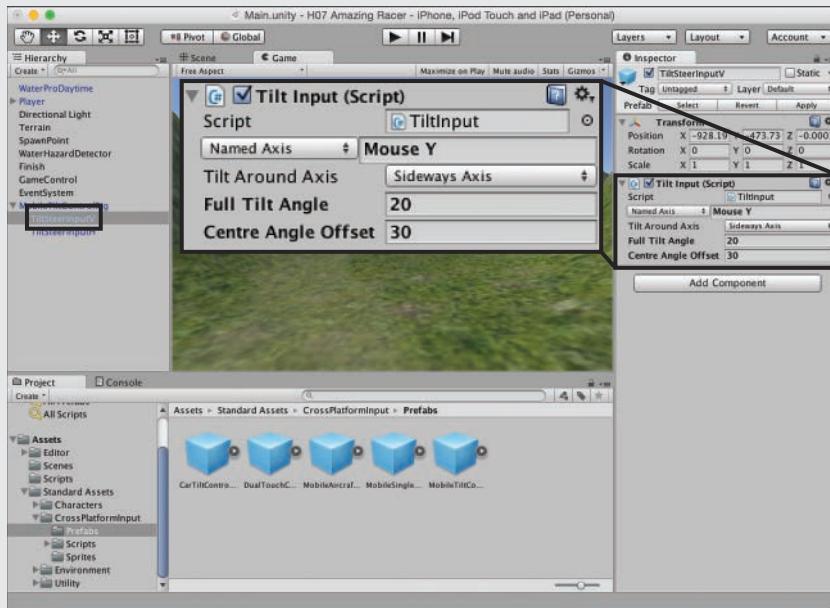


FIGURE 22.3

The tilt control settings.

Using a Touch Joystick

The next thing we need to do is provide controls for moving and jumping on a mobile device. There is a standard asset called `MobileSingleStickControl` that is ideal for this. It uses the Unity UI to put a virtual joystick and button on the screen.

CAUTION

Remember the Event System

A common cause of your control system not working at this stage is not having a UI Event System in your game. Without this, the system cannot respond to your touch inputs.

TRY IT YOURSELF

Add Joystick and Button

Let's use Unity MobileSingleStickControl package to create a virtual joystick:

1. Navigate to Assets > Standard Assets > CrossPlatformInput > Prefabs in the Project tab, and find the **MobileSingleStickControl** prefab. Drag this to your Hierarchy as you did for the **MobileTiltControlRig**.
2. Play the game with the Unity Remote attached. See if the joystick stays in place after you first release it. If so, great, Unity must have fixed the bug. If not, follow the steps below.

Fixing the Disappearing Joystick

At the time of writing, there's a bug in the provided assets that makes the joystick move off the bottom-left of the screen after it is first released. If you experience this bug, then follow these steps to fix it:

1. Expand the **MobileSingleStickControl** game object in your Hierarchy, and click on the **MobileJoystick** child object. Under **Joystick (Script)** in the inspector, double-click the word **Joystick** to edit the **Joystick.cs** script (see Figure 22.4).

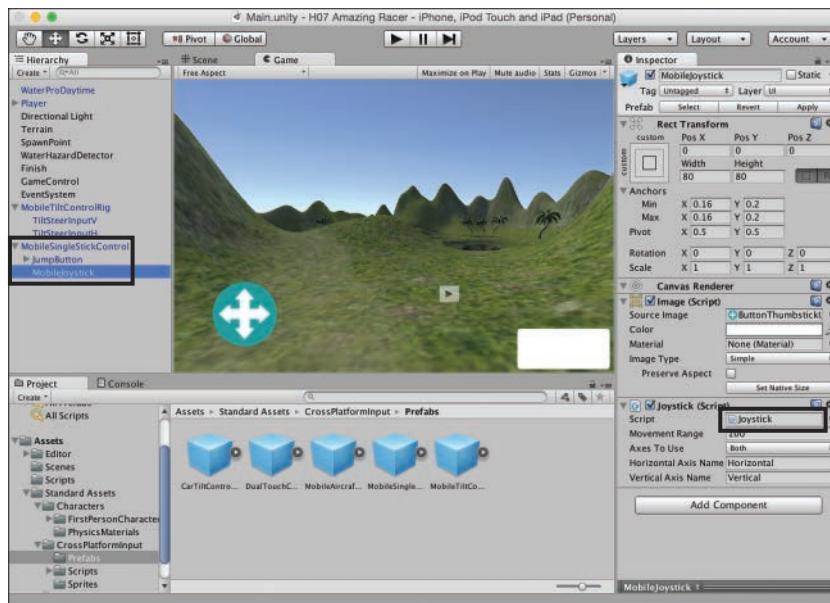
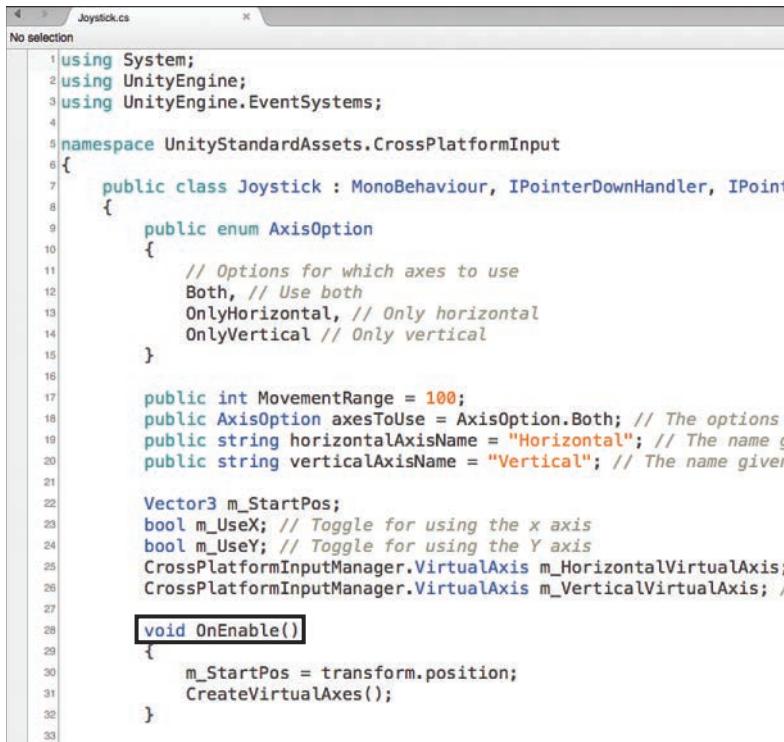


FIGURE 22.4
Finding the offending joystick code.

2. Look to see if there is a `void OnEnable()` method in the code (see Figure 22.5). If so, change its name to `void Start()`. This will fix the disappearing joystick bug.



```
1 using System;
2 using UnityEngine;
3 using UnityEngine.EventSystems;
4
5 namespace UnityStandardAssets.CrossPlatformInput
6 {
7     public class Joystick : MonoBehaviour, IPointerDownHandler, IPoint
8     {
9         public enum AxisOption
10        {
11             // Options for which axes to use
12             Both, // Use both
13             OnlyHorizontal, // Only horizontal
14             OnlyVertical // Only vertical
15         }
16
17         public int MovementRange = 100;
18         public AxisOption axesToUse = AxisOption.Both; // The options
19         public string horizontalAxisName = "Horizontal"; // The name g
20         public string verticalAxisName = "Vertical"; // The name given
21
22         Vector3 m_StartPos;
23         bool m_UseX; // Toggle for using the x axis
24         bool m_UseY; // Toggle for using the Y axis
25         CrossPlatformInputManager.VirtualAxis m_HorizontalVirtualAxis;
26         CrossPlatformInputManager.VirtualAxis m_VerticalVirtualAxis;
27
28         void OnEnable()
29         {
30             m_StartPos = transform.position;
31             CreateVirtualAxes();
32         }
33     }
}
```

FIGURE 22.5

The method that needs renaming to Start().

3. Save your script, and if asked about converting line endings, click Convert.
4. Run your game again with the Unity Remote app connected, and it should behave fine.

NOTE

Unity Remote and Dual Touch

The Unity Remote can be a little “touchy” with dual touch. You will get better results if you use the joystick or the jump button, but not both at the same time. Don’t worry, when deployed to an actual device this multitouch works fine!

Chaos Ball

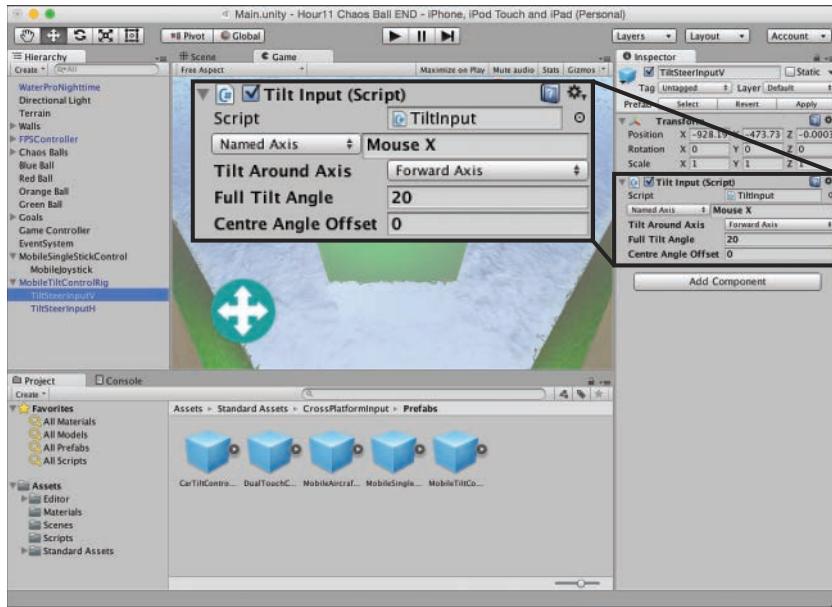
This game will use exactly the same control system as Amazing Racer. We also use a standard FirstPersonCharacter controller, which already talks to the virtual control layer. Therefore, all we need to do is convert our project and then add and tweak our controls.

To convert your project, follow these steps:

1. Convert your project to mobile by following the steps under “Converting Projects to Mobile” in the first section of this hour.
2. Drag both the **MobileSingleStickControl** and **MobileTiltControlRig** prefabs into your Hierarchy, exactly as you did in Amazing Racer.
3. Inspect the **MobileSingleStickControl** and delete the **JumpButton** as we don’t need it (see Figure 22.6). You will get a message about losing the prefab connection; just click Continue, as we want to make this control nonstandard.
4. Inspect the **TiltSteerInputV** child of **MobileTiltControlRig**, and change its settings to those in Figure 22.7. This will make the joystick control the movement of the player.
5. Inspect the **TiltSteerInputH** child object, and change the Named Axis to Mouse Y, the Tilt Around Axis to Sideways. That will make vertical movement of the stick control vertical looking (the reason it’s **TiltSteerInputH** is the device is on its side). Now set the Full Tilt Angle to 20, and the Centre Angle Offset to 30 to make the game more comfortable to play.
6. Play the game with the Unity Remote connected, and note that you can now control the paddle. It’s a bit chaotic, but that’s the point of this game!

**FIGURE 22.6**

Deleting the button we don't need.

**FIGURE 22.7**

The settings for TiltSteerInputV.

Captain Blaster

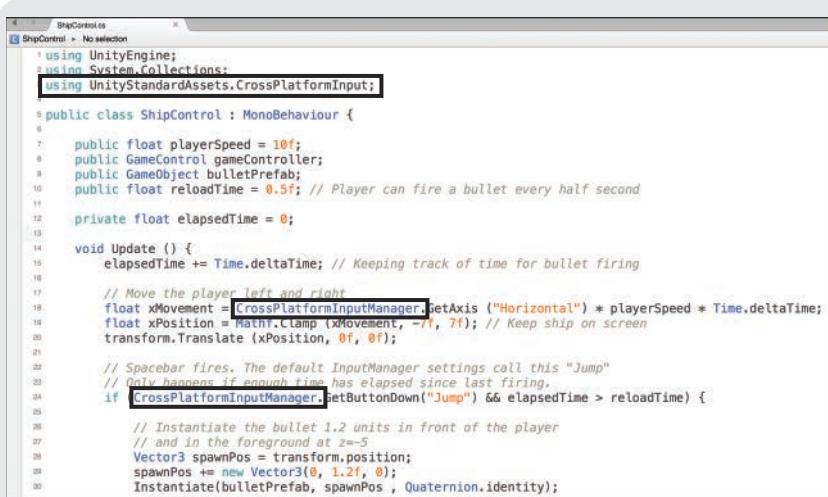
This game is easier to convert in some ways, as we only need to add the `MobileSingleStickControl` for movement and firing. However, we did not use a player prefab like `FirstPersonCharacter`, and our code was not written to read from the virtual controls. You will learn how to make a minor modification to our code, so that it reads from the virtual controls.

TRY IT YOURSELF

Convert Captain Blaster

Let's see what's involved in converting this 2D game to mobile control:

1. Convert your project to mobile by following the steps under “Converting Projects to Mobile” in the first section of this hour. Note you will need to import an asset pack this time.
2. Navigate to `Assets > Standard Assets > CrossPlatformInput > Prefabs` in your Project tab, and drag the `MobileSingleStickControl` prefab into your hierarchy.
3. Open your `ShipControl.cs` script. Import the `UnityStandardAssets.CrossPlatformInput` namespace, and replace both of your `Input.` calls with `CrossPlatformInputManager`. See Figure 22.8 for both changes. Now your code reads from the virtual controls.



```

4 ShipControl.cs
5 ShipControl > No selection
6
7 using UnityEngine;
8 using System.Collections;
9 using UnityStandardAssets.CrossPlatformInput;
10
11 public class ShipControl : MonoBehaviour {
12
13     public float playerSpeed = 10f;
14     public GameController gameController;
15     public GameObject bulletPrefab;
16     public float reloadTime = 0.5f; // Player can fire a bullet every half second
17
18     private float elapsedTime = 0f;
19
20     void Update () {
21         elapsedTime += Time.deltaTime; // Keeping track of time for bullet firing
22
23         // Move the player left and right
24         float xMovement = CrossPlatformInputManager.GetAxis ("Horizontal") * playerSpeed * Time.deltaTime;
25         float xPosition = Mathf.Clamp (xMovement, -7f, 7f); // Keep ship on screen
26         transform.Translate (xPosition, 0f, 0f);
27
28         // Spacebar fires. The default InputManager settings call this "Jump"
29         // Only handle if enough time has elapsed since last firing.
30         if (CrossPlatformInputManager.GetButtonDown ("Jump") && elapsedTime > reloadTime) {
31
32             // Instantiate the bullet 1.2 units in front of the player
33             // and in the foreground at z=-5
34             Vector3 spawnPos = transform.position;
35             spawnPos += new Vector3(0, 1.2f, 0);
36             Instantiate(bulletPrefab, spawnPos , Quaternion.identity);
37
38         }
39     }
40
41 }
```

FIGURE 22.8

Adjusting your code to read from the virtual controls.

4. Play the game with the Unity Remote attached. The joystick should now move the ship, and the jump button fires the gun. Well done!

Now that you know how to read from the virtual controls in code, you may as well do this in any game that you may want to make cross-platform in the future.

Gauntlet Runner

The conversion process for this game is almost identical to Captain Blaster, because this game also only needs the joystick control and also needs its code modifying. If you fancy a challenge, why not try getting this working yourself now using what you've learnt before reading on?

So, here's our solution. Follow the first two steps of the process in the earlier "Convert Captain Blaster" exercise. Now open Player.cs and make the changes highlighted in Figure 22.9 below. That's it, a nice easy conversion to finish on!

```

Player.cs
Player > Update()
1 using UnityEngine;
2 using System.Collections;
3 using UnityStandardAssets.CrossPlatformInput;
4
5 public class Player : MonoBehaviour {
6
7     public GameControl control;
8     private Animator anim;
9
10    public float strafeSpeed = 4f;
11    private bool jumping = false;
12
13    void Start () {
14        anim = GetComponent<Animator>();
15    }
16
17    void Update () {
18        float xMove = CrossPlatformInputManager.GetAxis ("Horizontal") * Time.deltaTime * strafeSpeed;
19        transform.Translate (xMove, 0f, 0f);
20
21        if (transform.position.x > 3) {
22            transform.Translate (3f, 0, 0);
23        } else if (transform.position.x < -3) {
24            transform.Translate (-3f, 0, 0);
25        }
26
27        if (CrossPlatformInputManager.GetButtonDown ("Jump")) {
28            anim.SetTrigger ("Jump");
29        }
30    }
}

```

FIGURE 22.9
Adjusting your code to read from the virtual controls.

Summary

In this hour, you rebuilt your four previous games to include mobile device controls. You started with *Amazing Racer*, where we learnt about converting our project to mobile and adding both a joystick and tilt control. From there, you modified *Chaos Ball* in a similar way. You changed gears in the next game, *Captain Blaster*, where you needed to modify your player script a little. In your final game, *Gauntlet Runner*, you reinforced what you learnt by making very similar changes again.

Q&A

- Q. Some of the games didn't seem to translate well to mobile. Is that normal?**
- A.** Yes, it is. Often, when a game is not made with mobile platforms in mind, it is difficult to transition it. Computers and gaming consoles have many more control options available to them than the simple mobile device. Always ask yourself when designing a game if a mobile version is possible in the future.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. How did you handle needing to use the screen to both jump and look in *Amazing Racer*?
2. What are the three main steps in converting a game to mobile?
3. What is the namespace we need to import to use virtual controls?
4. What is the command we need to use to read from virtual controls?

Answers

1. We used tilt control to look, and a virtual joystick to move and jump.
2. Switch the platform, check your code, and add the mobile controls.
3. UnityStandardAssets.CrossPlatformInput.
4. CrossMobileInputModule.

Exercise

In this hour, you worked through a lot of control mechanics on four different games. When developing games, major mechanics overhauls can sometimes take place. It is not uncommon to change how inputs or controls work to try to build a better user experience. As with any major change, always retest afterward to see what impact the changes had. For this exercise, go back through and replay these games again. This time, play them with mobile controls. Like always, take notes on what you like and what you don't like. When you finish playing, compare these notes to the notes you took when you originally made the games. (You kept those, right?) See what changes you can make to the controls, or the game, to make the *mobile* experience better. Try to implement some or all of those changes.

This page intentionally left blank

HOUR 23

Polish and Deploy

What You'll Learn in This Hour:

- ▶ How to manage scenes in a game
- ▶ How to save data and objects between scenes
- ▶ The different player settings
- ▶ How to deploy a game

In this hour, you learn all about polishing a game and deploying it. You start by learning how to move about different scenes. Then, you explore ways to persist data and game objects between scenes. From there, you take a look at the Unity player and its settings. You then learn how to build and deploy a game.

Managing Scenes

So far, everything you have done in Unity has been in the same scene. Although it is certainly possible to build large and complex games in this way, it is generally much easier to use multiple scenes. The idea behind a scene is that it is a self-contained collection of game objects. Therefore, when transitioning between scenes, all existing game objects are destroyed, and all new game objects are created. However, you can prevent this, as discussed in the next section.

NOTE

What Is a Scene? Revisited

What a scene is was discussed early on in this book. It is time, however, to revisit that concept with the knowledge you now possess. Ideally, a scene is like a level in a game. With games that get consistently harder or games that have dynamically generated levels, though, this is not necessarily true. Therefore, it can be good to think of scenes as a common list of assets. A game consisting of many levels that use the same objects can actually consist of one scene. It is only when you need to get rid of a bunch of objects, and load a bunch of new objects, that the idea of a new scene really becomes necessary. Basically stated, don't split levels into different scenes just because you can. Only create new scenes if required by the gameplay and asset management.

Establishing Scene Order

Transitioning between scenes is relatively easy. It just requires a little setup to function. The first thing you do is add the very scenes of your project to the project's build settings, as follows:

1. Open the build settings by clicking File > Build Settings.
2. With the Build Settings dialog open, click and drag any scenes you want in your final project into the Scenes in Build window (see Figure 23.1).

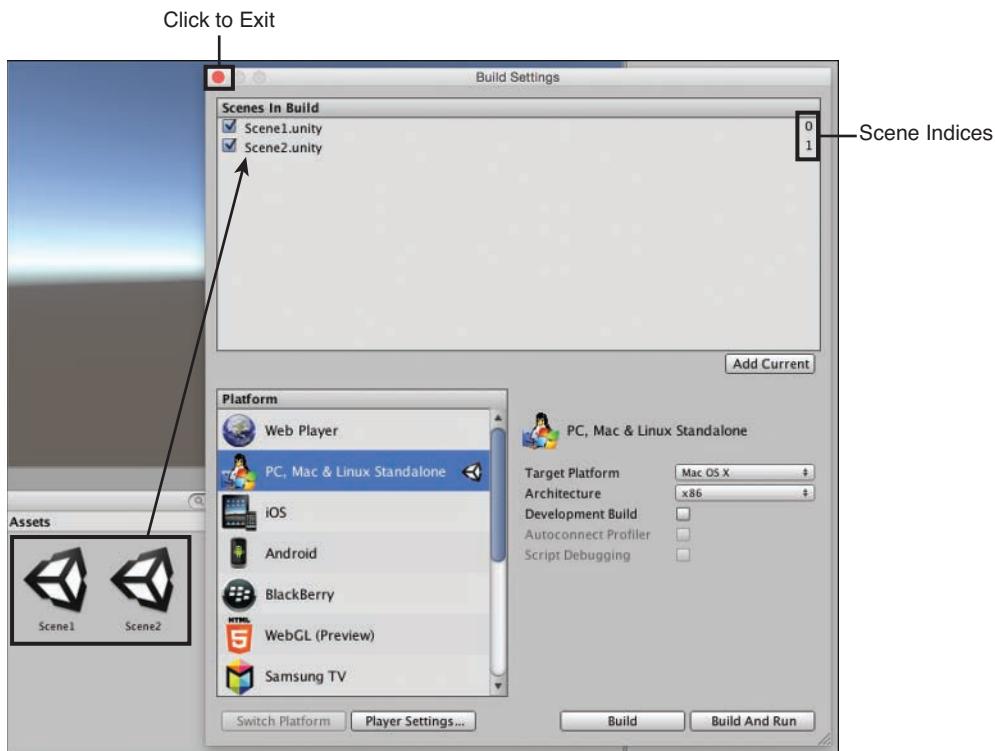


FIGURE 23.1
Adding scenes to the build settings.

3. Pay attention to the number that appears next to the scenes in the Scenes in Build window. These are used later. When done, click the X in the upper-right corner to exit the window. If you are on a PC, you have a similar button in the upper-right corner.

Now the scenes can be referenced and changed.

TRY IT YOURSELF ▼

Adding Scenes to Build Settings

In this exercise, you add scenes to the build settings of a project. Keep the project you make here; you will be using it in the next section:

1. Create a new project. Add a new folder under Assets named **Scenes**.
2. Click **File > New Scene** to create a new scene and then **File > Save Scene** to save it. Save the scene in the Scenes folder as **Scene1**. Repeat this step to save a **Scene2**.
3. Open the build settings (click **File > Build Settings**). Drag Scene1 into the Scenes in Build window first, and then drag Scene2 in. Scene1 should have an index of 0, and Scene2 should have an index of 1. Save the project for later.

Switching Scenes

Now that the scene order is established, switching between them is easy. To change scenes, use the method `LoadLevel()`, which is a part of the Application object. This method takes a single parameter that is either an integer representing the scene's index or a string representing the scene's name. Therefore, to load a scene that has a name of GameOverScene and has an index of 4, you could write either of these two lines:

```
Application.LoadLevel (4) ;           // Load by index  
Application.LoadLevel (GameOverScene) ; // Load by name
```

This method call immediately destroys all existing game objects and loads the next scene. Note that this command is immediate and irreversible, so make sure that it is what you want to do before calling it. The LevelManager prefab in the Exercise of Hour 14 used this method.

Persisting Data and Objects

Now that you have learned how to switch between scenes, you have undoubtedly noticed that data doesn't transfer during the switch. In fact, so far all of your scenes have been completely self-contained, with no need to save anything. In more complex games, however, saving data (often called *persisting*) becomes a real necessity. In this section, you learn how to keep objects from scene to scene and how to save data to a file to access later.

Keeping Objects

An easy way to save data in between scenes is just to keep the objects with the data alive. For example, if you have a player object that has scripts on it containing lives, inventory, score, and so on, the easiest way to ensure that this large amount of data makes it into the next scene is

just to make sure that it doesn't get destroyed. There is an easy way to accomplish this, and it involves a method called `DontDestroyOnLoad()`. The method `DontDestroyOnLoad()` takes a single parameter that is the game object that you want to save. Therefore, if you want to save a game object that was stored in a variable named `Brick`, you could write the following:

```
DontDestroyOnLoad (Brick) ;
```

Because the method takes a game object as a parameter, another great way for objects to use it is to call it on themselves using the `this` keyword. For an object to save itself, you put the following code in the `Start()` method of a script attached to it:

```
DontDestroyOnLoad (this) ;
```

Now when you switch scenes, your saved objects will be there waiting.

TRY IT YOURSELF

Persisting Objects

In this exercise, you save a cube from one scene to the next. This exercise requires the project created previously this hour. If you have not completed it yet, do so before continuing. Be sure to save this project; you will be using it again in the next section:

1. Load the project created previously. Ensure that `Scene1` is the currently loaded scene. Add a 3D Cube, and position it at `(0, 0, 0)`.
2. Create a `Scripts` folder. Move your `LevelManger` into this folder. Now create a new script in the folder named `DontDestroy`. Attach the script to the cube and replace the `Start()` method with the following:

```
void Start () {  
    DontDestroyOnLoad (this) ;  
}
```

3. Save and run the scene. Notice now that when you switch scenes, the cube stays. The cube is now persisted between scenes. Be sure to save this project for future use.

Saving Data

Sometimes, you need to save data to a file to access later. Some things you might need to save are the player's score, configuration preferences, or inventory. There are certainly many complex and feature-rich ways to save data, but a simple solution is something called the `PlayerPrefs`. `PlayerPrefs` is an object that exists to save basic data to a file locally on your system. You then use `PlayerPrefs` to pull the data back out.

Saving data to the PlayerPrefs is as simple as supplying some name for the data and the data itself. The methods you use to save the data depend on the type of data. For instance, to save an integer, you call the `SetInt()` method. To get the integer, you call the `GetInt()` method. Therefore, the code to save a value of 10 to the PlayerPrefs as the score and get the value back out would look like this:

```
PlayerPrefs.SetInt ("score", 10);
PlayerPrefs.GetInt ("score");
```

Likewise, there are methods to save strings, `SetString()`, and floats, `SetFloat()`. Using these methods, you can easily persist any data you want to a file.

TRY IT YOURSELF ▼

Using PlayerPrefs

In this exercise, you save data to the PlayerPrefs file. This exercise requires the project created previously this hour. If you have not completed it yet, do so before continuing. We will use the legacy GUI for this exercise. The focus is on PlayerPrefs not UI:

1. Open the project you created previously and ensure that Scene1 is loaded. Add a new script to the scripts folder named **SaveData** and attach it to the Main Camera. Add the following code to the script:

```
string playerName = "";

void OnGUI() {
    playerName = GUI.TextField (new Rect(5, 120, 100, 30), playerName);
    if (GUI.Button (new Rect (5, 180, 50, 50), "Save")) {
        PlayerPrefs.SetString ("name", playerName);
    }
}
```

2. Save Scene1 and load Scene2. Create a new script called **LoadData** and attach it to the Main Camera. Add the following code to the script:

```
string playerName = "";

void Start() {
    playerName = PlayerPrefs.GetString("name");
}

void OnGUI() {
    GUI.Label (new Rect(5, 220, 50, 30), playerName);
}
```

- ▼
- Save Scene2 and reload Scene1. Run the scene. Type your name into the text field and click the **Save** button. Now click the Load Scene2 button to load Scene2. Notice how the name you entered is written on the screen. The data was saved to PlayerPrefs and then reloaded from PlayerPrefs in a different scene.

CAUTION

Data Safety

Although using PlayerPrefs to save game data is very easy, it is also not very secure. The data is stored in an unencrypted file on the player's hard drive. Therefore, players could easily open the file and manipulate the data inside. This could give them an unfair advantage or break the game. Be aware that the PlayerPrefs, just as the name indicates, is intended for saving player preferences. It just so happens that it is useful for other things. True data security is a difficult thing to achieve and is definitely beyond the scope of this book. Just be aware that PlayerPrefs will work for what you need it for now, but in the future you want to look into more complex and secure means of saving player data.

Unity Player Settings

Unity provides several settings that affect how the game works once it is built. These settings are called the *player settings*, and they manage things like the game's icon and supported aspect ratios. There are many settings, and many of them are self-explanatory, but take your time looking through them and learning what they do. You can open the Player Settings window by clicking **Edit > Project Settings > Player**. The Player Settings window will open in the Inspector view.

Cross-Platform Settings

The first settings you see are the cross-platform settings (see Figure 23.2). These are the settings applied to the built game regardless of the platform (Windows, iOS, Android, Mac, and so on) you built it for. Most of the settings found in this section are self-explanatory. The product name is the name that will appear as the title of your game. The icon should be any valid texture image file. Note that the dimensions of the icon have to be a square power of 2, such as: 8 × 8, 16 × 16, 32 × 32, 64 × 64, and so on. If the icon doesn't match these dimensions, the scaling may not work properly, and the icon quality might be very low. You can also specify a custom cursor in the Cursor setting and define where the cursor hotspot is.

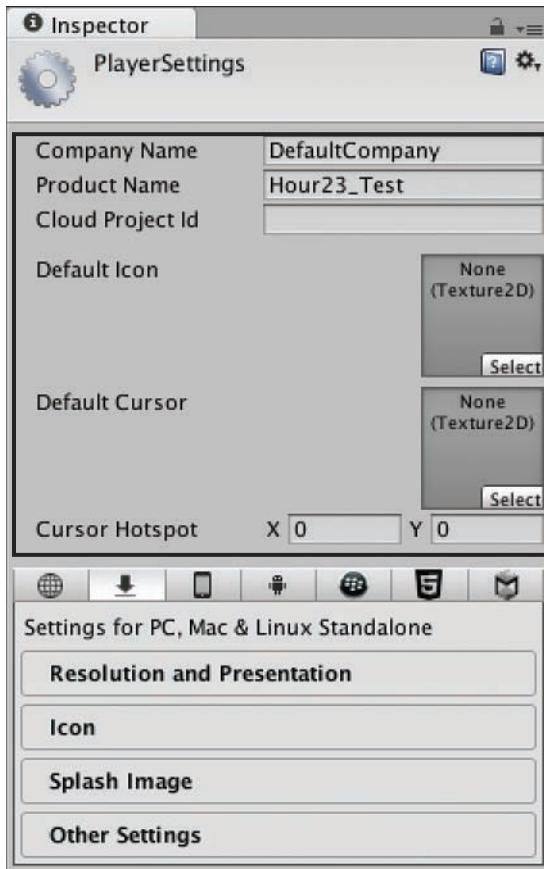


FIGURE 23.2
The cross-platform settings.

Per-Platform Settings

The per-platform settings are the settings specific to each platform. Even though there are several repeat settings in this section, you still have to set up each one of them for every platform you want to build your game for. You can select a specific platform by choosing its icon from the selection bar (see Figure 23.3).

**FIGURE 23.3**

The platform selection bar.

Many of these settings require a more specific understanding of the platform you are building on. These should not be modified until you better understand how that particular platform works. Other settings are rather straightforward and need to be modified only if you are trying to achieve a specific goal. For instance, the Resolution and Presentation settings deal with the dimensions of the game window. For desktop builds, these can be windowed or full screen, with a large array of different supported aspect ratios. By enabling or disabling the different aspect ratios, you allow or disallow different resolutions that the player can choose when playing the game.

The icon settings are auto-populated for you if you specify an icon image for the Default Icon property in the Cross-Platform Settings section. You can see that various sizes of the icon image will be generated based on a single provided image. This is why it is important for the provided image to have the correct dimensions. You can also provide a splash image for your game in the splash image settings. A splash image is an image that is added to the Player Settings dialog when the actual player first starts up the game.

NOTE

Too Many Settings

You probably noticed the large number of settings in the Player Settings that weren't covered in this section. The truth is that most of the properties are already set to default values so that you can just quickly build a game. The other settings all exist to achieve advanced functionality or polish. You shouldn't toy with most of the settings if you don't understand what they do, because they can lead

to strange behaviors or prevent your game from working at all. In short, only use the more basic settings for now until you get more comfortable game-building concepts and the different features you have use.

NOTE

Too Many Players

The term *player* is used a lot in this hour because there are two ways in which the term can be applied. The first, obviously, is the player who actually plays your game. This is a person. The second way the term can be used is to describe the Unity Player. The Unity Player is the window that the game is played in (like a movie player or a TV). This exists on the computer (or device). Therefore, when you hear *player*, it probably means a person, but when you hear *Player Settings*, it probably means the software that actually displays the game.

Building Your Game

Let's say that you've finished building your first game. You've completed all the work and tested everything in the editor. You have even gone through the Player Settings and set everything up the way you wanted. It is now time to build your game. You need to be aware of two settings windows during this process. The first is the Build Settings window, which is where you determine the final results of the build process. The second is the Game Settings window. These settings are seen by the actual player and are how players pick resolution and control configurations.

Build Settings

The Build Settings window contains the terms under which the game is built. It is here that you specify the platform the game will be built under as well as the various scenes in the game. You have seen this dialog once before, but now you should take a closer look at it.

To open the Build Settings dialog, click **File > Build Settings**. Once the Build Settings dialog opens, you can change and configure your game as you want. Figure 23.4 shows the Build Settings dialog and the various items on it.

As you can see, in the Platform section you can specify a new platform to build for. If you choose a new platform, you need to click **Switch Platform** to make the switch. Clicking the **Player Settings** button opens the Player Settings dialog in the Inspector view. You have seen the Scenes in Build section before. This is where you determine which scenes will make it into the game and their order. You also have the various build settings for the specific platform that you chose. The PC, MAC, and Linux Standalone settings are simple and should be self-explanatory. The only thing to note is the Development Build option, which will allow the game to run with a debugger and performance profiler.

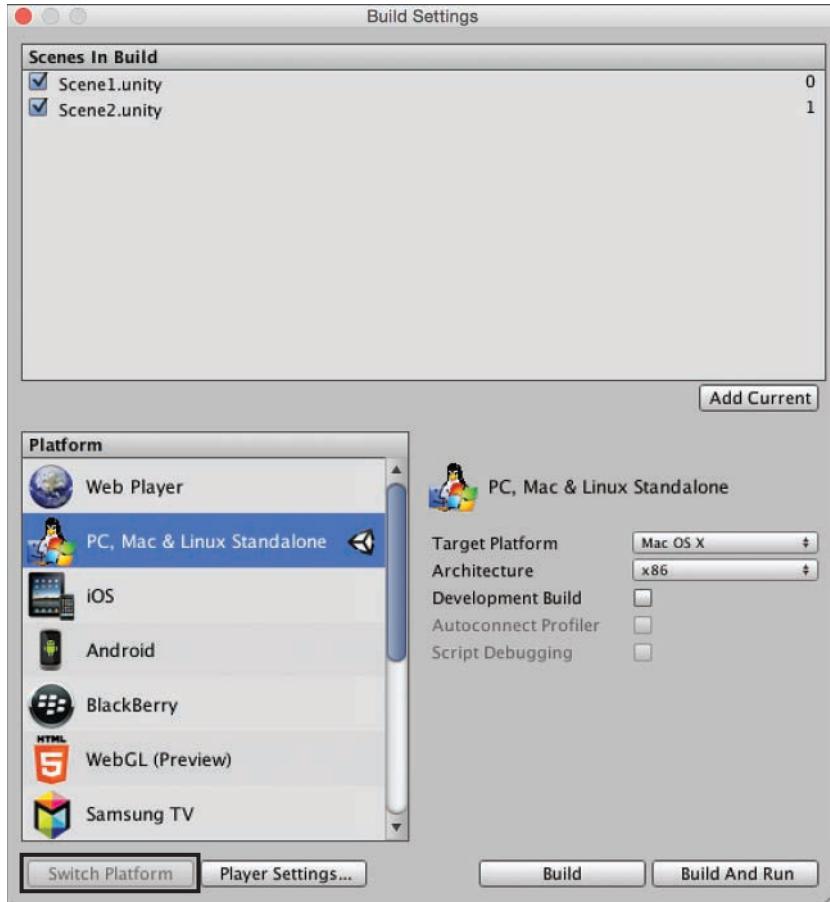


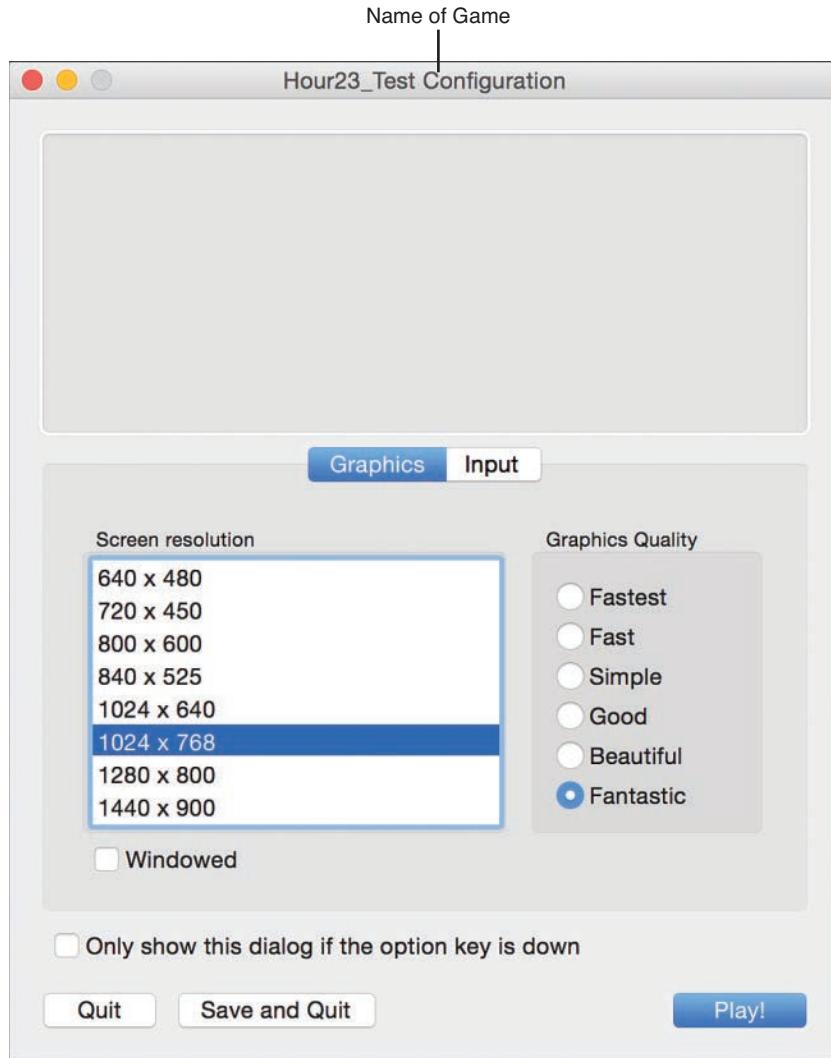
FIGURE 23.4
The Build Settings dialog.

When you are ready to build your game, you can either click **Build** to just build the game or **Build and Run** to run the game after it has finished building. The file that Unity creates will depend on the platform chosen.

Game Settings

When the built game is run from its actual file (not from within Unity), the player will be presented with a Game Settings dialog (see Figure 23.5). From this dialog, players choose options for their game experience.

The first things you may notice is that the name of the game appears in the title bar of the window. Also, any splash image you provided in the Player Settings dialog will appear at the top of

**FIGURE 23.5**

The Game Settings dialog.

this window. This first tab, Graphics, is where players specify the resolution at which they want to play the game. The list of available resolutions is determined by the aspect ratios you allowed or disallowed in the Player Settings dialog. Players can also choose to run the game in a window or full screen and can pick their quality settings.

Players can then switch over to the Input tab (see Figure 23.6). On this tab, players can remap any of the input axes to the buttons that they want.

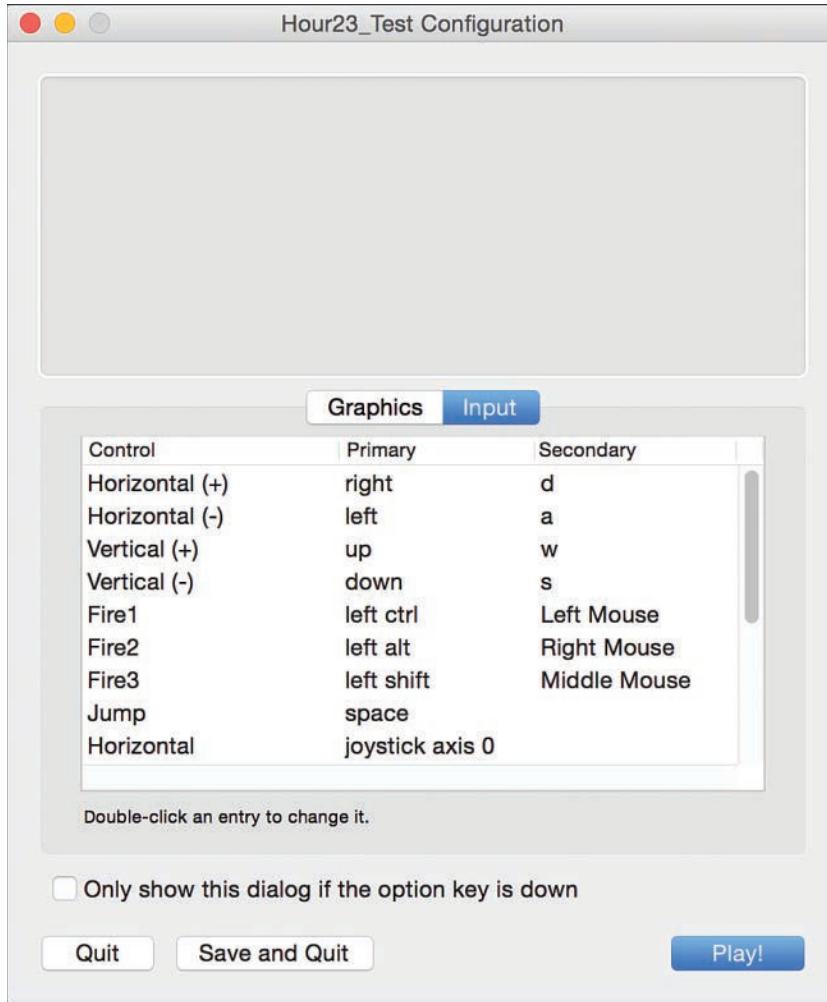


FIGURE 23.6
The input settings.

NOTE

Told You So!

You might recall earlier in this book where you were informed that you should always try to ensure that the input you are reading from a player is based on one of the input axes and not the specific keys. This is why. If you had looked for specific keys instead of axes, the player would have no choice but to use the control scheme you intended. If you think that this isn't a big deal, just remem-

ber that a lot of people out there (people with disabilities, for instance) use nonstandard input devices. If you deny them the ability to remap controls, they might not be able to play your games. Using axes instead of specific keys is a negligible amount of work on your part and can be the difference between players loving or hating your game.

After players choose the settings they want, they just click **Play!** They can finally begin enjoying your game.

Summary

In this hour, you learned all about polishing and building games in Unity. You started by learning how to change scenes in Unity using the `LoadLevel()` method. From there, you learned how to persist game objects and data. After that, you learned about the various player settings. Finally, you wrapped up the hour by learning to build your games.

Q&A

Q. A lot of these settings looked important. Why didn't we cover them?

A. Truth be told, most of those settings are unnecessary for you. The fact is that they aren't important . . . until they are important. Most of the settings are platform specific and are beyond the scope of this book. Instead of spending many pages going over settings you might never use, it is left up to you to learn about them if you ever need them.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. How do you determine the indices of each scene in your game?
2. True or False: Data can be saved using the `PlayerPrefs` object.
3. What dimensions should an icon for your game have?
4. True or False: The input settings in the game settings allows the player to remap all inputs in your game.

Answers

1. After you add the scenes to the list of Scenes in Build, they will have an index assigned to them.
2. True.

3. Game icons should be a square with sides that are powers of 2: 8×8 , 16×16 , 32×32 , and so on.
4. False. The player can only remap inputs that were established based on input axes, not specific key presses.

Exercise

In this exercise, you build a game for your desktop operating system, and experiment with the various features. There isn't much to this exercise, and you should spend most your time trying out various settings and watching their impact. Because this is just an example to get you building your games, there isn't a completed project to look at in the book assets:

1. Pick any project you have created previously, or create a new project.
2. Go into the Player Settings dialog and configure your player however you want.
3. Go into the Build Settings dialog and ensure that you have added your scenes to the Scenes in Build list. Ensure that the **PC**, **MAC**, and **Linux Standalone** platform is chosen. Build the game.
4. Locate the game file that you built and run it. Experiment with the different game settings and see how they affect the gameplay.

HOUR 24

Wrap Up

What You'll Learn in This Hour:

- ▶ What you've accomplished so far
- ▶ Where to go from here
- ▶ What resources are available to you

In this hour, you wrap up your journey with Unity. You start by looking at exactly what you've done so far. From there, you see where you can go to continue improving your skills. Then you are introduced to the various resources available to help you continue learning.

Accomplishments

When you have been working on something for a significant amount of time, you may sometimes forget everything that you have accomplished along the way. It is helpful to reflect on the skills you had when you began learning something and compare them to the skills you have now. There is a lot of motivation and satisfaction to be found in discovering your progress. Let's look at some numbers.

Your 19 Hours of Learning

First and foremost, you spent 19 hours (possibly more) intensely learning the various elements of game development with Unity 5. Here are some of the things you have learned:

- ▶ How to use the Unity editor and many of its windows and dialogs.
- ▶ About game objects, transforms, and transformations. You learned about 2D versus 3D coordinate systems and about local versus world coordinate systems. You became a pro at using Unity's built-in geometric shapes.
- ▶ About models. Specifically, you learned how models consist of textures and shaders applied to materials, which in turn are applied to meshes. You learned that meshes are made up of triangles that consist of many points in 3D space.

- ▶ How to build terrain in Unity. You sculpted unique landscapes and gave yourself the tools needed to build any kind of world you could ever dream of. (How many people can say that?) You improved those worlds with ambient effects and environmental detail.
- ▶ All about cameras and lights.
- ▶ To program in Unity. If you had never programmed before this book, that's a big deal. Good job!
- ▶ About collisions, physical materials, and raycasting. In other words, you took your first steps in object interactions through physics.
- ▶ About prefabs and instantiation.
- ▶ How to build UIs using Unity's powerful User Interface controls.
- ▶ How to control players through Unity's character controllers. On top of that, you built a custom 2D character controller to use in your own projects.
- ▶ How to make awesome particle effects using various particle systems. You learned to use the new Shuriken system, but you also got to try out some legacy system effects.
- ▶ The legacy animation system. This includes learning about the anatomy of animations and a little bit about how they are made.
- ▶ How to use Unity's new Mecanim animation system. While learning that, you learned how to remap the rigging on a model to use animations that weren't made specifically for it. You also learned how to edit animations to make your own animation clips.
- ▶ How to manipulate audio in your projects. You learned how to work with both 2D and 3D audio, in addition to how to loop and swap audio clips.
- ▶ How to work with games made for mobile devices. You learned how to test games with the Unity Remote and to utilize a device's accelerometer and multitouch screen.
- ▶ How to polish a game by using multiple scenes and data persistence. You learned how to build and play your games.

That's quite a list, and it's not even complete. As you read through this list, I hope you remembered experiencing and learning each of these items. You've learned a lot!

Four Complete Games

Over the course of this book, you created four games: *Amazing Racer*, *Chaos Ball*, *Captain Blaster*, and *Gauntlet Runner*. You designed each of these games. You worked through the concept, determined the rules, and came up with the requirements. Once done, you built all the entities of the games. Every object, player, world, ball, meteor, and more was put in the games specifically by

you. You wrote all the scripts and built all the interactivity into the game. Then, most importantly, you tested all the games. You determined their strengths and their weaknesses. You played them, and you had peers play them. You considered how they could be improved, and you even tried to improve them yourself. Take a look at some of the mechanics and game concepts you used:

- ▶ *Amazing Racer*: A 3D foot-racing game against the clock. This game utilized the built-in first-person character controller as well as fully sculpted and textured terrain. The game used water hazards, triggers, and lights.
- ▶ *Chaos Ball*: Another 3D game that truly earns its namesake of *Chaos*. This game featured a large amount of collision and physical dynamics. You utilized physics materials to build a bouncy arena. You even implemented corner goals that turned specific objects into kinematics.
- ▶ *Captain Blaster*: A retro-style 2D space shooter. This is the first game to use a scrolling background and 2D effects. It is also the first game you made where the player can lose. Third-party models and textures ensured that this game had a high level of graphical style.
- ▶ *Gauntlet Runner*: A 3D-ish running game where you had to collect power ups and avoid obstacles. This game utilized Mecanim animations and third-party models, as well as clever manipulations of texture coordinates to achieve a 3D scrolling effect.

Don't forget that you also went back and modified each of these games to work on a mobile device. You have gained experience in designing games, building them, testing them, and updating them for new hardware. Not bad. Not bad at all.

Over 50 Scenes

Over the course of this book, you created over 50 scenes while following along. Let that number sink in for a moment. That means that while reading through this book, you specifically got hands-on with at least 58 different concepts. That is quite a lot of experience for you to draw upon.

By now, you probably get the point of this section. You've done a lot, and you should be proud of that. You have personally used a huge part of the Unity game engine. That knowledge will serve you well as you go forward.

Where to Go from Here

Even though you have completed this book, you are far from done with your education in making games. In fact, it is fairly accurate to say that no one is ever truly done learning in an industry that moves as quickly as this one. That said, here is some advice on what you can do to keep going.

Make Games

No, seriously, make games. This cannot be overstated. If you are someone who is trying to learn more about the Unity game engine, someone who is trying to find a game job, or someone who has a game job and is looking to get better, make games. A common misconception with people newer to the game (or any software) industry is that knowledge alone will get you a job or improve your skills. This couldn't be further from the truth. Experience is king. Make games. They don't even have to be big games. Start by making several smaller games like the ones you've done in this book. In fact, trying a large game right away can lead to frustration and disappointment. No matter what you decide to do, though, make games (was that mentioned yet?).

Work with People

There are many local and online collaborative groups looking to make games for both business and pleasure. Join them. In fact, they would be lucky to have someone with as much Unity experience as you. Remember, you have four games under your belt already. Working with others teaches you a lot about group dynamic. Furthermore, working with others allows you to achieve higher levels of complexity in the games you can make. Try to find artists and sound engineers to make your games full of rich media goodness. You will find that working in teams is the best way to learn more about your strengths and weaknesses. It can be a great reality check as well as a confidence boost.

Write About It

Writing about your games and your game development endeavors can be a great boon to your personal progress. Whether you start a blog or just keep a personal notebook, your observations will serve you well in the present and in retrospect. Writing can also be a great way to hone your skills and collaborate with others. By putting your ideas out there, you can receive feedback and learn through the input of others.

Resources Available to You

Many resources are available to you to continue your education both on the Unity game engine and in game development in general. First and foremost is the Unity documentation. This manual is the official resource for all things Unity. It is important to know that this site (<http://docs.unity3d.com>) covers Unity from a technical approach. Don't think of the site as much of a learning tool as it is a manual.

Unity also provides a great assortment of online training on their Learn site. You can access this site from <http://unity3d.com/learn>. There, you will find many videos, projects, and other resources to help you improve your skills.

If you find that you have a question that you cannot answer with these two resources, the Unity community is very helpful. There is also the Unity Answers site, at <http://answers.unity3d.com>. This is where you ask specific questions and get direct answers from Unity pros.

Aside from the official Unity resources, several game development sites are available to you. Two of the more popular ones are <http://www.gamasutra.com> and <http://www.gamedev.net>. Both of these sites have large communities and regularly publish articles. Their subject matter is not limited to Unity, so they can provide a large and unbiased source of information.

Summary

In this hour, you reviewed everything you have done so far. You also looked forward. You started by examining all the things you have accomplished over the course of this book. Then, you looked at some of the things you can learn here to continue improving your skills. Finally, you looked at some of the free resources available to you on the Internet.

Q&A

- Q.** After reading this hour's materials, I can't help but feel that you think we should make games. Is that true?
- A.** Yes. I believe I mentioned it a few times. I cannot stress enough how important it is to continue to hone your skills through practice and creativity.

Workshop

Take some time to work through the questions here to ensure that you have a firm grasp of the material.

Quiz

1. Can Unity be used to make both 2D and 3D games?
2. Should you be proud of the things you have accomplished so far?
3. What is the single best thing you can do to continue increasing your skills in game development?
4. Have you learnt everything you need to about Unity?

Answers

1. Absolutely!
2. Absolutely again!

3. Keep making games, and sharing them with people.
4. No, you should never stop learning!

Exercise

The theme behind this final hour is that of retrospect and solidifying the things that you have learned. The final exercise for this book follows that same theme. It is common in the game industry to write something called a *post-mortem*. The idea behind a post-mortem is that you write an article about a game you have made, with the intention of other people reading it. In a post-mortem, you analyze the things that worked in your process and the things that didn't. You aim to inform others of the pitfalls that you discovered so that they won't fall into the same.

In this exercise, you write a post-mortem about one of the games you made. You don't necessarily have to have anyone read it. It is the writing of it that is important. Be sure to spend some time on this, because you might want to read it again further down the road. You will be amazed at the things you found difficult and at the things you found enjoyable.

After writing the post-mortem, print it out (unless you wrote it by hand) and put it in this book. Later, when you come across this book again, be sure to open the post-mortem and read it.

INDEX

A

accelerometers, 357-361
Add Grass Texture dialog, 67
Add Key frame (Animation window), 282
Add Terrain Texture dialog, 59
Add Tree dialog, 65
Albedo property (shader), 44
Alt Negative Button/Alt Positive Button property (axis), 147
Amazing Racer, 103-115, 395
adding scripts, 111-113
creating world, 106-108
design, 103-106
concept, 104
rules, 104-105
game control objects, adding, 109-111
playtesting, 114-115
requirements, 105-106
revisions, 368-371
disappearing joystick fixing, 370-371
tilt control, 368-369
using touch joystick, 369-370
scripts, connecting, 113-114
anchor button, 222, 223
anchors, canvas, 220-223
Angular Drag property (rigidbody), 162
Animation Clip Dropdown (Animation window), 282
Animation property (Texture module), 269

animations, 275-288
assets, 277-279
creating, 279-281, 283-284
Curves editor, 287-288
3D artists, 277
idle, 298-300, 308-309
models, preparing for, 276-277
preparation, 298
Record Mode, 285-287
rigs, 275-276
spritesheet slicing, 278-279
timing, 285
tools, 281-288
2D, 277-279
types, 277-281
walk, 300-302
walk turn, 302-305
window, 281-283
animators, 291-315
animation preparation, 298
assets, configuring, 296-305
blend trees, 310-312
creating, 305-314
idle animation, 298-300, 308-309
parameters, 309-310
rigging models, 293-296
rig preparation, 296-298
scripting, 314-315
states, 310-312
transitions, 312-314
walk animation, 300-302
walk turn animation, 302-305
Animator view, 307-308
apps, Unity Remote, 355
area lights, 86
arenas, Chaos Ball, 175-179
bouncy material, 178
texturing, 177-178
arithmetic operators, 130-131
Aspect drop-down menu (Game view), 16
asset packages, 6
assets, 8
animations, 277-279
configuring, animators, 296-305
importing, 78
terrain, 64
importing, 57-58
Asset Store, models, 39-40
assignment operators, 131
attaching scripts, 123-124
audio, 339-349
audio listener, 339-340
changing clips, 349
priorities, 342
scripting, 346-349
sources, 341-346
starting and stopping, 347-348
testing, 343, 344
Scene view, 343, 344
3D, 341, 345-346
2D, 341, 346
Audio Clip property (audio source), 341
audio clips, importing, 342
audio listeners, 77, 339-340
Audition mode (Scene view), 13

axis

- properties, 147-148
- rotation, 29

Axis property (axis), 148**B**

- background**, Captain Blaster, 240-241
- Background property** (cameras), 91
- baking objects, 83
- Baking property (point lights)**, 82
- base terrain settings, 69
- Best Fit property** (text objects, UIs), 226
- billboards, 67
- blend trees, 310-312
- blocks, methods, 143
- bool variable, 128
- Bounce Combine property** (physics material), 166
- Bounce Intensity property** (point lights), 82
- Bounce property** (Collision module), 266
- Bounciness property** (physics material), 166
- bouncy material, Chaos Ball arena, 178
- Box Collider 2D**, 213
- breaking prefabs, 197
- bugs, halos
- building games, 387-391
- Build Settings dialog**, 387-388
- built-in 3D objects, 36-37
- built-in methods, 127
- built-in objects, 25
- bullets, Captain Blaster, 245
- scripts, 254-255
- Bursts property** (Emission module), 262

buttons

- changing scenes via, 380-381
- UIs, 226-227
- properties, 226-227

Bypass Effects property (audio source), 341

- Bypass Listener Effects property** (audio source), 341
- Bypass Reverb Zones property** (audio source), 341

C

- calling methods**, 145-146
- cameras**, 18, 81, 91-95, 99
 - adding skyboxes to, 71
 - anatomy of, 91-92
 - Captain Blaster, 239-240
 - falling through the world, 77
 - layers, 95-99
 - lens flares, 73-74
 - multiple, 92-93
 - orthographic, 204-205
 - size of, 205
 - picture in picture, 93-95
 - properties, 91-92
 - screen-space, 231
 - sorting layer, 204, 209-211
 - split screens, 93-95
- canvas**, UI, 218-223
 - adding, 218
 - anchor button, 222, 223
 - anchors, 220-223
 - components, 223
 - EventSystem game object, 218
 - Rect Transform, 218-219, 220
 - Render Mode, 230-232
 - screen-space camera, 231
 - screen-space overlay, 230-231
 - world space, 231-232
- Canvas Scaler**, 223

Captain Blaster, 237-255, 395

- background, 240-241
- bullets, 245
- script, 254-255
- camera, 239-240
- controls, 247-255
- design, 237-238
- concept, 237-238
- requirements, 238
- rules, 238
- DestroyOnTrigger script**, 251
- improvements, 255
- meteors, 244-245
- script, 249-250
- spawn, 250-251
- players, 242-243
- revisions, 374-375
- ShipControl script**, 252-253
- triggers, 246
 - script, 251
- UI, 246-247
- world, 238-239
- Cast Shadows property** (Renderer module), 270
- Center property** (colliders), 164
- Chaos Ball**, 173-187, 395
 - arena, 175-179
 - bouncy material, 178
 - texturing, 177-178
 - chaos balls, 181-182
 - colored balls, 182-183
 - control objects, 183-187
 - design, 173-174
 - concept, 174
 - requirements, 174
 - rules, 174
 - game controller, 185-187
 - improving, 187
 - players, 179-180
 - revisions, 372-374
- character controllers**, 75-78
 - adding, 108
- char variable**, 128
- Circle Collider 2D**, 213
- class declaration section**, scripts, 125-126

classes, contents, 126-127
Clear Flags property
 (cameras), 91
Clipping Planes property
 (cameras), 91
code. See also scripts
 comments, 126
 scripting, 119
 scripts, 109, 120
 adding, 111-113
 attaching, 123-124
 basic, 125
 class declaration section,
 125-126
 classes contents, 126-127
 conditionals, 133-135
 connecting, 113-114
 creating, 120-123
 Game Control Script, 186
 GoalScript.cs, 184-185
 iteration, 136-137
 methods, 141-146
 operators, 130-133
 player input, 146-151
 using section, 125
 variables, 128-130
 VelocityScript.cs, 182
code listings
 Default Script Code, 125
 Demonstration of Class and
 Local Block Level, 129
 Game Control Script, 186
 GoalScript.cs, 184-185
 VelocityScript.cs, 182
collaborative groups, 396
colliders, 163-165
 complex, 165
 interaction matrix, 169
 Mesh, 165
 mixing and matching, 164
 physics materials, 165-166
 properties, 164
 trigger, 167-169
 2D, 212-214
 depth, 214
Collides With property (World
 mode), 267

collision, 161-171
 colliders, 163-165
 physics materials, 165-166
 trigger, 167-169
 raycasting, 169-171
 rigidbodies, 161-163
Collision Detection property
 (rigidbody), 162
Collision module (particle system),
 266-268
Collision Quality property (World
 mode), 267
Color by Speed module (particle
 system), 264-265
Color over Lifetime module
 (particle system), 264
Color property (images, UIs), 224
Color property (Color by Speed
 module), 265
Color property (point lights), 82
comments, 126
concept
 Amazing Racer, 104
 Captain Blaster, 237-238
 Chaos Ball, 174
 Gauntlet Runner, 317
conditionals, 133-135
Console window (editor), 126-127
Constraints property
 (rigidbody), 162
controllers
 character, 75-78
 game, 185-187
control objects, Chaos Ball,
 183-187
controls
 Captain Blaster, 247-255
 Gauntlet Runner, 329-335
 script, 330-332
 particle systems, 259
 virtual, 365-366
Cookie property (point lights), 82
cookies, 89-90
coordinate systems, 23-24
 world versus local coordinates,
 24-25
Create New Project dialog, 6,
 175-176
cross-platform input, 365-368
 projects to mobile conversion,
 366-368
 virtual controls, 365-366
cross-platform settings, players,
 384-385
Culling Mask property (cameras), 91
Culling Mask property (point
 lights), 83, 98
Current Frame (Animation
 window), 282
curve editor, particle systems,
 270-272
curves editor, animations,
 287-288
C# variable types, 128
Cycles property (Texture module),
 269

D

Dampen property (Collision
 module), 266
Dampen property (Limit Velocity
 over Lifetime module), 263
data
 persisting, 381-384
 saving, 382-384
Dead property (axis), 148
default particle system, 261-262
Default Script Code listing, 125
deltaPosition property (touch), 359
deltaTime property (touch), 359
Demonstration of Class and Local
 Block Level listing, 129
Depth property (cameras),
 92
Descriptive Name/Descriptive
 Negative Name property
 (axis), 147
design
 accelerometers, 357-358
 Amazing Racer, 103-106

concept, 104
 requirements, 105-106
 rules, 104-105
Captain Blaster, 237-238
 concept, 237-238
 requirements, 238
 rules, 238
Chaos Ball, 173-174
 concept, 174
 requirements, 174
 rules, 174
Gauntlet Runner, 317-318
 concept, 317
 requirements, 318
 rules, 318
UI, 217
DestroyOnTrigger script, **Captain Blaster**, 251
detail object settings, 70
dialogs
 Add Grass Texture, 67
 Add Terrain Texture, 59
 Add Tree, 65
 Build Settings, 387-388
 Create New Project, 6, 175-176
 Game Settings, 388-391
 Importing Packages, 57-58
 Project, 4-5
dimensions, 21-22
 coordinate systems, 23-24
 world versus local coordinates, 24-25
directional lights, 85-86
 cookies, 89
disappearing grass, 68
documentation, 396
Doppler Level property (3D audio), 346
double variable, 128-130
downloading
 models, 39-40
 Unity, 1-3
Drag property (**rigidbody**), 162
Draw Halo property (point lights), 82

Draw mode (**Scene view**), 13
draw order, 2D games, 209-212
 in layer, 212
 sorting layers, 209-211
dual touch, 371
Duration property (particle system), 261
Dynamic Friction property (physics material), 166
Dynamic Friction 2 property (physics material), 166

E

Edge Collider 2D, 213
Edit Collider property (colliders), 164
editor, 4-17
 Console window, 126-127
 Game view, 14-16
 Hierarchy view, 10-11
 Inspector view, 11-12
 Project view, 8-10
 Scene view, 13-14
 toolbars, 16-17
effects
 environment, 71-75
 fog, 73
 lens flares, 73-74
 skyboxes, 71-73
 water, 74-75
 particles, 259
Emission module (particle system), 262
emissive materials, 86
emitter versus particle settings, 268
environments, 63, 78. See also
 terrain; worlds

adding, 107
 billboards, 67
 character controllers, 75-78
 effects, 71-75
 fog, 73
 lens flares, 73-74
 skyboxes, 71-73
 water, 74-75

grass
 disappearing, 68
 painting, 66-68
 realistic, 68
 mobile development, setting up, 354-355
 terrain, settings, 68-70
trees
 generating, 63-70
 wind settings, 70
equality operators, 131-132
EventSystem game object, 218
External Forces module (particle system), 265

F

factories, methods, 143
Field of View property (cameras), 91
fingerId property (touch), 360
first project, creating, 5
Flare property (point lights), 83
flares, lens, 73-74
float variable, 128
Flythrough mode (**Scene view**), 18-19
fog, 73
Force over Lifetime module (particle system), 264
for loop, 137
formats, heightmaps, 54
Friction Combine property (physics material), 166
Friction Direction 2 property (physics material), 166

G

game control
 Captain Blaster, 248-249
game controller
 Chaos Ball, 185-187
game control objects, adding, 109-111

- Game Control script, Gauntlet Runner**, [330-332](#)
- Game Control Script listing**, [186](#)
- game overlay (Scene view)**, [14](#)
- games**
- adding terrain to, [49-51](#)
 - Amazing Racer, [103-115](#), [395](#)
 - adding scripts, [111-113](#)
 - connecting scripts together, [113-114](#)
 - creating world, [106-108](#)
 - design, [103-106](#)
 - game control objects, adding, [109-111](#)
 - playtesting, [114-115](#)
 - attaching scripts, [123-124](#)
 - building, [387-391](#)
 - Captain Blaster, [237-255](#), [395](#)
 - background, [240-241](#)
 - bullets, [245](#)
 - camera, [239-240](#)
 - controls, [247-255](#)
 - design, [237-238](#)
 - improvements, [255](#)
 - meteors, [244-245](#)
 - players, [242-243](#)
 - revisions, [374-375](#)
 - triggers, [246](#)
 - UI, [246-247](#)
 - world, [238-239](#)
 - Chaos Ball, [173-187](#), [395](#)
 - arena, [175-179](#)
 - chaos balls, [181-182](#)
 - colored balls, [182-183](#)
 - control objects, [183-187](#)
 - design, [173-174](#)
 - game controller, [185-187](#)
 - improving, [187](#)
 - players, [179-180](#)
 - revisions, [372-374](#)
 - creating, [396](#)
 - object, [5](#)
 - Gauntlet Runner, [317-336](#), [395](#)
 - controls, [329-335](#)
 - design, [317-318](#)
 - improving, [187](#)
 - players, [179-180](#)
 - revisions, [372-374](#)
 - entities, [321-329](#)
 - Game Control script, [330-332](#)
 - improvement, [335](#)
 - obstacles, [322](#)
 - player, [323-329](#)
 - move script, [333](#)
 - power ups, [321-322](#)
 - revisions, [375-376](#)
 - spawn script, [333-334](#)
 - trigger zone, [322](#)
 - script, [329](#)
 - world, [318-320](#)
 - organization, [9](#)
 - revisions, [365-376](#)
 - Amazing Racer, [368-371](#)
 - Captain Blaster, [374-375](#)
 - Chaos Ball, [372-374](#)
 - cross-platform input, [365-368](#)
 - Gauntlet Runner, [375-376](#)
 - 2D, [201-214](#)
 - adding sprites, [205-209](#)
 - basics, [201-204](#)
 - challenges, [202](#)
 - colliders, [212-214](#)
 - design principles, [201](#)
 - draw order, [209-212](#)
 - orthographic cameras, [204-205](#)
 - rigidbody, [212](#)
 - scene view, [202-204](#)
 - setting, [201-202](#)
 - writing about, [396](#)
 - Game Settings dialog**, [388-391](#)
 - Game view**, [14-16](#)
 - Gauntlet Runner**, [317-336](#), [395](#)
 - controls, [329-335](#)
 - script, [330-332](#)
 - design, [317-318](#)
 - concept, [317](#)
 - requirements, [318](#)
 - rules, [318](#)
 - entities, [321-329](#)
 - Game Control script, [330-332](#)
 - improvement, [335](#)
 - obstacles, [322](#)
 - player, [323-329](#)
 - script, [332-333](#)
 - power ups, [321-322](#)
 - move script, [333](#)
 - revisions, [375-376](#)
 - spawn script, [333-334](#)
 - trigger zone, [322](#)
 - script, [329](#)
 - world, [318-320](#)
- generating terrain**, [49-56](#)
- Geometric properties (colliders)**, [164](#)
- GetComponent**, using, [151-152](#)
- gizmo (scene)**, [14](#)
- Gizmos button (Game view)**, [16](#)
- Gizmo selector (Scene view)**, [14](#)
- GoalScript.cs listing**, [184-185](#)
- Graphical Raycaster**, [223](#)
- grass**
- disappearing, [68](#)
 - painting, [66-68](#)
 - realistic, [68](#)
- Gravity Modifier property (particle system)**, [262](#)
- Gravity property (axis)**, [148](#)
- ground, Gauntlet Runner**, [319](#)
 - scrolling, [319-320](#)
- H**
- halos**, [87-89](#)
- Hand tool**, [17-18](#)
- HDR property (cameras)**, [92](#)
- heightmaps**
- formats, [54](#)
 - sculpting, [51-54](#)
- Hierarchy view**, [10-11](#)
 - prefab instances, [190, 191](#)
- Horizontal and vertical Overflow property (text objects, UIs)**, [226](#)

I

idle animation, 298-300, 308-309

`if/else if` statement, 134-135

`if/else` statement, 134

`if` statement, 133-134, 135

images, UIs, 224-225

 properties, 224

importing

 assets, 78

 audio clips, 342

 models, 37-39

 sprites, 206

 mode, 206-208

 sizes, 208-209

 terrain assets, 57-58

Importing Packages dialog, 57-58

inheritance, prefabs, 190,

 196-197

Inherit Velocity property (particle system), 262

input

 basics, 147-148

 cross-platform, 365-368

 projects to mobile conversion, 366-368

 virtual controls, 365-366

key, 147, 149-150

mouse, 150-151

multi-touch, mobile devices, 359-361

scripting, 146-151

Input Manager, 147, 149

Inspector view, 11-12

 curve editor, 271

 rig preparation, 296

 script preview, 120, 121

Installing unity, 1-4

instances, prefabs, 190

 adding to scenes, 195

Instantiate() method, 197

instantiating prefabs through code, 197

Intensity property (point lights), 82

Interactable property (buttons, UIs), 226

Interpolate property (rigidbody), 162

int variable, 128

Invert property (axis), 148

invisible items, scenes, 97

Is Kinematic property (rigidbody), 162

Is Trigger property (colliders), 164

iteration, 136-137

 for loop, 137

 while loop, 136

J

Joy Num property (axis), 148

K

key codes, 149

Key frames (Animation window), 282

key input, 149-150

L

lakes, creating, 75

layers, 95-99

 order in, 212

 overloading, 95

 sorting, 204, 209-211

Layers drop-down menu, 17, 96

Layout drop-down menu, 17

lens flares, 73-74

license, Unity, activating, 1-3

Lifetime Loss property (Collision module), 266

lighting scenes, 13

lights, 81-90, 99

 area, 86

 baking objects, 82

cookies, 89-90

creating out of objects, 86

directional, 85-86

halos, 87-89

layers, 95-99

point, 82-84

 properties, 82-83

repeat properties, 81

spotlights, 84-85

Limit Velocity over Lifetime

 module (particle system), 263

listings

 Default Script Code, 125

 Demonstration of Class and

 Local Block Level, 129

 Game Control Script, 186

 GoalScript.cs, 184-185

 VelocityScript.cs, 182

LoadLevel() method, 381, 391

local components, accessing, 151-153

local coordinates, *versus* world coordinates, 24-25

logical operators, 132-133

Looping property (particle system), 261

Loop property (audio source), 342

loops

 for, 137

 while, 136

M

managing scenes, 379-381

maps, heightmaps

 formats, 54

 sculpting, 51-54

Mass property (rigidbody), 162

Material property (images, UIs), 224

Material property (colliders), 164

Material property (Renderer module), 270

materials, 41, 43
 emissive, 86
 models, applying to, 45-46
 UIs, 225
Max Distance property (3D audio), 346
Maximize on Play button (Game view), 16
Max Particles (particle system), 262
Max Particle Size property (Renderer module), 270
Mesh colliders, 165
meshes
 versus models, 36
 scaling of, 39
 simple modeling, 37
Metallic property (shader), 44
meteors, Captain Blaster, 244-245
 script, 249-250
 spawn, 250-251
methods, 141-146
 blocks, 143
 built-in, 127
 calling, 145-146
 as factories, 143
 identifying parts, 143
Instantiate(), 197
LoadLevel(), 381, 391
 names, 142
OnGUI(), 361
 parameter list, 142-143
 return type, 142
 writing, 144-145
Min Distance property (3D audio), 346
minimum requirements, Unity, 3
Min Kill Speed property (Collision module), 266
mobile development, 353-361
 accelerometers, 357-361
 environments, setting up, 354-355
 preparing for, 353-356
 Unity Remote app, 355

mobile devices
 multi-touch input, 359-361
 multitudes of, 354
 testing, 356
models, 35-40, 45
 applying textures, shaders, and materials, 45-46
 Asset Store, 39-40
 built-in 3D objects, 36-37
 downloading, 39-40
 importing, 37-39
 versus meshes, 36
 model asset workflow, 41
 preparing for animation, 276-277
 rigging, animators, 293-296
modules, particle systems, 259-270
 Collision, 266-268
 Color by Speed, 264-265
 Color over Lifetime, 264
 default, 261-262
 Emission, 262
 External Forces, 265
 Force over Lifetime, 264
 Limit Velocity over Lifetime, 263
 Renderer, 269-270
 Rotation by Speed, 265
 Rotation over Lifetime, 265
 shape, 263
 Size by Speed, 265
 Size over Lifetime, 265
 Sub Emitter, 269
 Texture Sheet, 269
 Velocity over Lifetime, 263
MonoDevelop, 120, 122, 123
mouse input, 150-151
move script, Gauntlet Runner, 333
 multiple cameras, 92-93
 multiple skyboxes, cameras, 71
multi-touch input, mobile devices, 359-361
mute audio button, 344
Mute property (audio source), 341

N

Name property (axis), 147
names, methods, 142
Navigation property (buttons, UIs), 227
Negative Button/Positive Button property (axis), 147
nested objects, transformations, 32-33
nesting, 11
Normal Direction property (Renderer module), 270
Normalized View Port Rect property (cameras), 91
Normal Map property (shader), 44

O

objects, 21, 25. *See also specific objects*
 baking, 83
 built-in, 25
 built-in 3D objects, 36-37
 consolidating, 176
 control, 183-187
 creating lights out of, 86
 detail settings, 70
 dimensions, 21-22
 EventSystem, 218
 finding, 153-156
 game control, adding, 109-111, 334-335
 keeping, 381-382
 layers, 95-99
 overloading, 95
 modifying components, 157
 nested, transformations, 32-33
 persisting, 381-384
 picture in picture, 94-95
 prefabs, 189
 breaking, 197
 creating, 193-194
 inheritance, 190, 196-197

- instances, 190
 - instantiating through code, 197
 - structure, 190-192
 - updating, 196-197
 - rotation, 29-30
 - scaling, 30-31
 - spin, 283-284
 - target, transforming, 157
 - textures, 41
 - transformations, 26-33
 - hazards, 31-32
 - nested objects, 32-33
 - transforming, 153
 - translation, 27-29
 - obstacles, *Gauntlet Runner*, 322
 - Occlusion Culling property (cameras), 92
 - Offset property (shader), 44
 - On Click () property (UIs), 227-229
 - OnGUI() method, 361
 - operating systems, supported, 3
 - operators, 130-133
 - arithmetic, 130-131
 - assignment, 131
 - equality, 131-132
 - logical, 132-133
 - order, in layer, 212
 - order, scenes, establishing, 380-381
 - organization, projects, 9
 - orthographic cameras, 204-205
 - size of, 205
 - Output property (audio source), 341
- P**
- painting
 - grass, 66-68
 - textures, terrain, 60
 - trees, 63-66
 - parameter list, methods, 142-143
 - parameters, animators, 309-310
- Particle Radius property (Plane mode)**, 266
 - particles**, 257
 - effects, 259
 - making collide, 267-268
 - particle settings versus emitter**, 268
 - particle systems**, 257-272
 - controls, 259
 - creating, 258
 - curve editor, 270-272
 - modules, 259-270
 - Collision, 266-268
 - Color by Speed, 264-265
 - Color over Lifetime, 264
 - default, 261-262
 - Emission, 262
 - External Forces, 265
 - Force over Lifetime, 264
 - Limit Velocity over Lifetime, 263
 - Renderer, 269-270
 - Rotation by Speed, 265
 - Rotation over Lifetime, 265
 - shape, 263
 - Size by Speed, 265
 - Size over Lifetime, 265
 - Sub Emitter, 269
 - Texture Sheet, 269
 - Velocity over Lifetime, 263
 - particles, 257
 - making collide, 267-268
 - Unity, 257-258
 - Pause button (Game view)**, 15
 - per-platform settings, players, 385-387
 - persisting objects, 382
 - phase property (touch), 360
 - physics**
 - collision, 161-171
 - colliders, 163-165
 - physics materials, 165-166
 - raycasting, 169-171
 - rigidbodies, 161-163
 - trigger, 167-169
 - 2D, 212-214
 - colliders, 212-214
 - rigidbody, 212
 - physics materials**, 165-166
 - picture in picture, 93-95
 - Pitch property (audio source)**, 342
 - Place Tree tool, 64-65
 - Planes property (Plane mode), 266
 - Planes/World property (Collision module)**, 266
 - Play button (Game view)**, 15
 - player input**
 - basics, 147-148
 - key, 147, 149-150
 - mouse, 150-151
 - scripting, 146-151
 - PlayerPrefs file, saving data to**, 382-383
 - players**
 - Captain Blaster, 242-243
 - Chaos Ball, 179-180
 - Gauntlet Runner, 323-329
 - script, 332-333
 - settings, 384-387
 - Play On Awake property (audio source)**, 342
 - Play On Wake property (particle system)**, 262
 - playtesting *Amazing Racer*, 114-115
 - point lights**, 82-84
 - properties, 82-83
 - scenes, adding to, 83
 - points, 23
 - Polygon Collider 2D**, 213
 - position property (touch), 360
 - power up, *Gauntlet Runner*, 321-322
 - script, 333
 - prefabs**, 189-197
 - breaking, 197
 - creating, 193-194
 - inheritance, 190, 196-197
 - instances, 190
 - adding to scenes, 195

instantiating through code, 197
 structure, 190-192
 updating, 196-197
Preserve Aspect property
 (images, UIs), 224
Preview (Animation window), 282
Prewarm property (particle system), 261
priorities, audio, 342
Priority property (audio source), 342
private variables, 129-130
Project Dialog, 4-5
Projection property (cameras), 91
projects. See also games
 adding terrain to, 49-51
 attaching scripts, 123-124
 converting to mobile, 366-368
 creating first, 5
 organization, 9
Project view, 8-10
 prefabs, 189-190, 191
properties
 Animation window, 282
 audio source, 341-342
 axis, 147-148
 buttons (UIs), 226-227
 cameras, 91-92
 character controllers, 77
 colliders, 164
 Collision module, 266
 Color by Speed module, 265
 default module, 261-262
 Emission module, 262
 fog, 73
 image component (UI), 224
 lights, 81
 Limit Velocity over Lifetime module, 263
 physics materials, 166
 Place Tree tool, 65
 Plane mode, 266
 Renderer module, 270
 rigidbodies, 162
 shader, 44
 text objects (UIs), 226

Texture module, 269
 touch, 359
 Velocity over Lifetime module, 263
 World mode, 267
public variables, 129-130

resources, 396-397
return type, methods, 142
Reverb Zone Mix property (audio source), 342
revisions, game, 365-376
 Amazing Racer, 368-371
 disappearing joystick fixing, 370-371
 tilt control, 368-369
 using touch joystick, 369-370

Captain Blaster, 374-375
 Chaos Ball, 372-374
 cross-platform input, 365-368
 projects to mobile conversion, 366-368
 virtual controls, 365-366
 Gauntlet Runner, 375-376

Rich Text property (text objects, UIs), 226

rigging models, animators, 293-296

rigidbodies, 161-163
 properties, 162
 2D, 212

rig preparation, animators, 296-298

rigs, animations, 275-276

rotation, objects, 29-30

Rotation by Speed module (particle system), 265

Rotation over Lifetime module (particle system), 265

rules

Amazing Racer, 104-105
 Captain Blaster, 238
 Chaos Ball, 174
 Gauntlet Runner, 318

R

Range property (point lights), 82
Rate property (Emission module), 262
raycasting, 169-171
reading
 mouse movement, 151
 specific key presses, 150
realistic grass, 68
Receive Shadows property (Renderer module), 270
Record Mode (Animation window), 282
Record Mode, animations, 285-287
Rect Transform, of canvas, 218-219, 220
Rect Transform Tool, 208
Renderer module (particle system), 269-270
Rendering Path property (cameras), 92
Render mode (Scene view), 13
Render Mode, canvas (UIs), 230-232
 screen-space camera, 231
 screen-space overlay, 230-231
 world space, 231-232
Render Mode property (point lights), 83
Render Mode property (Renderer module), 270
requirements
 Amazing Racer, 105-106
 Captain Blaster, 238
 Chaos Ball, 174
 Gauntlet Runner, 318

S

Samples (Animation window), 282
saving data, 382-384
 to PlayerPrefs file, 382-383
Scale Plane property (Plane mode), 266

scaling objects, 30-31
scenes, 11, 379, 395
 changing via buttons, 380-381
 character controllers, adding to, 76-78
 directional lights, adding to, 85-86
 establishing order, 380-381
 fog, adding to, 73
 Gauntlet Runner, 318-319
 gizmo, 14
 invisible items, 97
 lens flares, adding to, 74
 lighting, 13
 managing, 379-381
 point lights, adding to, 83
 prefabs instances, adding to, 195
Scene view, 17-19
 skyboxes, adding to, 72-73
 spotlights, adding to, 85
 switching, 381
Scene view, 13-14, 17-19
 audio testing, 343, 344
 Flythrough mode, 18-19
 2D games, 202-204
scope, variables, 129
screen-space camera, 231
screen-space overlay, 230-231
scripting, 119-137, 141-157
 accessing local components, 151-153
 animators, 314-315
 audio, 346-349
 conditionals, 133-135
 finding objects, 153-156
 iteration, 136-137
 language, 120
 methods, 141-146
 modifying object components, 157
 operators, 130-133
 arithmetic, 130-131
 assignment, 131
 equality, 131-132
 logical, 132-133
 player input, 146-151
scripts, 109, 120
 adding, 111-113
 attaching, 123-124
 basic, 125
 Captain Blaster
 bullets, 254-255
 DestroyOnTrigger, 251
 meteors, 249-250
 meteor spawn, 250-251
 ShipControl, 252-253
 triggers, 251
 class declaration section, 125-126
 classes contents, 126-127
 connecting, 113-114
 creating, 120-123
 Game Control Script, 186
 Gauntlet Runner
 controls, 330-332
 move script, 333
 players, 332-333
 spawn script, 333-334
 trigger zone, 329
 GoalScript.cs, 184-185
 using section, 125
 variables, 128-130
 VelocityScript.cs, 182
Scrubber (*Animation window*), 282
sculpting
 heightmaps, 51-54
 terrain, tools, 54-56
 worlds, 106-107
Send Collision Messages property (*Collision module*), 266
Sensitivity property (*axis*), 148
Separate Axis property (*Limit Velocity over Lifetime module*), 263
Set Native Size property (*images, UIs*), 224
settings, players, 384-387
shaders, 41, 42
 models, applying to, 45-46
 properties, 44
Shadow Type property (*point lights*), 82
shape module (*particle system*), 263
Shininess property (*shader*), 44
ShipControl script, Captain Blaster, 252-253
Simulation Space property (*particle system*), 262
Size by Speed module (*particle system*), 265
Size over Lifetime module (*particle system*), 265
Size property (*colliders*), 164
skyboxes, 71-73
 cameras, adding to, 71
 scenes, adding to, 72-73
Smoothness property (*shader*), 44
Snap property (*axis*), 148
Sorting Fudge property (*Renderer module*), 270
sorting layers, 204, 209-211
 creating, 209-211
Sort Order property (*Renderer module*), 270
Source Image property (*images, UIs*), 224
sources, audio, 341-346
Space property (*Velocity over Lifetime module*), 263
Spatial Blend property (*audio source*), 342
spawn script, Gauntlet Runner, 333-334
specific key presses, reading, 150
Speed property (*Limit Velocity over Lifetime module*), 263
Speed Range property (*Color by Speed module*), 265
split screens, 93-95
spotlights, 84-85
 cookies, 89-90
Spread property (*3D audio*), 346
Sprite Editor, 206-208
sprite mode, 206-208
sprites, 2D games
 adding, 205-209

importing, 206
 mode, 206-208
 sizes, 208-209
 missing, 212
 textures, 206

Start Color property (particle system), 261

Start Delay property (particle system), 261

starting audio, 347-348

Start Lifetime property (particle system), 261

Start Rotation property (particle system), 262

Start Speed property (particle system), 261

statement
 if, 133-134
 if/else, 134
 if/else if, 134-135

states, animators, 310-312

Static Friction property (physics material), 166

Static Friction 2 property (physics material), 166

Stats button (Game view), 16

Step button (Game view), 15

Stereo Pan property (audio source), 342

stopping audio, 347-348

string variable, 128

structure, prefabs, 190-192

Sub Emitter module (particle system), 269

supported operating systems, 3

switching scenes, 381

syntax, 128

T

tabs, adding, 7

Tag Manager, adding new layers to, 96-97

tapCount property (touch), 360

target objects, transforming, 157

Target Texture property (cameras), 92

terrain, 49, 61. *See also environments*

assets, 64
 base, settings, 69
 flattening, 55
 generation, 49-56
 heightmaps, sculpting, 51-54
 importing assets, 57-58
 sculpting, tools, 54-56
 settings, 68-70
 size, 51
 textures, 57-61
 creating, 61
 painting, 60

Terrain Settings tool, 68-69

testing
 Amazing Racer, playtesting, 114-115
 audio, 343, 344

testing (continued)
 2D, 346
 Scene view, 343, 344
 mobile devices, 356

Text objects, UIs, 225-226

Text property (text objects, UIs), 226

textures, 41-42
 baking objects, 83
 cookies, 89
 grass, 67
 models, applying to, 45-46
 sprite, 206
 terrain, 57-61
 creating, 61
 painting, 60

Texture Sheet module (particle system), 269

texturing Chaos Ball arena, 177-178

3D artists, animations, 277

3D audio, 341, 345-346

3D objects, 22
 built-in, 36-37

3D scene view, 13

3D Sound Settings property (audio source), 342

Tiles property (Texture module), 269

Tiling property (shader), 44

Timeline (Animation window), 282

timing, animations, 285

toolbars, editor, 16-17

tools
 animations, 281-288
 2D games (See 2D games)
 Hand, 17-18
 Place Tree, 65
 Rect Transform, 208
 sculpting, 54-56
 Terrain Settings, 68-69
 transform, 16

touch
 multi-touch input, mobile devices, 359-361
 tracking, 360-361

transformations
 hazards, 31-32
 nested objects, 32-33
 objects, 26-33, 153

transform component, 26
 accessing, 152-153

Transform gizmo toggles, 16

transform tools, 16
 Hand tool, 17-18
 Rect, 208

Transition property (buttons, UIs), 227

transitions, animators, 312-314

translation, objects, 27-29

trees
 painting, 63-66
 settings, 70

triangles, 36

triggers
 Captain Blaster, 246
 script, 251
 colliders, 167-169

trigger zone, Gauntlet Runner,
322
script, 329

2D animations, 277-279

2D audio, 341, 346

2D games, 201-214

adding sprites, 205-209

importing, 206

sizes, 208-209

sprite mode, 206-208

basics, 201-204

challenges, 202

colliders, 212-214

design principles, 201

draw order, 209-212

in layer, 212

sorting layers, 209-211

orthographic cameras, 204-205

rigidbody, 212

scene view, 202-204

setting, 201-202

2D objects, 22

2D scene view, 13

Type property (axis), 148

Type property (point lights), 82

U

UIs. See user interfaces (UIs)

Unity

2D colliders, 213

downloading, 1-3

editor, 4-17

installing, 1-4

interface, 6-8

license, activating, 1-3

minimum requirements, 3

modifying public variables in,
129-130

Unity particle systems, 257-258

Unity Remote app, 355, 371

unwrap texture, 42

updating prefabs, 196-197

**Use Gravity property (rigidbody),
162**

user input. See player input

user interfaces (UIs), 217-232

canvas, 218-223

adding, 218

anchor button, 222, 223

anchors, 220-223

components, 223

EventSystem game object,
218

Rect Transform, 218-219, 220

Render Mode, 230-232

Captain Blaster, 246-247

controls, 223-230

design, 217

elements, 223-230

buttons, 226-227

On Click () property, 227-229

images, 224-225

sorting, 230

text, 225-226

materials, 225

principles, 217-218

using section, scripts, 125

V

value curve, 261

variables, 128-130

creating, 128

private, 129-130

public, 129-130

scope, 129

Velocity over Lifetime module

(particle system), 263

VelocityScript.cs listing, 182

View Port Rect property

(cameras), 91

views

duplicating, 7

Game, 14-16

Hierarchy, 10-11, 190, 191

Inspector, 11-12

curve editor, 271

rig preparation, 296

script preview, 120, 121

Project, 8-10

prefabs, 189-190, 191

Scene, 13-14, 17-19

2D games, 202-204

Flythrough mode, 18-19

virtual control system, 365-366

**Visualization property (Plane
mode), 266**

**Volume property (audio source),
342**

**Volume Rolloff property
(3D audio), 346**

**Voxel Size property (World mode),
267**

W

walk animation, 300-302

walk turn animation, 302-305

water, 74-75

while loop, 136

wind, settings, 70

window (Animation), 281-283

**world coordinates, versus local
coordinates, 24-25**

worlds

Amazing Racer, 106-108

Captain Blaster, 238-239

Chaos Ball, arena, 175-176

Gauntlet Runner, 318-320

sculpting, 106-107

world space, 231-232

writing methods, 144-145

X

**XYZ property (Velocity over
Lifetime module), 263**

This page intentionally left blank

Dive Deeper into Unity with Video Instruction from Mike Geig

Title: Game Development Essentials II with Unity LiveLessons

ISBN: 978013389201

In this video training, Mike Geig covers key 2D and 3D game development concepts beyond the basics and scripting (programming) concepts for featured game engines. He builds on the success of his first Unity LiveLessons video, *Game Development Essentials with Unity*, to bring more intermediate level topics to the forefront so that developers can get the most out of this powerful game engine.



For more information and to watch free video lessons visit
informit.com/gamdev

Continue Learning With Ben Tristem

Join thousands of other students in Ben's 50 hour video course

This video course is a fantastic compliment to what you have learned in the book. As well as recapping the basics of scripting and using Unity, you will...

- Build seven more brand-new 2D and 3D games.
- Take the course anytime, anywhere, at your own pace.
- Interact direct with Ben in the discussion forums.
- Get help from a thriving community of students.
- Host your games online and share with your friends.



Simply visit <https://www.udemy.com/unitycourse> to find out more.



JOIN THE **INFORMIT** AFFILIATE TEAM!

You love our titles and you love to share them with your colleagues and friends...why not earn some \$\$ doing it!

If you have a website, blog, or even a Facebook page, you can start earning money by putting InformIT links on your page.

Whenever a visitor clicks on these links and makes a purchase on informit.com, you earn commissions* on all sales!

Every sale you bring to our site will earn you a commission. All you have to do is post the links to the titles you want, as many as you want, and we'll take care of the rest.

APPLY AND GET STARTED!

It's quick and easy to apply.

To learn more go to:

<http://www.informit.com/affiliates/>

*Valid for all books, eBooks and video sales at www.informit.com

