Cool projects that will push your skills to the limit

Unity 4 Game Development

Develop spectacular gaming content by exploring and utilizing Unity 4

HOTSHOT

Jate Wittayabundit

[PACKT] PUBLISHING

www.itbookshub.com

www.itbookshub.com

www.itbookshub.com

Table of Contents

- Unity 4 Game Development HOTSHOT
- Credits
- About the Author
- About the Reviewers
- www.PacktPub.com
  - Support files, eBooks, discount offers and more
    - Why Subscribe?
    - Free Access for Packt account holders
    - Instant Updates on New Packt Books
- Preface
  - What this book covers
  - What you need for this book
  - Who this book is for
  - Sections
  - Mission briefing
    - Why is it awesome?
    - Your Hotshot objectives
    - Mission checklist
  - Task 1
    - Prepare for lift off
    - Engage thrusters
    - Objective complete – mini debriefing
    - Classified intel
  - Mission accomplished
  - A Hotshot challenge / Hotshot challenges
  - Conventions
  - Reader feedback
  - Customer support
    - Downloading the example code

www.itbookshub.com

Errata
Piracy
Questions
1. Develop a Sprite and Platform Game
Mission briefing
Why is it awesome?
Your Hotshot objectives
Mission checklist
Setting up a 2D level and collider
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
The Sprite Renderer and Sorting Layer
Creating a 2D character and animation
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Controlling the character with the PlayerController_2D class
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Physics2D.Raycast
Creating a key, door, and replay button
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel

www.itbookshub.com

The Restart button
Mission accomplished
Hotshot challenges
2. Create a Menu for an RPG – Add Powerups, Weapons, and Armors
Mission briefing
Why is it awesome?
Your Hotshot objectives
Mission checklist
Customizing skin with GUISkin
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating a menu object
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating the STATUS tab
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating the INVENTORY tab
Engage thrusters
Objective complete – mini debriefing
Classified intel
GUI.tooltip
Creating the EQUIPMENT tab
Engage thrusters
Objective complete – mini debriefing
Classified intel

www.itbookshub.com

Hotshot challenges
3. Shade Your Hero/Heroine
Mission briefing
Why is it awesome?
Your Hotshot objectives
Mission checklist
Shader programming – Diffuse and Bump (normal) maps
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Shader programming – Ambient and Specular light
Engage thrusters
Objective complete – mini debriefing
Classified intel
Shader programming – Half Lambert, Rim Light, and Toon Ramp
Engage thrusters
Objective complete – mini debriefing
Mission accomplished
Hotshot challenges
4. Add Character Control and Animation to Our Hero/Heroine
Mission briefing
Why is it awesome?
Your Hotshot objectives
Mission checklist
Setting up character animation and level
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel

www.itbookshub.com

The Animations inspector
Creating an animator controller
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating a character control script
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating a third-person camera to follow our character
Engage thrusters
Objective complete – mini debriefing
Classified intel
Mission accomplished
Hotshot challenges
5. Build a Rocket Launcher!
Mission briefing
Why is it awesome?
Your Hotshot objectives
Mission checklist
Setting up a character animation and animator controller
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Blend Tree
Adding new features to the CharacterControl and CameraControl scripts
Engage thrusters
Objective complete – mini debriefing
Creating a MouseLook script and laser target scope

www.itbookshub.com

Objective complete – mini debriefing
Classified intel
Line Renderer
Creating a rocket prefab and particle effects
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating a rocket launcher and RocketUI
Engage thrusters
Objective complete – mini debriefing
Classified intel
BroadcastMessage
Mission accomplished
Hotshot challenges
6. Make AI Appear Smart
Mission briefing
Why is it awesome?
Your Hotshot objectives
Mission checklist
Creating the Waypoint and WaypointsContainer scripts
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Gizmos
Creating a custom editor for the WaypointsContainer script
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel

www.itbookshub.com

Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating a hit-point UI
Engage thrusters
Objective complete – mini debriefing
Classified intel
Mission accomplished
Hotshot challenges
7. Forge a Destructible and Interactive Virtual World
Mission briefing
Why is it awesome?
Your Hotshot objectives
Mission checklist
Creating a ragdoll object
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating a destructible wall
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating a rockslide and trigger area
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Creating the RocksTrigger and Rocks scripts

www.itbookshub.com

Engage thrusters
Objective complete – mini debriefing
Classified intel
Mission accomplished
Hotshot challenges
8. Let the World See the Carnage – Saving and Loading High Scores
Mission briefing
Why is it awesome?
Your Hotshot objectives
Mission checklist
Creating the UserData and Hiscore scripts
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Saving and loading the local high score
Engage thrusters
Objective complete – mini debriefing
Classified intel
Creating an XMLParser script
Engage thrusters
Objective complete – mini debriefing
Classified intel
Saving and loading server high score
Prepare for lift off
Engage thrusters
Objective complete – mini debriefing
Classified intel
Mission accomplished
Hotshot challenges
A. Important Functions

Awake()
Start()
Update()
FixedUpdate()
LateUpdate()
OnEnable()
OnDisable()
OnGUI()
OnDrawGizmos()
References
B. Coroutines and Yield
Coroutines
YieldInstruction
WaitForSeconds
WaitForFixedUpdate
StartCoroutine
Using StartCoroutine with method name (string)
StopCoroutine
StopAllCoroutines
References
C. Major Differences Between C# and Unity JavaScript
Unity script directives
Type names
Variable declaration
Variables with dynamically typed resolution
Multidimensional array declaration
Character literals not supported
Class declarations
Limited interface support
Generics
The foreach keyword

The new keyword
The yield instruction and coroutine
Casting
Properties with getters/setters
Changing struct properties by value versus by reference
Function/method definitions
References
D. Shaders and Cg/HLSL Programming
ShaderLab properties
Surface shaders
#pragma surface
Surface shaders input structure
The SurfaceOutput structure
Cg/HLSL Programming
Accessing shader properties in Cg/HLSL
Data types
Common methods to create shaders
References
Index

Unity 4 Game Development HOTSHOT

Unity 4 Game Development HOTSHOT

Copyright © 2014 Packt Publishing

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

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

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

First published: August 2011

Second edition: July 2014

Production reference: 1180714

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84969-558-9

www.packtpub.com

Cover image by Jate Wittayabundit (<jatewit@jatewit.com>)

Credits

Author

Jate Wittayabundit

Reviewers

John P. Doran

Thomas Finnegan

Chirag Raman

Commissioning Editor

Edward Gordon

Acquisition Editor

Joanne Fitzpatrick

Content Development Editor

Anila Vincent

Technical Editors

Mrunal Chavan

Gaurav Thingalaya

Copy Editors

Janbal Dharmaraj

Mradula Hegde

Deepa Nambiar

Karuna Narayanan

Project Coordinator

Mary Alex

Proofreaders

Maria Gould

Ameesha Green

Paul Hindle

Indexers

Rekha Nair

Priya Subramani

Tejal Soni

Abhinash Sahu

Production Coordinator

Komal Ramchandani

Cover Work

Komal Ramchandani

www.itbookshub.com

About the Author

Jate Wittayabundit was born in Bangkok, Thailand, in 1980. One thing that he always remembers about his childhood is that playing games was something very very special to him. He was allowed to play only during school breaks. The game that he played would be kept in a locked chest by his mom; it was *Super Mario Bros*, the first game he ever played. Something special in his childhood became something he dreamt to be as a boy in a country where nobody was familiar with computers at this time. So, he questioned how humans could create this thing.

”*Nothing is impossible*“, he believes!

However, there was no game development course at all in any Thai college or university at the time he chose to pursue his career in that field. Going abroad to study was a huge challenge, which he wasn’t ready for. He curtailed his dream, pursuing a Bachelor’s degree in Interior Architecture at King Mongkut’s University of Technology Thonburi to be able to improve his skills in arts, 3D visualization, and mathematics, which he thought were very important to support what he wanted to be. While he was studying Interior Architecture, he had a chance to use 3D Studio Max, FormZ, AutoCAD, Maya, Photoshop, After Effects, Premiere, and lots of 3D tools to create the architecture projects. Ever since, he has loved it and continues working with these tools. In 2003, after graduation, he started working as an interior architect and 3D visualizer for several companies in Thailand.

He also applied for a part-time 3D game course in Thailand and made a couple of friends who, like himself, had a similar passion to create games. They formed a team, making a side-scrolling game using a panda as the main character named *PAN PAN*. The game was built using Game Maker. As a team member, Jate was responsible for creating the graphics and cover art for the games, because he didn’t have any experience in programming at all. At that time, he sensed an upward trend in the game industry. In 2005, he decided to move to Ottawa, Canada, to study a brand new Game Development program. It was really tough for him at first, and he really wanted to quit because of the language barrier and the complexity of programming languages, as he had no basic knowledge at all. However, he had many good professors and friends to help him get through the course. He started to love programming, which he thought he would never understand, and in 2008, he graduated with honors in the Game Development program from Algonquin College.

After graduating from the Game Development program, he started working at Launchfire Interactive Inc. (www.launchfire.com) as a Flash Actionscript Programmer. At Launchfire, he developed many games and interactive content (for clients such as Dell and Alaska Airline) as well as learned how to created Flash 3D in Papervision3D and Away3D.

In 2009, he decided to move to Toronto—a big city—to get more experience of working in the game industry. He started working in a new position as a game developer and 3D artist at Splashworks.com Inc. (www.splashworks.com). At Splashworks, he had the chance to work with many different games and clients (such as Shockwave and Swiss Chalet). It also gave him a great chance to learn about Unity3D. He started using it from

September 2009 and just loved its fast and friendly UI. He really liked how easily Unity imported 3D objects and animations. Currently, he is working as a senior game developer creating many mobile games for many clients, including Sunkist, Nickelodeon, and American Girl.

He believes that being an architect is his strength and he is on the right track; it supports his concepts and ideas of how a real-world perception could apply in a virtual world. He loves to work on 3D software such as Zbrush or 3D Studio Max in his spare time and loves painting and drawing. He wants to try marrying his architectural and 3D skills with his game development skills to create the next innovation in gaming.

You can check out his work on his website, www.jatewit.com. He has also created a Zbrush character, http://www.zbrushcentral.com/showthread.php?t=90665&highlight=tyris.

First, I would like to thank the staff of Packt Publishing who, randomly or by getting interested in my work, gave me this great opportunity to write this book. Thanks to all the reviewers of this book for their feedback to make it possible and better for the readers. Thank you Susan Olszynko, International Marketing Manager at Algonquin College, who went far to get me from Thailand and gave me an assurance that I will never regret taking the Game Development course at Algonquin College. Also, I'm very grateful to my professor, Tony Davidson, for believing in me and directing me to my first career in Canada. Thanks to the professors and all my friends. I would like to thank my mom, dad, and family for supporting me anytime, no matter what happens. I would also like to thank my wife who always stayed by my side, supporting, helping, and encouraging me to complete this book. Also, I would like to thank my newborn baby for keeping me awake at night. Finally, thank you all for buying this book. I hope you will enjoy this book, and it becomes a part of your inspiration.

About the Reviewers

John P. Doran is a technical game designer who has been creating games for over 10 years. He has worked on an assortment of games in teams from just himself to over 70 in student, mod, and professional projects.

He previously worked at LucasArts on *Star Wars 1313* as the only junior designer in the team. He later graduated from DigiPen Institute of Technology in Redmond, WA, with a Bachelor of Science in Game Design.

John is currently working at DigiPen's Singapore campus as the lead instructor of the DigiPen-Ubisoft Campus Game Programming program, instructing graduate-level students in an intensive, advanced-level game programming curriculum. In addition to that, he also tutors and assists students on various subjects while giving lectures on C++, Unreal, Flash, Unity, and more.

He is the author of *Getting Started with UDK* and *Mastering UDK Game Development*, and he has also co-authored *UDK iOS Game Development Beginner's Guide*, all available from Packt Publishing.

Thomas Finnegan worked as a freelance game developer for a few years, having graduated from Brown College in 2010. Currently, he is teaching game development at the Minneapolis Media Institute. He has worked on everything from mobile platforms to web development and even on experimental devices. His past clients include Carmichael Lynch, Coleco, and Subaru. His most recent project is Battle Box 3D, a virtual table top. His first book, *Unity Android Game Development by Example Beginner's Guide*, *Packt Publishing*, saw its release in December 2013.

Chirag Raman began his career as an iOS application developer and a visiting game programming faculty at the University of Mumbai. During this time, he also worked as a project engineer at the Indian Institute of Technology, Bombay, on projects employing Blender 3D for interactive content creation. He subsequently pursued a Master's degree from the Entertainment Technology Center at Carnegie Mellon University, where he was fortunate to gain hands-on experience with the User Experience and Creative Services team at Microsoft and the Next Generation Experience - New Media Group at Walt Disney Parks and Resorts.

After graduating from Carnegie Mellon University, Chirag worked for a while as a Research Associate with the Computer Vision group at Disney Research, Pittsburgh. His interests lie in combining the Arts and Computer Science to craft synergistic experiences for people. He currently works as the lead iOS UX developer for an upcoming entertainment industry startup in New York, pursuing his passion to employ science to innovate user experience.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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

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

http://PacktLib.PacktPub.com

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

Why Subscribe?

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

Free Access for Packt account holders

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

www.itbookshub.com

Instant Updates on New Packt Books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter, or the *Packt Enterprise* Facebook page.

Preface

Only Unity fits the bill of being a game engine that allows you to create an entire 2D and 3D game for free and with phenomenal community support. *Unity 4 Game Development Hotshot* will equip you with the skills to create professional-looking games at no cost.

This book will teach you how to exploit the complete array of Unity 3D and 2D technology in order to create an advanced gaming experience for the user, with eight exciting and challenging projects that provide step-by-step explanations, diagrams, and screenshots to help you achieve that goal.

Every project is designed to push your Unity skills to the very limits and beyond. You will start with the creation of a 2D platform game with the character sprite sheet. Then, you will be using Unity's immediate mode GUI system (IMGUI) to create a cool RPG menu that allows you to customize the character skills and equipment. You will create a shader and animation controller to make your character look more like a character from a next-gen game than a simple sprite.

Now for some damage—rocket launchers! Typically, this is the most powerful weapon in any first-person shooter game. You will create a rocket launcher that has fire and smoke particles and most importantly, causes splash damage for the all-important area effect. We will create AI-controlled enemies to eliminate the rocket using the powerful editor script. Then, we will create an interactive world that is destructible. The final touch will be for you to upload your scores both online and locally so that everyone can see the carnage.

What this book covers

Project 1, *Develop a Sprite and Platform Game*, creates a 2D sprite platformer project using the new 2D sprite feature in Unity. This includes the new sprite object and the new 2D physics engine that integrated Box2D. There will be an explanation about how to set up and prepare a sprite sheet and how to set up the 2D camera, 2D physics, and 2D character animation.

Project 2, *Create a Menu for an RPG – Add Powerups, Weapons, and Armors*, will teach you how to create a cool and complex UI that is mostly used in an RPG using the new Unity GUI (IMGUI). This includes creating a hit-point bar, buttons for the user to go to different menus, a scrolled area and scrollbar for the items, and dragging-and-dropping a character's skills and equipments.

Project 3, *Shade Your Hero/Heroine*, will write a custom shader using the surface shader. We will create an ambient and diffuse shader, bump shader, specular shader, half lambert shader, toon ramp shader, and rim light shader.

Project 4, *Add Character Control and Animation to Our Hero/Heroine*, will teach the basics of character animation using the new Mecanim animation system in Unity. We will begin by setting up the walk, run, idle, jump, and fall animation clips and then move on to setting up the animator controller to enable transition between the animation clips, creating a custom third-person camera, and adding the character controller to control our character.

Project 5, *Build a Rocket Launcher!*, will continue using our third-person camera from the previous project. This time, we will add the ability for our project to shoot a rocket via a rocket launcher. We will also add the laser target effect to make user aiming easier. Then, we will add particle effects for the rocket and bomb using the Shuriken Particle effect.

Project 6, *Make AI Appear Smart*, will show you how to create AI that is smart enough using the waypoint concept. This includes having AI follow a path and adding/removing the waypoint from the editor using the editor script.

Project 7, *Forge a Destructible and Interactive Virtual World*, starts with setting up the ragdoll object. Then, we will be creating a destroyable wall and destructible rock from multiple cubes, which will be triggered using the event and delegate concept.

Project 8, *Let the World See the Carnage – Saving and Loading High Scores*, will show you how to save and load your high scores locally and through the server. This includes PlayerPref, Serialized/Deserialized data, and WWWForm.

Appendix A, *Important Functions*, includes details of the important functions that are mostly used, such as Awake() and Start(), sourced from Unity's scripting document.

Appendix B, *Coroutines and Yield*, includes a brief explanation of coroutines/yield and how to use them; this is sourced from Unity's scripting document and the Unity forum.

Appendix C, *Major Differences Between C# and Unity JavaScript*, shows the differences between C# and Unity JavaScript using examples that are sourced from Unity's answer

forum and documentation.

Appendix D, *Shaders and Cg/HLSL Programming*, explains the structure of the shader programming and the basic function using Cg/HLSL and so on, sourced from Unity's documentation and the NVIDIA website.

What you need for this book

The following are the requirements for this book:

- You will need Unity 4.6 or higher, which you can download from http://www.unity3d.com/download/
- 3D Studio Max (optional), which can be downloaded from http://usa.autodesk.com/3ds-max/trial
- TexturePacker (optional), which can be downloaded from http://www.codeandweb.com/texturepacker

Who this book is for

This book is for experienced users who already have some basic knowledge of how to use the Unity game engine and intermediate users who want to explore Unity above and beyond the basic techniques.

www.itbookshub.com

Sections

A Hotshot book has several sections that will be elaborated as the book progresses.

Mission briefing

This section explains what you will build, with a screenshot of the completed project.

Why is it awesome?

This section explains why the project is cool, unique, exciting, and interesting. It describes the advantages the project will give you.

Your Hotshot objectives

This section explains the major tasks required to complete your project, which are as follows:

- Task 1
- Task 2
- Task 3
- Task 4, and so on

Mission checklist

This section mentions prerequisites for the project (if any), such as resources or libraries that need to be downloaded.

Each **task** is explained using individual sections.

Task 1

Prepare for lift off

This section explains any preliminary work that you need to do before beginning work on the task.

Engage thrusters

This section lists the steps required in order to complete the task.

www.itbookshub.com

Objective complete – mini debriefing

This section explains how the steps performed in the previous section (*Engage thrusters*) allow us to complete the task.

Classified intel

This section provides extra information that is relevant to the task.

After all the tasks are completed, the following sections will be present

Mission accomplished

This section explains the task you accomplished in the project.

A Hotshot challenge / Hotshot challenges

This section explains additional things that can be done or tasks that can be performed using the concepts explained in this project.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: “Next, we trigger the jump animation state by using `_animator.SetTrigger("Jump");`.”

A block of code is set as follows:

```
void Init ( bool isEquipment ) : void {
  _isEquipment = isEquipment;
  …
}
```

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

```
// C# user:

Vector2 scrollPostion = Vector2.zero;
void OnGUI() {
scrollPostion = GUI.BeginScrollView(new Rect(0,0,100,40), scrollPostion,
new Rect(0,0,80,120));
GUI.Button(new Rect(0,0,80,40),"Button 1");
GUI.Button(new Rect(0,40,80,40),"Button 2");
GUI.Button(new Rect(0,80,80,40),"Button 3");
GUI.EndScrollView();
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: “Finally, we have attached the sound effect and a **Restart** button to our game.”

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

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

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

www.itbookshub.com

Customer support

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

Downloading the example code

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

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/5589OT_GraphicsBundle.pdf.

Errata

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

Piracy

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

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

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

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Develop a Sprite and Platform Game

Even in today's world, people remember Mario, Sonic, and Mega Man. Of course, Mario was first introduced in the eighties, followed by Mega Man and Sonic, but even now the new generation loves these games. Yes, we are talking about the old style 2D platform games, which are still quite popular today, especially among indie game developers.

In this book, we will start the first project with a 2D platform game because there are some basic tricks for a 2D platform game that will help you—those who haven't got into the 3D world yet—to understand more before jumping into the 3D world for the later projects.

There is a huge improvement on Unity after the release of **4.3**. The best feature for most game developers is the 2D feature in Unity, which will help us to speed up the 2D game development time. In this project, we will be using a similar concept to the Unity's 2D Platformer project in the assets store. This includes the new **sprite** object and the new 2D physics engine that integrated Box2D (the 2D physics game engine that is used in many games such as Angry Birds).

Note

We can also download the Unity 2D project from the Unity Asset Store: https://www.assetstore.unity3d.com/#/content/11228

For more information about Box2D, visit the following URLs: http://box2d.org/about/ and http://en.wikipedia.org/wiki/Box2D

Mission Briefing

We'll be creating a 2D platform or side-scrolling game, which is similar to Mario or other games that we mentioned previously; it will have a simple character that can be made to move, jump, and collect a key item for passing the level, and a **Restart** button to play the game again. The following screenshot demonstrates this:

We will use a 2D character sprite sheet (which is the set of many small images in a large image, as shown in the previous figure), and the new 2D feature in Unity to create our 2D character. The new 2D feature in Unity helps us to manage the sprite sheet and set up the animation for each character's movement without writing any script.

Note

There are many tools that can help you to create a sprite sheet, such as TexturePacker. It basically creates an image file with the data that contains all the information such as name of the sprite, position, and sprite area. In this process, we will have two files—the image file (`.jpeg` or `.png`) and datafile (`.txt` or `.xml`)— which might be a bit confusing for beginners. So, Unity 4.3 solved this problem by including the **Asset Import Setting**, which we can just use to import the asset and set the **Texture Type** to **Sprite**, and then set the mode to either **Multiple** (the image that contains many sprites packed in one file) or **Single** sprite.

www.itbookshub.com

For more information about TexturePacker and a sprite sheet, visit the following URL: http://www.codeandweb.com/texturepacker and http://www.codeandweb.com/what-is-a-sprite-sheet.

The purpose of this project is to familiarize you with the new 2D feature and language syntax in Unity, which is very important when creating a playable game.

We will begin by setting up the sprite for our character, game level, and item in the game. Next, we will create our character sprite object, create the animator controller, and set up each animation clip for idle, walk, jump, and fall. We will also create the script, which will control the character to show the correct animation (idle, walking, and jumping). This script will allow us to control the character action by pressing the arrow key to move and space bar to jump. Also, we will learn how to set up the custom input manager.

Next, we will create the level and platform using the sprite object with `BoxCollider2D` and `PolygonCollider2D` attached, which will collide with the character that has `Rigidbody2D` attached to make the movement realistic.

To end the game, we will create a trigger event by having a door and key game object. The player needs to collect the key to be able to open the door and end the game. We will also add sound to make our game lively, but we are not finishing it yet. The game needs to be replayable. Lastly, we will add a **Replay** or **Play again** button to replay our game by using `Destroy` and `Instantiate` to reset our character's position and key item.

Why is it awesome?

When we are done with this project, we will get a good understanding of how to create a sprite and 2D platform game by using the new 2D feature in the Unity engine. Also, we will be able to create our own 2D platform style game such as Sonic, Mario, and Mega Man, and reuse some of our techniques, scripts, and concepts to create a 3D game at a later stage.

Your Hotshot objectives

This project will be split into five tasks. As we are not creating any enemies in our game, we don’t have to deal with any complex scripting. It will be a simple step-by-step process from beginning to end. The following is the outline of the tasks:

- Setting up a 2D level and collider
- Creating a 2D character and animation
- Controlling the character with the `PlayerController_2D` classes
- Creating a key, door, and replay button

Mission checklist

Before we start, we will need to get the latest Unity version (http://unity3d.com/unity/download/) that includes **MonoDevelop**, which we will use for our scripting editor. We will also need a few graphics for our character, key, and door as well as a collection of sound FX. These can be downloaded as ZIP files from Packt's website: http://www.packtpub.com/support?nid=8267.

Browse the preceding URL and download the `Chapter1.zip` package and unzip it. Inside the `Chapter1` folder, there are two unity packages: `Chapter1Package.unitypackage` (we will use this package for this project) and `Chapter1Package_Completed.unitypackage` (this is the completed project package that includes both C# and Unity JavaScript).

Setting up a 2D level and collider

In this section, we will begin with setting up all sprite assets including the character sprite sheet, background, platform, door, and key sprite. Then, we will create a custom tag and layer to use in our game. Next, we will set up our level by using the sprite game object and attaching the 2D collider including `BoxCollider2D` and `PolygonCollider2D`.

Prepare for lift off

Before we start creating this project, we will create the project in Unity by performing the following steps:

1. Create a new project by going to **File | New Project** to bring up the Project Wizard window. Next, click on the **Create new Project** tab and set the **Project Directory** as you want, as we can see in the following screenshot:

As we can see from the preceding screenshot, we don't import any Unity assets package because we won't be using any of those packages in this project. This helps to reduce the size of the project because we don't have any unnecessary assets. We also set up the default of Unity mode to use **2D** instead of **3D**, which basically will set the camera to use the 2D view instead of the 3D view. It also sets the default asset importing to use **Sprite** instead of the **Texture** type.

Tip

By setting the default mode to **2D** or **3D** in Unity editor, we can go to **Edit | Project Settings | Editor** and change the **Default Behavior Mode** to **2D** or **3D**, as we can see in the following screenshot:

We can also switch between the 2D and 3D views by clicking on the **2D** tab in the **Scene** view, as shown the following screenshot:

2. Then, go to **Assets | Import Package | Custom Package…** and select the `Chapter1Package.unitypackage` file from the folder that we just unzipped and click on **Import** to import all assets, as we can see in the following screenshot:

3. To make it easy for us, we will go to **SimplePlatform | Scenes** in the **Project** view. We will see there is one scene in this folder named **SimplePlatform**; double-click to open it as shown in the following screenshot:

We will see that nothing in the **Hierarchy** has changed. No worries, we will use this scene as our base scene to save all the progress that we will make in this project.

4. At the last step, we will go to **SimplePlatform** | **Resources** | **Sprites** | **Level** and click on the **Background** file inside this folder in the **Project** view. Then, we will go to the **Inspector** view, set it as the following screenshot, and click on **Apply**:

5. As we can see, this time we set the **Sprite Mode** to **Single**. This means that this file only contains one sprite. Now we have finished the setup part. Next, we will start building the level.

Engage thrusters

To start building the level, first we need to add new tags and layers. So, let's get on with it:

1. Let's go to **Edit** | **Project Settings** | **Tags and Layers** to bring up the **Tags and Layers** inspector view and set it as follows:

Tags	
Size	`5`
Element 0	`Ground`
Element 1	`RestartButton`
Element 2	`Key`
Element 3	`Door`
Layers	
User Layer 8	`Player`
User Layer 9	`Ground`

   Tags make it easy to find the object, we need `GameObject.FindWithTag("TagName")` or `gameObject.tag`. More details on tags can be found at http://docs.unity3d.com/Documentation/Components/Tags.html.

   ### Note

   Layers are used to determine which object will be rendered or used for raycasting. Sometimes, we can use layers to separate the objects to be rendered. For example, we could have two cameras and set the first camera to render only the foreground objects and then another camera to render only the background object (such as, UI, or minimap). More details on layers can be found at http://docs.unity3d.com/Documentation/Components/Layers.html.

2. Next, we create the background by going to **SimplePlatform** | **Resources** | **Sprites** | **Level** in the **Project** view, click on **Background** file, and drag it to the **Hierarchy** view, as shown in the following screenshot:

3. Then, we click on the **Background** object in the **Hierarchy** view, go to its **Inspector** view, and set the parameters as follows:

Layer	**Level**
Transform	
Position	**X**: `0`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `3`, **Y**: `3`, and **Z**: `1`
Sprite Renderer	
Color	**R**: `171`, **G**: `245`, **B**: `255`, and **A**: `255`
Order in Layer	`-1`

4. Next, we will create the **Level** empty game object to contain our level by going to **GameObject** | **Create Empty**; name it **Level**, and reset all the transformation to the default value by going to its **Inspector** view and right-clicking on the gear wheel icon and choosing **Reset**, as shown in the following screenshot:

5. Then, we will add the floor and wall by going to **SimplePlatform** | **Resources** | **Sprites** | **Level** in the **Project** view and clicking on the arrow in front of the **Platform** file. Then, we will click on the **Floor** object and drag it inside the **Level**

game object in the **Hierarchy** view, and do the same thing with the **Wall** object, as shown in the following screenshot:

6. Next, we will click on the **Floor** object in the **Hierarchy** view and add **Box Collider 2D** by going to **Component | Physics2D | Box Collider 2D**. Then, we will go to **Inspector** and set the values of the attributes as follows:

Tag	**Ground**
Layer	**Ground**
Transform	
Position	**X**: `0`, **Y**: `-9.2`, and **Z**: `0`
Scale	**X**: `5`, **Y**: `5`, and **Z**: `1`
Sprite Renderer	
Order in Layer	`2`
Box Collider 2D	
Material	**Ground** (drag the Physics2D Material here)

7. Then, we duplicate the **Floor** object by pressing *Ctrl* + *D* (on Windows) or *command* + *D* (on Mac) and set the second **Floor** object and change the transform position as follows:

Transform	
Position	**X**: `0.25`, **Y**: `11`, and **Z**: `0`

Next, we will set the **Wall** object, click on this object, and add the **Box Collider 2D** by going to **Component | Physics2D | Box Collider 2D**. Then, we will go to

Inspector and set it as follows:

Transform	
Position	**X**: `-14`, **Y**: `-1.46`, and **Z**: `0`
Scale	**X**: `5`, **Y**: `5`, and **Z**: `1`
Sprite Renderer	
Order in Layer	`1`
Box Collider 2D	
Material	**Edge** (drag the Physics 2D Material here)

8. Then, we duplicate the **Wall** object by pressing *Ctrl* + *D* (on Windows) or *command* + *D* (on Mac) three times and set the second, third, and fourth **Wall** object and change the transform position as follows:

Transform (second `Wall` object)	
Position	**X**: `14`, **Y**: `-1.47`, and **Z**: `0`
Transform (third `Wall` object)	
Position	**X**: `-17.8`, Y: `-1.47`, and **Z**: `0`
Transform (forth `Wall` object)	
Position	**X**: `17.8`, **Y**: `-1.47`, and **Z**: `0`

9. Now, we will add `LargePlatform`. Let's go to **SimplePlatform** | **Resources** | **Prefabs** in the **Project** view, and click on the arrow in front of the **LargePlatform** prefab. Then, we will click on the **Platform_5**, **Platform_4**, **Platform_6**, and **Platform_1** objects and set **Tag** and **Layer** to **Ground**. Then, we will drag **LargePlatform** inside the **Level** game object in the **Hierarchy** view, go to its **Inspector**, and set it as follows:

Transform	
Position	**X**: `7.34`, **Y**: `0.7`, and **Z**: `0`

10. Next, we add **Platform_1**. Let's go to **SimplePlatform** | **Resources** | **Prefabs** in the **Project** view, click on the **Platform_1** object, and set **Tag** and **Layer** to **Ground**. We will see the **Change Layer** pop up; click on the **No, this object only** button. Then, we will drag it inside the **Level** game object in the **Hierarchy** view, go to its **Inspector**, and set it as follows:

Transform	
Position	**X**: `9.5`, **Y**: `-4`, and **Z**: `0`

11. Then, we do the same thing for **Platform_3**. Let's go to **SimplePlatform** | **Resources** | **Prefabs** in the **Project** view, click on the **Platform_3** object, and set **Tag** and **Layer** to **Ground**. We will see the **Change Layer** pop up; click on the **No, this object only** button. Then, we will drag it inside the **Level** game object in the **Hierarchy** view. Go to **Inspector**, and set it as follows:

Transform	
Position	**X**: `-6.28`, **Y**: `2.33`, and **Z**: `0`
Scale	**X**: `1.4`, **Y**: `1.4`, and **Z**: `0`

12. Next, we will duplicate **Platform_3** by pressing *Ctrl* + *D* or *command* + *D* to create the second **Platform_3** object and set it as follows:

Transform	
Position	**X**: `7.25`, **Y**: `4.53`, and **Z**: `0`
Scale	**X**: `-1.4`, **Y**: `1.4`, and **Z**: `0`

We will see the level in the **Scene** and **Hierarchy** view similar to the following screenshot:

13. We can see that we already have the **Platform_1** and **Platform_3** prefab. Now, we will create the **Platform_2** prefab from scratch. First, we will go to **SimplePlatform | Resources | Sprites | Level** in the **Project** view and click on the arrow in front of the **Platform** file. Then, we will click on the **Platform_2** object and drag it inside the **Level** game object in the **Hierarchy** view. Click on this object and add **Polygon Collider 2D** by going to **Component | Physics2D | Polygon Collider 2D**. Then, we will go to **Inspector** and set the values of the attributes as follows:

Tag	**Ground**
Layer	**Ground**
Transform	
Position	**X**: `-7`, **Y**: `-1.17`, and **Z**: `0`
Box Collider 2D	
Material	**Ground** (drag the Physics2D Material here)

14. Now, we go to the **Scene** view. We will see that our **Platform_2** sprite has **Polygon Collider 2D** that matches the shape of this sprite, as shown in the following screenshot:

15. From the preceding screenshot, we will see that there are too many vertices for this sprite. So, we need to decrease and adjust the vertices to match the sprite.

Tip

Removing the vertex in **Polygon Collider 2D** means decreasing the time to calculate unnecessary vertices' data, which will also help to speed up the performance in the runtime.

16. Next, we will adjust the vertices in **Polygon Collider 2D**. To make it easy to adjust, we will go to **Platform_2 Inspector** and click on the arrow in front of **Sprite Renderer** to hide the 2D outline gizmos, as shown in the following screenshot:

17. Now, we will remove the vertices in **Collider Info** from `50` to `7`, which will optimize the content and help the overall performance in the scene. We can do this by pressing *Ctrl* (on Windows) or *command* (on Mac) . We will see the red dot. We can now click on the vertex to remove the vertices from `50` to `7`, similar to the following screenshot:

18. Next, we will adjust the vertices to match the sprite by pressing *Shift*. We will see the green dot. We can go to each vertex to move it.

    Tip

    We can also add the vertex by pressing Shift and then clicking on the line.

    For more information about Polygon Physics 2D, visit http://docs.unity3d.com/Documentation/Components/class-PolygonCollider2D.html.

19. We will get the result as shown in the following screenshot (the position of each vertex shows as the circle):

20. Then, we will create the **Edge** game object, which is basically to prevent the character from getting stuck on the platform edge. Let's go to **GameObject** | **Create Empty**, name it `Edge`, and drag it inside **Platform_2**.
21. Then, we will add **Box Collider 2D** to the **Edge** object by going to **Component** | **Physics2D** | **Box Collider 2D**. Go to **Inspector** and set it as follows:

Transform	
Position	**X**: `1.47`, **Y**: `-0.2`, and **Z**: `0`
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `333.4`
Box Collider 2D	
Material	**Edge** (drag the Physics 2D Material here)
Size	**X**: `0.21` and **Y**: `0.5`

22. Next, we will duplicate the **Edge** game object by pressing *Ctrl* + *D* or *command* + *D* to create the second **Edge** object and set the values of the attributes as follows:

Transform	
Position	**X**: `-1.52`, **Y**: `-0.02`, and **Z**: `0`
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `9.2`
Box Collider 2D	
Material	**Edge** (drag the Physics 2D Material here)
Size	**X**: `0.21` and **Y**: `0.67`

23. We got our **Platform_2** game object. Next, we will create the prefab of this game object by dragging the **Platform_2** game object from the **Hierarchy** view to the **SimplePlatform** | **Resources** | **Prefabs** in the **Project** view.
24. Next, we will create two more **Platform_2** objects. Let's press *Ctrl* + *D* or *command* + *D* twice to create the second and third **Platform_2** object and change the transform position as follows:

Transform (second `Platform_2` object)	
Position	**X**: `-10`, **Y**: `-3.27`, and Z: `0`
Transform (third `Platform_2` object)	
Position	**X**: `-9.25`, **Y**: `-5.87`, and **Z**: `0`
Scale	**X**: `1.3`, **Y**: `1.3`, and **Z**: `1`

25. Finally, we will see the result of our level as the following screenshot, and we are now ready for the next step:

Objective complete – mini debriefing

Basically, what we have done here is create a **Background** object and the **Level** object, and set each platform that attached **Box Collider 2D** and **Polygon Collider 2D**, which we will use to check the collision and make our player walkable on the platform. We also set the **Order in Layer** in the **Sprite Renderer** component to get the correct depth from the background to foreground, as shown in the following diagram:

We also adjust and remove the vertices of **Polygon Collider 2D** by holding the *Shift* or *Ctrl*/*command* key. Then, we create the **Edge** object that uses a different Physics2D Material from the platform object, which is used to prevent our player from getting stuck on the platform's edge.

Classified intel

In 3D Unity mode, the camera can be set to either **Orthographic** or **Perspective**. The difference between both projections is that with the **Orthographic Projection**, the object won't scale by a factor inversely proportional to the distance from the camera. So in our scene, we will see only one side of the cube that faces the camera. On the other hand, in the **Perspective Projection**, we will see that the face of the cube will scale down depending on the distance from the camera, which is similar to real life:

On the other hand, in the 2D mode, we won't see any significant difference between **Orthographic Projection** and **Perspective Projection**. In some cases, we will see only one side if the cube is directly facing the camera. However, if we want to zoom in or out the 2D camera in the **Orthographic Projection**, we can only adjust the **Size** parameter. The Z transform position will not do anything, which is different from the **Perspective Projection**. We can either adjust the Z transform position or the **Field of View** parameter to zoom in or out as shown in the following screenshot:

The Sprite Renderer and Sorting Layer

In the **Sprite Renderer**, we will see that there are two **Layer** properties, **Sorting Layer** and **Order in Layer**, which will help us sort the depth of our sprite and tell Unity which sprite should be drawn first (the lower number will get drawn first). We only use the **Order in Layer** property to sort the depth of our sprite in the same layer group. So, here's when **Sorting Layer** comes in handy.

If we go to **Edit | Project Settings | Tags & Layers**, we will see that there is **Sorting**

Layers. This is the new feature in Unity so that we can set our sprites in this layer group and tell Unity which layer group should be drawn first. For example, we can set the background layer to contain all the background sprites and then set the foreground layer to contain all the foreground sprites. This way we will always know that the foreground sprites will be drawn after the background sprites, as shown in the following screenshot:

For more information about **Sprite Renderer** and **Sorting Layer**, visit http://docs.unity3d.com/Documentation/Components/class-SpriteRenderer.html.

Creating a 2D character and animation

In this step, we will start with setting up our sprites. Next, we will create our 2D character from the sprite sheet in our **SimplePlatform** package. Then, we will create the animator controller and the animation clip for idle, walking, falling, and jumping.

Prepare for lift off

Let's begin with setting up our character from the sprite sheet:

1. Now, we will go to **SimplePlatform** | **Resources** | **Sprites** | **Characters** and click on the **Spritesheet** file inside this folder in the **Project** view. Then, we will go to the **Inspector** view, set it as shown the following screenshot, and click on **Apply**:

2. In the **Inspector** view, we click on the **Sprite Editor** button, which will open the **Sprite Editor** window. Then, we click on the **Slice** drop-down button to bring up the **Slice** window. We will leave the parameters as it is and then click on the **Slice** button to slice all the sprites as the following screenshot:

3. Now, we will see the white frame on each sprite, click on the first sprite (on the top-left corner) to bring up the sprite window, and rename from `Spritesheet_0` to `Walk_0`, as shown in the following screenshot:

4. Next, we will keep going by clicking on each sprite from top-left to bottom-right corner and rename the sprites as follows:

Sprite name	Renamed sprite
`Spritesheet_0`	`Walk_0`
`Spritesheet_1`	`Walk_1`
`Spritesheet_2`	`Walk_2`
`Spritesheet_3`	`Walk_3`
`Spritesheet_4`	`Walk_4`
`Spritesheet_5`	`Walk_5`
`Spritesheet_6`	`Walk_6`
`Spritesheet_7`	`Idle_0`

Spritesheet_8	Idle_1
Spritesheet_9	Jump_0
Spritesheet_10	Jump_1
Spritesheet_11	Jump_2

Then, we will click on the **Apply** button at the top-right corner of the **Sprite Editor** window, as shown in the following screenshot:

5. If we go to the **Project** view, we will see that the **Spritesheet** file has an arrow in front of it. If we click on the arrow, we see all the sprites that we just renamed, as shown in the following screenshot:

6. Finally, we will go to **SimplePlatform** | **Resources** | **PhysicsMaterials** in the **Project** view. We will see that there is **Physics2D Material** for **Ground** and **Edge**, which control **Friction** and **Bounciness** between **Player** and **Level**. We still need to create one more **Physics2D Material** player for our player. So, right-click and go to **Create** | **Physics2D Material**, name it `Player`, and set **Friction** to `1.2` and **Bounciness** to `0`, as shown in the following screenshot:

Project
Create
Favorites
Assets
SimplePlatfor
Resources
Animati
Physics
Prefabs
Sound
Sprites
Char
Level
UI
Scenes
Scripts
Assets ▸ SimplePlatform ▸
Create
Reveal in Finder
Open
Delete
Import New Asset...
Import Package
Export Package...
Find References In Scene
Select Dependencies
Refresh ⌘R
Reimport
Shader
Compute Shader
Prefab
Material
Cubemap
Lens Flare
Render Texture
Animator Controller
Animation
Animator Override C
Avatar Mask
Physic Material
Physics2D Material
Assets ▸ SimplePlatform ▸
Edge
Ground
Player
Inspector
Player
Friction 1.2
Bounciness 0

Engage thrusters

Now, we can start creating our character sprite. Let's get started:

1. Let's go to **SimplePlatform** | **Resources** | **Sprites** | **Characters** in the **Project** view and click on the arrow in front of the **Spritesheet** file. Then, we will click on both the **Idle_0** and **Idle_1** objects and drag it to the **Hierarchy** view. We will see the pop-up window asking to create a new animation, set the filename to `Idle`, put it in **SimplePlatform** | **Resources** | **Sprites** | **Animation**, and click on **Save**, as shown in the following screenshot:

2. Next, we will go to **SimplePlatform** | **Resources** | **Sprites** | **Animation** and click on the **Idle_0** object and rename it to `PlayerAnimator`.
3. Then, we go back to **SimplePlatform** | **Resources** | **Sprites** | **Characters** in the **Project** view and click on the arrow in front of the **Spritesheet** file. After that, we will click on all **Walk** objects from **Walk_0** to **Walk_7** (eight objects in total), and drag them to the **Idle_0** game object in the **Hierarchy** view. We will see the pop-up window asking us to create the new animation again, set the filename to `Walk`, put it in **SimplePlatform** | **Resources** | **Sprites** | **Animation**, and click on **Save**.
4. In the **Spritesheet** file in the **Project** view, we will click on **Jump_0** and **Jump_1** and drag them to the **Idle_0** game object in the **Hierarchy** view. We will see the pop-up window asking us to create the new animation again, set the filename to `Jump`, put it in **SimplePlatform** | **Resources** | **Sprites** | **Animation**, and click on **Save**.
5. We will do the same as the last step, but this time we will click on **Jump_1** and **Jump_2**, and drag them to the **Idle_0** game object in the **Hierarchy** view. We will see the pop-up window asking us to create the new animation again, set the filename

to `Fall`, put it in **SimplePlatform** | **Resources** | **Sprites** | **Animation**, and click on **Save**.

6. Next, we will go to **SimplePlatform** | **Resources** | **Sprites** | **Animation** in the **Project** view. Then, we will click on the **Fall** and **Jump** animation, go to their **Inspector** view, and uncheck **Loop Time** for both clips, as shown in the following screenshot:

7. Then, we go to the **Hierarchy** view, rename **Idle_0** to `Player`, go to its **Inspector** view, and set the values of the attributes as follows:

Tag	**Player**
Layer	**Default**
Transform	
Position	**X**: `-10`, **Y**: `4.6`, and **Z**: `0`
Animator	
Apply Root Motion	Uncheck this option
Culling Mode	**Based On Renderers**

8. Next, we will set up **PlayerAnimator**. Let's go to **SimplePlatform** | **Resources** | **Sprites** | **Animation** in the **Project** view and double-click the **PlayerAnimator** object to bring up the **Animator** view. Then, we click on each clip and layout the position similar to the following screenshot:

9. Then, we click on each clip and go to **Inspector** to set the speed of each animation clip. We will start with the **Idle** clip and move ahead; let's go to each clip inspector and set it up as follows:

Clip	**Speed**
Idle	0.15
Walk	0.8
Jump	0.5
Fall	0.5

10. Next, we will add a few parameters to control when each animation will be played. Let's go to the **Parameters** tab at the bottom-left corner of the **Animator** view, click on the plus icon four times, and choose the name and type of parameter as follows:

Name	**Type**
Speed	Float
Jump	Trigger
Fall	Trigger
Ground	Trigger

These parameters will be used as a condition to control the changing state of each animation clip on our character.

11. Next, we will create the transition between each animation state. First, we will start with our base animation **Idle** and transition to **Walk**. Right-click on the **Idle** clip and choose **Make Transition**, which will bring up the arrow, and click-and-drag again on top of the **Walk** clip. We will see the arrow line link from the **Idle** clip to the **Walk** clip, as shown in the following screenshot:

12. Then, we want to click on the arrow line that we just created and go to **Inspector | Conditions** and set **Speed** to a value greater than 0.1.
13. Next, we will create the transition back from the **Walk** to **Idle** animation. So, let's right-click on the **Walk** clip, choose **Make Transition**, and then click-and-drag on **Idle**. Then, we will click on the arrow line again to go to the **Inspector** and set a value less than 0.1 for **Speed** in the **Conditions** view.
14. Then, we set up the rest of the animation. This time we want the transition from **Walk** to **Fall**. So, let's right-click on the **Walk** clip, choose **Make Transition**, and then click-and-drag on **Fall**. After that, we will click on the arrow line again to go to **Inspector** and set the **Conditions** view to **Fall**.
15. Also, we will create the transition from **Fall** back to **Walk**, go to the transition **Inspector**, click on the plus icon to add another condition and set **Speed** as well as **Ground** to a value greater than 0.1.
16. Next, we want the **Idle** clip transition to **Fall**. Let's create the transition arrow from **Idle** to **Fall**, go to **Inspector**, and set the **Conditions** view to **Fall**.
17. We also need to create the transition from **Fall** back to **Idle**, set up its **Inspector** view, click on the plus icon to add another condition, and set **Speed** to a value less than 0.1.
18. Then, we will create the transition arrow from **Idle** to **Jump** and set up its **Inspector** view by setting its **Conditions** view to **Jump**.
19. Next, we create the transition arrow from **Walk** to **Jump** and set up its **Inspector** view by setting its **Conditions** view to **Jump**.
20. Then, we need the transition arrow from **Jump** to **Fall** and set up its **Inspector** view by setting its **Conditions** view to **Fall**.
21. The current **Animator** view will look similar to the following screenshot:

22. Finally, we will need to attach **Rigidbody 2D** and **Box Collider 2D** to our **Player** game object. Let's go back to the **Hierarchy** view, click on the **Player** game object, go to **Component | Physics 2D | Rigidbody 2D** to add **Rigidbody 2D**, go to **Component | Physics 2D | Box Collider 2D** to add **Box Collider 2D**, and then set up **Inspector** as follows:

Rididbody 2D	
Fixed Angle	Check the box
Interpolate	**Interpolate**
Box Collider 2D	
Material	**Player** (drag the Physics2D Material here)

Tip

Rigidbody – Mass and Drag / Rigidbody2D – Mass and Linear Drag

In Unity **Rigidbody** or **Rigidbody2D**, **Mass** doesn't make the object fall faster or slower. The speed of the object will depend on gravity and drag. **Mass** will only be used when the object is colliding with another, as the higher mass will push the lower mass more. (Make sure you keep **Mass** between **0.1** and never more than **10**). The links to the documentation are as follows: http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody2D-mass.html and http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody-mass.html.

Drag or **Linear Drag** can be used to slow down the object. The higher the drag, the

more the object will slow down. The links to the documentation are as follows: http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody2D-drag.html and http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody-drag.html.

Objective complete – mini debriefing

We just set up the **Player** sprite from the **Spritesheet** file that included multiple sprites by using the Unity **Sprite Import Settings** to slice each sprite and rename it to make it easier to use in each animation clip. We also create the animator controller, which is a part of the Mecanim animation system, to switch the state between each animation clip by using the float value and trigger condition.

The Mecanim animation system is the new animation system in Unity, which helps us to simplify the complex interactions between each animation with a visual programming tool. It also provides an easy workflow to set up the humanoid character animation. For more information on the Mecanim animation system, visit http://docs.unity3d.com/Documentation/Manual/MecanimAnimationSystem.html.

We will get in more details about Mecanim in Project 4, *Add Character Control and Animation to Our Hero/Heroine*.

Finally, we add **RigidBody 2D** and **Box Collider 2D** to our player. Then, we tell **Rigidbody 2D** to use the **Fixed Angle** and **Interpolate** mode. Also, we set the **Box Collider 2D Material** to use the **Player Physics 2D Material** that we've created.

Tip

Fixed Angle and Interpolate mode

Why do we need to perform **Fixed Angle** of **RigidBody 2D**? Is this also similar to the freezing of the rotation and position of **Rigidbody**? To use **Fixed Angle** of **RigidBody 2D**, we basically tell Unity that it will ignore the calculation of the rotation of the object. This will make our character always stay at the same angle while moving. In this way, we can also save the CPU cycles because Unity will ignore the unnecessary calculation and only calculate the one it needs. For the **Interpolate mode**, we've set it to **Interpolate** to ensure smooth motion, which use the object's position of the previous frame to calculate the next position. This will also prevent jerky movement. For more information on Interpolate, visit http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody2D-interpolation.html.

Classified intel

In this step, we create the **Player** physics 2D material object and assign it to **Box Collider 2D** for our **Player** game object. So, what is **Physics2D Material**? If we take a close look at the inspector of the **Physics2D Material** object, we will see that there are two properties: **Friction** and **Bounciness**. **Friction** is the force that resists the relative motion of this collider. **Bounciness** is the degree to which collisions rebound from the surface (`0` means no bounce and `1` means perfect bounce with no loss energy).

Note

For more details of **Physics2D Material**, visit http://docs.unity3d.com/Documentation/Components/class-PhysicsMaterial2D.html.

We've set **Friction** for the **Player** physics 2D material to `1.2`, which means that we will use **1** to calculate the coefficient of friction in this collider. If we remember the previous step, we also assigned the **Edge** and **Ground** physics 2D material object on our platform and edge. If we take a look at the **Ground** physics 2D material's **Friction** value, we will see that the value is equal to **1**. This means that when we stop moving our character, there won't be any much force left to push the character forward.

On the other hand, the **Edge** physics 2D material object has the **Friction** value set to `0`, which means that there is no friction in this collider. So, it will result our character not being able to stop on this collider. This helps us to prevent our character from getting stuck on the edge of the platform, as shown in the following screenshot:

Controlling the character with the PlayerController_2D class

In this section, we will create a new script to control the movement of our character and a sprite animation for each action of our character. We will use MonoDevelop as our scripting editor, which comes with Unity. MonoDevelop is mainly designed for the C# and .NET environments, so if you are comfortable with C#, you will probably love it. However, if you use **Unity JavaScript**, it also has many tools to help us write the script faster and debug better, such as finding as well as replacing words in the whole project by pressing *command* + *Shift* + *F* in Mac or *Ctrl* + *Shift* + *F* in Windows and autocomplete (or Intellisense), to name a few. Moving from Unity JavaScript to C# or C# to Unity JavaScript is also a comparatively smooth transition.

Note

In this version of this book, we will show examples for both Unity JavaScript and C#. You can check out the major difference between Unity JavaScript and C# in the Appendix C, *Major differences Between C# and Unity JavaScript*.

Prepare for lift off

Now, we are just about to start coding, but first let's make sure that we have everything ready:

1. Next, we want to make sure that Unity uses MonoDevelop as our main Scripting editor (**Unity** | **Preferences** in Mac or **Edit** | **Preferences** in Windows).
2. We will see a **Unity Preferences** window. In the **External Tools** tab, go to the **External Script Editor** and make sure that the **MonoDevelop** option is selected. Click on **Browse…**, go to **Applications** | **Unity** | **MonoDevelop.app** in Mac or **{unity install path}** | **Unity** | **MonoDevelop** | **MonoDevelop.exe** in Windows, and we are done:

If we develop a game for Android, we can also set the **Android SDK Location** path here as shown in the previous screenshot.

Engage thrusters

Now, we are ready to create the `PlayerController_2D` script. Let's get started:

1. First, go to **Assets** | **Create** | **Javascript** (for Unity JavaScript developers) or **Assets** | **Create** | **C# Script** (for C# developers) and name our script as `PlayerController_2D`.
2. Double-click on the script; it will open the **MonoDevelop** window.
3. Now, we will see three windows in the **MonoDevelop** screen:
   - On the top-left is **Solution**; we can see our project folder here, but it will only show the folder that contains a script
   - On the bottom-left, we will see a **Document Outline**; this window will show all the functions, classes, and parameters in the file
   - The last window on the right will be used to type our code
4. Let's get our hands dirty with some code—first start with defining the following functions to initialize the variables: the `Awake()` and `Start()` function.

   ### Tip

   `Awake ()`

   `Awake ()` is called when the script instance is being loaded. It used to initialize any variable or game state before calling the `Start()` function. In the `Awake()` function, we usually put any `GetComponent()` or `Find()` object function, which will make it easier to set up all the parameters during `Start()`.

5. We need to remove the autogenerated code and replace it with the following code:

```
// Unity JavaScript user:

#pragma strict
@script RequireComponent(AudioSource)
@script RequireComponent(BoxCollider2D)
@script RequireComponent(Rigidbody2D)
// Distance from the character position to the ground
private final var RAYCAST_DISTANCE : float = 0.58f;
// This number should be between 0.35 to 0.49
private final var CHARACTER_EDGE_OFFSET : float = 0.40f;
var doorOpenSound : AudioClip;
var getKeySound : AudioClip;
var jumpSound : AudioClip;
var moveForce : float = 80f;
var jumpForce : float = 350f;
var maxSpeed : float = 3f;
var layerMask : LayerMask;
private var _animator : Animator;
private var _boxCollider2D : BoxCollider2D;
private var _isFacingRight : boolean = true;
private var _isJump : boolean = false;
private var _isFall : boolean = false;
private var _isGrounded : boolean = false;
private var _gameEnd : boolean = false;
```

```
private var _height : float;
private var _lastY : float;
private var _horizontalInput : float;
function Awake() {
  _animator = GetComponent.<Animator>();
  _boxCollider2D = GetComponent.<BoxCollider2D>();
  _height = _boxCollider2D.size.y;
}
function Start () {
  _isFacingRight = true;
  _isJump = false;
  _isFall = false;
  _isGrounded = false;
  _gameEnd = false;
  _lastY = transform.position.y;
  Camera.main.transform.position = new Vector3(transform.position.x,
transform.position.y, Camera.main.transform.position.z);
}
```

Note

`#pragma strict`

In Unity JavaScript, we can use `#pragma strict` to tell Unity to disable dynamic typing (`var name = 5`) and force us to use static typing (`var name : int = 5`). This will make it easy for us to debug. For example, if we forgot to use static typing, you will see an error message from the Unity console window. Using strict typing also makes the code run faster as the complier doesn't have to go and do the type lookups.

```
// C# user:

using UnityEngine;
using System.Collections;
[RequireComponent(typeof(AudioSource))]
[RequireComponent(typeof(BoxCollider2D))]
[RequireComponent(typeof(Rigidbody2D))]
public class PlayerController_2D : MonoBehaviour
{
  // Distance from the character position to the ground
  const float RAYCAST_DISTANCE = 0.58f;
  // This number should be between 0.35 to 0.49
  const float CHARACTER_EDGE_OFFSET = 0.40f;

  public AudioClip doorOpenSound;
  public AudioClip getKeySound;
  public AudioClip jumpSound;
  public float moveForce = 80f;
  public float jumpForce = 350f;
  public float maxSpeed = 3f;
  public LayerMask layerMask;
  Animator _animator;
  BoxCollider2D _boxCollider2D;
  bool _isFacingRight = true;
  bool _isJump = false;
  bool _isFall = false;
  bool _isGrounded = false;
```

```
  bool _gameEnd = false;
  float _height;
  float _lastY;
  float _horizontalInput;
  void Awake() {
    _animator = GetComponent<Animator>();
    _boxCollider2D = GetComponent<BoxCollider2D>();
    _height = _boxCollider2D.size.y;
  }
  void Start () {
    _isFacingRight = true;
    _isJump = false;
    _isFall = false;
    _isGrounded = false;
    _gameEnd = false;
    _lastY = transform.position.y;
    Camera.main.transform.position = new Vector3(transform.position.x,
transform.position.y, Camera.main.transform.position.z);
  }
}
```

Tip

Downloading the example code

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

Note

`@script RequireComponent(Component)` and `[RequireComponent(typeof(Component))]`

We add `RequireComponent` to force the script to automatically add the required component as a dependency when adding this class to the game object. For more details, visit http://docs.unity3d.com/Documentation/ScriptReference/RequireComponent.html.

6. Next, we will add the `Flip()` function after the `Start()` function to make our character sprite show the correct graphics when moving left or right. The code for Unity JavaScript and C# users are as follows:

```
// Unity JavaScript user:

private function Flip () {
  _isFacingRight = !_isFacingRight;
  var scale : Vector3 = transform.localScale;
  scale.x *= -1;
  transform.localScale = scale;
}


// C# user: (Put the code inside the class)
```

```
void Flip () {
  _isFacingRight = !_isFacingRight;
  Vector3 scale = transform.localScale;
  scale.x *= -1;
  transform.localScale = scale;
}
```

7. Next, we will add another function, which will use `Physics2D.Raycast` to check whether the player is on the ground or not. Let's create the `Grounded()` function as follows:

```
// Unity JavaScript user:

private function Grounded () {
  var distance : float = _height*RAYCAST_DISTANCE;
  var hitDirectionV3 : Vector3 = transform.TransformDirection(-
Vector3.up);
  var hitDirection : Vector2 = new
Vector2(hitDirectionV3.x,hitDirectionV3.y);
  var rightOrigin : Vector2 = new Vector2(transform.position.x +
(_boxCollider2D.size.x*CHARACTER_EDGE_OFFSET), transform.position.y);
  var leftOrigin : Vector2 = new Vector2(transform.position.x -
(_boxCollider2D.size.x*CHARACTER_EDGE_OFFSET), transform.position.y);
  var origin : Vector2 = new Vector2(transform.position.x,
transform.position.y);
  if (Physics2D.Raycast (origin, hitDirection, distance,
layerMask.value)) {
    _isGrounded = true;
  } else if (Physics2D.Raycast (rightOrigin, hitDirection, distance,
layerMask.value)) {
    _isGrounded = true;
  } else if (Physics2D.Raycast (leftOrigin, hitDirection, distance,
layerMask.value)) {
    _isGrounded = true;
  } else {
    if (_isGrounded) {
      if (Mathf.Floor(transform.position.y) == _lastY) {
        _isGrounded = true;
      } else {
        _isGrounded = false;
      }
    }
  }
  _lastY = Mathf.Floor(transform.position.y);
}


// C# user: (Put the code inside the class)

void Grounded () {
  float distance = _height*RAYCAST_DISTANCE;
  Vector3 hitDirectionV3 = transform.TransformDirection (-Vector3.up);
  Vector2 hitDirection = new
Vector2(hitDirectionV3.x,hitDirectionV3.y);
  Vector2 rightOrigin = new Vector2(transform.position.x +
```

```
(_boxCollider2D.size.x*CHARACTER_EDGE_OFFSET), transform.position.y);
  Vector2 leftOrigin = new Vector2(transform.position.x -
(_boxCollider2D.size.x*CHARACTER_EDGE_OFFSET), transform.position.y);
  Vector2 origin = new Vector2(transform.position.x,
transform.position.y);
  if (Physics2D.Raycast (origin, hitDirection, distance,
layerMask.value)) {
    _isGrounded = true;
  } else if (Physics2D.Raycast (rightOrigin, hitDirection, distance,
layerMask.value)) {
    _isGrounded = true;
  } else if (Physics2D.Raycast (leftOrigin, hitDirection, distance,
layerMask.value)) {
    _isGrounded = true;
  } else {
    if (_isGrounded) {
      if (Mathf.Floor(transform.position.y) == _lastY) {
        _isGrounded = true;
      } else {
        _isGrounded = false;
      }
    }
  }
  _lastY = Mathf.Floor(transform.position.y);
}
```

8. Next, we will include the script in the `Update()` function and `LateUpdate()` function that get called after the `Update()` function. These functions will be used to check the input from the user and update the camera to follow our character movement. The code for these functions are as follows:

```
// Unity JavaScript user:

function Update() {
  if (!_gameEnd) {
    _horizontalInput = Input.GetAxis("Horizontal");
    if ((!_isFacingRight && (_horizontalInput > 0)) ||
      (_isFacingRight && (_horizontalInput < 0))) {
      Flip();
    }
    Grounded();
    if(Input.GetButtonDown("Jump") && _isGrounded) {
      _isJump = true;
    }
  }
}
function LateUpdate() {
  if (!_gameEnd) {
    Camera.main.transform.position = new Vector3(transform.position.x,
transform.position.y, Camera.main.transform.position.z);
  }
}


// C# user: (Put the code inside the class)
```

```
void Update() {
  if (!_gameEnd) {
    _horizontalInput = Input.GetAxis("Horizontal");
    if ((!_isFacingRight && (_horizontalInput > 0)) ||
      (_isFacingRight && (_horizontalInput < 0))) {
      Flip();
    }
    Grounded();
    if(Input.GetButtonDown("Jump") && _isGrounded) {
      _isJump = true;
    }
  }
}
void LateUpdate() {
  if (!_gameEnd) {
    Camera.main.transform.position = new Vector3(transform.position.x,
transform.position.y, Camera.main.transform.position.z);
  }
}
```

9. Then, we create a `FixedUpdate()` function, which will handle the animation state changing (**Idle**, **Walk**, **Jump**, or **Fall**) and add force to move our character horizontally and vertically. Let's add this function as follows:

```
// Unity JavaScript user:

function FixedUpdate () {
  if (!_gameEnd) {
    _animator.SetFloat("Speed", Mathf.Abs(_horizontalInput));
    var xSpeed : float = Mathf.Abs(_horizontalInput *
rigidbody2D.velocity.x);
    if (xSpeed < maxSpeed) {
      rigidbody2D.AddForce(Vector2.right * _horizontalInput *
moveForce);
    }
    if (Mathf.Abs(rigidbody2D.velocity.x) > maxSpeed) {
      var newVelocity : Vector2 = rigidbody2D.velocity;
      newVelocity.x = Mathf.Sign(newVelocity.x) * maxSpeed;
      rigidbody2D.velocity = newVelocity;
    }
    if(_isJump) {
      _animator.SetTrigger("Jump");
      audio.volume = 0.3f;
      audio.PlayOneShot(jumpSound);
      rigidbody2D.AddForce(new Vector2(0f, jumpForce));
      _isJump = false;
    }
    if (!_isGrounded) {
      if ((rigidbody2D.velocity.y <= 0f) && !_isFall) {
        _animator.SetTrigger("Fall");
        _isFall = true;
      }
    }
    if (_isGrounded) {
      if (_isFall) {
```

```
        _animator.SetTrigger("Ground");
        _isFall = false;
      } else {
        var animationStateInfo : AnimatorStateInfo =
_animator.GetCurrentAnimatorStateInfo(0);
        if ((rigidbody2D.velocity.y < 0f) &&
(animationStateInfo.IsName("Base Layer.Jump"))) {
          _animator.SetTrigger("Fall");
          _isFall = true;
        }
      }
    }
  }
}


// C# user: (Put the code inside the class)

void FixedUpdate () {
  if (!_gameEnd) {
    #region Setting player horizontal movement
    _animator.SetFloat("Speed", Mathf.Abs(_horizontalInput));
    float xSpeed = Mathf.Abs(_horizontalInput *
rigidbody2D.velocity.x);
    if (xSpeed < maxSpeed) {
      rigidbody2D.AddForce(Vector2.right * _horizontalInput *
moveForce);
    }
    if (Mathf.Abs(rigidbody2D.velocity.x) > maxSpeed) {
      Vector2 newVelocity = rigidbody2D.velocity;
      newVelocity.x = Mathf.Sign(newVelocity.x) * maxSpeed;
      rigidbody2D.velocity = newVelocity;
    }
    #endregion
    #region If the player should jump
    if(_isJump) {
      _animator.SetTrigger("Jump");
      audio.volume = 0.3f;
      audio.PlayOneShot(jumpSound);
      rigidbody2D.AddForce(new Vector2(0f, jumpForce));
      _isJump = false;
    }
    #endregion
    #region If the player should fall
    if (!_isGrounded) {
      if ((rigidbody2D.velocity.y <= 0f) && !_isFall) {
        _animator.SetTrigger("Fall");
        _isFall = true;
      }
    }
    #endregion
    #region If the player is grounded
    if (_isGrounded) {
      if (_isFall) {
        _animator.SetTrigger("Ground");
        _isFall = false;
```

```
      } else {
        AnimatorStateInfo animationStateInfo =
_animator.GetCurrentAnimatorStateInfo(0);
        if ((rigidbody2D.velocity.y < 0f) &&
(animationStateInfo.IsName("Base Layer.Jump"))) {
          _animator.SetTrigger("Fall");
          _isFall = true;
        }
      }
    }
    #endregion
  }
}
```

Tip

`#region YourComment - #endregion`

In C#, we can put any script between `#region Your Code Discription` and `#endregion` to specify a block of code that we can expand or collapse by clicking on the plus and minus sign in MonoDevelop.

We can see how to use `#region ….. #endregion` in the following screenshot:

Lastly, we will add the `OnDrawGizmos()` function in this class. It is a very nice function that allows us to debug the game, the result of which we won't see in the real game. Let's add the following block of code:

```
// Unity JavaScript user:

function OnDrawGizmos() {
  _boxCollider2D = GetComponent.<BoxCollider2D>();
  _height = _boxCollider2D.size.y;
  var distance : float = (_height * RAYCAST_DISTANCE);
  var rightOrigin : Vector3 = new Vector3(transform.position.x +
(_boxCollider2D.size.x*CHARACTER_EDGE_OFFSET), transform.position.y,
transform.position.z);
```

```
  var leftOrigin : Vector3 = new Vector3(transform.position.x -
(_boxCollider2D.size.x*CHARACTER_EDGE_OFFSET), transform.position.y,
transform.position.z);
  Gizmos.color = Color.red;
  Gizmos.DrawRay(transform.position, transform.TransformDirection (-
Vector3.up) * distance);
  Gizmos.DrawRay(rightOrigin, transform.TransformDirection (-
Vector3.up) * distance);
  Gizmos.DrawRay(leftOrigin, transform.TransformDirection (-Vector3.up)
* distance);
}


// C# user: (Put the code inside the class)
void OnDrawGizmos() {
  _boxCollider2D = GetComponent<BoxCollider2D>();
  _height = _boxCollider2D.size.y;
  float distance = (_height * RAYCAST_DISTANCE);
  Vector3 rightOrigin = new Vector3(transform.position.x +
(_boxCollider2D.size.x*CHARACTER_EDGE_OFFSET), transform.position.y,
transform.position.z);
  Vector3 leftOrigin = new Vector3(transform.position.x -
(_boxCollider2D.size.x*CHARACTER_EDGE_OFFSET), transform.position.y,
transform.position.z);
  Gizmos.color = Color.red;
  Gizmos.DrawRay(transform.position, transform.TransformDirection (-
Vector3.up) * distance);
  Gizmos.DrawRay(rightOrigin, transform.TransformDirection (-
Vector3.up) * distance);
  Gizmos.DrawRay(leftOrigin, transform.TransformDirection (-Vector3.up)
* distance);
}
```

10. Now, save it and go back to Unity; drag-and-drop the **PlayerController_2D** script to **Player**, click on **Player**, and go to the **Inspector** window. Click on **Idle Sprite** and **Walk Sprite** to expand it and then set the following:

Player Controller_2D (Script)	
Door Open Sound	`doorOpen`
Get Key Sound	`getkey`
Jump Sound	`jump`
Layer Mask	**Ground**

Yes! We are done. Let’s click on the **Play** button to play the game. We will see our player moving his hand back and forth. Next, press the *A* or left arrow / *D* or right arrow to move the player to the left or to the right; now we see that he is walking. We can also press the Space bar to make our character jump. Isn’t that cool?

Objective complete – mini debriefing

We just created a script that controls the movement of our character and his animation. First, we created the parameters to use in our script. We also used the `const` keyword in C# and the `final` keyword in Unity JavaScript to create the constant variables.

Note

The `const` and `final` keywords

We used the `const` keyword in C# to specify that the value of the field or local variable is constant, which means it can't be modified from anywhere else.

However, there is no `const` keyword in Unity JavaScript, so we use the `final` keyword instead; there is a slight difference between the `final` and `const` keywords. The `const` keyword can only applied to a field whose value is to be known at compile time, such as `const float`. So, this means that we can't use the `const` keyword with `Vector3`, `Rect`, `Point`, and all the struct that are included in Unity. On the other hand, we will use `readonly` instead. However, the `final` keyword can be used in the both cases.

For more details, visit http://tutorials.csharp-online.net/CSharp_FAQ%3A_What_are_the_differences_between_CSharp_and_Java_cons

Then, we get `_animator`, `_boxCollider2D`, and `_height` in the `Awake()` function to check the animation state while it is moving. Then, we set all the variables to their default value. Next, we created the `Flip()` function, which will set `_isFacingRight` to `true` or `false` depending on the character movement. This function also sets the local `x` scale to `1` if the character is facing right and `-1` if it's facing left.

The next function that we created is the `Grounded()` function that uses `Physics2D.Raycast` to check whether our character is on the ground or not.

In the `Update()` function, we get the character's horizontal movement by using `Input.GetAxis("Horizontal")`. Then, we check for the movement direction and call the `Flip()` function. After that, we call the `Grounded()` function to check whether the character is on the ground or not. If not, we can make the character jump by using `Input.GetButtonDown("Jump")`. Next, we update our camera position by using `Camera.main` to get access to the **Main Camera** object and set its position relative to **Player**.

Tip

`Camera.main` lets us access a **Camera** object that has the **MainCamera** tag from anywhere we want.

In the `FixedUpdate()` function, we change the animation state from **Idle** to **Walk** by using `_animator.SetFloat("Speed", Mathf.Abs(_horizontalInput));`. Next, we get the `x` speed and check that the speed is lower than the maximum speed. Then, we add the force to **Rigidbody2D** by using `rigidbody2D.AddForce(Vector2.right * _horizontalInput * moveForce);`. We also make sure that the velocity doesn't reach the maximum limit

after adding the force by assigning `newVelocity` to `rigidbody2D.velocity`.

Next, we trigger the jump animation state by using `_animator.SetTrigger("Jump");`. We also add the force to make our character jump using `rigidbody2D.AddForce(new Vector2(0f,jumpForce);`. Then, we trigger the fall animation state by using `_animator.SetTrigger("Fall");`. We also trigger the ground animation state by using `_animator.SetTrigger("Ground");` if our character is falling.

Then, we check for the current animation state by getting `animatorStateInfo` using `_animator.GetCurrentAnimatorStateInfo(0);`. Then, if the current state is the jump state and `y` velocity lower than `0`, we will change the animation state to the fall state using `if ((rigidbody2D.velocity.y < 0) && (animationStateInfo.IsName("BaseLayer.Jump")).`

At last, we use the `OnDrawGizmos()` function to see the visual of the ray that was drawn in the `Grounded()` function using `Physics2D.Raycast`. We can also use this function to debug the game without removing any code when releasing the game. Because all the code in the `OnDrawGizmos()` function won't be shown in the real game, it's a convenient way to debug our game. As we can see from the following figure, the red arrows represent where the raycast is, which will only show in the scene view:

Classified intel

In Unity, we can set a custom Input Manager by going to **Edit** | **Project Settings** | **Input**. In **Inspector**, click on **Axes** and you will see that the value of **Size** is `15`, which is the array length of all the inputs. If we want more than 15 inputs, we can put the number here (the default is `15`). Next, we will see all 15 names from **Horizontal** to **Jump** as a default setting. Each one will have its own parameters, which we can set up as follows:

For more information of each parameter, go to the following link: http://docs.unity3d.com/Manual/class-InputManager.html.

The **Negative Button** and **Positive Button** here will send the negative and positive value, which in most cases is used to control directions such as, left, right, up, and down. There is a **Dead** parameter, which will set any number that is lower than this parameter to `0`, which is very useful when we use a joystick.

Also, setting **Type** to **Key or Mouse Button** and enabling the **Snap** parameter will reset axis values to zero after it receives opposite inputs.

Physics2D.Raycast

If we take a look at the `Grounded()` function and check out the following highlighted code, we will see that we have cast the ray a bit longer than the sprite collider:

```
if (Physics2D.Raycast (origin, hitDirection, distance, layerMask.value)) {
  _isGrounded = true;
} else if (Physics2D.Raycast (rightOrigin, hitDirection, distance,
layerMask.value)) {
  _isGrounded = true;
} else if (Physics2D.Raycast (leftOrigin, hitDirection, distance,
layerMask.value)) {
  _isGrounded = true;
}
```

From the highlighted code, we draw `Raycast` from the middle of our character downward by `0.58` units. Why don't we set it to `0.5`? We set it this way to make our character stay on the edge of the floor surface. If we set it to `0.5`, we will see that the character will not hit the ground. This is because of the way **Box Collider 2D** and **Polygon Collider 2D** detect other collider objects in Unity, as we can see in the following screenshot:

One last thing for `Gizmos`: if we want to see our gizmos in the game scene, we can click on the **Gizmos** tab on the top-right corner of the game scene:

Creating a key, door, and replay button

In this step, we will create the end point (door). We will also create a trigger `Collider`, which will check for the condition that if the player collected the item, he/she can win the game; of course, the item's the key to open our door. At last, we will create the restart button so that we can restart the game after the game ends.

Prepare for lift off

Let’s make sure that we have the sound effects setting correct by going to the **SimplePlatform | Resources | Sound** folder in the **Project** view. We will see **buttonClick.aiff**, **doorOpen.wav**, **getKey.aiff**, and **jump.wav**. Unity, by default, translates every sound that we import in our project to 3D, but we don’t really need it as we are creating a 2D game. So, we will click on each sound in the **Sound** folder in the **Project** view, go to their **Inspector** window, and make sure that the **3D Sound** is unchecked. If not, uncheck it and click on the **Apply** button:

Engage thrusters

Here, we will create the object's **Key** and **Door** object first, and then we will create the **RestartButton** using **GUITexture**. Let's do this as follows:

1. Let's go to **SimplePlatform** | **Resources** | **Sprites** | **Level** in the **Project** view and drag the **Key** object to the **Hierarchy** view. At the **Hierarchy** view, we will click on the **Key** game object and then go to **Component** | **Physics 2D** | **Circle Collider 2D** to add the **Circle Collider 2D** component. Next, we will go to its **Inspector** view, and set it as follows:

Tag	**Key**
Transform	
Position	**X**: `10`, **Y**: `-2`, and **Z**: `0`
Scale	**X**: `0.7`, **Y**: `0.7`, and **Z**: `1`
Sprite Renderer	
Color	**R**: `255`, **G**: `142`, **B**: `0`, and **A**: `255`
Circle Collider 2D	
Is Trigger	Check the box

2. Next, we click on the **Key** game object in the **Hierarchy** view and drag inside **SimplePlatform** | **Resources** | **Prefabs** in the **Project** view to create the **Key** prefab object.
3. Then, we will create the **Door** game object by going to **SimplePlatform** | **Resources** | **Sprites** | **Level** in the **Project** view and dragging the **DoorClose** object to the **Hierarchy** view. At the **Hierarchy** view, we will click on the **DoorClose** game object, go to **Component** | **Physics 2D** | **Box Collider 2D**, and then go to **Component** | **Miscellaneous** | **Animator**.
4. Next, we will click on the **DoorClose** game object, rename it to `Door`, and go to **Inspector**, and set it as follows:

Tag	**Door**
Transform	
Position	**X**: `0.65`, **Y**: `-2.38`, and **Z**: `0`
Scale	**X**: `1.5`, **Y**: `1.5`, and **Z**: `1`
Circle Collider 2D	
Is Trigger	Check the box

Animator	
Controller	**DoorAnimator**
Apply Root Motion	Uncheck the box
Culling Mode	**Based On Renderers**

5. Now we've got the **Key** and **Door** game object. We need to go back to the **PlayerController_2D** script and add a function to make this work. Let's open the **PlayerController_2D** script and add the `OnTriggerEnter2D()` function as shown in the following highlighted script:

```
// Unity JavaScript user:

private var _hasKey : boolean = false;
   …
function Start () {
…
  _hasKey = false;
}
…
function OnTriggerEnter2D (hit : Collider2D) : IEnumerator {
  if (!_gameEnd) {
    if (hit.tag == "Key") {
      if (!_hasKey) {
        _hasKey = true;
        audio.volume = 1.0f;
        audio.PlayOneShot(getKeySound);
        Destroy (hit.gameObject);
      }
    }

    if (hit.tag == "Door") {
      if (_hasKey) {
        _gameEnd = true;
        _animator.enabled = false;
        audio.volume = 1.0f;
        audio.PlayOneShot(doorOpenSound);
        var doorAnimator : Animator = hit.GetComponent.<Animator>();
        doorAnimator.SetTrigger("DoorOpen");
        yield WaitForSeconds(1);
        doorAnimator.SetTrigger("DoorClose");
        Destroy (gameObject);
      }
    }
  }
}


// C# user: (Put the code inside the class)

bool _hasKey = false;
…
void Start () {
```

```
…
  _hasKey = false;
}
…
IEnumerator OnTriggerEnter2D (Collider2D hit) {
  if (!_gameEnd) {
    if (hit.tag == "Key") {
      if (!_hasKey) {
        _hasKey = true;
        audio.volume = 1.0f;
        audio.PlayOneShot(getKeySound);
        Destroy (hit.gameObject);
      }
    }
    if (hit.tag == "Door") {
      if (_hasKey) {
        _gameEnd = true;
        _animator.enabled = false;
        audio.volume = 1.0f;
        audio.PlayOneShot(doorOpenSound);
        Animator doorAnimator = hit.GetComponent<Animator>();
        doorAnimator.SetTrigger("DoorOpen");
        yield return new WaitForSeconds(1);
        doorAnimator.SetTrigger("DoorClose");
        Destroy (gameObject);
      }
    }
  }
}
```

6. Next, we click on the **Player** game object in the **Hierarchy** view and drag the object inside **SimplePlatform** | **Resources** | **Prefabs** in the **Project** view to create the **Player** prefab object.
7. Inside **SimplePlatform** | **Resources** | **Prefabs** in the **Project** view, we will see the **RestartButton_C#** and **RestartButton_JS** prefab. Now, we will create a new **RestartButton** from those prefabs, drag the **RestartButton_C#** prefab (for C# user), and drag the **RestartButton_JS** prefab (for Unity JavaScript user) to the **Hierarchy** view.

Tip

From now on, we will call this prefab `RestartButton` instead of `RestartButton_C#` or `RestartButton_JS`.

8. Then, let's click on the **RestartButton** prefab, go to its **Inspector** view, and set the values of the attributes as follows:

Tag	**RestartButton**
Texture Button (Script)	
Key	Drag the `Key` prefab object from the `Project` view
Player	Drag the `Player` prefab object from the `Project` view

Note

Make sure that you use the prefab object from the **Project** view, not the **Hierarchy** view. If not, the script might not work properly.

9. For the last step, we will go back to the **PlayerController_2D** script and the `_restartButton` variable to make this work. Let's open the **PlayerController_2D** script and add the `OnTriggerEnter2D()` function as shown in the highlighted script:

```
// Unity JavaScript user:
…
private var _restartButton : GUITexture;
…
function Awake () {
  …
  _restartButton = GameObject.FindWithTag("RestartButton").guiTexture;
}
function Start () {
  …
  _restartButton.enabled = false;
}
…
function OnTriggerEnter2D (hit : Collider2D) : IEnumerator {
  if (!_gameEnd) {
    …
    if (hit.tag == "Door") {
      if (_hasKey) {
       …
        _restartButton.enabled = true;
      }
    }
  }
}


// C# user: (Put the code inside the class)
…
GUITexture _restartButton;
…
void Awake () {
  …
  _restartButton = GameObject.FindWithTag("RestartButton").guiTexture;
}
void Start () {
…
  _restartButton.enabled = false;
}
…
IEnumerator OnTriggerEnter2D (Collider2D hit) {
  if (!_gameEnd) {
   …
    if (hit.tag == "Door") {
      if (_hasKey) {
        …
```

```
            _restartButton.enabled = true;
        }
    }
  }
}
```

Ok, now we are done; click on **Play** to see what we have. Now, when we collect the key and go inside the door, we will see a **Restart** button appear; click on this button and the game will restart.

Objective complete – mini debriefing

We just created a **Key** and **Door** object and placed them in our level. We also created the function that will trigger when the character hits the **Key** and **Door** objects. For the **Door** object, we used **DoorAnimator** for **Animator Controller**. If we double-click on **DoorAnimator** and check the **Animator** view, we will see that there are only two animation states and two trigger parameters to switch between the opening and closing of the door, which is a concept similar to **PlayerAnimator**, as shown in the following screenshot:

We switched the state to **DoorOpen** when the character got the **Key** object and hit the **Door** object. Then, we waited for one second to remove our character from the scene and switch animation back to **DoorClose** by using **Yield** and **Destroy**.

Lastly, we added the **RestartButton** prefab to the scene. This prefab is the **GUITexture** object, which we can click to reset the game after the player goes to the door.

Classified intel

In our script, we need to wait for a second between opening the door and ending the game. We could do this by looping or performing some other task for a second, but that would stop the animations, the sound, and everything else. We get around this by using the `yield` command and the `Coroutines()` function.

Coroutines

The `yield` command tells Unity to stop running our function and come back later (in our game, one second later as we call `yield WaitForSecond(1)` (Unity JavaScript) or `yield return new WaitForSecond(1)` (C#). By using the `yield` command, our function becomes `Coroutines` and now it must return `IEnumerator` (Unity needs this so that it can tell when to start our function again). This means Coroutines can't return a value like a normal function. We can change most functions in our `MonoBehaviours` script into `Coroutines`, apart from the ones that already run in every frame, such as `Update()`, `FixedUpdate()`, and `OnGUI()`. We can get more information about Coroutines from the following Unity script reference:
http://unity3d.com/support/documentation/ScriptReference/Coroutine.html

The Restart button

If we take a look at the `TextureButton` script, we will see that it uses the mouse event function, which will check for the mouse roll over, mouse roll out, and mouse up events. In the `OnMouseUp()` function, we will see that we create the new player and key by using `Instantiate` to clone both prefab objects and set it back in the scene.

We can also add `Application.LoadLevel(LevelName)` to reset our game, which is much easier than using `Instantiate`, but `Application.LoadLevel` will destroy all the game objects in the scene and reload again.

In this case, we use `Instantiate` in our game because we only have one scene and don't want to load the whole game level again. However, we can also put `DontDestroyOnLoad()` in the `Awake()` function of the object that we don't want to destroy, but it needs a bit of setting up. So, there is no right or wrong. It depends on what we want to use or where we want the project to go.

Mission accomplished

We just created a simple 2D platform game, and it is our first piece in getting started with Unity. In this project, we learned how to manage a sprite animation by using the new 2D feature in Unity and the Animator Controller from the Mecanim Animation system. We have gone through the MonoDevelop scripting editor. Also, we learned the basics of how to use Input Manager, Physics2D Raycast, Gizmos, and Collider2D. Finally, we attached the sound effect and a **Restart** button to our game. Let’s take a look at what we have:

Hotshot challenges

Now we have a game that looks good, but it’s not complete yet. So, why don’t you try to do something by using the knowledge gained from this project to add more fun to your game and make it look better? Let’s try the following:

- Add a background music and more sound effects
- Design a challenging level, such as create a movable platform, collect more items at open the door, or even have a longer level
- Add obstacles that can make your character die, lose hit points, or restart to another position
- Add hit point for our character
- Create an animated background or level by using the sprite
- Create a parallax background by adding more layers for the background or foreground objects

Chapter 2. Create a Menu for an RPG – Add Powerups, Weapons, and Armors

Here, we are in the second project. When we talk about traditional role-playing games, we will probably be thinking about the development of the character, such as the attributes, skills, powers, levels, or experiences. When we are playing an RPG, we typically have to open the menu or UI to adjust and manage our main character, such as increase the character attribute, change the weapon, or choose skills. The menu is very important in an RPG. So, in this project, we will create the menu window for a simple RPG-like game using a `GUI` class in Unity.

Note

From Unity 4.6, there will be the new GUI system for a Unity user, which is faster in performance, easier to use, and has many more features such as 3D canvas, dynamic GUI, and event system. This project won't go over the new GUI system because the book is based on Unity 4.3. However, the project will show you how to use the old GUI system or the **Immediate Mode GUI** (**IMGUI**) system. The good thing to know is that the custom editor in Unity is based on the IMGUI system, which will help you to get the basic understanding to create the custom editor. More details on the new GUI system (uGUI) can be found at the following links:

- http://blogs.unity3d.com/2014/05/28/overview-of-the-new-ui-system/
- https://www.youtube.com/watch?feature=player_embedded&v=EOX6itCuKOc

Mission briefing

We'll create a simple menu, yet complex enough for the RPG. In this project, we will continue using some assets from Project 1, *Develop a Sprite and Platform Game*. So, we won't have to recreate the character. The menu will include a **STATUS** tab that will show us the current attributes, skills, and equipments of our character. Next is the **INVENTORY** tab that will contain all the items that our character has as well as the information for each item when the user rolls over the tab.

The last tab is the **EQUIPMENT** tab with which the user will be able to change the weapons, armors, accessories, and skills, as shown in the following screenshot:

The purpose of this project is to understand the `GUI` class in Unity and create our custom user interface, which is different from **GUITexture** that we used to create our **RestartButton** in Project 1, *Develop a Sprite and Platform Game*. There is also **GUIText**, which is used to display the text of any font that we import in the screen coordinate. Both are types of rendering components that can be used once per object. So, if we try to create a complete menu, we will need many **GUITexture/GUIText** objects and the scripts to

handle them. On the other hand, the `GUI` class is operating inside one function `OnGUI`, and we only deal with one object and only create a script that will display all buttons in the **Menu** tab.

Note

The `OnGUI` function acts in a similar way to an `Update` function, but `OnGUI` gets called more than once for rendering and handling the `GUI` events, meaning that the `OnGUI` implementation might be called several times per frame (one call per event), which means it is recalculated at least twice per frame, which can make it slow than another `GUIElement` (`GUITexture` or `GUIText`). More importantly, each `GUI` object will also create one or more draw calls, which means that this can cause a performance hit easily. So, it's not recommended to use `OnGUI` for the in-game UI (especially in game development), instead, we can use `GUIElement`.

Draw calls is how many materials from each object are being drawn to the screen. Around 200 to 300 draw calls is considered acceptable on a mobile platform.

Also, there are many remarkable UI tools in the assets store that are easier to implement and they have a great performance for mobile development, such as the new GUI system (uGUI) on Unity 4.6, NGUI (recommended), and EZGUI.

On the other hand, the `GUI` class can also be used with the `editor` class to create a custom inspector or window in Unity, which is very helpful. (We will learn about this in a later project.)

In this project, we will apply the custom GUI graphics to Unity by using `GUISkin`. We can have multiple styles for our GUI graphics in Unity. Let's say that we have multiple types of fonts that we want to use in our menu; Unity has a way to do this. We can create a `GUISkin` element and apply our custom skin to the area that we want to show the font in. That is the great thing about Unity.

First, we will create the `Item`, `ItemsContainer`, and `SkillsContainer` classes to contain our items and equipments. Then, we will create a menu scripting class that will bring up a new menu window in the game scene when the player presses the *M* key. Next, we will generate a script to create three tab buttons, which will take the player to each window, **STATUS**, **INVENTORY**, and **EQUIPMENT**.

In the **STATUS** tab, we will create a script that will show the image of our character, hit points, magic points, skills, and all attributes of this character. Next, we will create the **INVENTORY** tab, which will contain all the items that the player can scroll up and down to choose an item. Finally, we will create the **EQUIPMENT** tab that the player can use to manage and change the equipments and skills of the character by clicking on it.

Next, we will create a menu game object and item game objects, and apply scripts to them. Then, we will add parameters and textures to our menu and start playing the game. Lastly, we will add the `item` script to the **Key** element on the scene. Then, we will add an item and show this in the menu when we will collect the **Key** element and remove it when we use the **Key** element to open the door.

Why is it awesome?

When we will complete this project, we will be able to create our custom UI for our RPG; this GUI can be used not only in an RPG, but we can also create the user interface for every genre. We will get a good understanding of the `GUI` class in Unity, which is very powerful to create an awesome user interface such as with *Dungeon Hunter*, and *Final Fantasy*, and can be used in the `editor` class too.

Your Hotshot objectives

Because we are creating a menu for an RPG-styled game, we need a menu that is a little more complex than the usual menu. So, this menu will be split into five tasks. The following is an outline of the tasks:

- Customizing skin with **GUISkin**
- Creating a menu object
- Creating the **STATUS** tab
- Creating the **INVENTORY** tab
- Creating the **EQUIPMENT** tab

Mission checklist

Before we start, we will need to get the project folder and assets from this book's website, http://www.packtpub.com/support?nid=8267, which includes the finished project from Project 1, *Develop a Sprite and Platform Game*, and the assets that we will need to use in this project.

Browse the preceding URL and download the `Chapter2.zip` package and then unzip it. Inside the `Chapter2` folder, there are two unity packages, which are `Chapter2Package.unitypackage` (we will use this package for this project) and `Chapter2Package_Completed.unitypackage` (this is the completed project's package, which includes both C# and Unity JavaScript).

Customizing skin with GUISkin

Those who are familiar with HTML will probably have a good understanding of using a repetitive image for a background to reduce memory usage. Unity uses the same idea to create a graphic for the user interface, which will save a lot of memory and size for our game. In this section, we will take a look at the **GUISkin** feature, which is the main key to creating a custom skin in Unity.

Prepare for lift off

We will begin by creating a new project in Unity. Let's start our project by performing the following steps:

1. First, create a new project and name it `MenuInRPG`, similar to what we did in [Project 1](), *Develop a Sprite and Platform Game*. Click on the **Create new Project** button, as shown in the following screenshot:

2. Next, import the assets package by going to **Assets | Import Package | Custom Package…**; choose `Chapter2Package.unityPackage`, which we just downloaded, and then click on the **Import** button in the pop-up window link, as shown in the following screenshot:

3. Wait until it's done, and you will see the `MenuInRPGGame` and `SimplePlatform` folders in the **Window** view. Next, click on the arrow in front of the `SimplePlatform` folder to bring up the drop-down options and you will see the **Scenes** folder and the **SimplePlatform_C#** and **SimplePlatform_JS** scenes, as shown in the following screenshot:

4. Next, double-click on the **SimplePlatform_C#** (for a C# user) and **SimplePlatform_JS** (for a Unity JavaScript user) scenes, as shown in the preceding screenshot, to open the scene that we will work on in this project.
5. When you double-click on either of the **SimplePlatform** scenes, Unity will display a pop-up window asking whether you want to save the current scene or not. As we want to use the **SimplePlatform** scene, just click on the **Don't Save** button to open up the **SimplePlatform** scene, as shown in the following screenshot:

6. Then, go to the `MenuInRPGGame/Resources/UI` folder and click on the first file to make sure that the **Texture Type** and **Format** fields are selected correctly, as shown in the following screenshot:

Why do we set it up in this way? This is because we want to have a UI graphic to look as close to the source image as possible. However, we set the **Format** field to **Truecolor**, which will make the size of the image larger than **Compress**, but will show the right color of the UI graphics.

7. Next, we need to set up the **Layers** and **Tags** configurations; for this, go to **Edit | Project Settings | Tags** and set them as follows:

Tags	
Element 0	**UI**
Element 1	**Key**
Element 2	**RestartButton**
Element 3	**Floor**
Element 4	**Wall**
Element 5	**Background**
Element 6	**Door**

Layers	
User Layer	**Background**
User Layer	**Level**
Use Layer	**UI**

8. At last, we will save this scene in the `MenuInRPGGame/Scenes` folder, and name it `MenuInRPG` by going to **File** | **Save Scene as…** and then save it.

Engage thrusters

Now we are ready to create a GUI skin; for this, perform the following steps:

1. Let's create a new `GUISkin` object by going to **Assets** | **Create** | **GUISkin**, and we will see **New GUISkin** in our **Project** window. Name the GUISkin object as `MenuSkin`. Then, click on **MenuSkin** and go to its **Inspector** window. We will see something similar to the following screenshot:

2. You will see many properties here, but don't be afraid, because this is the main key to creating custom graphics for our UI. **Font** is the base font for the GUI skin. From **Box** to **Scroll View**, each property is **GUIStyle**, which is used for creating our custom UI. The **Custom Styles** property is the array of **GUIStyle** that we can set up to apply extra styles. **Settings** are the setups for the entire GUI.
3. Next, we will set up the new font style for our menu UI; go to the **Font** line in the **Inspector** view, click the circle icon, and select the **Federation Kalin** font.
4. Now, you have set up the base font for **GUISkin**. Next, click on the arrow in front of the **Box** line to bring up a drop-down list. We will see all the properties, as shown in the following screenshot:

Note

For more information and to learn more about these properties, visit http://unity3d.com/support/documentation/Components/class-GUISkin.html.

Name is basically the name of this style, which by default is **box** (the default style of `GUI.Box`). Next, we will be setting our custom UI to this **GUISkin**; click on the arrow in front of **Normal** to bring up the drop-down list, and you will see two parameters —**Background** and **Text Color**.

5. Click on the circle icon on the right-hand side of the **Background** line to bring up the **Select Texture2D** window and choose the **boxNormal** texture, or you can drag the **boxNormal** texture from the `MenuInRPG/Resources/UI` folder and drop it to the **Background** space.

Tip

We can also use the search bar in the **Select Texture2D** window or the **Project** view to find our texture by typing `boxNormal` in the search bar, as shown in the following screenshot:

6. Then, under the **Text Color** line, we leave the color as the default color—because we don't need any text to be shown in this style—and repeat the previous step with **On Normal** by using the **boxNormal** texture.
7. Next, click on the arrow in front of **Active** under **Background**. Choose the **boxActive** texture, and repeat this step for **On Active**.
8. Then, go to each property in the **Box** style and set the following parameters:
   - **Border**: **Left**: `14`, **Right**: `14`, **Top**: `14`, **Bottom**: `14`
   - **Padding**: **Left**: `6`, **Right**: `6`, **Top**: `6`, **Bottom**: `6`

   For other properties of this style, we will leave them as default.
9. Next, we go to the following properties in the **MenuSkin** inspector and set them as follows:

Label	
Normal | **Text Color**	**R** `27`, **G**: `95`, **B**: `104`, **A**: `255`
Window	
Normal | **Background**	**myWindow**
On Normal | **Background**	**myWindow**
Border	**Left**: `27`, **Right**: `27`, **Top**: `55`, **Bottom**: `96`
Padding	**Left**: `30`, **Right**: `30`, **Top**: `60`, **Bottom**: `30`
Horizontal Scrollbar	
Normal | **Background**	**horScrollBar**
Border	**Left**: `4`, **Right**: `4`, **Top**: `4`, **Bottom**: `4`
Horizontal Scrollbar Thumb	

Normal | Background	**horScrollBarThumbNormal**
Hover | Background	**horScrollBarThumbHover**
Border	**Left**: `4`, **Right**: `4`, **Top**: `4`, **Bottom**: `4`
Horizontal Scrollbar Left Button	
Normal | Background	**arrowLNormal**
Hover | Background	**arrowLHover**
Fixed Width	`14`
Fixed Height	`15`
Horizontal Scrollbar Right Button	
Normal | Background	**arrowRNormal**
Hover | Background	**arrowRHover**
Fixed Width	`14`
Fixed Height	`15`
Vertical Scrollbar	
Normal | Background	**verScrollBar**
Border	**Left**: `4`, **Right**: `4`, **Top**: `4`, **Bottom**: `4`
Padding	**Left**: `0`, **Right**: `0`, **Top**: `0`, **Bottom**: `0`
Vertical Scrollbar Thumb	
Normal | Background	**verScrollBarThumbNormal**
Hover | Background	**verScrollBarThumbHover**
Border	**Left**: `4`, **Right**: `4`, **Top**: `4`, **Bottom**: `4`
Vertical Scrollbar Up Button	
Normal | Background	**arrowUNormal**
Hover | Background	**arrowUHover**
Fixed Width	`16`
Fixed Height	`14`
Vertical Scrollbar Down Button	

Normal | **Background**	**arrowDNormal**
Hover | **Background**	**arrowDHover**
Fixed Width	16
Fixed Height	14

We have finished setting up of the default styles.

10. Now we will go to the **Custom Styles** property and create our custom **GUIStyle** to use for this menu; go to **Custom Styles** and under **Size**, change the value to 6. Then, we will see **Element 0** to **Element 5**.
11. Next, we go to the first element or **Element 0**; under **Name**, type `Tab Button`, and we will see **Element 0** change to **Tab Button**. Set it as follows:

Tab Button (or **Element 0**)	
Name	**Tab Button**
Normal	
Background	**tabButtonNormal**
Text Color	**R**: 27, **G**: 62, **B**: 67, **A**: 255
Hover	
Background	**tabButtonHover**
Text Color	**R**: 211, **G**: 166, **B**: 9, **A**: 255
Active	
Background	**tabButtonActive**
Text Color	**R**: 27, **G**: 62, **B**: 67, **A**: 255
On Normal	
Background	**tabButtonActive**
Text Color	**R**: 27, **G**: 62, **B**: 67, **A**: 255
Border	**Left**: 12, **Right**: 12, **Top**: 12, **Bottom**: 4
Padding	**Left**: 6, **Right**: 6, **Top**: 6, **Bottom**: 4
Font Size	14
Alignment	**Middle Center**
Fixed Height	31

The settings are shown in the following screenshot:

For the **Text Color** value, we can also use the eyedropper tool next to the color box to copy the same color, as we can see in the following screenshot:

12. We have finished our first style, but we still have five styles left, so let's carry on with **Element 1** with the following settings:

Exit Button (or Element 1)	
Name	**Exit Button**
Normal | Background	**buttonCloseNormal**

Hover | Background	**buttonCloseHover**
Fixed Width	26
Fixed Height	22

The settings for **Exit Button** are showed in the following screenshot:

13. The following styles will create a style for **Element 2**:

Text Item (or Element 2)	
Name	**Text Item**
Normal | Text Color	**R**: 27, **G**: 95, **B**: 104, **A**: 255
Alignment	**Middle Left**
Word Wrap	Check

The settings for **Text Item** are shown in the following screenshot:

14. To set up the style for **Element 3**, the following settings should be done:

Text Amount (or Element 3)	
Name	**Text Amount**
Normal | **Text Color**	**R**: `27`, **G**: `95`, **B**: `104`, **A**: `255`
Alignment	**Middle Right**
Word Wrap	Check

The settings for **Text Amount** are shown in the following screenshot:

15. The following settings should be done to create **Selected Item**:

Selected Item (or **Element 4**)	
Name	**Selected Item**
Normal | Text Color	**R**: 27, **G**: 95, **B**: 104, **A**: 255
Hover	
Background	**itemSelectHover**
Text Color	**R**: 27, **G**: 95, **B**: 104, **A**: 255
Active	
Background	**itemSelectHover**
Text Color	**R**: 27, **G**: 95, **B**: 104, **A**: 255
On Normal	
Background	**itemSelectActive**
Text Color	**R**: 27, **G**: 95, **B**: 104, **A**: 255
Border	**Left**: **6**, **Right**: **6**, **Top**: 6, **Bottom**: 6

Margin	**Left**: 2, **Right**: 2, **Top**: 2, **Bottom**: 2
Padding	**Left**: 4, **Right**: 4, **Top**: 4, **Bottom**: 4
Alignment	**Middle Center**
Word Wrap	Check

The settings are shown in the following screenshot:

16. To create the **Disabled Click** style, the following settings should be done:

Disabled Click (or **Element 5**)	
Name	**Disabled Click**
Normal	
Background	**itemSelectNormal**
Text Color	**R**: 27, **G**: 95, **B**: 104, **A**: 255

Border	**Left**: 6, **Right**: 6, **Top**: 6, **Bottom**: 6
Margin	**Left**: 2, **Right**: 2, **Top**: 2, **Bottom**: 2
Padding	**Left**: 4, **Right**: 4, **Top**: 4, **Bottom**: 4
Alignment	**Middle Center**
Word Wrap	Check

The settings for **Disabled Click** are shown in the following screenshot:

And now, we have finished this step.

Objective complete – mini debriefing

Basically, what we have done in this project is we have created the custom GUI skin to use for our menu. First, we tell the GUI that we want to use the font name **Federation Kalin** as our default font for this **GUISkin** by setting up the **Font** type at the first line of the skin's inspector. Then, we changed all the default skin textures to use our custom UI graphics from the `UI` folder by setting up all the necessary properties and parameters in the **Box**, **Label**, **Window**, **Horizontal Scrollbar**, **Horizontal Scrollbar Thumb**, **Horizontal Scrollbar Left Button**, **Horizontal Scrollbar Right Button**, **Vertical Scrollbar**, **Vertical Scrollbar Thumb**, **Vertical Scrollbar Up Button**, and **Vertical Scrollbar Down Button** styles. Then, we created six **Custom Styles**—**Tab Button**, **Exit Button**, **Text Item**, **Text Amount**, **Selected Item**, and **Disabled Click**, which will be used for scripting in the next section.

Note

The **Custom Style** feature is basically **GUIStyle** that we can add to **GUISkin**. This style allows us to create a custom style that will act differently from the default styles (**Box**, **Label**, **Window**, and so on) in this **GUISkin**.

Classified intel

In this section, we applied the UI graphics to **GUISkin**. You might have a question here—how does it work? Here, we will go through the basic concept of how to create a custom UI in Photoshop and get the right texture to use in our **GUISkin**.

First, let's take a look at the **myWindow.png** in our `MenuInRPGGame/Resources/UI` folder. If we select this file, we will see something similar to the following capsule-shaped image. You might be curious—how are we going to create a window graphics with this capsule shape? Well, the trick is the properties of **Border** in which we can set the parameters, **Left**, **Right**, **Top**, and **Bottom**. It uses these parameters to set the number of pixels that will be shown in the fixed image. On the other hand, the pixels in the middle will get repeated depending on the width and height of the settings similar to the **HTML** style or **scale9grid** in Flash (the concept is to draw the pixel perfectly on the four corners and then repeat them in the middle to match the size we need).

The following figure shows us how the Unity **GUIStyle** works:

First, we set the parameters for the **Border** attribute. These parameters will offset the pixels of the current UI graphics from **0** to the number that we will assign. For example, if we want to draw a rectangular window, which is 320 pixels in width and 240 pixels in height, and we set the borders, **Left** to `27`, **Right** to `27`, **Top** to `55`, and **Bottom** to `96`, this will tell Unity **GUIStyle** to always draw the graphics from pixel **0** to pixel **27** on the left-hand side with the same scale as the source texture. What will happen from pixel 28? Basically, it will repeat pixel 27 until it hits the right-hand side border, which is also set to 27 pixels from the right-hand side. So, this means that we tell **GUIStyle** to draw graphics from the source texture, that is from pixel 0 to pixel 27, and then repeat the texture from pixel 28 to pixel 293, then switch back and draw pixel 294 to pixel 320 from the source

texture, which is the offset of 27 pixels from the right-hand side. This also applies to the top and bottom borders, as we can see on the left-hand side of the preceding figure.

From this concept, we can save a lot of memory because instead of using a 320 x 240 pixel image, we just use 54 x 151 pixels. However, in some cases, we don't want any repeating pixels for our UI such as fixed button graphics—for example, our **Exit Button** style—or any fixed texture, and so on, as we can see in the following figure:

We can also set the **Fixed Width** and **Fixed Height** properties in **GUIStyle** to match our image size. For instance, we have the **Exit Image** button, which is 26 pixels wide and 22 pixels high. So, we just set the **Fixed Width** value to `26` and **Fixed Height** to `22`. On the other hand, we can also set either **Fixed Width** or **Fixed Height** in **GUIStyle**—as we already did in our **Tab Button** of **Custom Styles**—as we can see in the following figure:

We set the **Fixed Height** value to `31`, and we leave the **Fixed Width** value at `0`, which means that the height of the style will be 31 pixels always but the width can vary from zero to infinity.

Creating a menu object

Continuing from the first step, we will now create our menu game object in the scene that can open and close the menu window. Pressing the *M* key will open the menu window, and clicking on the **x** button in the window will close the menu window. We will also create three tab buttons for the player to be able to see through the different pages, **STATUS**, **INVENTORY**, and **EQUIPMENT**, as we can see in the following screenshot:

Engage thrusters

We will begin by creating the menu using the following steps:

1. First, we want to create an empty game object in our scene and name it `menu`; go to **GameObject | Create Empty** and name it `MenuObject` and set its **Tag** and **Layer** to **UI**. We will use this object for our menu.
2. Next, we will create the menu script that will control our entire menu; go to **Assets | Create | Javascript** (for Unity JavaScript users) or **Assets | Create | C#** (for C# users), name it `Menu`, double-click on it to launch **MonoDevelop**, and we will get our hands dirty with the code.
3. Open the `Menu` script file and type the following variables:

```
// Unity JavaScript user

#pragma strict
public enum TAB {STATUS,INVENTORY,EQUIPMENT};
var customSkin : GUISkin;
var heroTexture : Texture;
var statBox1Texture : Texture;
var statBox2Texture : Texture;
var skillBoxTexture : Texture;
private final var TOOLBARS : String[] =
[ TAB.STATUS.ToString(), TAB.INVENTORY.ToString(),
TAB.EQUIPMENT.ToString() ];
private final var HERO_RECT : Rect = new Rect (19, 35, 225, 441);
private final var CLOSE_BTN_RECT : Rect = new Rect (598, 8, 26, 22);
private final var TAB_BTN_RECT : Rect = new Rect (35, 15, 480, 40);
private var _currentTool : TAB = TAB.STATUS;
private var _windowRect : Rect = new Rect (10, 10, 640, 480);
private var _isMenuOpen : boolean = false;
```

Note

`final`: We use the word `final` here because we want the value to be assigned only once. We can say that this is the constant value and it is used to prevent the error from reassigning the value.

```
// C# user:

using UnityEngine;
using System.Collections;
public class Menu : MonoBehaviour {
  public enum TAB {STATUS,INVENTORY,EQUIPMENT};
  public GUISkin customSkin;
public Texture heroTexture;
  public Texture statBox1Texture;
  public Texture statBox2Texture;
  public Texture skillBoxTexture;
  readonly string[] TOOLBARS =
{TAB.STATUS.ToString(),
TAB.INVENTORY.ToString(),TAB.EQUIPMENT.ToString()};
  readonly Rect HERO_RECT = new Rect (19, 35, 225, 441);
```

```
  readonly Rect CLOSE_BTN_RECT = new Rect (598, 8, 26, 22);
  readonly Rect TAB_BTN_RECT = new Rect (35, 15, 480, 40);
TAB _currentTool = TAB.STATUS;
  Rect _windowRect = new Rect (10, 10, 640, 480);
bool _isMenuOpen = false;
…
}
```

Note

`readonly`: We use `readonly` instead of `const` here because `string[]` and `Rect` aren't the constant expressions in C# and are not known at the compile time. So, it will produce a compile error if we use `const`. Also, `readonly` is basically similar to the word `final` in Java, which can be initialized only once.

Here, we just created the necessary variables for our menu window, as shown in the following screenshot:

Note

The result of **statBox1Texture**, **statBox2Texture**, and **skillBoxTexture** will be shown in the *Creating the STATUS tab* section.

4. Next, we will set `_isMenuOpen` to `false` in the `Start()` function, because we don't want our menu to show until the player presses the *M* key, so type the code as follows:

```
// Unity JavaScript user:

function Start () : void {
  _isMenuOpen = false;
```

```
}


// C# user:

void Start () {
  _isMenuOpen = false;
}
```

5. Then, we go to the `OnGUI` function, and we will use the `Event` class to check if the user is pressing the correct key.

Note

The `OnGUI` function acts in a similar way to the `Update` function, but `OnGUI` gets called more than once, for rendering and handling the `GUI` events, meaning that `OnGUI` implementation might be called several times per frame (one call per event).

```
// Unity JavaScript user:

function OnGUI () : void {
  GUI.skin = customSkin;
  var e : Event = Event.current;
  if ((e.isKey) && (e.keyCode == KeyCode.M) && (!_isMenuOpen)) {
    _isMenuOpen = true;
    Time.timeScale = 0.0f;
  }
}


// C# user:

void OnGUI () {
  GUI.skin = customSkin; //Assigning custom MenuSkin
  Event e = Event.current;
  if ((e.isKey) && (e.keyCode == KeyCode.M) && (!_isMenuOpen)) {
    _isMenuOpen = true;
    Time.timeScale = 0.0f;
  }
}
```

6. Next, we will create a draggable window menu, which will contain all the buttons and background textures; add this code after the `OnGUI` function as follows:

```
// Unity JavaScript user:

private function DoMyWindow (windowID : int) : void {
  _currentTool = GUI.Toolbar (TAB_BTN_RECT, parseInt(_currentTool),
TOOLBARS, GUI.skin.GetStyle("Tab Button"));
  GUI.DrawTexture(HERO_RECT, heroTexture);
  if (GUI.Button (CLOSE_BTN_RECT, "", GUI.skin.GetStyle("Exit
Button"))) {
    _isMenuOpen = false;
    Time.timeScale = 1.0f;
  }
  GUI.DragWindow();
```

```
}


// C# user:

void DoMyWindow (int windowID ) {
  _currentTool = (TAB) GUI.Toolbar (TAB_BTN_RECT, (int)_currentTool,
TOOLBARS, GUI.skin.GetStyle("Tab Button"));
  GUI.DrawTexture(HERO_RECT, heroTexture);
  if (GUI.Button (CLOSE_BTN_RECT, "", GUI.skin.GetStyle("Exit
Button"))) {
    _isMenuOpen = false;
    Time.timeScale = 1.0f;
  }
  GUI.DragWindow();
}
```

7. Then, we go back to the `OnGUI` function. We will add the code to create a draggable window by using `GUI.Window` and passing the `DoMyWindow()` function that we just created. We also make sure that the window is always on the screen by checking the `x` and `y` positions of `_windowRect`, so add the following highlighted code to the `OnGUI` function.

```
// Unity JavaScript user:

function OnGUI () : void {
  …
  if ((e.isKey) && (e.keyCode == KeyCode.M) && (!_isMenuOpen)) {
     …
  }
  if (_isMenuOpen) {  //Open menu if 'true'
    _windowRect = GUI.Window (0, _windowRect, DoMyWindow, "");
    _windowRect.x = Mathf.Clamp(_windowRect.x, 0.0f, Screen.width -
_windowRect.width);
    _windowRect.y = Mathf.Clamp(_windowRect.y, 0.0f, Screen.height -
_windowRect.height);
  }
}


// C# user:

void OnGUI () {
  …
  if ((e.isKey) && (e.keyCode == KeyCode.M) && (!_isMenuOpen)) {
     …
  }
  if (_isMenuOpen) {  //Open menu if 'true'
    _windowRect = GUI.Window (0, _windowRect, DoMyWindow, "");
    _windowRect.x = Mathf.Clamp(_windowRect.x, 0.0f,
      Screen.width - _windowRect.width);
    _windowRect.y = Mathf.Clamp(_windowRect.y, 0.0f, Screen.height -
_windowRect.height);
  }
}
```

As we want everything inside our menu window, we used the `DoMyWindow()` function to take `GUI.Window` as one parameter. Inside the `DoMyWindow()` function, we create all the buttons and textures. Then, we make our window draggable by adding `GUI.DragWindow()`. With that, we are done with coding for this step.

8. Next, go back to Unity, click on the `Menu` script file and drag-and-drop it to `MenuObject` in the **Hierarchy** view. Next, click on `MenuObject` in the **Hierarchy** view, open the **Inspector** view under the `menu` (script), and set the parameters as follows:

Parameters	**Settings**
Custom Skin	Drag-and-drop **MenuSkin**
Hero Texture	Drag-and-drop **player**
Stat Box 1Texture	Drag-and-drop **stat1**
Stat Box 2Texture	Drag-and-drop **stat2**
Skill Box Texture:	Drag-and-drop **skillBox**

9. Then, we can click on the **Play** button to see the result. In the game scene, we can press the *M* key to bring up our window and click on the **x** button at the top-right corner of the screen to close it, as shown in the following screenshot:

Objective complete – mini debriefing

We just created a menu window, which can be opened by pressing the *M* key and closed by clicking on the **x** button at the top-right corner of the menu window. We also have our character texture nicely placed inside our menu window. Next, we made this window draggable and made sure it is always on the screen by using the following code:

```
_windowRect.x = Mathf.Clamp(_windowRect.x, 0.0f, Screen.width -
_windowRect.width);
_windowRect.y = Mathf.Clamp(_windowRect.y, 0.0f, Screen.height -
_windowRect.height);
```

Basically, we set the minimum limit of our window in the `x` position to `0` and the maximum to the screen width subtracted by the window width; we also set the minimum limit of `y` position to `0` and the maximum to the screen height subtracted by the window height.

We will see this result when we click on **Play the game** and try to drag this window off the screen. Lastly, we created a tab that can be clicked to change to a different page.

Classified intel

In this step, we used the `GUI` class to create our window, box, and buttons, but we can also use a `GUILayout` class to create the same thing as we did with the `GUI` class. The only difference between these two classes is that `GUI` will need to take a `Rect` object to specify the size and position of the UI. On the other hand, `GUILayout` doesn't need to take the `Rect` object. It will automatically adjust the size according to the source it has. Let's say, we want to create a box that contains a text or image, `GUILayout` will automatically adjust the height and width to nicely fit your text or image. For the position, `GUILayout` will automatically set the first position to the top-left corner of the screen, which is (**0**, **0**), and it will continue to the right-hand side or down depending on the `GUILayout` object that we already have on the screen. However, the downside is that we will not be able to create a fixed position or size for the `GUILayout` class. Also, because the `GUILayout` class will automatically calculate the position and size for us, this means that it will be more expensive to use than the `GUI` class. However, the `GUILayout` class is very convenient and the UI editor in Unity is based on this class, we can use it to create the custom editor script combined with the `EditorGUILayout` (we will learn about the custom editor in a later project). Both of these classes are very powerful. We can use them in different situations.

You can see more details of the `GUI` class at the following URL:

http://unity3d.com/support/documentation/ScriptReference/GUI.html

You can see the details of the `GUILayout` class at the following URL:

http://unity3d.com/support/documentation/ScriptReference/GUILayout.html

The `GUI.DragWindow()` function allows us to create a draggable window by specifying the drag area on our window. You can visit the following URL to see the details:

http://unity3d.com/support/documentation/ScriptReference/GUI.DragWindow.html

This function takes one `Rect` parameter, which is the area that allows the user to drag the window around. However, we didn't assign the `Rect` parameters to our `GUI.DragWindow()` function, which means that we can drag the whole window in an area.

Creating the STATUS tab

In this step, we will create a **STATUS** page for our menu, which will show all attributes of the character, including hit points, magic points, levels, experiences, experience needed for the next level, attacks, defense, agility, intelligence, luck, and the current equipments and skills of that character, as shown in the following screenshot:

Engage thrusters

We will start with assigning the **STATUS** parameters for our character and displaying them on the menu by performing the following steps:

1. Let's go back to our `menu` script in MonoDevelop, and include variables before the `Start()` function as highlighted in the following code.

```
// Unity JavaScript user:

…
private var _isMenuOpen : boolean = false;
var fullHP : int = 9999;
var fullMP : int = 999;
var currentHP : int = 9999;
var currentMP : int = 999;
var currentLV : int = 99;
var currentEXP : int = 9999999;
var currentNEXT : int = 99999;
var currentATK : int = 999;
var currentDEF : int = 999;
var currentAGI : int = 999;
var currentINT : int = 999;
var currentLUC : int = 999;
private final var MAX_HP : int = 9999;
private final var MAX_MP : int = 999;
private final var MAX_LV : int = 99;
private final var MAX_EXP : int = 9999999;
private final var MAX_NEXT : int = 99999;
private final var MAX_ATK : int = 999;
private final var MAX_DEF : int = 999;
private final var MAX_AGI : int = 999;
private final var MAX_INT : int = 999;
private final var MAX_LUC : int = 999;
private final var STAT_1_RECT : Rect = new Rect (252, 77, 331, 125);
private final var STAT_2_RECT : Rect = new Rect (252, 244, 331, 142);
private final var HP_RECT : Rect = new Rect (313, 75, 120, 25);
private final var MP_RECT : Rect = new Rect (313, 100, 120, 25);
private final var LV_RECT : Rect = new Rect (313, 124, 120, 25);
private final var EXP_RECT : Rect = new Rect (313, 150, 120, 25);
private final var NEXT_RECT : Rect = new Rect (313, 177, 120, 25);
private final var ATK_RECT : Rect = new Rect (529, 75, 50, 25);
private final var DEF_RECT : Rect = new Rect (529, 100, 50, 25);
private final var AGI_RECT : Rect = new Rect (529, 124, 50, 25);
private final var INT_RECT : Rect = new Rect (529, 150, 50, 25);
private final var LUC_RECT : Rect = new Rect (529, 177, 50, 25);
private final var STAT_BOX_RECT : Rect = new Rect (237, 67, 360, 147);
private final var WEAPON_BOX_RECT : Rect = new Rect (237, 230, 360,
207);
private final var WEAPON_LABEL_RECT : Rect = new Rect (252, 264, 180,
40);
private final var ARMOR_LABEL_RECT : Rect = new Rect (252, 324, 180,
40);
private final var ACCESS_LABEL_RECT : Rect = new Rect (252, 386, 180,
40);
```

```
private final var SKILL_TEX_RECT : Rect = new Rect (464, 288, 119,
117);
private final var SKILL_BOX_RECT : Rect = new Rect (460, 284, 127,
125);
function Start () : void {
…


// C# user:

…
bool _isMenuOpen = false;
public int fullHP = 9999;
public int fullMP = 999;
public int currentHP = 9999;
public int currentMP = 999;
public int currentLV = 99;
public int currentEXP = 9999999;
public int currentNEXT = 99999;
public int currentATK = 999;
public int currentDEF = 999;
public int currentAGI = 999;
public int currentINT = 999;
public int currentLUC = 999;
const int MAX_HP = 9999;
const int MAX_MP = 999;
const int MAX_LV = 99;
const int MAX_EXP = 9999999;
const int MAX_NEXT = 99999;
const int MAX_ATK = 999;
const int MAX_DEF = 999;
const int MAX_AGI = 999;
const int MAX_INT = 999;
const int MAX_LUC = 999;
readonly Rect STAT_1_RECT = new Rect (252, 77, 331, 125);
readonly Rect STAT_2_RECT = new Rect (252, 244, 331, 142);
readonly Rect HP_RECT = new Rect (313, 75, 120, 25);
readonly Rect MP_RECT = new Rect (313, 100, 120, 25);
readonly Rect LV_RECT = new Rect (313, 124, 120, 25);
readonly Rect EXP_RECT = new Rect (313, 150, 120, 25);
readonly Rect NEXT_RECT = new Rect (313, 177, 120, 25);
readonly Rect ATK_RECT = new Rect (529, 75, 50, 25);
readonly Rect DEF_RECT = new Rect (529, 100, 50, 25);
readonly Rect AGI_RECT = new Rect (529, 124, 50, 25);
readonly Rect INT_RECT = new Rect (529, 150, 50, 25);
readonly Rect LUC_RECT = new Rect (529, 177, 50, 25);
readonly Rect STAT_BOX_RECT = new Rect (237, 67, 360, 147);
readonly Rect WEAPON_BOX_RECT = new Rect (237, 230, 360, 207);
readonly Rect WEAPON_LABEL_RECT = new Rect (252, 264, 180, 40);
readonly Rect ARMOR_LABEL_RECT = new Rect (252, 324, 180, 40);
readonly Rect ACCESS_LABEL_RECT = new Rect (252, 386, 180, 40);
readonly Rect SKILL_TEX_RECT = new Rect (464, 288, 119, 117);
readonly Rect SKILL_BOX_RECT = new Rect (460, 284, 127, 125);
void Start () {
…
```

We basically created variables for the `Rect` positions for each UI texture and text labels for each attribute.

2. Next, we will create a new function in this `menu` script after the `DoMyWindow()` function and call it `CheckMax()`, which will limit the maximum and minimum values of the attributes that are highlighted in the following code.

```
// Unity JavaScript user:

…
private function DoMyWindow (windowID : int) : void {
…
}
private function CheckMax () : void {
  fullHP = Mathf.Clamp(fullHP, 0, MAX_HP);
  fullMP = Mathf.Clamp(fullMP, 0, MAX_MP);
  currentHP = Mathf.Clamp(currentHP, 0, fullHP);
  currentMP = Mathf.Clamp(currentMP, 0, fullMP);
  currentLV = Mathf.Clamp(currentLV, 0, MAX_LV);
  currentEXP = Mathf.Clamp(currentEXP, 0, MAX_EXP);
  currentNEXT = Mathf.Clamp(currentNEXT, 0, MAX_NEXT);
  currentATK = Mathf.Clamp(currentATK, 0, MAX_ATK);
  currentDEF = Mathf.Clamp(currentDEF, 0, MAX_DEF);
  currentAGI = Mathf.Clamp(currentAGI, 0, MAX_AGI);
  currentINT = Mathf.Clamp(currentINT, 0, MAX_INT);
  currentLUC = Mathf.Clamp(currentLUC, 0, MAX_LUC);
}


// C# user:

…
void DoMyWindow (int windowID ) {
  …
}
void CheckMax () {
  fullHP = Mathf.Clamp(fullHP, 0, MAX_HP);
  fullMP = Mathf.Clamp(fullMP, 0, MAX_MP);
  currentHP = Mathf.Clamp(currentHP, 0, fullHP);
  currentMP = Mathf.Clamp(currentMP, 0, fullMP);
  currentLV = Mathf.Clamp(currentLV, 0, MAX_LV);
  currentEXP = Mathf.Clamp(currentEXP, 0, MAX_EXP);
  currentNEXT = Mathf.Clamp(currentNEXT, 0, MAX_NEXT);
  currentATK = Mathf.Clamp(currentATK, 0, MAX_ATK);
  currentDEF = Mathf.Clamp(currentDEF, 0, MAX_DEF);
  currentAGI = Mathf.Clamp(currentAGI, 0, MAX_AGI);
  currentINT = Mathf.Clamp(currentINT, 0, MAX_INT);
  currentLUC = Mathf.Clamp(currentLUC, 0, MAX_LUC);
}
```

3. Then, we will create a `StatusWindow()` function, which will show all the statuses in the menu window, so add this function after `CheckMax()`.

```
// Unity JavaScript user:
```

```
…
private function CheckMax () : void {
  …
}
private function StatusWindow() : void {
  GUI.Box (STAT_BOX_RECT, "");
  GUI.Box (WEAPON_BOX_RECT, "");
  GUI.DrawTexture(STAT_1_RECT, statBox1Texture);
  GUI.DrawTexture(STAT_2_RECT, statBox2Texture);
  GUI.DrawTexture(SKILL_BOX_RECT, skillBoxTexture);
  CheckMax();
  GUI.Label(HP_RECT, currentHP.ToString() + "/" + fullHP.ToString(),
"Text Amount");
  GUI.Label(MP_RECT, currentMP.ToString() + "/" + fullMP.ToString(),
"Text Amount");
  GUI.Label(LV_RECT, currentLV.ToString(), "Text Amount");
  GUI.Label(EXP_RECT, currentEXP.ToString(), "Text Amount");
  GUI.Label(NEXT_RECT, currentNEXT.ToString(), "Text Amount");
  GUI.Label(ATK_RECT, currentATK.ToString(), "Text Amount");
  GUI.Label(DEF_RECT, currentDEF.ToString(), "Text Amount");
  GUI.Label(AGI_RECT, currentAGI.ToString(), "Text Amount");
  GUI.Label(INT_RECT, currentINT.ToString(), "Text Amount");
  GUI.Label(LUC_RECT, currentLUC.ToString(), "Text Amount");
}


// C# user:

…
void CheckMax () {
  …
}
void StatusWindow() {
    GUI.Box (STAT_BOX_RECT, "");
    GUI.Box (WEAPON_BOX_RECT, "");
    GUI.DrawTexture(STAT_1_RECT, statBox1Texture);
    GUI.DrawTexture(STAT_2_RECT, statBox2Texture);
    GUI.DrawTexture(SKILL_BOX_RECT, skillBoxTexture);
    CheckMax();
    GUI.Label(HP_RECT, currentHP.ToString() + "/" + fullHP.ToString(),
"Text Amount");
    GUI.Label(MP_RECT, currentMP.ToString() + "/" + fullMP.ToString(),
"Text Amount");
    GUI.Label(LV_RECT, currentLV.ToString(), "Text Amount");
    GUI.Label(EXP_RECT, currentEXP.ToString(), "Text Amount");
    GUI.Label(NEXT_RECT, currentNEXT.ToString(), "Text Amount");
    GUI.Label(ATK_RECT, currentATK.ToString(), "Text Amount");
    GUI.Label(DEF_RECT, currentDEF.ToString(), "Text Amount");
    GUI.Label(AGI_RECT, currentAGI.ToString(), "Text Amount");
    GUI.Label(INT_RECT, currentINT.ToString(), "Text Amount");
    GUI.Label(LUC_RECT, currentLUC.ToString(), "Text Amount");
}
```

4. Then, we need to call `StatusWindow()` from our `DoMyWindow()` to make it show up on our menu window; for this, go back to the `DoMyWindow()` function and add the highlighted code as follows:

```
// Unity JavaScript user:

…
private function DoMyWindow (windowID : int) : void {
  _currentTool = GUI.Toolbar (TAB_BTN_RECT, parseInt(_currentTool),
TOOLBARS, GUI.skin.GetStyle("Tab Button"));
  switch (_currentTool) {
    case TAB.STATUS : //Status
      StatusWindow();
      break;
      }
  GUI.DrawTexture(HERO_RECT, heroTexture);
  …
}
…


// C# user:

…
void DoMyWindow (int windowID ) {
  _currentTool = (TAB) GUI.Toolbar (TAB_BTN_RECT, (int)_currentTool,
TOOLBARS, GUI.skin.GetStyle("Tab Button"));
  switch (_currentTool) {
    case TAB.STATUS : //Status
      StatusWindow();
      break;
      }
  GUI.DrawTexture(HERO_RECT, heroTexture);
  …
}
```

Tip

`switch-case`: To make the code run efficiently, why don't we use the `if-else` statement? Well, the `switch-case` statement makes the code easier to read, and also, this performs faster because the complier doesn't need to compare the value of each state before jumping to the next one, which might be the case when we use the `if-else` statement. The last state might take more time to access because of waiting for the previous state to finish.

Basically, for the `switch-case` statement, all the items will get access at the same time. In our case, we will add another two states later in the next step. Even though there is a slight boost in our code, it recommended to use the `switch-case` if we have more than two states.

We can go back to Unity and click on **Play** to see the result; we will see that all the attributes are shown in the window. However, we will still see that there is no text shown under the **Weapon**, **Armor**, and **Accessory** sections.

5. Next, we will create a new class named `Item`, which basically contains the information of the item, icon, and name. For this, go to **Assets** | **Create** | **Javascript** (for Unity JavaScript users) or **Assets** | **Create** | **C#** (for C# users), name it `Item`,

double-click on it to launch MonoDevelop, and replace the code as follows.

```
// Unity JavaScript user:

#pragma strict
var icon : Texture;
var info : String;
private var _guiContent : GUIContent;
function get guiContent () : GUIContent {
  return new GUIContent(this.name, this.icon, this.info);
}


// C# user:

using UnityEngine;
using System.Collections;

[System.Serializable]
public class Item : MonoBehaviour {
  public Texture icon;
  public string info;
  GUIContent _guiContent;
  public GUIContent guiContent {
    get { return new GUIContent(this.name, this.icon, this.info); }
  }
}
```

This class is basically to set the information of each item and then return the information as `GUIContent` via the `guiContent()` function, which will be shown in our information window.

6. Then, we need to create a class called `ItemsContainer` to contain all the items, initializations, and to set the scroll view (which we will use in the next step); go to **Assets** | **Create** | **Javascript** (for Unity JavaScript users) or **Assets** | **Create** | **C#** (for C# users), name it `ItemsContainer`, double-click on it to launch **MonoDevelop**, and replace the code as follows:

```
// Unity JavaScript user:

#pragma strict
class ItemsContainer {
private final var UNEQUIP : String = "UNEQUIP";
  private final var NONE : String = "NONE";
  private var _guiContent : GUIContent;
  private var _isEquipment : boolean;
  function get guiContent () : GUIContent {
    return _guiContent;
  }
  function Init ( isEquipment : boolean ) : void {
    _isEquipment = isEquipment;
    _guiContent = (_isEquipment) ? new GUIContent(UNEQUIP) : new
GUIContent(NONE);
  }
}
```

```
// C# user (put the code inside the class):

using UnityEngine;
using System.Collections;
[System.Serializable]
public class ItemsContainer {
  const string UNEQUIP = "UNEQUIP";
  const string NONE = "NONE";
  GUIContent _guiContent;
  bool _isEquipment;
  public GUIContent guiContent {
    get { return _guiContent; }
  }
  public void Init ( bool isEquipment ) {
    _isEquipment = isEquipment;
    _guiContent = (_isEquipment) ? new GUIContent(UNEQUIP) : new
GUIContent(NONE);
  }
}
```

Tip

`get`: We use the `get` keyword here because we don't want to set a value outside of this class. This is very helpful to prevent an error from setting the value of this variable outside of this class.

7. Next, we still need to create the last class, which is similar to the `ItemsContainer` class, but this one will contain the skill information; go to **Assets** | **Create** | **Javascript** (for Unity JavaScript users) or **Assets** | **Create** | **C#** (for C# users), name it `SkillsContainer`, double-click on it to launch **MonoDevelop**, and replace the code as follows:

```
// Unity JavaScript user:

#pragma strict
class SkillsContainer {
  private var _guiContent : GUIContent;
  private var _skillBoxTexture : Texture;
  function get guiContent () : GUIContent {
    return _guiContent;
  }
  function Init ( skillBoxTexture : Texture ) : void {
    _guiContent = new GUIContent("");
    _skillBoxTexture = skillBoxTexture;
  }
}


// C# user (put the code inside the class):

using UnityEngine;
using System.Collections;
[System.Serializable]
```

```
public class SkillsContainer {
  GUIContent _guiContent;
  Texture _skillBoxTexture;
  public GUIContent guiContent {
    get { return _guiContent; }
  }
  public void Init ( Texture skillBoxTexture ) {
    _guiContent = new GUIContent("");
    _skillBoxTexture = skillBoxTexture;
  }
}
```

8. Then, we will add variables and initialize it to our menu script; open the menu script and type the following highlighted code:

```
// Unity JavaScript user:

…
private final var SKILL_BOX_RECT : Rect = new Rect (460, 284, 127,
125);
var weapons : ItemsContainer;
var armors : ItemsContainer;
var accessories : ItemsContainer;
var items : ItemsContainer;
var skills : SkillsContainer;
function Start () : void {
  _isMenuOpen = false;
  weapons.Init(true);
  armors.Init(true);
  accessories.Init(true);
  items.Init(false);
  skills.Init(skillBoxTexture);
}
…


// C# user:

…
readonly Rect SKILL_BOX_RECT = new Rect (460, 284, 127, 125);
public ItemsContainer weapons;
public ItemsContainer armors;
public ItemsContainer accessories;
public ItemsContainer items;
public SkillsContainer skills;
void Start () {
  _isMenuOpen = false;
  weapons.Init(true);
  armors.Init(true);
  accessories.Init(true);
  items.Init(false);
  skills.Init(skillBoxTexture);
}
…
```

9. Lastly, we go to the `StatusWindow()` function in this `SkillsCointainer` class, and

add the following highlighted code:

```
// Unity JavaScript user:

private function StatusWindow() : void {
  …
  GUI.Label(LUC_RECT, currentLUC.ToString(), "Text Amount");
  GUI.Label(WEAPON_LABEL_RECT, weapons.guiContent, "Text Item");
  GUI.Label(ARMOR_LABEL_RECT, armors.guiContent, "Text Item");
  GUI.Label(ACCESS_LABEL_RECT, accessories.guiContent, "Text Item");
  GUI.Label(SKILL_TEX_RECT, skills.guiContent, "Text Item");
}


// C# user (put the code inside the class):

void StatusWindow() {
  …
  GUI.Label(LUC_RECT, currentLUC.ToString(), "Text Amount");
  GUI.Label(WEAPON_LABEL_RECT, weapons.guiContent, "Text Item");
  GUI.Label(ARMOR_LABEL_RECT, armors.guiContent, "Text Item");
  GUI.Label(ACCESS_LABEL_RECT, accessories.guiContent, "Text Item");
  GUI.Label(SKILL_TEX_RECT, skills.guiContent, "Text Item");
}
```

10. Now, we can go back to Unity, click on **Play**, and press the *M* key to bring up our menu window. We will see all the attributes for our character, as shown in the following screenshot:

Objective complete – mini debriefing

We just created the `Item`, `ItemsContainer`, and `SkillsContainer` classes to contain our items and skills information, which will be shown on the equipment box. All these classes will be used in the next step to create the item's scroll view. Then, in the `Start()` function, we initialized items, equipments, and skills. In this function, we also set the equipment to the default state, which is the `UNEQUIP` state.

Next, we created the `StatusWindow()` function to show the **STATUS** page when the players see our menu when they first click on the **STATUS** tab. We also created a `CheckMax()` function to make sure that the number of the character attributes is not over or under the limit.

Classified intel

In this section, we used `GUIContent` to contain information of our items and skills, then passed it to the `GUI.Label()` function. Basically, if we take a look at each `GUI` class's function, we will see that they can take many variables such as `Rect`, `string`, `Texture`, `GUIContent`, and `GUIStyle`. We already know `Rect`, `string`, and `Texture`. Also, `GUIStyle` is the name of the style from our `MenuSkin` object that we created, but what is `GUIContent`? It is basically a class that contains the necessary variables (image, text, and tooltip) to apply to our GUI. For example, if we want our button to have an icon, name, and information when the user rolls over it, we can add the following code:

```
GUI.Button(Rect(0,0,100,20), GUIContent("My Button Name", icon, "This is
the button info").
```

The first parameter is the string that will be the text or in our case, it is the button name, which is seen on the button, and next is the graphic's texture or icon that will also be seen on this button. The last string is the information that will be stored in this button, which we call tooltip. We can show this tooltip when the user rolls over this button by calling `GUI.tooltip`. This will automatically show the current button's tooltip that the user rolls over. We will use it in the next section.

Note

For more details about `GUIContent` and `GUI.tooltip`, check out the following website:

http://unity3d.com/support/documentation/ScriptReference/GUIContent.html

Creating the INVENTORY tab

So, we are now in the second page of our menu window, which is the **INVENTORY** page. In this section, we will create an item `scroll` that the player can use to scroll up and down to select the item and see its name, amount, and information about the item.

Engage thrusters

We will start with adding the parameters by performing the following steps, which we will use to store the data in our **INVENTORY** page:

1. Open the `ItemsContainer` class; now, we will create the item's array object and the scroll view that can contain all the items; firstly, add the following highlighted code to create new variables:

```
// Unity JavaScript user:

#pragma strict
import System.Collections.Generic;
…
private var _isEquipment : boolean;
private final var ITEM_BOX_POS : Rect = new Rect (257, 87, 320, 200);
private final var TOOL_BOX_RECT : Rect = new Rect (0, 0, 280, 40);
private final var EQUIP_BOX_POS : Rect = new Rect (257, 300, 320, 120);
var items : List.<Item>;
private var _itemCount : int;
private var _selectedItem : int;
private var _scrollPosition : Vector2;
function get guiContent () : GUIContent {
…


// C# user:

using UnityEngine;
using System.Collections;
using System.Collections.Generic;
…
bool _isEquipment;
readonly Rect ITEM_BOX_POS = new Rect (257, 87, 320, 200);
readonly Rect TOOL_BOX_RECT = new Rect (0, 0, 280, 40);
readonly Rect EQUIP_BOX_POS = new Rect (257, 300, 320, 120);
public List<Item> items;
int _itemCount;
int _selectedItem;
Vector2 _scrollPosition;
public GUIContent guiContent {
…
```

Note

`List.<T>` or `List<T>`: The `List<T>` array is basically an array of type `<T>`, which is easier to add, remove, or insert later in the next step than using the built-in array. Also the performance is a lot better than using an array. To use the `List<T>` array, we need to add the following code at the beginning of the class:

```
import System.Collections.Generic (Unity Javascript) using
System.Collections.Generic (C#)
```

2. Next, we will go to the `Init()` function and add the highlighted code as follows to

set up the value of those variables that we've just created:

```
// Unity JavaScript user:

function Init ( isEquipment : boolean ) : void {
  _isEquipment = isEquipment;
  _guiContent = (_isEquipment) ? new GUIContent(UNEQUIP) : new
GUIContent(NONE);
  _selectedItem = 0;
  _scrollPosition = Vector2.zero;
  _itemCount = items.Count;
}


// C# user:

public void Init ( bool isEquipment ) {
  _isEquipment = isEquipment;
  _guiContent = (_isEquipment) ? new GUIContent(UNEQUIP) : new
GUIContent(NONE);
  _selectedItem = 0;
  _scrollPosition = Vector2.zero;
  _itemCount = items.Count;
}
```

3. Then, we will add the `SetupScrollBar()` function after the `Init()` function as shown the following highlighted code:

```
// Unity JavaScript user:

function Init ( isEquipment : boolean ) : void {
…
}
function SetupScrollBar () {
  var position : Rect = (_isEquipment) ? EQUIP_BOX_POS : ITEM_BOX_POS;
  var view : Rect = TOOL_BOX_RECT;
  var itemCount = _itemCount;
  var isNoItem : boolean = false;
  if (_isEquipment) {
    itemCount++;
  }  else {
    if (itemCount == 0) {
      itemCount++;
      isNoItem = true;
    }
  }
  view.height *= itemCount;
  _scrollPosition = GUI.BeginScrollView (position, _scrollPosition,
view);
  var itemsContent : GUIContent[] = new GUIContent[itemCount];
  if (itemCount > 1) {
    for (var i: int = 0; i < itemCount; i++) {
       itemsContent[i] = (_isEquipment) ? ((i == 0) ? new
GUIContent(UNEQUIP) : items[i-1].guiContent) : items[i].guiContent;
    }
  } else {
```

```
    itemsContent[0] = (isNoItem) ? new GUIContent(NONE) :
((_isEquipment) ? new GUIContent(UNEQUIP) : items[0].guiContent);
  }
  _selectedItem = GUI.SelectionGrid (view, _selectedItem, itemsContent,
1, GUI.skin.GetStyle("Selected Item"));
  GUI.EndScrollView ();
  _guiContent = itemsContent[_selectedItem];
}


// C# user:

public void Init ( bool isEquipment ) {
  …
}

public void SetupScrollBar () {
  Rect position = (_isEquipment) ? EQUIP_BOX_POS : ITEM_BOX_POS;
  Rect view = TOOL_BOX_RECT;
  int itemCount = _itemCount;
bool isNoItem = false;
  if (_isEquipment) {
    itemCount++;
  }  else {
    if (itemCount == 0) {
      itemCount++;
     isNoItem = true;
    }
  }
  view.height *= itemCount;
  _scrollPosition = GUI.BeginScrollView (position, _scrollPosition,
view);
  GUIContent[] itemsContent = new GUIContent[itemCount];
  if (itemCount > 1) {
     for (int i = 0; i < itemCount; i++) {
        itemsContent[i] = (_isEquipment) ? ((i == 0) ? new
GUIContent(UNEQUIP) : items[i-1].guiContent) : items[i].guiContent;
    }
  } else {
    itemsContent[0] = (isNoItem) ? new GUIContent(NONE) :
((_isEquipment) ? new GUIContent(UNEQUIP) : items[0].guiContent);
  }
  _selectedItem = GUI.SelectionGrid (view, _selectedItem, itemsContent,
1, GUI.skin.GetStyle("Selected Item"));
  GUI.EndScrollView ();
  _guiContent = itemsContent[_selectedItem];
}
```

There is a bug in `GUI.SelectionGrid` in Unity 4.0 to 4.2. When you roll over the items, the tooltip always returns an empty string. If you are using Unity 4.0 to 4.2, you can replace these lines of code with the following code:

```
// Unity JavaScript user:

_selectedItem = GUI.SelectionGrid (view, _selectedItem, itemsContent,
1, GUI.skin.GetStyle("Selected Item"));
```

The preceding code should be replaced with the following code:

```
for (var j: int = 0; j < itemsContent.Length; j++) {
   if (_selectedItem == j) {
     GUI.Label(new Rect (0, j*40, 280,
40),itemsContent[j],GUI.skin.GetStyle("Disabled Click"));
   }  else {
     if (GUI.Button(new Rect (0, j*40, 280,
40),itemsContent[j],GUI.skin.GetStyle("Selected Item"))) {
       _selectedItem = j;
     }
   }
}
```

For a C# user, the following code exists:

```
_selectedItem = GUI.SelectionGrid (view, _selectedItem, itemsContent,
1, GUI.skin.GetStyle("Selected Item"));
```

The preceding code should be replaced with the following code:

```
for (int j = 0; j < itemsContent.Length; j++) {
   if (_selectedItem == j) {
     GUI.Label(new Rect (0, j*40, 280,
40),itemsContent[j],GUI.skin.GetStyle("Disabled Click"));
   }  else {
     if (GUI.Button(new Rect (0, j*40, 280,
40),itemsContent[j],GUI.skin.GetStyle("Selected Item"))) {
       _selectedItem = j;
    }
   }
}
```

In this function, first, we check the equipments or items to get the `Rect` position of the item box. Then, we add the `itemCount` item if it's the equipment, which is standing for the `UNEQUIP` selection. Next, we get the height of the scroll view by calculating the number of items from the `itemCount` item. Then, we create the scroll view by using `GUI.BeginScrollView`. Next, we create a `GUIContent` array to contain our items and check if it's the equipment that we add as the first item to `UNEQUIP` then add the rest. Next, we create `GUI.SelectionGrid` and apply the selected item to `_guiContent`, which will be used in the next step in the `Menu` script.

Note

In this step, we've used the `?` and `:` syntaxes. These syntaxes are basically a shortcut to get variables from the `if-else` statement. Consider the following example:

```
var something;if (condition) { something  = true; } else { something
= false; }
```

We can change this to something like the following code:

```
var something = (condition) ? true : false;
```

4. Next, we go back to the `Menu` script and add the variables as highlighted in the

following code:

```
// Unity JavaScript user:

…
var skills : SkillsContainer;
private final var ITEM_BOX_RECT : Rect = new Rect (237, 67, 360, 247);
private final var ITEM_TIP_BOX_RECT : Rect = new Rect (237, 330, 360,
107);
private var _scrollPosition : Vector2 = Vector2.zero;
function Start () : void {
…


// C# user:

…
public SkillsContainer skills;
readonly Rect ITEM_BOX_RECT = new Rect (237, 67, 360, 247);
readonly Rect ITEM_TIP_BOX_RECT = new Rect (237, 330, 360, 107);
Vector2 _scrollPosition = Vector2.zero;
void Start () {
…
```

5. Then, we will add the `ItemWindow()` function after the `StatusWindow()` function as highlighted in the following code:

```
// Unity JavaScript user:

private function StatusWindow() : void {
  …
}
private function ItemWindow() : void {
  GUI.Box (ITEM_BOX_RECT, "");
  GUI.Box (ITEM_TIP_BOX_RECT, "");
  items.SetupScrollBar();
  var info : String = (items.guiContent.tooltip == "") ? "Show items
information here" : items.guiContent.tooltip;
  var style : GUIStyle = GUI.skin.GetStyle("Label");
var tooltip : String = ( GUI.tooltip != "") ? GUI.tooltip : info;
  var height : float = style.CalcHeight(GUIContent(tooltip), 330.0f) +
20;
  _scrollPosition = GUI.BeginScrollView (new Rect (257, 343, 320, 75),
_scrollPosition, new Rect (0, 0, 280, height));
  GUI.Label(new Rect (0, 0, 280, height), tooltip);
  GUI.EndScrollView ();
}


// C# user:

void StatusWindow() {
  …
}
void ItemWindow() {
  GUI.Box (ITEM_BOX_RECT, "");
```

```
  GUI.Box (ITEM_TIP_BOX_RECT, "");
  items.SetupScrollBar();
  string info = (items.guiContent.tooltip == "") ? "Show items
information here" : items.guiContent.tooltip;
  GUIStyle style = GUI.skin.GetStyle("Label");
  string tooltip = ( GUI.tooltip != "") ? GUI.tooltip : info;
  float height = style.CalcHeight(new GUIContent(tooltip), 330.0f) +
20;
  _scrollPosition = GUI.BeginScrollView (new Rect (257, 343, 320, 75),
_scrollPosition, new Rect (0, 0, 280, height));
  GUI.Label(new Rect (0, 0, 280, height), tooltip);
  GUI.EndScrollView ();
}
```

In this function, we first set up the item box and the item tooltip box. Then, we create the item scroll view by calling `items.SetupScrollBar()`. Lastly, we get the selected item tooltip, check if the tooltip isn't an empty string if it's assigned to our string. Then, we calculate the height of this tooltip box by using the `Label` style and create the scroll view for it.

6. Then, we go to the `DoMyWindow()` function inside the `switch` statement and add the highlighted code inside this function to show the **INVENTORY** window when we click on the **INVENTORY** tab, as follows:

```
// Unity JavaScript user:

private function DoMyWindow (windowID : int) : void {
  …
  switch (_currentTool) {
  case TAB.STATUS : //Status
    …
    break;
  case TAB.INVENTORY : //Items
    ItemWindow();
    break;
    }
  …
}


// C# user:

void DoMyWindow (int windowID ) {
  …
  switch (_currentTool) {
  case TAB.STATUS : //Status
    …
    break;
  case TAB.INVENTORY : //Items
    ItemWindow();
    break;
    }
  …
}
```

7. Let's go back to Unity, we can click on **Play** to see the result. We will see only **NONE** in the inventory box and the **Show items information here** text in the tooltip box. So, we need to create the item `GameObject` and add it our `MenuObject` by going to **GameObject | Create Empty** to create the empty game object and name it `Golden Key`.
8. Next, click on the `Golden Key` object that we just created and add the `item` script to this object, then go to its **Inspector** view and set up the attributes as follows:

`Item` **(script)**	
Icon	**keyIcon**
Info	`This golden key looks very old. Looks like, it was made by someone in the Ancient Rome.`

9. Next, drag the `Golden Key` object from the **Hierarchy** view to the `MenuInRPGGame/Resources/Prefab` folder in the **Project** view to create a prefab of this item. Then, we will remove the `Golden Key` object from the **Hierarchy** view by right-clicking on the object and choosing **Delete**.
10. Lastly, click on the `MenuObject` item in the **Hierarchy** view to bring up its **Inspector** view at the `Menu` (script); go to **Items** and add it as follows:

Items	
Size	`1`
Element 0	`Golden Key` (drag from the **Project** view)

Now, click on **Play** and press the *M* key to bring up the menu window. Click on the **INVENTORY** tab and we will see our item page. Isn't that cool?

Tip

We can add more variety of items to the **INVENTORY** page by adding the `GameObject` item that has the `item` script attached, and give a name, icon, and information to the `Menu` (script) as we did in step 10. We can see the result, as shown in the following screenshot:

STATUS
INVENTORY
EQUIPMENT
Golden Key
Key 1
Key 2
Key 3
Key 4
THE
PLAYER
This golden key looks very old. Looks like, it was made by someone in the Ancient Rome.

Objective complete – mini debriefing

We just created our **INVENTORY** page that we'll be able to view by clicking on the **INVENTORY** tab. First, we created the item array by using `List<T>` to contain all the items. Then we added the `SetupScrollBar()` function in the `ItemsContainer`class in this function, we created a scroll view for the items by using `GUI.BeginScrollView()` and `GUI.EndScrollView()`, and created a scrollable area that contains all the items. We also used `GUI.SelectionGrid` to align all items in the list from which the player can select any item. Lastly, we passed the `GUIContent` selection to use in the `Menu` script.

Then, in the `Menu` script in the `Start()` function, we initialized our items, armors, accessories, weapons, and skills objects. Next, we created the `ItemWindow()` function, which we used to control our item page. In this function, we created a scroll view. Then, we got the current tooltip from `GUIContent` that we get from `ItemsContainer` and checked whether there is any information or not. Next, we checked `GUI.tooltip` for any stored string; if nothing is stored here, we assigned the current tooltip from our selected items to `GUI.tooltip`, which showed the result that if we roll over each item, the current information will change to the roll-overed item. On the other hand, if we roll over from our items list, the result will show the selected item's information. Next, we get the `Label` style height from the current `GUI.tooltip` item. Then, we created another scroll view to show this tooltip information in the box area.

In the `DoMyWindow()` function, we added the **INVENTORY** state to show the item page when the user clicks on the **INVENTORY** tab by calling the `ItemWindow()` function.

At last, we created the `GameObject` prefab that has the `item` script attached to it and added it to our item page.

Classified intel

In this step, we were using `GUI.SelectionGrid` to create a list of the items. By using `GUI.SelectionGrid`, we were able to create a list of buttons using one line of code that have a fixed height and space, which was very convenient. We can see more details on how to use `GUI.SelectionGrid` at the following URL:

http://unity3d.com/support/documentation/ScriptReference/GUI.SelectionGrid.html

GUI.tooltip

We also used the `GUI.tooltip` parameter to show our items' information when the player rolls over each item and showed the selected item's information if the player rolls out. So, how does `GUI.tooltip` work? Basically, `GUI.tooltip` will return the string from each button that contains a tooltip string when the player rolls over it. However, if the player rolls out or that button doesn't have any tooltip stored in it, this parameter will automatically return a blank string, similar to the following code that we used in the `ItemWindow()` function in the `Menu` script's class.

A Unity JavaScript user can use the following code:

```
var tooltip : String = ( GUI.tooltip != "") ? GUI.tooltip : info;
```

A C# user can use the following code:

```
string tooltip = ( GUI.tooltip != "") ? GUI.tooltip : info;
```

We basically tell `GUI.tooltip` that we will assign the rollover tooltip information to the label when the player rolls over. And, if the player rolls out, we will show the selected item's information for which the default is the first item, as we can see in the following screenshot:

From the preceding screenshot, we can see that the left-hand side shows that when we roll over the second key, the information box shows the tooltip of the second key. The right-hand side shows that when we roll out from the second key, the information box shows the tooltip of the selected key, which is the first key.

Note

You can see more details about `GUI.tooltip` at the following URL:

http://unity3d.com/support/documentation/ScriptReference/GUI-tooltip.html

Creating the EQUIPMENT tab

This is the last step of our menu. We will create a tab with which the player can change the weapons, armors, accessories, and skills, which will also update variables in the **STATUS** tab, as we can see in the following screenshot:

Engage thrusters

We will start this section by going to the `SkillsContainer` script and adding new variables. This is done by performing the following steps:

1. Open the `SkillsContainer` script and add the following highlighted code:

```
// Unity JavaScript user:

…
private var _skillBoxTexture : Texture;
private final var SKILL_BOX_POS : Rect = new Rect (253, 286, 330, 140);
private final var SKILL_BOX_RECT : Rect = new Rect (0, 0, 125, 117);
var textures : Texture[];
private var _textureCount : int;
private var _selectedSkill : int;
private var _scrollPosition : Vector2;
function get guiContent () : GUIContent {
…


// C# user:

…
Texture _skillBoxTexture;
readonly Rect SKILL_BOX_POS = new Rect (253, 286, 330, 140);
readonly Rect SKILL_BOX_RECT = new Rect (0, 0, 125, 117);
public Texture[] textures;
int _textureCount;
int _selectedSkill;
Vector2 _scrollPosition;
public GUIContent guiContent {
…
```

2. Next, we go to the `Init()` function and add the following highlighted code to set up all variables:

```
// Unity JavaScript user:

function Init ( skillBoxTexture : Texture ) : void {
  _guiContent = new GUIContent("");
  _skillBoxTexture = skillBoxTexture;
  _selectedSkill = 0;
  _scrollPosition = Vector2.zero;
  _textureCount = textures.Length;
}


// C# user:

public void Init ( Texture skillBoxTexture ) {
  _guiContent = new GUIContent("");
  _skillBoxTexture = skillBoxTexture;
  _selectedSkill = 0;
  _scrollPosition = Vector2.zero;
```

```
  _textureCount = textures.Length;
}
```

3. Then, we will set up the scroll view for our skill textures, which is very similar to the one we set in the `ItemsContainer` class, but in this function, we will have a horizontal scroll view instead. So, add the new function after the `Init()` function and name it `SetupSkillScrollBar()`; add the highlighted code as follows:

```
// Unity JavaScript:

function Init ( skillBoxTexture : Texture ) : void {
  …
}
function SetupSkillScrollBar () : void {
  var itemCount = _textureCount+1;
  var itemsContent : GUIContent[] = new GUIContent[itemCount];
  for (var i: int = 0; i < itemCount; i++) {
     itemsContent[i] = (i == 0) ? new GUIContent(_skillBoxTexture) :
new GUIContent(textures[i-1]);
  }
var newView : Rect = SKILL_BOX_RECT;
  newView.width *= itemCount;
_scrollPosition = GUI.BeginScrollView (SKILL_BOX_POS, _scrollPosition,
newView);
  newView,y = 4;
  _selectedSkill = GUI.SelectionGrid (newView, _selectedSkill,
itemsContent, itemCount, GUI.skin.GetStyle("Selected Item"));
  GUI.EndScrollView ();
  _guiContent = (_selectedSkill > 0) ? itemsContent[_selectedSkill] :
new GUIContent("");
}


// C# user:

public void Init ( Texture skillBoxTexture ) {
  …
}
public void SetupSkillScrollBar () {
  int itemCount = _textureCount+1;
  GUIContent[] itemsContent = new GUIContent[itemCount];
for (int i = 0; i < itemCount; i++) {
   itemsContent[i] = (i == 0) ? new GUIContent(_skillBoxTexture) : new
GUIContent(textures[i-1]);
}
Rect newView = SKILL_BOX_RECT;
newView.width *= itemCount;
  _scrollPosition = GUI.BeginScrollView (SKILL_BOX_POS,
_scrollPosition, newView);
  newView.y = 4;
  _selectedSkill = GUI.SelectionGrid (newView, _selectedSkill,
itemsContent, itemCount, GUI.skin.GetStyle("Selected Item"));
  GUI.EndScrollView ();
   _guiContent = (_selectedSkill > 0) ? itemsContent[_selectedSkill] :
new GUIContent("");
```

```
}
```

4. Next, we will go back to our `Menu` script and add the new variables for the **EQUIPMENT** window, so add the highlighted code as follows:

```
// Unity JavaScript user:

…
private var _scrollPosition : Vector2 = Vector2.zero;
private final var EQUIP_BOX_RECT : Rect = new Rect (237, 67, 360, 207);
private final var EQUIP_WEAPON_RECT : Rect = new Rect (237, 280, 360,
157);
private final var EQUIP_STAT_RECT : Rect = new Rect (252, 81, 331,
142);
private final var EQUIP_SKILL_RECT : Rect = new Rect (460, 121, 127,
125);
private final var EQUIP_RECTS : Rect[] = [new Rect (252, 101, 180, 40),
new Rect (252, 161, 180, 40), new Rect (252, 221, 180, 40), new Rect
(464, 125, 119, 117) ];
private final var EQUIP_WINDOW_RECT : Rect = new Rect (500, 0, 70,
100);
private var _equipBooleans : boolean[] = new boolean[4];
…


// C# user:

…
Vector2 _scrollPosition = Vector2.zero;
readonly Rect EQUIP_BOX_RECT = new Rect (237, 67, 360, 207);
readonly Rect EQUIP_WEAPON_RECT = new Rect (237, 280, 360, 157);
readonly Rect EQUIP_STAT_RECT = new Rect (252, 81, 331, 142);
readonly Rect EQUIP_SKILL_RECT = new Rect (460, 121, 127, 125);
readonly Rect [] EQUIP_RECTS = {  new Rect (252, 101, 180, 40), new
Rect (252, 221, 180, 40), new Rect (464, 125, 119, 117)};
bool[] _equipBooleans = new bool[4];
…
```

5. Then, we create `EquipWindow()`, which will contain the script that is used to create the **EQUIPMENT** window's UI. In this function, we also check whether each equipment label is clickable or not. Here, we use the `switch-case` statement again to check which state we are in when the player clicks on each equipment label in the box at the top of the screen. It will basically show the correct equipments or skills in the box at the bottom of the screen, which also allows the player to select a new equipment or skill. So, let's type the following code after the `ItemWindow()` function:

```
// Unity JavaScript user:

…
private function ItemWindow() : void {
  …
}
private function EquipWindow() : void {
  GUI.Box (EQUIP_BOX_RECT, "");
  GUI.Box (EQUIP_WEAPON_RECT, "");
```

```
  GUI.DrawTexture(EQUIP_STAT_RECT, statBox2Texture);
  GUI.DrawTexture(EQUIP_SKILL_RECT, skillBoxTexture);
  var equipContent : GUIContent[] = [weapons.guiContent,
armors.guiContent, accessories.guiContent, skills.guiContent];
  for (var i : int = 0; i < _equipBooleans.Length; i++) {
    if (_equipBooleans[i] == true) {
          GUI.Label(EQUIP_RECTS[i], equipContent[i], "Disabled Click");
          switch (i) {
        case 0:  weapons.SetupScrollBar();  break;
        case 1:  armors.SetupScrollBar();    break;
        case 2:  accessories.SetupScrollBar();  break;
        case 3:  skills.SetupSkillScrollBar();  break;
      }
    } else {
        if (GUI.Button(EQUIP_RECTS[i], equipContent[i], "Selected
Item")) {
        _equipBooleans[i] = true;
        for (var j : int = 0; j < _equipBooleans.Length; j++) {
             if (i != j) {_equipBooleans[j] = false; }
         }
      }
    }
  }
}


// C# user:

…
void ItemWindow() {
  …
}
void EquipWindow() {
  GUI.Box (EQUIP_BOX_RECT, "");
  GUI.Box (EQUIP_WEAPON_RECT, "");
  GUI.DrawTexture(EQUIP_STAT_RECT, statBox2Texture);
  GUI.DrawTexture(EQUIP_SKILL_RECT, skillBoxTexture);
  GUIContent[] equipContent = {weapons.guiContent, armors.guiContent,
accessories.guiContent, skills.guiContent};
  for (int i = 0; i < _equipBooleans.Length; i++) {
    if (_equipBooleans[i] == true) {
          GUI.Label(EQUIP_RECTS[i], equipContent[i], "Disabled Click");
          switch (i) {
        case 0:  weapons.SetupScrollBar();  break;
        case 1:  armors.SetupScrollBar();    break;
        case 2:  accessories.SetupScrollBar();  break;
        case 3:  skills.SetupSkillScrollBar();  break;
      }
    } else {
        if (GUI.Button(EQUIP_RECTS[i], equipContent[i], "Selected
Item")) {
        _equipBooleans[i] = true;
        for (int j = 0; j < _equipBooleans.Length; j++) {
          if (i != j) { _equipBooleans[j] = false; }
        }
      }
```

```
    }
  }
}
```

6. Then, we go back to the `DoMyWindow()` function and add the following highlighted code in the `switch-case` statement to inform the **EQUIPMENT** window when the user clicks on the **EQUIPMENT** tab:

```
// Unity JavaScript user:

private function DoMyWindow (windowID : int) : void {
  …
  switch (_currentTool) {
    …
  case TAB.INVENTORY : //Items
     …
     break;
     case TAB.EQUIPMENT : //Equip
     EquipWindow();
     break;
  }
  …
}


// C# user:

void DoMyWindow (int windowID ) {
  …
  switch (_currentTool) {
    …
  case TAB.INVENTORY : //Items
    …
    break;
        case TAB.EQUIPMENT : //Equip
    EquipWindow();
    break;
  }
…
}
```

7. Now, we need to create three prefabs for weapons, armors, and accessories; so, go back to Unity and create the first `GameObject` (weapon) by going to **GameObject** | **Create Empty** to create the empty game object and name it `Dark Fist`, and add the `item` script to this object by going to its **Inspector** view, and then click on **Add Component** and choose **Scripts** | **Item**.
8. Next, we right-click on this object and choose **Duplicate** twice to create two other objects and name those two objects as `Dark Armor` and `Dark Mask`.
9. Now, we need to put the information in each item that we just created; go to the **Inspector** view of each item and set them as follows:
   - For **Dark Fist**, add the following information:

`Item` **(script)**

Icon	**weapon** (drag the **weapon** texture here)
Info	`This is the dark weapon.`

- For **Dark Armor**, add the following information:

`Item` **(script)**	
Icon	**armor** (drag the **armor** texture here)
Info	`This is the dark armor.`

- For **Dark Mask**, add the following information:

`Item` **(script)**	
Icon	**accessory** (drag the **accessory** texture here)
Info	`This is the dark accessory.`

Then, we drag all three objects to the `MenuInRPGGame/Resources/Prefab` folder in the **Project** view to create a prefab of those objects. Then, we will remove all three objects from the **Hierarchy** view by right-clicking on each object and choosing **Delete**.

10. Next, we will click on the **MenuObject** item in the **Hierarchy** view to bring up its **Inspector** view. Then, we set up the following properties:
    - For weapons, set up the following properties:

Items	
Size	`1`
Element 0	**Dark Fist** (drag from the **Project** view)

    - For armors, set up the following properties:

Items	
Size	`1`
Element 0	**Dark Armor** (drag from the **Project** view)

    - For accessories, set up the following properties:

Items	
Size	`1`
Element 0	**Dark Mask** (drag from the **Project** view)

- For skills, set up the following properties:

Textures	
Size	4
Element 0	**skill1** (drag from the **Project** view)
Element 1	**skill2** (drag from the **Project** view)
Element 2	**skill3** (drag from the **Project** view)
Element 3	**skill4** (drag from the **Project** view)

Now, we will finish the last step of the menu; click on **Play**, open the menu window, click on the **EQUIPMENT** tab, and roll over it, and click on the **UNEQUIP** label or the skill box. We will be able to change the character's equipments, as we can see in the following screenshot:

Objective complete – mini debriefing

We just finished the last tab of our menu window. In this step, we created the **EQUIPMENT** button that will bring up the selection window from which the player can choose the type of equipment or skill. It will update the current equipment's status on the **STATUS** tab too.

If we take a look at our `EquipWindow()` function, we will see that we call the `SetupScrollBar()` function in the `ItemsContainer` class similar to what we did in the `ItemWindow()` function, but this time it is the equipment. We know it is the equipped object because we call the `Init()` function and pass the value `isEquipment` to the `Start()` function of the `Menu` script.

For a Unity JavaScript User, the following code can be used:

```
function Init ( isEquipment : boolean ) : void {
  _isEquipment = isEquipment;
  …
}
```

For a C# User, the following code can be used:

```
void Init ( bool isEquipment ) : void {
  _isEquipment = isEquipment;
  …
}
```

Then, in the `SetupScrollBar()` function in the `ItemsContainer` class, we just check that value to return the information and show it on the scroll view correctly.

For a Unity JavaScript user, the following highlighted code can be used:

```
if (itemCount > 1) {
  for (var i : int = 0; i < itemCount; i++) {
    itemsContent[i] = (_isEquipment) ? ((i == 0) ? new GUIContent(UNEQUIP)
: items[i-1].guiContent) : items[i].guiContent;
  }
}  else {
    itemsContent[0] = (isNoItem) ? new GUIContent(NONE) : ((_  isEquipment)
? new GUIContent(UNEQUIP) : items[0].guiContent);
}
```

For a C# user, the following highlighted code can be used:

```
if (itemCount > 1) {
  for (int i = 0; i < itemCount; i++) {
    itemsContent[i] = (_isEquipment) ? ((i == 0) ? new GUIContent(UNEQUIP)
: items[i-1].guiContent) : items[i].guiContent;
  }
}  else {
    itemsContent[0] = (isNoItem) ? new GUIContent(NONE) : ((_isEquipment) ?
new GUIContent(UNEQUIP) : items[0].guiContent);
}
```

Next, we call the `SetupSkillScrollBar()` function in the `SkillsContainer` class, which

has a very similar concept to the `SetupScrollBar()` function in the `ItemsContainer` class, but this time it contains the texture instead of item information, as shown in the following script:

```
itemsContent[i] = (i == 0) ? new GUIContent(_skillBoxTexture) :new
GUIContent(textures[i-1]);
```

In `SetupSkillScrollBar()`, we also set up the `GUI.SelectionGrid` parameter to show the result in a horizontal view instead of a vertical view by passing the `itemCount` item to the `GUI.SelectionGrid()` function, as we can see in the following code:

```
_selectedSkill = GUI.SelectionGrid (newView, _selectedSkill, itemsContent,
itemCount, GUI.skin.GetStyle("Selected Item"));
```

At last, we create the equipment prefabs and add it to our `Menu` script to show all items that the users can equip themselves when a weapon, armor, or accessory is selected.

Tip

We can create and add more weapons, armors, or accessories similar to the **INVENTORY** page by creating a new prefab and adding it to the `items` array in the **Weapons**, **Armors**, or **Accessories** properties in the `Menu` script inspector, as shown in the following screenshot:

Classified intel

We have created a vertical and horizontal scroll view by using `GUI.BeginScrollView()` to begin the scroll view and ended it with `GUI.EndScrollView()`. Basically, we can use this function, which is very convenient to use, when we want to create a scrollable area that contains any type of a `GUI` object, because this function will automatically create a scrollable area from the two `Rect` parameters that we set up.

For example, in order to create a vertical scroll area at position **x**: `0`, **y**: `0`, **Width**: `100` pixels, and **Height**: `40` pixels, which contains three buttons with each button having 40 pixels height, we'd use code similar to the following snippets:

```
// Unity JavaScript user:

var scrollPostion : Vector2 = Vector2.zero;
function OnGUI() {
   scrollPostion = GUI.BeginScrollView(new Rect(0,0,100,40), scrollPostion,
new Rect(0,0,80,120));
   GUI.Button(new Rect(0,0,80,40),"Button 1");
   GUI.Button(new Rect(0,40,80,40),"Button 2");
   GUI.Button(new Rect(0,80,80,40),"Button 3");
   GUI.EndScrollView();
}


// C# user:

Vector2 scrollPostion = Vector2.zero;
void OnGUI() {
  scrollPostion = GUI.BeginScrollView(new Rect(0,0,100,40),scrollPostion,
new Rect(0,0,80,120));
  GUI.Button(new Rect(0,0,80,40),"Button 1");
  GUI.Button(new Rect(0,40,80,40),"Button 2");
  GUI.Button(new Rect(0,80,80,40),"Button 3");
  GUI.EndScrollView();
}
```

From the preceding code, we can see that the `GUI.BeginScrollView()` function returns `Vector2`, which is the vertical and horizontal position of this scroll view. It also takes two `Rect` objects; the first `Rect` object is the area that the player will see, or we can call it a mask area. The second `Rect` object is the area of our content, which is based on the content that we included between the `GUI.BeginScrollView()` and `GUI.EndScrollView()` functions, which are the three lines of `GUI.Button`. We can see more details of this function at the following URL:

http://unity3d.com/support/documentation/ScriptReference/GUI.BeginScrollView.html

The following figure shows how the `GUI.BeginScrollView()` function works in a visual representation:

Masking area
Scrolling area
Contain 3 buttons

Mission accomplished

In this project, we just created a nice menu, which has the features for an RPG menu. This menu can move around the screen, and we can change the equipment of the character, too. We used a `GUI` class, `GUISkin`, and the `OnGUI` function to create this menu. In the `GUI` class, we used `GUI.window` to create our main menu, `GUI.box` to create the background box area, `GUI.DrawTexture` to show our character's graphics, `GUI.Button` to create a button, `GUI.ToolBar` to create a tab button, `GUI.SelectionGrid` to create a list of clickable items, `GUI.BeginScrollView` and `GUI.EndScrollView` to create a scrolling area, and lastly, we used `GUI.Label` to create a text label. We also used `GUIContent` to contain the information of our button or label. Let's take a look at what we learned from this project from the following screenshot:

We can also go back to the **STATUS** tab to see the result when we equip all the equipments, as seen in the following screenshot:

STATUS
INVENTORY
EQUIPMENT
THE
PLAYER
HP 9999/9999
EN 999/999
LV 99
EXP 9999999
NEXT 99999
ATK 999
DEF 999
AGI 999
INT 999
LUC 999
WEAPON
Dark Fist
ARMOR
Dark Armor
ACCESSORY
Dark Mask
SKILLS
SKILL-1

Hotshot challenges

Now, we have a nice menu, but we still have room to improve this menu to work better. So, why don’t you try something to make this menu much more interesting? Try the following challenges:

- Add a new tab called **OPTION** that the player can use to adjust music and volume
- Create more items or any equipment to make the menu much more interesting
- Add the ability to update the character’s graphics when we change the equipment or skill of the character
- Pause the game when we bring up our menu
- Create your own custom UI graphic and use it instead of the one mentioned in this project

Note

The completed package that comes with this project package has included pausing the game while the menu is showing and adding the new item in the **INVENTORY** page when the key is collected.

Chapter 3. Shade Your Hero/Heroine

In the last two projects, we learned how to create a UI using the GUI object as well as a 2D platform game that used the 2D sprite texture to create our 2D character, and we also got to know a bit about the 3D world in the first project. So, in this project, we will be using a full 3D character. We will take a close look at how to apply a material and shader to the model.

In Unity, there are three different types of shaders. First, surface shaders is the best option if we want our shader to be affected by lights and shadows. Next, Vertex and Fragment Shaders will be required when our shader doesn't need to interact with lighting or needs an effect that surface shaders can't handle. Finally, ShaderLab and Fixed Function Shaders use the old hardware that don't support programmable shaders (GLSL and HLSL/CG), which is mostly used for fallback from a fancy shader. Each type has its advantages. However, we will use surface shaders in this project, which is very convenient and easiest to use when we want to deal with the lighting and shadow. (For more details, check out Appendix D, *Shaders and CG/HLSL Programming*.)

Note

Computer Graphics (**CG**) is a high-level shader language developed by NVIDIA in close collaboration with Microsoft to programme the vertex and pixel shader. It is similar to **High Level Shader Language** or **High Level Shading Language** (**HLSL**), which is a proprietary shading language developed by Microsoft for use with the Microsoft Direct3D API. These references are taken from http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html and http://msdn.microsoft.com/en-us/library/bb509635(v=VS.85).aspx.

We will also get an understanding of shader programming in Unity and create custom shaders by using surface shaders. We can then use the Cg shading language or HLSL to write vertex and fragment shaders.

Note

Surface shaders in Unity is basically the repetitive code that makes it easier to write the shaders that interact with lighting than using low-level vertex/pixel shader programs (Vertex and Fragment Shaders), such as lighting calculation. However, we still need to know Cg/HLSL to write surface shaders. For more information, refer to http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaders.html.

When we are writing shaders, we need to deal with the vertex and pixel shader programming. We will need to get the vertex data from our geometry, calculate the data, and pass it out to the pixel level. At the pixel level, we will calculate the color of the geometry, light, and shadow, and then we will get the result. This can be very complex and repetitive, especially when we have to deal with lighting. It can be a nightmare. Unity 3 has come up with a new style of writing the shader program that is shorter and simpler—surface shaders—which helps us to reduce the time to write the repetitive code over and over again. However, we still need to know the basics of Cg/HLSL programming, but we

won’t go too deep into how to create a shader from scratch or how Cg/HLSL works. We will just introduce some functions and variables that are required for our custom shader.

Mission briefing

We will create a basic custom shader and apply this shader to the character model that we already have. That's it. We might say, "Hey! Why is it so short?" Well, it's short to say but it takes a long time to explain the whole concept of writing a shader.

First, we will import the character model to Unity and start applying a build material and take a look at the default shader in Unity.

Then, we will start creating a shader as follows:

- Create the diffuse and bump maps using built-in lighting models
- Apply a custom light model, which will include the ambient color and specular color
- Apply **Half Lambert**, **Toon Ramp**, and **Rim Light**

Lastly, we will apply all of them together to create our custom Cel shading style, as shown in the following screenshot:

Why is it awesome?

When we complete this project, we will be able to write a cool shader effect similar to most AAA games without scratching our head and trying to write per-pixel lighting calculations. By going through this project, we will get a basic understanding of how surface shaders in Unity works. Then, we can adapt, add, or tweak the parameters, or even create a new custom cool lighting model, which can be used to create a more advanced shader in the future such as reflection.

Your Hotshot objectives

As we are not shader programmers, we don’t want to deal with the low-level vertex/pixel shader programs. So, we will just go through the basic structure of how to create a surface shader step by step, as follows:

- Shader programming – Diffuse and Bump (normal) maps
- Shader programming – Ambient and Specular light
- Shader programming – Half Lambert, Rim Light, and Toon Ramp

Mission checklist

Before we start, we will need to get the project folder and assets from this book's website, http://www.packtpub.com/support?nid=8267, which includes the finished package from the first project and the assets that we need to use in this project.

Browse to the preceding URL and download the `Chapter3.zip` package and unzip it. Inside the `Chapter3` folder, there are two unity packages, which are `Chapter3Package.unitypackage` (we will use this package for this project) and `Chapter3Package_Completed.unitypackage` (this is the completed project package).

Shader programming – Diffuse and Bump (normal) maps

In this first step, we will import `Chapter3.unitypackage` (which is already included in the FBX model and textures) and create a shader program, which will include all the properties that we can edit from the **Material Inspector**. We will start by assigning the Diffuse and Bump (normal) maps. Then, we will use the built-in lighting models, `Lambert` and `BlinnPhong`, which are located in the `Lighting.cginc` file inside the Unity application, to see our result.

Note

Lambert or diffuse reflection will cause all closed polygons to reflect light equally in all directions when rendered. This algorithm is named after Johann Heinrich Lambert, who invented it.

Blinn-Phong or Blinn-Phong reflection is a shading model that is a modification of the Phong reflection model and was developed by Jim Blinn.

The **Phong** reflection model is a shading model that includes a model for the reflection of light from surfaces. It also has a compatible method of estimating pixel colors using the interpolation of surface normals across rasterized (or bitmap) polygons. This was developed by Bui Tuong Phong.

The references used are as follows:

- http://www.opengl-redbook.com
- http://en.wikipedia.org/wiki/Lambertian_reflectance
- http://en.wikipedia.org/wiki/Phong_shading
- http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model

We can find `Lighting.cginc` from the following locations:

- **For Windows**: `{unity install path}/Data/CGIncludes/Lighting.cginc`
- **For Mac**: `/Applications/Unity/Unity.app/Contents/CGIncludes/Lighting.cginc`

The following screenshot shows an example of the `Lambert` and `BlinnPhong` lighting models and how they are different from each other. We will see that the Blinn-Phong is shiny, but the Lambert is matte.

Diffuse (Lambert)
Blinn-Phong

Prepare for lift off

Now, we can start the shader programming by implementing the following steps:

1. Let's create a new project named `Shader`, similar to that in the last project, and click on the **Create Project** button, as shown in the following screenshot:

2. Import the assets package by going to **Assets | Import Package | Custom Package…**, choose `Chapter3.unityPackage`, which we downloaded earlier, and then click on the **Import** button in the pop-up window, as shown in the following screenshot:

Wait until it's done, and you will see the **FBX**, **Materials**, and **Textures** folders, as we can see in the following screenshot:

3. Next, double-click on the **Shader** scene, as shown in the preceding screenshot, to open the scene that we will work on in this project. When you double-click on the **Shader** scene, Unity will bring up the pop up and ask whether we want to save the current scene or not, similar to what we saw in the last project. Just click on the **Don't save** button to open up the **Shader** scene.
4. Then, go to the **FBX** folder and click on **Heroine** in this folder to bring up its **Inspector** view. In the **Inspector** view, we will see three tabs: **Model**, **Rig**, and **Animations**. In the **Model** tab, we will make sure that **Scale Factor** is set to `1`. This is to make sure that our model size matches the physics calculation in Unity.
5. Next, in the **Rig** and **Animations** tabs, we set **Animation Type** in the **Rig** tab to **None** and uncheck the **Import Animation** option in the **Animations** tab, as we don't have any animation in this model; so, we don't need to attach it to our model, as we can see in the following screenshot (we will take a look at the **Rig** and **Animations** tabs in the next project):

6. **Scale Factor** can be set to rescale the whole **FBX** file. In Unity, the Physics Engine is scaled as 1 unit equals 1 meter. So, we can set the model scale to match Unity's Physics Engine to get an accurate result when we use the physics calculation.

Tip

Our model height is around 1.7 meters, so it is about 1.7 units in Unity. We can also measure the model by using the cube game object in Unity, which is 1 x 1 x 1 unit, or turn on the grid to measure the model size related to Unity's unit scale:

7. Then, go to the **Textures** folder and click on **Normal** to bring up its **Inspector** view. In the **Inspector** view, change **Texture Type** to **Normal Map**, check **Generate from Grayscale**, set the **Bumpiness** to `0.07`, and click on the **Apply** button, as shown in the following screenshot:

8. We set **Texture Type** to **Normal map**, which can adjust **Bumpiness** and **Filtering** to get the bump effect we want by checking **Create from Grayscale**.

Engage thrusters

Now, we will put the 3D model into our scene and start writing our custom shader. Follow the ensuing steps:

1. First, we drag the **Heroine** model in the **FBX** folder from the **Project** view to the **Hierarchy** view.
2. Next, we will click on the **Heroine** model in the **Hierarchy** view to bring its **Inspector** view up. Then, we will go to the **Inspector** view and set the value of **Y** under **Rotation** to `180`, as shown in the following screenshot:

3. If we go to the material component, we will see **Diffuse** applied to the shader in this material, which has two properties, **Main Color** and **Base (RGB)**. **Main Color** takes the color that we edited and then applies it to our model. **Base (RGB)** takes the texture that is used for our model. Both properties can be edited and adjusted in the Unity editor to get the best look for our model, as shown in the following screenshot:

4. Now, we will start coding by going to **Assets** | **Create** | **Shader** and naming it `MyShader`. Then, we double-click or right-click on it and choose **Sync MonoDevelop Project** to open **MonoDevelop**.

   ### Note

   The **Sync MonoDevelop Project** step might not work if we haven't set **MonoDevelop** as our default editor. (This was discussed in the first project.)

   In Unity 4.x, we can double-click on the shader file and Unity will automatically

open the shader file on **MonoDevelop** for us.

In **MonoDevelop**, you will see the default setup of the shader script, as shown in the following screenshot:

```
MyShader.shader
Shader "Custom/MyShader" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert

        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex);
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

On the other hand, if you create the shader inside **MonoDevelop**, the default setup of the shader script will be different from the preceding screenshot and similar to the following screenshot:

```
MyShader.shader*
Shader "MyShader" {
    Properties {
        _Color ("Main Color", Color) = (1,1,1,1)
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _BumpMap ("Bump (RGB) Illumin (A)", 2D) = "bump" {}
    }
    SubShader {
        UsePass "Self-Illumin/VertexLit/BASE"
        UsePass "Bumped Diffuse/PPL"
    }
    FallBack "Diffuse"
}
```

Default Shader File Create in MonoDevelop

5. Next, go to the first line in the `MyShader.shader` file and modify the existing code as follows:

```
Shader "Custom/MyShader (Toon Rim)" {
```

6. In this line, we change the folder position and name our shader, which will appear in the **Shader** dropdown when we select the **Shader** properties in the object's **Inspector** view.
7. Then, go back to Unity and click on the **Heroine** model in the **Hierarchy** view to bring the **Inspector** view up.
8. In the **Shader** properties in the material component, we will click on the small arrow on the right-hand side to bring up the dropdown and then go to **Custom | MyShader (Toon Rim)**, as shown in the following screenshot:

9. Then, we go back to **MonoDevelop** again, go to the next line of the `MyShader.shader` file, and start modifying the `Properties` section, as follows:

```
Properties {
  _MainTex ("Diffuse (RGBA)", 2D) = "white" {}
  _BumpMap ("Bumpmap", 2D) = "bump" {}
}
```

10. Then, we go to the `SubShader` section to modify and add the following highlighted code:

```
SubShader {
  Tags { "RenderType"="Opaque" }
  LOD 300
  CGPROGRAM
  #pragma surface surf Lambert

  sampler2D _MainTex;
  sampler2D _BumpMap;

  struct Input {
    float2 uv_MainTex;
    float2 uv_BumpMap;
  };
  void surf (Input IN, inout SurfaceOutput o) {
    half4 c = tex2D (_MainTex, IN.uv_MainTex);
    o.Albedo = c.rgb;
    o.Alpha = c.a;
    o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
```

```
    }
    ENDCG
}
```

We just take the bump texture to create a normal map by getting the normals from the texture using `UnpackNormal` and set it to the normal of `SurfaceOutput (o.Normal)`.

Note

Normals, or surface normals, are a technique used to add details on the surface of the object without using more polygons by faking the lighting of bumps and dents.

11. Finally, we go back to Unity and apply the texture to our model. Let's click on the **Heroine** model in the **Hierarchy** view to bring its **Inspector** view up. In the **Inspector** view, we will go to the material component and set the following:
    - **Texture**: Drag-and-drop **Diffuse** in the **Textures** folder from the **Project** view to this thumbnail
    - **Bumpmap**: Drag-and-drop **Normal** in the **Textures** folder from the **Project** view to this thumbnail

    You will see the **Inspector** view, as shown in the following screenshot:

12. Now, click on play and behold the result:

DIFFUSE
DIFFUSE + NORMAL

Objective complete – mini debriefing

Let's take a look at what we did here.

First, we added a new property, `_BumpMap`, which is used to get the surface normal of our character's geometry.

Note

For more details on surface normals, refer to http://en.wikipedia.org/wiki/Normal_(geometry).

Properties can be created using the following syntax:

```
name ("display name", property type) = default value
```

The components of this code snippet are as follows:

- `name`: This is the name of the property inside the shader script
- `display`: This is the name that will be shown in the **Material Inspector**
- `property type`: This is the type of the property that we can use in our shader programming, which can be `Range`, `Color`, `2D`, `Rect`, `Cube`, `Float`, or `Vector`
- `default`: This is the default value of our property

Note

Every time you add new properties in the `Properties` section, you will need to create the same parameter inside `CGPROGRAM` in the `SubShader` section, as shown in the following code:

```
Properties { _BumpMap ("Bumpmap", 2D) = "bump" {} }
SubShader {
  …
  CGPROGRAM
  #pragma surface surf Lambert
  sampler2D _BumpMap;
  …
  ENDCG
}
```

This is to make sure that our properties can be passed and used in the `CGPROGRAM` section.

We can see more details about this and see what each parameter does at http://docs.unity3d.com/Documentation/Components/SL-Properties.html.

Then, we set the **LOD** (**Level of Detail**) for our shader to `300`. The LOD is the setup that will limit our shader to use the maximum graphic details of this `SubShader` section to the number that we set. We used 300 because we have included a bump map in our shader, which is the same number of the Unity built-in setup for the diffuse bump. You can get more information on shader LOD at http://unity3d.com/support/documentation/Components/SL-ShaderLOD.html.

We added the `sampler2D _BumpMap;` line, which is the same property that gets passed from the `Properties` section:

```
_BumpMap ("Bumpmap", 2D) = "bump" {}
```

Note

`sampler2` is basically a two-dimensional texture, which is the type of parameter used in Cg/HLSL. We can get more information about the Cg parameter from https://developer.nvidia.com/content/cg-tutorial-chapter-3-parameters-textures-and-expressions.

Next, we added `float2 uv_BumpMap` in `struct Input {}`, which will be used to calculate the color information from `_BumpMap`. The `uv_BumpMap` parameter is the texture coordinate, which is basically similar to `vector2`.

In the `surf()` function, we have the following:

```
half4 c = tex2D (_MainTex, IN.uv_MainTex);
o.Albedo = c.rgb;
o.Alpha = c.a;
o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
```

Note

The `surf(Input IN, inout SurfaceOutput o)` function is basically the function that will get the input information from `struct Input {}`. Then, we will assign the new parameter to `SurfaceOutput o`. This parameter will get passed and used next in the vertex and pixel processor.

We can get more details on `Input struct` and the default parameter of `SurfaceOutput struct` at http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaders.html.

Talking about `SurfaceOutput struct`, it is the default struct in Unity, which allows us to pass the parameters easily without creating one.

Tip

`SurfaceOutput struct` is the default struct in Unity, which is located in the `Lighting.cginc` file inside the Unity application. We can also create a custom `SurfaceOutput struct` to pass other variables that are not a part of the default struct by creating a new `SurfaceOutputCustom {…}` struct and passing it to the `surf()` function like `surf (Input IN, inout SurfaceOutputCustom o)`.

The `tex2D()` function will return the color value (`Red`, `Green`, `Blue`, `Alpha`) or (`R`,`G`,`B`,`A`) from the sample state, `_MainTex`, and the texture coordinate, `IN.uv_MainTex`, which we will then assign to `o.Albedo` and `o.Alpha`, respectively. The `o.Albedo` parameter will store the color information (RGB) and the `o.Alpha` parameter will store the alpha information.

Note

Albedo, or the reflection coefficient, is defined as the ratio of the reflected radiation from the surface to the incident radiation upon it. It also refers to the diffuse reflectivity or the reflecting power of a surface.

More information can be found at http://en.wikipedia.org/wiki/Albedo.

The next line is to get the normal information, which is the vector that contains the position (*x*, *y*, and *z*). Then, we used the `tex2D()` function to get the color values (R,G,B,A) from the sample state, `_BumpMap`, and the texture coordinates from `IN.uv_BumpTex`. Then, we used the `UnpackNormal()` function to get the normal as the result of the `tex2D()` function.

Classified intel

Talking about shader programming, there are a lot of things to get to know and understand, for example, how the shader works. We will take a look at the basic structure of shader programming in Unity. The following diagram explains how shaders work:

Note

The preceding diagram is from Amir Ebrahimi and Aras Pranckevčius, who presented the Shader Programming course at Unite 2008, and it represents how a shader works in Unity. We can get more information from the following website (warning: this presentation might be difficult to understand, as it showed how to create the shader without using any surface shader and it used the old version of Unity): http://unity3d.com/unite/archive/2008.

Let's get back to the diagram—we can see that the shader file that we are writing works on both the vertex and pixel (fragment) levels. Then, it will show the result to the frame buffer, but what are vertex and pixel shaders? These are the different types of processors in the GPU. First, the vertex processor gets the vertex data, which contains the position and color of each vertex in the 3D model; then, it draws a triangle from these vertices and passes the data to the pixel processor. The pixel processor will get that value and translate it to the per-pixel screen. It is similar to taking vector art from Illustrator or Flash and translating it to pixel art in Photoshop. Then, it interpolates color data to each pixel, as shown in the following diagram:

From the explanation, we know that we need to deal with the vertex and pixel shader programming when we want to write a shader program. For example, if we want to create a shader, we will need to get the vertex data from our geometry, calculate the data, and pass it out to the pixel level. At the pixel level, we will calculate the color of the geometry, light, and shadow, and then we will get the result.

However, this can be very complex and repetitive when we have to handle lighting manually. That’s why Unity created surface shaders so that we don’t have to deal with various types of lightning, rendering, and so on.

If you check out the **ShaderLab** link in Unity, you will see that there are many things to do, but don’t be afraid because we don’t need to understand everything that’s there to create our custom shader. In the next step, we will create the custom lighting models in surface shaders.

Shader programming – Ambient and Specular light

In this step, we will add Ambient and Specular light to our shader script as well as create our custom lighting models.

Note

The custom lighting model is basically the function that will be used to calculate our surface shader, which is the output of the `surf()` function's interaction with the lights.

The `surf()` function is the function that will take any UVs or data we need as input and fill in the output structure `SurfaceOutput` (the predefined structure, such as `Albedo`, `Normal`, `Emission`, `Specular`, `Gloss`, and `Alpha`).

Engage thrusters

Let's get started. Go to **MonoDevelop**, open the **MyShader** file, and go to the `Properties` section and add the highlighted script:

```
Properties {
  _AmbientColor ("Ambient Color", Color) = (0.5, 0.5, 0.5, 1.0)
  _SpecularColor ("Specular Color", Color) = (0.17, 0.42, 0.75, 1.0)
  _Shininess ("Shininess", Range (0.01, 1)) = 0.078125
  _MainTex ("Diffuse (RGBA)", 2D) = "white" {}
  _BumpMap ("Bumpmap", 2D) = "bump" {}
}
```

Next, go to the `SubShader` section and modify it as well as add the following highlighted code:

```
SubShader {
  Tags { "RenderType"="Opaque" }
  LOD 400
  CGPROGRAM
  #pragma surface surf Lambert
  fixed4 _AmbientColor;
  fixed4 _SpecularColor;
  half _Shininess;
  sampler2D _MainTex;
  sampler2D _BumpMap;

  struct Input {
    float2 uv_MainTex;
    float2 uv_BumpMap;
  };

  void surf (Input IN, inout SurfaceOutput o) {
    fixed4 c = tex2D (_MainTex, IN.uv_MainTex);
    o.Albedo = c.rgb * _AmbientColor.rgb;
    o.Alpha = c.a * _AmbientColor.a;
    o.Specular = _Shininess;
    o.Gloss = c.a;
    o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
  }
  ENDCG
}
```

Basically, we just added three properties for Ambient as well as Specular color and shininess. Then, in the `surf()` function, we changed `half4` to `fixed4` to increase the performance by reducing the precision of our texture. We also multiplied `c.rgb` with `_AmbientColor.rgb` and `c.a` with `_AmbientColor.a`. This will add the color value from the `_AmbientColor` properties with our main texture. Then, we passed `_Shininess` to `o.Specular` and `c.a` to `o.Gloss` to control the amount of gloss on our model.

If we open the diffuse texture in Photoshop and go to the alpha channel, we will see the black and white color area, which is used to specify the amount of glossiness on our character. As we can see in the following screenshot, the armor path is white and the

character skin is almost black:

We will add our custom lighting, which will show the Specular and Ambient color similar to 3D Studio Max, which uses the gray color as the base color.

Tip

The good thing about using the gray color as the base color is that we can add more brightness to our main texture such as Specular and Ambient light.

We can go to the end of the `surf()` function, and before `ENDCG`, add the following highlighted code:

```
void surf (Input IN, inout SurfaceOutput o) {
  …
}
```

```
inline fixed4 LightingRampSpecular (SurfaceOutput s, fixed3 lightDir,
fixed3 viewDir, fixed atten) {
  //Ambient
  fixed3 ambient = s.Albedo * _AmbientColor.rgb;
  //Diffuse
  fixed NdotL = saturate(dot (s.Normal, lightDir));
  fixed3 diffuse = s.Albedo * _LightColor0.rgb * NdotL;
  //Specular
  fixed3 h = normalize (lightDir + viewDir);
  float nh = saturate(dot (s.Normal, h));
  float specPower = pow (nh, s.Specular * 128) * s.Gloss;
  fixed3 specular = _LightColor0.rgb * specPower * _SpecularColor.rgb;
  //Result
  fixed4 c;
  c.rgb = (ambient + diffuse + specular) * (atten * 2);
```

```
  c.a = s.Alpha + (_LightColor0.a * _SpecularColor.a * specPower* atten);
  return c;
}

ENDCG
```

Next, go to the `#pragma surface surf Lambert` line and change it to the following code:

```
#pragma surface surf RampSpecular
```

We can now go back to Unity and click on **Play** to see our result with the specular reflection, as shown in the following screenshot:

Objective complete – mini debriefing

In this step, we first added the new properties, `_AmbientColor`, `_SpecularColor`, and `_Shininess`, which will be used to calculate in our custom lighting models function to get the specular reflection.

Next, we increased `LOD` to `400` because we wanted to increase the LOD for our custom lighting model that will calculate the specular lighting.

Next, in the `surf()` function, we changed the first line from `half4 c = tex2D (_MainTex, IN.uv_MainTex);` to `fixed4 c = tex2D (_MainTex, IN.uv_MainTex);` to increase the performance of our shader. As the return value from the `tex2D()` function is the color value (R,G,B,A), which has a range from `0` to `1`, it will be expensive to use `half` or `float`.

Tip

What are the `half` and `fixed` parameters for? When we are writing a shader in Cg/HLSL, there are three types of parameters that we can use: `fixed`, `half`, and `float`. These parameters determine the precision of computations. The parameter `fixed` is low precision (11 bits, should be in the range of -2.0 to +2.0, and has a precision value of 1/256), `half` is medium precision (16 bits, in the range of -60000 to +60000, and 3.3 decimal digits of precision), and `float` is high precision (32 bits and similar to float on the computer architecture).

The reference is taken from http://docs.unity3d.com/Documentation/Components/SL-ShaderPerformance.html.

However, it follows a trend wherein the more precision we have, the more calculation we need. If we use `float` every time for our shader, it will cause the game to slow down. So, if we want to improve the performance of our game, we should use the lowest precision possible while still having enough output. It's the task of the programmer to decide which aspects need to be precise and which do not.

Then, in the `surf()` function, we also multiplied `c.rgb` with `_AmbientColor.rgb` and `c.a` with `_AmbientColor.a`, which will get passed to `o.Albedo` and `o.Alpha`, respectively. This will add the color value from the `_AmbientColor` properties with our main texture. Then, we passed `_Shininess` to `o.Specular` and `c.a` to `o.Gloss` to control the amount of gloss on our model.

Next, we created our custom lighting function, which is `inline half4 LightingRampSpecular (SurfaceOutput s, half3 lightDir, half3 viewDir, half atten)`. This function passes four parameters, surface output, light direction, view direction, and light attenuation, which we will use to calculate the output for our shader. Then, we changed `#pragma surface surf` from `Lambert` to `RampSpecular`, which means that we changed our lighting calculated from the built-in `Lambert` model to `RampSpecular` in our custom lighting function, `LightingRampSpecular`.

Note

Why is the name of this function not `RampSpecular`? First, we call this function by using `#pragma surface surf RampSpecular`, but to have this function working properly, we need to add `Lighting` as a prefix to the name of our custom lighting function so that Unity will know that this function is a custom lighting function. This is how surface shaders are set up in Unity. You can find out more detail on this from http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaderLighting.html.

In the `LightingRampSpecular()` function, we first got the ambient color value by getting `s.Albedo`, which is the `o.Albedo` parameter from the `surf()` function, and then multiplied `s.Albedo` by `_AmbientColor.rgb`, where `_AmbientColor` is the color information from the `Properties` section at the beginning of our code.

Note

The `fixed`, `half`, and `float` parameters in Cg/HLSL can contain one, two, three, or four values of floating numbers such as `1.0`, `(1.0, 1.0)`, `(1.0, 1.0, 1.0)` or `(1.0, 1.0, 1.0, 1.0)` by calling it `fixed`, `fixed2`, `fixed3`, `fixed4`; `half`, `half2`, `half3`, `half4`; or `float`, `float2`, `float3`, `float4`, respectively. We can also access the value in these parameters by using `(x, y, z, w)` or `(r, g, b, a)`. For example, if you have `fixed4 color = (1.0, 0.5, 0.3, 0.8);` and you want to create another parameter that will contain only three values `(1.0, 0.5, 0.3)` from `fixed4 color`, you can name it like the following: `fixed3 newColor = color.rgb;`. However, if you want the `newColor` value equal to `(0.5, 1.0, 0.3)`, you can name it `fixed3 newColor = color.grb;`.

Then, we calculated the diffuse color by getting the dot product of the surface normal of the object, `s.Normal`, and we passed `o.Normal` from the `surf()` function and the light direction `fixed NdotL = dot (s.Normal, lightDir);`. Then, we used this value to multiply it with the object diffuse texture, `s.Albedo`, and light color, `_LightColor0.rgb`, which is similar to the `Lambert` model.

Next, we calculated the specular color by first getting the normalize vector of the light direction and view direction using the following line of code:

```
fixed3 h = normalize (lightDir + viewDir);
```

In `float nh = saturate(dot (s.Normal, h));`, we calculated the dot product of the surface normal and normalize vector and made sure that the returned number isn't greater than `1` or lower than `0` by using `saturate()`.

Then, we used `nh` to calculate the specular power by powering it with the `s.Specular` and `s.Gloss` properties using the following line of code:

```
float specPower = pow (nh, s.Specular * 128) * s.Gloss;
```

Next, we got the specular color by multiplying the light color, specular power, and the specular color properties using the following line of code:

```
_LightColor0.rgb * specPower * _ SpecularColor.rgb;
```

This is similar to the Blinn-Phong model.

In the last step, we added ambient, diffuse, and specular together and doubled the lighting attenuation value to get a smooth specular effect:

```
c.rgb = (ambient + diffuse + specular) * (atten * 2);.
```

This way, we get the result that the default color is gray, and then we can adjust the color from white to black the way we want as the following screenshot shows (this shader is similar to the standard materials in 3D Studio Max):

The major part of the code is in the Cg/HLSL language, so you might not be familiar with it. However, you can still get an idea of how it works by trying to see more examples and taking a look at the Cg/HLSL language at https://developer.nvidia.com/content/cg-tutorial-appendix-e-cg-standard-library-functions.

We can also see an example of the custom lighting model from http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaderExamples.html.

Classified intel

How exactly do the surface shaders work?

First, we get the parameters from the `Input struct`, and these parameters will get passed to the `SurfaceOutput struct` inside the `surf()` function. Then, the return value of the `SurfaceOutput struct` will go to the lighting model function to calculate both the vertex and pixel (fragment) shaders.

Lastly, the result from the lighting model function will be passed to the frame buffer, as shown in the following diagram:

Shader programming – Half Lambert, Rim Light, and Toon Ramp

In this last step, we will add the last three properties, `_RimColor`, `_RimPower`, and `_Ramp` to get the toon shader result. The `_RimColor` and `_RimPower` properties basically control the back lighting effect of our character. The `_Ramp` properties will be the ramp textures that are used to calculate the lighting effect based on the angle between the light direction and surface normal of the object. We will also get the Half Lambert lighting effect to make a smooth lighting effect.

Engage thrusters

This is the last section, after which you will be able to see the result of your custom shader:

1. Go to **MonoDevelop**, open the `MyShader` file, go to the `Properties` section, and add the highlighted script as follows:

   ```
   Properties {
     _BumpMap ("Bumpmap", 2D) = "bump" {}
     _RimColor ("Rim Color", Color) = (0.07, 0.2, 0.42, 1.0)
     _RimPower ("Rim Power", Range(0.01,8)) = 7
     _Ramp ("Ramp Texture", 2D) = "gray" {}
   }
   ```

2. Go to the `SubShader` section and add the highlighted code as follows:

   ```
   SubShader {
     Tags { "RenderType"="Opaque" }
     LOD 400
     CGPROGRAM
     #pragma surface surf RampSpecular exclude_path:prepass

     fixed4 _AmbientColor;
     fixed4 _SpecularColor;
     half _Shininess;

     sampler2D _MainTex;
     sampler2D _BumpMap;

     fixed4 _RimColor;
     half _RimPower;
     sampler2D _Ramp;

     struct Input {
       float2 uv_MainTex;
       float2 uv_BumpMap;
       fixed3 viewDir;
     };

   }
   ```

   We can see that we put `exclude_path:prepass` after the `#pragma` line. This means that this lighting model will only work on the forward lighting (no deferred lighting). This is because we don't have the angle between the light direction and normal to calculate in the prepass, which is needed for the deferred lighting.

3. Add the following highlighted code inside the `surf()` function to create the Rim Light effect using `Emission`:

   ```
   void surf (Input IN, inout SurfaceOutput o) {
     o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
     fixed rim = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
     o.Emission = (_RimColor.rgb * pow (rim, _RimPower));
   }
   ```

4. Finally, go to the custom lighting function `LightingRampSpecular()`. In this, we will calculate the diffuse color by using the **Half Lambert** or **Warp Lambert** method to get the lighting warp around our model. Then, we get the ramp texture from the property and multiply it with the light color and our main color texture. Let's modify and add the following code:

```
inline fixed4 LightingRampSpecular (SurfaceOutput s, fixed3 lightDir,
fixed3 viewDir, fixed atten) {
  //Ambient
  fixed3 ambient = s.Albedo * _AmbientColor.rgb;
  //Ramp – Diffuse (Half Lambert)
  fixed NdotL = saturate(dot (s.Normal, lightDir));
  fixed halfLambert = NdotL * 0.5 + 0.5;
  fixed3 ramp = tex2D(_Ramp, float2(halfLambert, halfLambert)).rgb;
  fixed3 diffuse = s.Albedo * _LightColor0.rgb * ramp;
  //Specular

}
```

5. Finally, we go back to Unity and apply the ramp texture to our model. Let's click on the `Heroine` model in the **Hierarchy** view to bring up its **Inspector** view. In the **Inspector** view, we will go to the material component in the new property, **Ramp Texture**, and set the following:

   - **Ramp Texture**: Drag-and-drop **Ramp** in the **Textures** folder from the **Project** view to this thumbnail

   After doing this, we will see the **Inspector** view as shown in the following screenshot:

6. Now, we can click on **Play** to see the result, as shown in the following screenshot:

Note

We can also move or rotate our camera to see our character with the shader in a different angle.

Objective complete – mini debriefing

In this section, first we added three properties, `_RimColor`, `_RimPower`, and `_Ramp`, in the `Properties` section, which are used to calculate the Rim Light as well as the Toon Ramp shader styles.

Then, we placed `exclude_path:prepass` after `#pragma surface surf RampSpecular`. This means that we set our shader to compile without the deferred rendering. Why would we want to do this? As our Toon Ramp shader needs the angle data between the light direction and surface normals to calculate the lighting that can't be calculated in the deferred rendering, we excluded it.

Note

In Unity, we can choose three types of rendering paths: Vertex Lit, Forward, and Deferred Lighting. Vertex Lit is basically the lowest lighting quality and doesn't support any real-time shadows. Forward is shader-based, which is the default setting in Unity and only supports real-time shadows from one-directional light. Deferred Lighting is the rendering path with the most lighting and shadow quality, which is available only in Unity Pro with no support for mobile devices. We can get more information about the rendering path from http://docs.unity3d.com/Documentation/Manual/RenderingPaths.html.

Next, we added `half3 viewDir;` in `struct Input {}`, which allows us to get the user view direction vector. This parameter will be used to calculate the specular reflection on our model.

Inside the `surf()` function, we calculated the rim power or the brightness of our backlight, which is `fixed rim = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));`, by using the saturation of the dot product of the view direction normalize and surface normals. In the next line, `o.Emission = (_RimColor.rgb * pow (rim, _ RimPower));`, we multiplied the Rim Light color with the power of the rim power that we got. Then, we assigned the result to `o.Emission` to show the rim light effect on our object.

In the `LightingRampSpecular()` function, we changed the calculation of the lighting by using the Half Lambert model, which makes our object brighter with the light that warps around the object by dividing it by half and adding `0.5`:

```
fixed diff = NdotL * 0.5 + 0.5;
```

Note

Half Lambert lighting is a technique first developed in the original *Half-Life*. It is designed to prevent the rear of an object from losing its shape and looking too flat. Half Lambert is a completely non-physical technique and gives a purely perceived visual enhancement and is an example of a forgiving lighting model.

This reference is taken from http://developer.valvesoftware.com/wiki/Half_Lambert.

We can see the difference between each lighting model in the following diagram:

Next, we used `halfLambert` to calculate the ramp texture, `_Ramp`, to get the color result using the `tex2D()` function, as follows:

```
fixed3 ramp = tex2D (_Ramp, float2(halfLambert, halfLambert)).rgb;
```

Then, we multiplied this value with the diffuse color and light color, as follows:

```
fixed3 rampDiffuse = s.Albedo * _LightColor0.rgb * ramp;
```

We will get a result that is different from the previous section, as shown in the following screenshot:

Mission accomplished

In this project, we learned the basic concepts of shader programming using a surface shader and created the custom lighting model for the shader. Some of you might find shader programming to be very complex with a lot of things to learn; well, yes, that's true! There is no easy way to write the code for shader programming. However, if you want to know more about shader programming, you should definitely learn the Cg/HLSL language, which will help you to understand more about the structure and syntax of the shader language. Now, let's see our result in the following screenshot:

Note

We can also get more detail on shader programming in Appendix D, *Shaders and Cg/HLSL Programming*, and from Unity at the following websites:

- **NVIDIA CG**: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html
- **Unify Community**: http://wiki.unity3d.com/index.php?title=Shaders
- **Unity Shader Reference**: http://docs.unity3d.com/Documentation/Components/SL-Reference.html
- **Unity ShaderLab forum**: http://forum.unity3d.com/forums/16-ShaderLab

Hotshot challenges

We have learned the basic concepts of how to write a custom shader using a surface shader in Unity. Why don’t we try out something to get more familiar with it by playing with the properties to get a different type of rendering style?

- Adjust a value in the material editor in our shader to create a different lighting color and effect
- Create a new ramp texture and apply it to the shader to see the new result of just changing the ramp texture
- Try taking out some properties and using new properties such as cubes
- Try changing some parameters in the custom lighting function by adding a different method to calculate the lighting direction
- Adjust some equations by changing plus to multiply or have more properties to get the different types of rendering techniques
- You can also create your own custom lighting models

Chapter 4. Add Character Control and Animation to Our Hero/Heroine

Here we are in part two of hero/heroine. In this project, we will make our character come to life using the **Mecanim animation system** (the new animation system in Unity 4.x, which allows us to easily set up the complex character animation) and some custom scripts to control our character to walk, jump, and run with smooth transition from one animation to another.

Why Mecanim? The advantage of the Mecanim animation system in Unity is that we can use the animation from different characters and retarget the animation. This is a very convenient way to reuse the same animation on a different character.

We will learn the basics behind the Mecanim animation system and the **Animator Controller** window for our character as well as understand the concept of the third-person controller and camera. This way, we will obtain a good understanding of the new animation system in Unity and can adapt it to use for the specific extra animation later on. The following screenshot shows the state machine of the Mecanim system:

For more details on the Mecanim animation system, we can go to the following Unity website: http://docs.unity3d.com/Documentation/Manual/MecanimAnimationSystem.html.

Note

There is also the old legacy animation system in Unity (which we will not cover in this project), which we can use to animate characters and objects. We can get more details on this from the following Unity websites:

- http://docs.unity3d.com/Documentation/Manual/Animations40.html

- http://docs.unity3d.com/Documentation/Manual/AnimationEditorGuide40.html
- http://docs.unity3d.com/Documentation/Manual/AnimationScripting40.html

Mission briefing

In this project, we will start by setting up the animation clip from the imported FBX model with animation (walk, run, jump, and fall), which is created using other 3D software, such as 3D Studio Max, Maya, Blender3D. Then, we will create **Animator Controller** and set up the Mecanim animation system using the **Animator** view.

Next, we will add **Character Controller**, instead of the Physics component, **Rigidbody**. **Character Controller** will give us the ability to access collision detection as well as the `Move()` function, which is very easy to use.

Note

We will use the `Move()` function to move our character while playing the animation. This function can be accessed from the `CharacterController` class when we add the **Character Controller** component to our game object. The `Move()` function will return `CollisionFlags bitmask`, which will tell us which part of our character hits other collider objects.

Then, we will create the third-person character controller script to control our character's walk, run, and jump by using the `Move()` function in the `Character Controller` class. In this step, we will also create the `OnAnimatorMove()` function to tell Mecanim to play the correct animation clip. Next, we will create the camera script to follow our character. We will then attach the script to our character and make it move on the level.

Why is it awesome?

After we complete this project, we will know how to set up the animation clip from an FBX file. We will also be able to set up the rig using the Humanoid animation type for our character as well as set the animation clip to use in our project. We will be able to create a custom controller script to control our character in the 3D world and blend the animation from idle to walk, walk to run and jump, and so on by using a state machine in the Mecanim animation system. We will also learn how to create a third-person camera to follow our character.

This project will give you an understanding of how to create the third-person character control script and the basic setup of the Mecanim animation system. By the end, you will be able to apply this technique and use it for other characters.

Your Hotshot objectives

Unity is already provided with a built-in third-person controller script, but it only works on the old legacy animation system. For the new Mecanim, we will need to create a new third-person controller. To do this, we need to perform the following:

- Setting up character animation and level
- Creating an animator controller
- Creating a character control script
- Creating a third-person camera to follow our character

Mission checklist

Before we start, we will need to get the project folder and assets from http://www.packtpub.com/support?nid=8267, which includes the executed file from Project 1, *Develop a Sprite and Platform Game*, and the assets that we need to use in this project.

Browse to the preceding URL, download the `Chapter4.zip` package, and unzip it. Inside the `Chapter4` folder, there are two Unity packages: `Chapter4Package.unitypackage` (we will use this package for this project) and `Chapter4Package_Completed.unitypackage` (this is the complete chapter package).

Setting up character animation and level

In Unity, we can import the FBX format file with the rigging animation and set it up either for one clip per FBX or multiple clips per FBX, however we want.

For more information on how to import the FBX, visit http://docs.unity3d.com/Documentation/Manual/HOWTO-importObject.html.

The concept behind the multiple clips is that we have one file that includes all small clips from walking, running, or jumping. Then, we divide it to each type of animation by telling Unity the range of frames for this animation. For example, if we create a walking cycle animation from frames 1 to 30, we can just tell Unity that we want to use the range of frames from 1 to 30 for the walking animation. This concept is very flexible to adjust and change the animation clip on the fly.

Prepare for lift off

In this section, we will begin by preparing the animation for our character, which is the same model and shader from the last project, but this character will include all the necessary animations that we need for this section. Perform the following steps:

1. Create a new project with the name `CharacterAnimation`, as shown in the following screenshot:

2. To import the assets package, go to **Assets** | **Import Package** | **Custom Package…**, choose **Chapter4.unityPackage**, which we downloaded earlier, and then click on the **Import** button in the pop-up window, as shown in the following screenshot:

3. Wait until it has completed importing the package, and you will see the **Chapter4** and **Resources** folders in the Window view. Go to **Chapter4** | **Scene** | **CharacterAnimation** and double-click to open the scene, as shown in the following screenshot:

Project
Create
Favorites
Assets
Chapter4
Animation
Scene
Scripts
Resources
FBX
Materials
Materials
Prefabs
Shader
Textures
Assets ▸ Chapter4 ▸ Scene
CharacterAnimation

Engage thrusters

Now, we are ready to start this section:

1. Let's go to **Resources | FBX**; click on **Heroine_animate** to bring up the **Inspector** view, as shown in the following screenshot:

   There are two ways to import the animations to use in Unity. The first method is the one that we currently use. We import a single model that contains all animations and split the animation by setting the duration of the frame. In the second method, we don't have to set up the animation frame from start to end. Unity will automatically export the animation clip for you. (We will use this method in the next project.)

   However, this method will need to import the multiple model files, each file having a different animation clip such as idle, walk, and run. Also, we need to follow the naming convention for Unity to be able to import the animation clip properly.

   ### Note

   For more details, visit the following website: http://docs.unity3d.com/Documentation/Manual/Splittinganimations.html.

   For example, we have imported the base FBX model named `Heroine_animate` without any animation clips. Then, we will import another FBX model that contains only the idle animation; we should name it `Heroine_animate@idle`, as we can see in the following screenshot:

2. Then, click on the **Rig** tab; for the **Animation Type**, choose **Humanoid** and click on **Apply** button. Now, we will see the **Configure…** button is clickable; click on this button, as shown in the following screenshot:

3. Next, we will see the character with all the bones in the **Scene** view. In the **Inspector** view, we will see that all bones have been placed to each circle area of the green avatar, as we can see in the following screenshot:

4. Here, we want to make sure that our character maps correctly with the T-Pose. So, we go to **Inspector** for all the bones. We will see the drop-down buttons: **Mapping** and **Pose**. At **Mapping**, choose **Automap**, and then at **Pose**, choose **Enforce T-Pose** as shown in the following screenshot:

Usually, Mecanim will automatically map all the bones for us, like the one we just did. However, there might be some cases in which we might have to set up the bones ourselves by dragging the bone from the project view to each circle area in the green avatar.

Note

We can also map the bones by dragging-and-dropping the bone from the **Hierarchy** view to each bone in the **Inspector** view. However, for the retargeting animations, it helps to have the bone setup on the target similar to the previous source model. For more details, go to http://docs.unity3d.com/Documentation/Manual/ConfiguringtheAvatar.html.

5. Next, we will click on the **Done** button to finish the avatar setup and go back to the **Inspector** view.

Note

If we take a look at **Inspector**, we will see the top tabs: **Mapping** and **Muscles**. The **Mapping** tab is basically the one we set up right now. However, we can also set up the **Muscles** tab, which allows us to set up the range of the bone (the minimum and maximum movement of each bone).

For more details, visit http://docs.unity3d.com/Documentation/Manual/MuscleDefinitions.html.

6. Then, we will click on the **Animations** tab to set up the animation clip for our character to use in the game. We will see a small window here with the word **Clips,** as shown in the following screenshot:

7. Here, we will see one clip named **Animation**, which starts from frame **2** and ends at frame **361**. So, we want to rename the **Animation** clip to `idle` by changing the text in the textbox underneath the clip window to `idle`. Then, we can go down and type in `7` in the **Start** input textbox and `210` in the **End** input textbox, as shown in the following screenshot:

8. Next, we will set up the rest of parameters for this idle animation as follows:

Loop Pose	Click on the checkbox

Root Transform Rotation	
Bake into Pose	Click on the checkbox
Root Transform Position (Y)	
Bake into Pose	Click on the check box
Root Transform Position (XZ)	
Bake into Pose	Click on the check box

Then, we click on the **Apply** button and save this setup, as shown in the following screenshot:

In the last setup, we tell this clip to be a looping animation and set all the transforms (rotation, Y position, and XZ position) to use **Baked into Pose**. This will make sure that all the root transforms will be constant and delta root rotation, Y position, and XZ position equal zero. This means that our character, `gameObject`, will not rotate or move at all by `AnimationClip`.

9. Next, we will set up other animation clips. Let's go back to the **Clips** window and click the plus sign four times to create four more animation clips and set them up as follows:

Start	`230`
End	`280`

Loop Pose	Click on the checkbox
Root Transform Rotation	
Bake into Pose	Click on the checkbox
Root Transform Position (Y)	
Bake into Pose	Click on the checkbox
Root Transform Position (XZ)	
Bake into Pose	Click on the checkbox
run	
Start	290
End	320
Loop Pose	Click on the checkbox
Root Transform Rotation	
Bake into Pose	Click on the checkbox
Root Transform Position (Y)	
Bake into Pose	Click on the checkbox
Root Transform Position (XZ)	
Bake into Pose	Click on the checkbox
jump	
Start	325
End	339
Root Transform Rotation	
Bake into Pose	Click on the checkbox
Root Transform Position (Y)	
Bake into Pose	Click on the checkbox
Root Transform Position (XZ)	
Bake into Pose	Click on the checkbox
fall	

Start	340
End	360
Root Transform Rotation	
Bake into Pose	Click on the checkbox
Root Transform Position (Y)	
Bake into Pose	Click on the checkbox
Root Transform Position (XZ)	
Bake into Pose	Click on the checkbox

10. Now we have set up the walk and run clips similar to our idle clip. On the other hand, we set the jump and fall clips a bit differently because we only want to play both clips once and stay at the last frame, so we don't need to check **Loop Pose**:

11. Next, we want to add the level to our scene. Go to the **Project** view under the **Prefabs** folder and drag the **Level** prefab to the **Hierarchy** view, as shown in the following screenshot:

12. Before we finish this step, we will add our character to the scene. Go to the **Resources | FBX** in the **Project** view and drag **Heroine_animate** to the **Hierarchy** view:

13. In the preceding screenshot, we can see the outline, which shows the white boxes on the character that represent extra bones to control the extra objects on our character. In this case, the extra objects are the sword, sword sheath, dragonhead on the shoulder, and the back plate.

    ### Note

    The extra bone meshes are usually exported from other 3D software, depending on the artist or animator who sets it up. Sometimes, we can use these meshes for collision detection for the attack action if we have a fighting animation attached to the character.

14. In this case, we don’t need to show the mesh here. We can either remove or hide it, but we will choose to remove it because we don’t need to use the **Mesh Renderer** component for this project. Let’s do this by clicking on **Heroine_animate** in the **Hierarchy** view. Then, we will use the search box in the **Hierarchy** view to search and put the bones in this box, as shown in the following screenshot:

15. Let’s click on the first bone, **back_plate_bone**, to bring up its inspector and go to the **Inspector** view, right-click on the **Mesh Renderer** component, and then click on **Remove Component** to remove it, as we can see in the following screenshot:

16. Then, we go to the next bones, **shoulder_bones**, **sword_bone** and **sword_sheath_bone**, perform tasks similar to what we performed for **back_plate_bone**, and we will see all the white boxes disappear, as shown in the following screenshot:

Objective complete – mini debriefing

Basically, what we have done here is set up our character to use the **Humanoid** animation type and create the avatar for our character. Then, we set up the animation clip for idle, walk, run, jump, and fall, as well as set up each animation to **Bake into Pose** to make sure that **AnimationClip** won't rotate or move our **gameObject** character.

Then, we also created the level for our scene and added our character, which included the animation clips that we have set up. Lastly, we removed the **Mesh Renderer** of the extra bones because we don't want to show it in our scene.

Note

If we click on **Heroine_animate** in the **Hierarchy** view, we will see that there is the **Animator** component attached in the **Inspector** view. This **Animator** component is the new component that Mecanim uses to control our character, which will contain **Animation Controller** (we will create this in the next step), **Avatar** (which is **Heroine_animateAvatar** that we just created), **Apply Root Motion** (we will talk about this later in the project), **Animate Physics**, and **Culling Mode**.

For more details on each parameter, visit http://docs.unity3d.com/Documentation/Components/class-Animator.html.

Classified intel

As we know, most 3D software uses the *y* axis as the upward direction, and Unity is no exception. However, 3D Studio Max uses the *z* axis as the upward direction.

Tip for 3Ds Max users

As we know, 3D Studio Max uses the *z* axis as the upward direction, but Unity uses the *y* axis for the same purpose. Usually, when we export a 3D model without any animation attached, we set the *x* rotation of the character pivot to -90 degrees. However, if we set the model with animation this way, we will have a problem of the biped having the wrong rotation. The good thing is that we don't need to do anything; just leave the rotation of the pivot as default, as shown in the following screenshot:

This is because our character has two objects attached to it—the model and bones. Then, when we import it to Unity, the FBX Importer in Unity will basically handle it for us by creating the container and add both objects and its children, which will solve the problem of wrong rotation and set the default rotation of the model as **X** to `0`, **Y** to `0`, and **Z** to `0`, as we can see in the following screenshot:

Some of you might be curious—how do we know when to rotate the pivot or not rotate the pivot in 3D Studio Max? Well, it's very simple; just remember that with any 3D model that is static and not complicated or has only one mesh object included, we should rotate its pivot. On the other hand, if we have a character model with rigging, or maybe a simple mesh for detecting the collision, we can just leave it as it is.

Note

We can also fix the rotation of the imported model in Unity by creating the empty object as a parent of the imported model. For more information on how to fix the rotation of the imported model in Unity, we can go to http://docs.unity3d.com/Documentation/Manual/HOWTO-FixZAxisIsUp.html.

The Animations inspector

If we click on **Heroine_animate** in the **Project** view and go to the **Animations** tab in the **Inspector** view, we will see the clip window that contains all the clips that we created. Here, we can select any clip, for example, the idle clip. Then, we can go down to see the ruler bar under the clip's name. We can drag the arrow marker to adjust the **Start** and **End** points of the clip instead of typing in the number in the textbox.

As soon as we drag the marker, we will see the **Looping** fitness curves for all parameters (**Rotation** and **Position**). The green line shows the best loop match followed by yellow and red, as shown in the following screenshot:

Note

This feature is very convenient to adjust the looping animation. For example, if we have the animation clip from the motion capture data, we can easily find a good match for the looping animation.

For more details, visit http://docs.unity3d.com/Documentation/Manual/LoopingAnimationClips.html.

At last, we can use the play button in the **Preview** view to see each animation clip in action and adjust the speed of our clip by dragging the slider bar close to the time icon, before applying it, as shown in the following screenshot:

Creating an animator controller

In this step, we will create the **Animator Controller**, which we will use to control our character animation using the state machine and parameters to create a condition to change the type of animation. This animation will be controlled using the script that we will create in the next step.

Engage thrusters

Let’s get started:

1. Click on **Animation** under the **Chapter4** folder in the **Project** view, right-click and go to **Create | Animator Controller**, and rename it to `MyAnimatorController`.
2. Double-click on **MyAnimatorController** to bring up the **Animator** view, as seen in the following screenshot:

3. Double-click on **Base Layer** in the top-left corner and change the name of this layer to **Base**, as shown in the following screenshot:

4. Next, we go to **Resources | FBX** in the **Project** view, click on the arrow in front of **Heroine_animate**, and drag the idle clip to the **Animator** view, as shown in the following screenshot:

5. Then, we drag four more clips (**walk**, **run**, **jump**, and **fall)** to the **Animator** view, similar to the **idle** clip, and place it in the positions shown in the following screenshot:

6. Now, we will click on each clip and go to the **Inspector** view to set up our parameter. We will start with the **idle** clip; let's go to each clip **Inspector** and set it up as follows:

idle	

Speed	`0.7`	
Foot IK	Check the box	
walk		
Speed	`1.5`	1.5
Foot IK	Check the box:	check the box
run		
Speed	`1.7`	1.7
Foot IK	Check the box	
jump		
Speed	`4`	
Foot IK	Check the box	
fall		
Speed	`0.7`	

We just set the speed of each animation as well as the foot IK, which we will use to control our character.

7. Next, we will add a few parameters to control when each animation will be played. Let's go to the **Parameters** tab at the bottom-left corner of the **Animator** view and click on the plus icon thrice and choose the name and type of parameter as follows:

Name	Type
Speed	`Float`
IsRun	`Bool`
MotionState	`Int`

These parameters will be used in the script to change the state of each animation clip.

8. Next, we will create the transition between each animation state. First, we will start with our base animation, **idle**, and transition to **walk**. Right-click on the **idle** clip and choose **Make Transition**, which will bring up the arrow, and click-and-drag on top

of the **walk** clip. We will see the link from the **idle** clip to the **walk** clip, as shown in the following screenshot:

9. Then, we want to click on the arrow line that we just created, then go to the **Inspector**, uncheck the **Atomic** checkbox, and go to the **Conditions** view; click on the plus icon to add another condition and set up as shown in the following screenshot:

The **Atomic** checkbox ensures that this animation will finish playing before another animation can be played. However, in our case, we want the animation to be interrupted anytime when the character jumps or falls down, which we will set up in the next step.

Note

More details about animation transition can be found at http://docs.unity3d.com/Manual/class-Transition.html.

10. Next, we want to create another transition back from **walk** to the **idle** animation. So, let's right-click on the **walk** clip, choose **Make Transition**, and then click-and-drag on **idle**. Then, we will click on the arrow line again to go to the **Inspector** view and set it up as follows:

Atomic	Uncheck the box

Conditions	**Speed**	**Less**	`0.1`

11. Then, we set up the rest of the animation. This time, we want the transition to go from **walk** to **run**. So, let's right-click on the **walk** clip, choose **Make Transition**, and then click-and-drag on **run**. Then, we will click on the arrow line again to go to the **Inspector** and set it up as follows:

Atomic	Uncheck the box	
Conditions	**IsRun**	**true**

Also, we will create the transition from **run** back to **walk**. Go to the transition inspector and set it up as follows:

Atomic	Uncheck the box	
Conditions	**IsRun**	**false**

12. Next, we want the **run** clip to transition back to **idle**. Let's create the transition arrow from **run** to **idle**, go to **Inspector**, and set it up as follows:

Atomic	Uncheck the box		
Conditions	**Speed**	**Less**	`0.1`

We also need to create the transition from idle to run. Set up its **Inspector** view as follows:

Atomic	Uncheck the box	
Conditions		
Speed	**Greater**	`0.1`
IsRun	**true**	

13. Now, we finished the ground animation setup. Next, we will set up the **jump** and **fall** animations, and we will use the **MotionState** parameter to control each state of the animation. Let's set up the transition from **fall** to **idle**; set **Inspector** as follows:

Atomic	Uncheck the box	
Conditions		
Speed	**Less**	`0.1`
MotionState	**Equals**	`0`

Also, we need to create the transition from **idle** to **fall**; set **Inspector** as follows:

Atomic	Uncheck the box		
Conditions	**MotionState**	**Equals**	2

14. Next, we will create another transition from **fall** to **walk**; set **Inspector** as follows:

Atomic	Uncheck the box	
Conditions		
Speed	**Greater**	0.1
IsRun	**false**	
MotionState	**Equals**	0

Now, the transition from **walk** to **fall**; set **Inspector** as follows:

Atomic	Uncheck the box		
Conditions	**MotionState**	**Equals**	2

15. Then, the last transition from the **fall** clip. We will set it from **fall** to **run**; set **Inspector** as follows:

Atomic	Uncheck the box	
Conditions		
Speed	**Greater**	0.1
IsRun	**true**	
MotionState	**Equals**	0

Now, the transition from **run** to **fall**; set **Inspector** as follows:

Atomic	Uncheck the box		
Conditions	**MotionState**	**Equals**	2

The current **Animator** view will look like the following screenshot:

16. Next, we will set up the **jump** state animation. As we already know, we want our character to be able to jump from any state when the character hits the ground. So, we will do this by creating the transition from the **Any State** clip (the green color box) to **jump** by right-clicking on **Any State** and clicking-and-dragging again on **jump**. Then, we will set **Inspector** as follows:

Atomic	Leave it checked		
Conditions	**MotionState**	**Equals**	1

We can see here that this time, we leave the **Atomic** box checked, which means that our transition to the **jump** state won't get interrupted by any other animation. We also need to edit the curve and tweak the blending time to `0` to make the transition from **Any State** instantly change to the **jump** animation when the character jumps, as shown in the following screenshot:

We can zoom in/out the curve editor by scrolling the middle mouse button or pan by clicking the middle mouse button and moving the cursor.

17. Lastly, we will need the transition from **jump** to **fall**. We have set it this way because we don't know how high the character will jump; so, we want to show the **jump** animation clip until the character starts falling. Simultaneously, we switch to the **fall** animation clip. Let's right-click on **jump** and create the transition to the **fall** clip and set **Inspector** as follows:

Atomic	Leave it checked		
Conditions	**MotionState**	**Equals**	2

Now, we have finished setting up the **Animator Controller** in the **Animator** view, which will look like the following screenshot:

18. Before we finish this section, we need to add **MyAnimatorController** (which we just created) to our character. Let's click on the **Heroine_animate** object in the **Hierarchy** view and bring up the inspector. Go to **Controller** and put **MyAnimatorController**, as shown in the following screenshot:

If we click on the play button to play the game while opening the **Animator** window, we can change the condition parameter and see the animation changing in action:

Objective complete – mini debriefing

In this step, we have created the **Animator Controller**, which we used to control the animation state of our character. First, we named the layer to `Base`. Then, we dragged all the animation clips and linked them with the transition. In the transition, we set up three parameters to give the condition for each changing state, which are **Speed**, **IsRun**, and **MotionState**.

The **Speed** parameter is a condition to check whether the character is moving, which uses the `float` object. **IsRun** is a `boolean` object, which is used to check whether the player hit the key to run (in this case, we will use the *Shift* key to toggle it in the next step). The last parameter, **MotionState**, is the integer that is used to check which state the character is in. In this case, we use `0 = Ground`, `1 = Jump`, `2 = Fall`, and `3 = Jump Hold` (this state will hold the jump animation to the last frame).

At last, we have put the **Animator Controller** to our character in the scene.

Classified intel

From the transition between each clip, we will see that there is the **Exit Time** parameter in the **Conditions** view that is always set to default. However, we never use it in this section. What is it and how do we use it?

The **Exit Time** parameter is basically to tell the animation to change to the other state by the percentage of the current animation. For example, if we set it to `0.9`, the next animation state will trigger when the current animation has already played 90 percent through. We can also use the curve editor to set and see the matching of both clips. Have a look at the following screenshot:

For more details, visit http://docs.unity3d.com/Documentation/Manual/AnimationTransitions.html.

In **Transitions**, we can also add more than one transition between each transition, which we can sometimes use to check some conditions such as direction greater than 1 and direction less than -1. We will see the three arrows on the transition line as shown in the following screenshot:

In this case, it might be difficult to see which transition is on. We can use the **Solo** and **Mute** toggles to see each transition while we are playing the game. This is very useful to debug when we have many transitions on the same state.

For more details on **Solo** and **Mute**, visit http://docs.unity3d.com/Manual/class-Transition.html.

Creating a character control script

In this step, we will create a script that controls our character's movements such as forward and backward movements, turning left or right, and other actions such as jumping and falling by using the **CharacterController** component and the `OnAnimatorMove()` function.

Tip

We can also use the **Rigidbody** and **CapsuleCollider** components instead of **CharacterController**. However, in this book, we want to show the example of using **CharacterController** because there are many convenient functions to use to control the character such as `Move()`, `collisionFlags` (checking where the character is collided), and `slopeLimit` (the character can move up on the slope by a limited degree). If we use **Rigidbody** and **CapsuleCollider**, we have to create those functions ourselves.

Engage thrusters

Now, we will create the script to control our character. Perform the following steps:

1. We will create the **CharacterControl** script that will control our entire menu; go to **Assets** | **Create** | **Javascript** (for Unity JavaScript users) or go to **Assets** | **Create** | **C#** (for C# user), name it `ChracterControl`, double-click on it to launch **MonoDevelop**, and we will get our hands dirty with the code.
2. Open the **ChracterControl** script file and type the following variables:

```
// Unity JavaScript user:

#pragma strict
@script RequireComponent(Animator)
@script RequireComponent(CharacterController)
public enum MOTION_STATE {GROUND,JUMP,FALL,JUMP_HOLD}
private final var GRAVITY : float = 20.0f;
private final var MIN_AIR_TIME : float = 0.15f; // 0.15 sec.
var rotationSpeed : float = 1.0f;
var walkSpeed : float = 2.0f;
var runSpeed : float = 5.0f;
var jumpSpeed : float = 8.0f;
private var _animator : Animator;
private var _hDirection : float;
private var _vDirection : float;
private var _moveDirection : Vector3;
private var _movement : Vector3;
private var _isMoveBack : boolean;
private var _isRun : boolean;
private var _isJumping : boolean;

private var _motionState : MOTION_STATE;
private var _baseCurrentState : AnimatorStateInfo;
private var _characterController : CharacterController;
private var _moveSpeed : float;
private var _verticalSpeed : float;
private var _inAirTime : float;
private var _inAirStartTime : float;
public function get IsMoveBackward () : boolean {
  return _isMoveBack;
}
public function get  IsGrounded () : boolean {
  return _characterController.isGrounded;
}
public function get  IsFall () : boolean {
  return (_inAirTime > MIN_AIR_TIME);
}

// C# user:

using UnityEngine;
using System.Collections;
[RequireComponent(typeof(Animator))]
[RequireComponent(typeof(CharacterController))]
```

```
public class CharacterControl : MonoBehaviour {
  public enum MOTION_STATE {GROUND,JUMP,FALL,JUMP_HOLD}
  const float GRAVITY = 20.0f;
  const float MIN_AIR_TIME = 0.15f; // 0.15 sec.
  public float rotationSpeed = 1.0f;
  public float walkSpeed = 2.0f;
  public float runSpeed = 5.0f;
  public float jumpSpeed = 8.0f;
  Animator _animator;
  float _hDirection;
  float _vDirection;
  Vector3 _moveDirection;
  Vector3 _movement;
  bool _isMoveBack;
  bool _isRun;
  bool _isJumping;
  MOTION_STATE _motionState;
  AnimatorStateInfo _baseCurrentState;
  CharacterController _characterController;
  float _moveSpeed;
  float _verticalSpeed;
  float _inAirTime;
  float _inAirStartTime;
  public bool IsMoveBackward {
    get { return _isMoveBack; }
  }
  public bool IsGrounded {
    get { return _characterController.isGrounded; }
  }
  public bool IsFall {
    get { return (_inAirTime > MIN_AIR_TIME); }
  }
  …
}
```

Here, we have all the necessary parameters to use in our script. In the first line, we want to make sure that we have the **CharacterController** and **Animator** components attached when we use this script. Then, we have the `enum` variable, which we will use to set the state of the animation clip. (The enums are integer values that start from 0,1,2….) Next, we have the animation speed to control how fast the character rotation and movement should be.

We have the `GRAVITY` property because we will use it to calculate the `Move()` function in the `CharacterController` class. In the last three functions, we basically have the `get` function to return if our character is moving backward, on the ground, or is falling. For the `IsFall()` function, we basically check whether our character stays in the air longer than `MIN_AIR_TIME`.

3. Next, we will start creating the first function, `Awake()`, using the following code to get the `_animator` and `_characterController` components:

```
// Unity JavaScript user:

function Awake () {
```

```
  _animator = GetComponent.<Animator>();
  _characterController = GetComponent.<CharacterController>();
}


// C# user:

void Awake () {
  _animator = GetComponent<Animator>();
  _characterController = GetComponent<CharacterController>();
}
```

4. Then, we create the `Start()` function and set up all the variables as follows:

```
// Unity JavaScript user:

function Start () {
  _moveDirection = Vector3.zero;
  _motionState = MOTION_STATE.GROUND;
  _isRun = false;
  _isMoveBack = false;
  _moveSpeed = 0.0f;
  _verticalSpeed = 0.0f;
  _inAirTime = 0.0f;
  _inAirStartTime = Time.time;
}


// C# user:

void Start () {
  _moveDirection = Vector3.zero;
  _motionState = MOTION_STATE.GROUND;
  _isRun = false;
  _isMoveBack = false;
  _moveSpeed = 0.0f;
  _verticalSpeed = 0.0f;
  _inAirTime = 0.0f;
  _inAirStartTime = Time.time;
}
```

5. Next, we will create the `Update()` function. In this function, we will calculate all movements for our character. First, we will use `input` and `_targetDirection` by using the `main camera` transform to calculate `_moveDirection` as follows:

```
// Unity JavaScript user:

function Update () {
  _hDirection = Input.GetAxis("Horizontal");
  _vDirection = Input.GetAxis("Vertical");
  var cameraTransform : Transform = Camera.main.transform;
  var _forward : Vector3 =
cameraTransform.TransformDirection(Vector3.forward);
  _forward.y = 0f;
  var _right : Vector3 = new Vector3(_forward.z, 0f, -_forward.x);
  if (_vDirection < 0) { _isMoveBack = true; }
```

```
  else { _isMoveBack = false; }
  var _targetDirection : Vector3 = (_hDirection * _right) +
(_vDirection * _forward);
  if (_targetDirection != Vector3.zero) {
    _moveDirection = Vector3.Slerp(_moveDirection, _targetDirection,
rotationSpeed * Time.deltaTime);
    _moveDirection = _moveDirection.normalized;
  } else {
    _moveDirection = Vector3.zero;
  }
}
```

```
// C# user:
void Update () {
  _hDirection = Input.GetAxis("Horizontal");
  _vDirection = Input.GetAxis("Vertical");
  Transform cameraTransform = Camera.main.transform;
  Vector3 _forward =
cameraTransform.TransformDirection(Vector3.forward);
  _forward.y = 0f;
  Vector3 _right = new Vector3(_forward.z, 0f, -_forward.x);
  if (_vDirection < 0) { _isMoveBack = true; }
  else { _isMoveBack = false; }
  Vector3 _targetDirection = (_hDirection * _right) + (_vDirection *
_forward);
  if (_targetDirection != Vector3.zero) {
    _moveDirection = Vector3.Slerp(_moveDirection, _targetDirection,
rotationSpeed * Time.deltaTime);
    _moveDirection = _moveDirection.normalized;
  } else {
    _moveDirection = Vector3.zero;
  }
}
```

Note

`Vector3.Slerp()` is the function that we can use to interpolate between two vectors spherically by the amount of time, and the return vector's magnitude will be the difference between the magnitudes of the first vector and the second vector. This function is usually used when we want to get the smooth rotation from one vector to another vector in a fixed amount of time. You can see more details at http://docs.unity3d.com/Documentation/ScriptReference/Vector3.Slerp.html.

6. We are still in the `Update()` function. Next, we will calculate `_verticalSpeed` and `_inAirTime` by checking whether our character is grounded. Let's continue from the previous step and put the highlighted code as follows:

```
// Unity JavaScript user:

function Update () {
  …
  } else {
    _moveDirection = Vector3.zero;
  }
```

```
  if (IsGrounded) {
    _isJumping = false;
    _verticalSpeed = 0.0f;
    _inAirTime = 0.0f;
    _inAirStartTime = Time.time;
  } else {
    _verticalSpeed -= GRAVITY * Time.deltaTime;
    _inAirTime = Time.time - _inAirStartTime;
  }
}


// C# user:
void Update () {
  …
  } else {
    _moveDirection = Vector3.zero;
  }
  if (IsGrounded) {
    _isJumping = false;
    _verticalSpeed = 0.0f;
    _inAirTime = 0.0f;
    _inAirStartTime = Time.time;
  } else {
    _verticalSpeed -= GRAVITY * Time.deltaTime;
    _inAirTime = Time.time - _inAirStartTime;
  }
}
```

7. Next, if our character is able to jump, we will check for the jump action and set `_verticalSpeed` to `jumpSpeed`. We also check that if the user holds *Shift*, `_isRun` is set to `true`. Then, we set `_moveSpeed` depending on the character's action (run or walk). Let's put the highlighted code as follows:

```
// Unity JavaScript user:

function Update () {
  …
  } else {
    _verticalSpeed -= GRAVITY * Time.deltaTime;
    _inAirTime = Time.time - _inAirStartTime;
  }
  if (!_isJumping && (_motionState == MOTION_STATE.GROUND)) {
    _isRun = (Input.GetKey (KeyCode.LeftShift) || Input.GetKey
(KeyCode.RightShift));
    _moveSpeed = (_isRun) ? runSpeed : walkSpeed;
    if (Input.GetButtonDown ("Jump")) {
      _verticalSpeed = jumpSpeed;
         _isJumping = true;
      _inAirTime = 0.0f;
      _inAirStartTime = Time.time;
    }
  }
}
```

```
C# user:
void Update () {
  …
  } else {
    _verticalSpeed -= GRAVITY * Time.deltaTime;
    _inAirTime = Time.time - _inAirStartTime;
  }
if (!_isJumping && (_motionState == MOTION_STATE.GROUND)) {
    _isRun = (Input.GetKey (KeyCode.LeftShift) || Input.GetKey
(KeyCode.RightShift));
    _moveSpeed = (_isRun) ? runSpeed : walkSpeed;
    if (Input.GetButtonDown ("Jump")) {
      _verticalSpeed = jumpSpeed;
        _isJumping = true;
      _inAirTime = 0.0f;
      _inAirStartTime = Time.time;
    }
  }
}
```

8. The last section of code that we will add to the `Update()` function is to control our character's movement and rotation, which is a very important part to create our character when we press the button to control it. Let's add the following highlighted code in continuation with the last step:

```
// Unity JavaScript user:

function Update () {
  …
      _inAirStartTime = Time.time;
    }
  }
  _movement = (_moveDirection * _moveSpeed) + new Vector3 (0,
_verticalSpeed, 0);
  _movement *= Time.deltaTime;

  if (_movement != Vector3.zero) {
    _characterController.Move(_movement);
  }
  if (_moveDirection != Vector3.zero) {
    transform.rotation = Quaternion.LookRotation(_moveDirection);
  }
}

// C# user:

void Update () {
      …
      _inAirStartTime = Time.time;
    }
  }
  _movement = (_moveDirection * _moveSpeed) + new Vector3 (0,
_verticalSpeed, 0);
  _movement *= Time.deltaTime;
```

```
  if (_movement != Vector3.zero) {
    _characterController.Move(_movement);
  }

  if (_moveDirection != Vector3.zero) {
    transform.rotation = Quaternion.LookRotation(_moveDirection);
  }
}
```

9. Now, in the last function of this **CharacterControl** script, we will add the `OnAnimationMove()` function to control the state of our animation. So, let's add the following code:

```
// Unity JavaScript user:

function OnAnimatorMove (){
  if (_animator) {
    if (_isJumping) {
      if (IsGrounded) {
        _motionState = MOTION_STATE.FALL;
      } else {
        if (_motionState == MOTION_STATE.GROUND) {
          _motionState = MOTION_STATE.JUMP;
        } else if (_motionState == MOTION_STATE.JUMP) {
          _motionState = MOTION_STATE.JUMP_HOLD;
        }
      }
    } else {
      if (IsFall) {
        _motionState = MOTION_STATE.FALL;
      } else {
        _motionState = MOTION_STATE.GROUND;
      }
    }
    var velocity: Vector3 = new Vector3(_hDirection,0.0f,_vDirection);
    velocity.y = 0.0f;
    _animator.SetFloat ("Speed", velocity.sqrMagnitude);
    _animator.SetBool ("IsRun", _isRun);
    _animator.SetInteger ("MotionState", parseInt(_motionState));
  }
}

// C# user:
void OnAnimatorMove (){
  if (_animator) {
    if (_isJumping) {
      if (IsGrounded) {
        _motionState = MOTION_STATE.FALL;
      } else {
        if (_motionState == MOTION_STATE.GROUND) {
            _motionState = MOTION_STATE.JUMP;
        } else if (_motionState == MOTION_STATE.JUMP) {
          _motionState = MOTION_STATE.JUMP_HOLD;
        }
      }
```

```
    } else {
      if (IsFall) {
        _motionState = MOTION_STATE.FALL;
      } else {
        _motionState = MOTION_STATE.GROUND;
      }
    }
    Vector3 velocity = new Vector3(_hDirection,0.0f,_vDirection);
    velocity.y = 0.0f;
    _animator.SetFloat ("Speed", velocity.sqrMagnitude);
    _animator.SetBool ("IsRun", _isRun);
    _animator.SetInteger ("MotionState", (int)_motionState);
  }
}
```

Note

The `OnAnimatorMove()` function is the callback function that will be called each frame after the state machines and the animations have been evaluated. We can say that this is very similar to the `FixedUpdate()` function. However, the `OnAnimatorMove()` function will make sure that all the parameters will be got and set correctly before we do something.

For more details, visit http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnAnimato

10. Lastly, we will go back to Unity editor; drag-and-drop the **CharacterControl** script that we just created on **Heroine_animate** in the **Hierarchy** view. Then, we click on the **Heroine_animate** object in the **Hierarchy** view and go to its **Inspector** view and set it up as follows:

Character Controller	
Skin Width	`0.01`
Center	**X**: `0`, **Y**: `0.85`, and **Z**: `0`
Radius	`0.2`
Height	`1.7`

Click on play and now our character can move, jump, and run.

Objective complete – mini debriefing

In this section, we created a script called the **CharacterControl** script that we can use to control the character's movement by using the `Move()` function in the **CharacterController** component. This function needs the `Vector3` direction passed in and returns **CollisionFlags**, which are very convenient to track the position of the character that has been collided.

Note

We can get **CollisionFlags** by accessing the **CharacterController** component, which in our case, can be done using `_characterController.collisionFlags`. For example, we can use this flag to check if the character only hits the ceiling by using the following expression:

```
If (_characterController.collisionFlags == CollisionFlags.Above) {…}
```

For more detail, visit http://docs.unity3d.com/Documentation/ScriptReference/CollisionFlags.None.html.

We also applied `GRAVITY` while the character falls down from the jump or from the platform where there is no collider. Also, we used the `OnAnimatorMove()` function to control the animation state.

Classified intel

If we click on play, we will be able to control our character. However, we might see that there is a sliding movement or it is still playing the **idle** animation when our character begins to walk or run. This is because the transition blending is too high. So, when we click to walk or run, the animation doesn’t change instantly as it should do. This problem can be solved by setting the **Transition** blending curve in the inspector to `0` or less transition. Have a look at the following screenshot:

Creating a third-person camera to follow our character

From the last section, we got a controllable character with animation, but the camera isn’t actually following the character at all. So, in this section, we will create the third-person camera to follow our character.

Engage thrusters

Now, we will create the script to control our character:

1. We will create the **CameraControl** script that will control our entire menu; go to **Assets** | **Create** | **Javascript** (for Unity JavaScript users) or **Assets** | **Create** | **C#** (for C# users), name it `CameraControl`, double-click on it to launch **MonoDevelop**, and then we will get our hands dirty with the code.
2. Open the **CameraControl** script file and type in the following code:

```
// Unity JavaScript user:

#pragma strict
@script RequireComponent (CharacterControl)
var smoothTime : float = 0.1f;
var maxSpeed : float = 150.0f;
var heightSmoothTime : float = 0.1f;
var distance : float = 2.5f;
var height : float = 0.75f;
private var _heightVelocity : float = 0.0f;
private var _angleVelocity : float = 0.0f;
private var _velocity : Vector3;
private var _targetTransform : Transform;
private var _cameraTransform : Transform;
private var _maxRotation : float;
private var _characterControl : CharacterControl;
private var _targetHeight : float = Mathf.Infinity;
private var _centerOffset : Vector3 = Vector3.zero;


// C# user:
using UnityEngine;
using System.Collections;
[RequireComponent(typeof(CharacterControl))]
public class CameraControl : MonoBehaviour {
  public float smoothTime = 0.1f;
  public float maxSpeed = 150.0f;
  public float heightSmoothTime = 0.1f;
  public float distance = 2.5f;
  public float height = 0.75f;
  float _heightVelocity = 0.0f;
  float _angleVelocity = 0.0f;
  Vector3 _velocity;
  Transform _targetTransform;
  Transform _cameraTransform;
  float _maxRotation;
  CharacterControl _characterControl;
  float _targetHeight = Mathf.Infinity;
  Vector3 _centerOffset = Vector3.zero;
  …
}
```

   Using the preceding code, we created all the variables required for this script.

3. Next, we will start creating the first function, `Awake()`, using the following code to

get the camera transform and the **CharacterController** component:

```
// Unity JavaScript user:

function Awake () {
  _cameraTransform = Camera.main.transform;
  _targetTransform = transform;
  _characterControl = GetComponent.<CharacterControl>();
}

// C# user:

void Awake () {
  _cameraTransform = Camera.main.transform;
  _targetTransform = transform;
  _characterControl = GetComponent<CharacterControl>();
}
```

4. Then, we create the `Start()` function as follows, to get the center position of the target, which is the character we are pointing at:

```
// Unity JavaScript user:

function Start () {
  var collider : Collider = _targetTransform.collider;
  _centerOffset = collider.bounds.center - _targetTransform.position;
}

// C# user:
void Start () {
  Collider collider = _targetTransform.collider;
  _centerOffset = collider.bounds.center - _targetTransform.position;
}
```

5. Next, we will create the `AngleDistance()` function to get the angle distance between the current angle and target; let's add the following code:

```
// Unity JavaScript user:

function AngleDistance  ( a : float, b : float) : float {
  a = Mathf.Repeat(a, 360);
  b = Mathf.Repeat(b, 360);
  return Mathf.Abs(b - a);
}

// C# user:
float AngleDistance  (float a, float b) {
  a = Mathf.Repeat(a, 360);
  b = Mathf.Repeat(b, 360);
  return Mathf.Abs(b - a);
}
```

Note

`Mathf.Repeat(t,l)` is the function that we can use to loop the `t` value but not higher than `l` and not lower than `0`.

`Mathf.Abs(n)` is the function that will return the absolute number `n`.

For more details, visit the following websites:

- http://docs.unity3d.com/Documentation/ScriptReference/Mathf.Repeat.html
- http://docs.unity3d.com/Documentation/ScriptReference/Mathf.Abs.html

6. Then, we need to create the `SetupRotation()` function to update the rotation of our camera. Type in the following code:

```
// Unity JavaScript user:

function SetUpRotation ( centerPos : Vector3 ) {
  var cameraPos : Vector3 = _cameraTransform.position;
  var offsetToCenter : Vector3 = centerPos - cameraPos;
  var yRotation : Quaternion = Quaternion.LookRotation(new
Vector3(offsetToCenter.x, offsetToCenter.y + height,
offsetToCenter.z));
  var relativeOffset : Vector3 = Vector3.forward * distance +
Vector3.down * height;
  _cameraTransform.rotation = yRotation *
Quaternion.LookRotation(relativeOffset);
}

// C# user:
void SetUpRotation ( Vector3 centerPos ) {
  Vector3 cameraPos = _cameraTransform.position;
  Vector3 offsetToCenter = centerPos - cameraPos;
  Quaternion yRotation = Quaternion.LookRotation(new
Vector3(offsetToCenter.x, offsetToCenter.y + height,
offsetToCenter.z));
  Vector3 relativeOffset = Vector3.forward * distance + Vector3.down *
height;
  _cameraTransform.rotation = yRotation *
Quaternion.LookRotation(relativeOffset);

}
```

7. Next, we will create the `LateUpdate()` function to update the camera position and rotation after all the objects have their `Update()` functions called. So, let's add the following code:

Note

The `LateUpdate()` function will be called after the `Update()` function has been called. This function will make sure that all the calculation in the `Update()` function is finished before we start the `LateUpdate()` function. We use the `LateUpdate()` function for the camera calculation because we don't want the target position of the camera to lerp (linear interpolation) or get affected by any concurrent physics or location calculations. It should be calculated after the character's orientation and position has been determined in `Update()`.

For more details on this function, refer to http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.LateUpdate

```
// Unity JavaScript user:

function LateUpdate () {
  var targetCenter : Vector3 = _targetTransform.position +
_centerOffset;
  var originalTargetAngle : float = _targetTransform.eulerAngles.y;
  var currentAngle : float = _cameraTransform.eulerAngles.y;
  var targetAngle : float = originalTargetAngle;
  if (AngleDistance (currentAngle, targetAngle) > 160 &&
_characterControl.IsMoveBackward) {
    targetAngle += 180;
  }
  currentAngle = Mathf.SmoothDampAngle(currentAngle, targetAngle,
_angleVelocity, smoothTime, maxSpeed);
  _targetHeight = targetCenter.y + height;
  var currentHeight : float = _cameraTransform.position.y;
  currentHeight = Mathf.SmoothDamp (currentHeight, _targetHeight,
_heightVelocity, heightSmoothTime);
  var currentRotation : Quaternion = Quaternion.Euler (0, currentAngle,
0);
  _cameraTransform.position = targetCenter;
  _cameraTransform.position += currentRotation * Vector3.back *
distance;
  var newCameraPos : Vector3 = _cameraTransform.position;
  newCameraPos.y = currentHeight;
  _cameraTransform.position = newCameraPos;
  SetUpRotation(targetCenter);
}

// C# user:
void LateUpdate () {
  Vector3 targetCenter = _targetTransform.position + _centerOffset;
  float originalTargetAngle = _targetTransform.eulerAngles.y;
  float currentAngle = _cameraTransform.eulerAngles.y;
  float targetAngle = originalTargetAngle;
  if (AngleDistance (currentAngle, targetAngle) > 160 &&
_characterControl.IsMoveBackward) {
    targetAngle += 180;
  }
  currentAngle = Mathf.SmoothDampAngle(currentAngle, targetAngle, ref
_angleVelocity, smoothTime, maxSpeed);
  _targetHeight = targetCenter.y + height;
  float currentHeight = _cameraTransform.position.y;
  currentHeight = Mathf.SmoothDamp (currentHeight, _targetHeight, ref
_heightVelocity, heightSmoothTime);
  Quaternion currentRotation = Quaternion.Euler (0, currentAngle, 0);
  _cameraTransform.position = targetCenter;
  _cameraTransform.position += currentRotation * Vector3.back *
distance;
  Vector3 newCameraPos = _cameraTransform.position;
  newCameraPos.y = currentHeight;
  _cameraTransform.position = newCameraPos;
  SetUpRotation(targetCenter);
}
```

Note

`Mathf.SmoothDampAngle()` is the function that gradually changes the angle given in degrees towards the target angle over time.

`Mathf.SmoothDamp()` is the function that gradually changes the value towards the target value over time.

For more details, visit the following websites:

- http://docs.unity3d.com/Documentation/ScriptReference/Mathf.SmoothDampAn
- http://docs.unity3d.com/Documentation/ScriptReference/Mathf.SmoothDamp.htı

8. Finally, we will go back to the Unity editor and drag-and-drop the **CameraControl** script that we just created on **Heroine_animate** in the **Hierarchy** view. Then, we can click on the play button to see the result.

Objective complete – mini debriefing

In this section, we created a third-person camera script to follow our character. This script also allows us to set the distance from our character and the height of our camera position by using some code from the built-in third-person camera script and adapting it to match our character.

Classified intel

Why do we need the `LateUpdate()` function instead of the `Update()` function in this script? Well, we use it to guarantee that the player position is already updated when we are performing the camera calculations. If we perform the calculation in the `Update()` function, the camera position might be calculated before the player position is updated. This will result in jitter.

We can also explain it this way: we wait for the input from the user and then get the direction in which the character will move from the `Update()` function. Then, we use the position of the character as the target position that our camera will follow and calculate the camera position in the `LateUpdate()` function.

Note

What's the difference between `Update()`, `LateUpdate()`, and `FixedUpdate()` functions? `Update()` is called for every frame. `LateUpdate()` is also called for every frame but after all the `Update()` functions have been called. `FixedUpdate()` is called for every fixed frame rate, which means that no matter how our game is running, slow or fast, the `FixedUpdate()` function will always call at the same rate. For example, if our game is slow, the `FixedUpdate()` function gets called more than once before calling the `Update()` function.

The following link is useful for `Update()` and `FixedUpdate()`: http://answers.unity3d.com/questions/10993/whats-the-difference-between-update-and-fixedupdat.html.

This way, we will be able to track each movement of our character and the camera will follow the direction smoothly without any jitter, as we can see in the following diagram:

Mission accomplished

In this project, we learned how to set up the animation using the Mecanim system as well as the basic **Animator Controller** window and how to use the **Animator** view to set up the state machine. We also created parameters to use as a condition to change to each animation state and the transition between each clip.

Next, we created our **CharacterControl** script to control our character. We also learned how to use the `Move()` function in the `CharacterController` script and how to speed up or slow down the animation clip by setting the speed of the clip. We also learned how to adjust the transition blending between each animation clip to change the state instantly. Then, we used the `OnAnimationMove()` function to set the animation state and parameters to control our animation clip.

Lastly, we created `CharacterCamera` to follow our character by using the `LateUpdate()` function to track the position of the character.

We will see a result similar to the following screenshot:

Hotshot challenges

Now we know how to create a custom character control script, camera, and animation from our custom script. Even though our custom script works really well with this character, it still has a lot of things that we can improve to make our script much more flexible. Let's do something to make our script better and much more flexible. Give the following ideas a try:

- Add your own character with a different animation, even if the character has more than five animation clips
- Use a different method to make the camera not follow the character when our character jumps (or basically just rotate the camera)
- Change some parameters such as distance or height in the `CameraControl` script to see how the game will look
- Create more action for the character such as slide or crawl and control them by using the state machine
- Add the backward walk or run by setting the negative speed for those animation clips and using the `_isBackward` property to check it
- Download the Mecanim Locomotion Starter Kit package from Unity Asset Store (https://www.assetstore.unity3d.com/#/content/7673) and use the **Mocap** data inside this package and retarget to our character

Chapter 5. Build a Rocket Launcher!

In this project, we will learn how to create a rocket launcher. Here, we will first reuse the `CameraControl` and `CharacterControl` classes from Project 4, *Add Character Control and Animation to Our Hero/Heroine*, and we will tweak the camera to make the shooting view more similar to *Resident Evil 4* or *Resident Evil 5*:

We will also take the robot character model and animation from the old FPS tutorial package from Unity, which we can download from the following website:

http://armedunity.com/files/file/32-fps-tutorial-unity-4x/

We will build the rocket launcher from scratch, add particle effects, and switch the camera view between the normal movement and aim. We will still use the **Mecanim** animation system to create the animation state, which makes it easy to set up the character animation.

Mission briefing

This project will start with how to set up the character animation by setting the animation clip using Mecanim and how to adapt the third-person controller (`CharacterControl`) and camera controller (`CameraControl`) scripts to make our character shoot, walk, run, jump, fall, and remain idle. Then, we will create a rocket prefab and the rocket launcher script to fire our rocket and add the particle effects for explosion, fire trail, and smoke that will appear from the barrel of the rocket launcher when the player shoots. We will use the **Shuriken Particle** system to create these effects.

Then, we will create the laser target to aim and the rocket **GUITexture** to track the number of rockets we are left with after each shot as well as the **reload** button to refill the rockets.

Why is it awesome?

This project will teach you how to deal with the extra bone in setting the humanoid character in Mecanim and how to use the blend tree and substate machine diagrams in Mecanim. You will understand how to switch the camera view and settings in the Shuriken Particle system, which is very powerful in Unity.

By the end of this project, you will have the basic knowledge required to create a third-person shooter style game with the aiming camera view and laser.

Your Hotshot objectives

In Project 4, *Add Character Control and Animation to Our Hero/Heroine*, we learned how to create a third-person controller script to control our character. For this project, we will use a similar concept: adapt and reuse the script to create the third-person shooter to fire a rocket from the rocket launcher. We will complete the following tasks:

- Setting up a character animation and animator controller
- Adding new features to the `CharacterControl` and `CameraControl` scripts
- Creating a `MouseLook` script and laser target scope
- Creating a rocket prefab and particle effects
- Creating a rocket launcher and RocketUI

Mission checklist

Before we start, we will need to get the project folder and assets from http://www.packtpub.com/support?nid=8267, which includes the finished project and the assets that we need to use in this project.

Browse to the preceding URL and download the `Chapter5.zip` package and unzip it. Inside the `Chapter5` folder, there are two Unity packages, which are `Chapter5Package.unitypackage` (we will use this package for this project) and `Chapter5Package_Completed.unitypackage` (this is the complete chapter package).

Setting up a character animation and animator controller

In Project 4, *Add Character Control and Animation to Our Hero/Heroine*, we imported the FBX file with multiple animation clips attached. However, in this project, we shall import the animation clips using another method. This method entails using multiple `FBX` files, with each file containing a separate animation clip. The concept behind this method is that you can create separate model files and name them using the convention `modelName@animationName.fbx`. For example, if you have the main model, `robot.fbx`, you will need to name this as `robot@idle.fbx` to create an **idle** animation clip.

Tip

The advantage of this method is that we can have multiple animators working on the same model with the different animation files, and we can put them all together in Unity. For more details, visit the following link:

http://docs.unity3d.com/Documentation/Manual/Splittinganimations.html

Prepare for lift off

In this section, we will begin by preparing an animation for our character, which will use the humanoid type to set up the animation; this can be done by performing the following steps:

1. Import the assets package by going to **Assets** | **Import Package** | **Custom Package…** and choosing `Chapter5.unityPackage`, which we downloaded earlier, and then clicking on the **Import** button in the pop-up window, as shown in the following screenshot:

2. Wait until the package is completely imported and you will see the `Chapter5` and `Resources` folders in the **Window** view. Then, go to the `Chapter5/Scene/RocketLaucher` scene and double-click on it to open the scene, as shown in the following screenshot:

3. Next, go to the `Resources/FBX/Robot Artwork` folder and click on `robot` to bring up the **Inspector** view, as shown in the following screenshot:

4. Then, go to its **Inspector** view and click on the **Rig** tab and make sure that the **Inspector** view is set as per the following screenshot:

We can see that we have the extra bone on **Rig** in the **Extra Transforms to Expose** window because this robot has been set up with the extra bone, which is the **gun** skin.

Tip

For a humanoid character, the bone structure is similar to the human bone structure. Mecanim takes advantage of this similarity by automatically setting up the basic skeleton structure, such as body, head, and limbs. However, in our character, the **gun** bone isn't usually in the human bones structure, so we need to set this as an extra bone.

Engage thrusters

Now, we are ready to start this section; perform the following steps:

1. Go to the `Chapter5/Animation` folder in the **Project** view and then double-click on **MyAnimatorController** to bring up the **Animator** view, as shown in the following screenshot:

2. Next, we will add the blend tree by right-clicking on the **Animator** window and navigating to **Create State | From New Blend Tree,** and then we'll name it `Locomotion`, as shown in the following screenshot:

3. Then, we will double-click on the **Locomotion** state and click on the **Blend Tree** state and go to its **Inspector** view. Set the **Blend Type** parameter to `1D`, **Parameter** to **Speed**, and in the **Motion** section, click on the **+** button and choose **Add Motion Field** twice.
4. After that, we will click on the circle icon and choose the **walk** clip as the first motion. Then, choose the **run** clip as the second motion, as shown in the following screenshot:

5. Now we will set up the **Threshold** and **Animation time** values for each clip to control the blending between the **walk** and **run** clips by using **Speed** as **Parameter**. This is done by going to the graph under **Parameter**, as shown in the following screenshot:

6. Next, we set up the transition from **fall** to **Locomotion**. So, let's right-click on the **fall** clip, choose **Make Transition**, and then click-and-drag on the **Locomotion** clip. Then, click on the arrow line to go to the **Inspector** view and set up the view as follows:

Atomic	Uncheck the box		
Conditions	**Speed**	**Greater**	`0.1`
	MotionState	**Equals**	`0`

7. Also, we will create the transition back from **Locomotion** to **fall**. Go to the transition's **Inspector** view and set up the view as follows:

Atomic	Leave this checked		
Conditions	**MotionState**	**Equals**	`2`

8. Next, we will set the blending time to `0` from **fall** to **idle** and from back **idle** to **fall**, as shown in the following screenshot:

9. Now we need to add the transitions from **idle** to **Locomotion**. So, let's right-click on the **idle** clip, choose **Make Transition**, and then click-and-drag this clip on the **Locomotion** clip. Then, we will click on the arrow line to go to the **Inspector** view and set up the view as follows:

Atomic	Leave this checked		
Conditions	**Speed**	**Greater**	`0.1`

10. Also, we will create the transition from **Locomotion** back to **idle**. Go to the transition's **Inspector** view and set up the view as follows:

Atomic	Leave this checked		
Conditions	**Speed**	**Less**	0.1

11. Next, we will create the substate machine, which will be used to control the shooting and aiming animations. Let's right-click on the **Animator** window and choose the **Create Sub-State Machine** option and then name it `Shooting`, as shown in the following screenshot:

12. Let's double-click on the **Shooting** substate machine to open it and go to the `Resources/FBX/Robot Artwork` folder in the **Project** view; click on the arrow in front of **robot@shoot**; then drag the **shoot** and **aiming** clips to the **Animator** view, as shown in the following screenshot:

13. Now, we need to create the transition from the **aiming** state to **shoot**. Let's right-click on the **aiming** clip, choose **Make Transition**, and then click-and-drag this on the **shoot** clip. Then, we will click on the arrow line again to go to the **Inspector** view and set up the view as follows:

Atomic	Uncheck the box	
Conditions	**IsShot**	**true**

14. Also, we will create the transition back from **shoot** to **aiming**. Go to the transition's **Inspector** view and set up the view as follows:

Atomic	Uncheck the box	

Conditions	**Exit Time**	0.8

15. Before we go to the **Base Layer** transition, we need to add the transition from **aiming** to **(Up) Base**. Then we will see the pop-up window; choose **idle**, as shown in the following screenshot:

16. After that, we will click on the arrow line again to go to the **Inspector** view and set up the view as follows:

Atomic	Uncheck the box		
Conditions	**MotionState**	**Equals**	0

17. Now, we go to the **Base Layer** transition and add the transition from **idle** state to **Shooting**. Let's right-click on the **idle** clip, choose **Make Transition**, and then click-and-drag the **idle** clip on to the **Shooting** clip. Then we will see the pop-up window; choose **aiming**, as shown in the following screenshot:

18. Click on the arrow line again to go to the **Inspector** view and set up the view as follows:

Atomic	Uncheck the box		
Conditions	**MotionState**	**Equals**	4
	IsShot	**false**	

19. Finally, the last transition is from **Locomotion** to **Shooting**. Let's right-click on the **Locomotion** clip, choose **Make Transition**, and then move the cursor and click on the **Shooting** clip. Then, we will see the pop-up window, choose **aiming**, as shown in the following screenshot:

20. Then, we will click on the arrow line again to go to the **Inspector** view and set up the view as follows:

Atomic	Uncheck the box		
Conditions	**MotionState**	**Equals**	4
	IsShot	**false**	

21. Now, we will get all the transitions, as shown in the following screenshot:

Objective complete – mini debriefing

Basically, what we have done here is set up the animator controller for our robot character, which is similar to the way we set up our character in Project 4, *Add Character Control and Animation to Our Hero/Heroine*. However, in this project, we've used the **Blend Tree** settings to set up the **walk** and **run** animations.

In **Blend Tree**, we can add as many animations as we want. The animation will change the state depending on the parameter we set up. In our case, we used **Speed** as our **Parameter** and set up `2` as the minimum value for the **walk** animation and `5` as the maximum for the **run** animation. We use the minimum value of `2` because we want our robot to change the animation state from **idle** to **walk** at the **Speed** value of `2`, which will be created in the next step. Also, the value `5` is the maximum speed for our character to run. The settings for **Speed** are shown in the following screenshot:

We also added a new substate machine for our **Shooting** animation. This substate machine contains two animations, which are the **aiming** and **shoot** clips. The substate machine is used to identify the separate stages that consist of a number of states. In our case, we used this because we knew that the **shoot** state only connects to the **aiming** state, as we can see in the following screenshot:

For more information on **Sub-State Machine**, visit the following link:

http://docs.unity3d.com/Documentation/Manual/NestedStateMachines.html

Classified intel

From this project, we have learned many things about how to set up the animator controller including **Blend Tree**. So, what is **Blend Tree**?

Blend Tree

Blend Tree is a very powerful state that allows us to blend multiple animations smoothly by using the blending parameters. For example, we can blend the **walk straight**, **walk left**, and **walk right** states by using the **Direction** setting as a blending parameter, or we can get a bit crazy by having two **Blend Tree** settings for the **run** and **walk** clips. Then, each clip will have its own **Blend Tree** setting to control the direction of the animation, as we can see in the following screenshot:

We can also set the **Blend Type** parameter in **Blend Tree** from **1D** to **2D** for advanced setting by using two parameters to check for the blending of states, as we can see in the following screenshot:

For more information on **Blend Tree**, visit the following links:

- http://docs.unity3d.com/Manual/class-BlendTree.html
- http://docs.unity3d.com/Manual/BlendTree-1DBlending.html
- http://docs.unity3d.com/Manual/BlendTree-2DBlending.html
- http://docs.unity3d.com/Manual/BlendTree-AdditionalOptions.html

Adding new features to the CharacterControl and CameraControl scripts

In this step, we will adapt and add new features to the `CharacterControl` and `CameraControl` scripts that we created in Project 4, *Add Character Control and Animation to Our Hero/Heroine*. We will add the script to control our character‘s movement by gradually changing the speed from **idle** to **walk** and **walk** to **run**. Then, we will add the ability to switch between shooting and movement, which will also change the camera’s view position.

Engage thrusters

Let's get started! To add new features to the `CharacterControl` and `CameraControl` scripts, perform the following steps:

1. Go to the `Chapter5/Scripts/C#` (for C# users) or `Chapter5/Scripts/Javascript` (for Unity JavaScript users) folder in the **Project** view and double-click on the `CharacterControl` script file to open it, as we can see in the following screenshot:

2. Open the `ChracterControl` script file; add the following highlighted script in the variable area as follows:

```
// Unity JavaScript user:

public enum MOTION_STATE {GROUND,JUMP,FALL,JUMP_HOLD,AIM}
…
private var _currentEulerAngle : Vector3;
private var _isAiming : boolean;
private var _isShot : boolean;
…
function get IsFall () : boolean {
  return (_inAirTime > MIN_AIR_TIME);
}
function get IsAiming () : boolean {
  return _isAiming;
}
function get GetCharacterEulerAngle() : Vector3{
  return _currentEulerAngle;
}


// C# user:

public class CharacterControl : MonoBehaviour {
  public enum MOTION_STATE {GROUND,JUMP,FALL,JUMP_HOLD,AIM}
  …
  Vector3 _currentEulerAngle;
  bool _isAiming;
  bool _isShot;
  …
  public bool IsFall {
    get { return (_inAirTime > MIN_AIR_TIME); }
  }
```

```
  public bool IsAiming {
    get { return _isAiming; }
  }
  public Vector3 GetCharacterEulerAngle {
    get { return _currentEulerAngle; }
  }
  …
}
```

3. Next, we will go to the `Start()` function and add the following highlighted code:

```
// Unity JavaScript user:

function Start () {
  …
  _inAirStartTime = Time.time;
  _currentEulerAngle = transform.eulerAngles;
  _isAiming = false;
}


C# user:

void Start () {
  …
  _inAirStartTime = Time.time;
  _currentEulerAngle = transform.eulerAngles;
  _isAiming = false;
}
```

4. Then, we go to the `Update()` function to set the running and aiming animation controller. Let's go to the `If (IsGround)` function and replace the old code from `if (!_isJumping && (_motionState == MOTION_STATE.GROUND)) {…}` to `if (_moveDirection != Vector3.zero) {…}`, as shown in the following highlighted script:

```
// Unity JavaScript user:
If (IsGround) {
  …
} else {
  …
}
if (!_isJumping) {
  _isAiming = (Input.GetKey(KeyCode.E));
  if (_isAiming) {
    _isShot = Input.GetButtonDown("Fire1");
  } else {
    if (_motionState == MOTION_STATE.GROUND) {
      _isRun = (Input.GetKey (KeyCode.LeftShift) || Input.GetKey
(KeyCode.RightShift));
      if (_isRun) {
        if (_moveDirection != Vector3.zero) {
          _moveSpeed += 0.15f;
          _moveSpeed = Mathf.Clamp(_moveSpeed,0.0f,runSpeed);
        }
      } else {
```

```
        if (_moveDirection != Vector3.zero) {
          _moveSpeed -= 0.15f;
          _moveSpeed = Mathf.Max(walkSpeed,_moveSpeed);
        } else {
          _moveSpeed = 0.0f;
        }
      }
      if (Input.GetButtonDown ("Jump")) {
        _verticalSpeed = jumpSpeed;
        _isJumping = true;
        _inAirTime = 0.0f;
        _inAirStartTime = Time.time;
      }
    }
  }
} else {
  // TODO – Reset Aiming
}

if (!_isAiming) {
  _movement = (_moveDirection * _moveSpeed) + new Vector3 (0,
_verticalSpeed, 0);
  _movement *= Time.deltaTime;
  if (_movement != Vector3.zero) {
    _characterController.Move(_movement);
  }
transform.eulerAngles = _currentEulerAngle;
  if (_moveDirection != Vector3.zero) {
    transform.rotation = Quaternion.LookRotation(_moveDirection);
  }
  _currentEulerAngle = transform.eulerAngles;
}


// C# user:
If (IsGround) {
  …
} else {
  …
}
if (!_isJumping) {
  _isAiming = (Input.GetKey(KeyCode.E));
  if (_isAiming) {
    _isShot = Input.GetButtonDown("Fire1");
  } else {
    if (_motionState == MOTION_STATE.GROUND) {
      _isRun = (Input.GetKey (KeyCode.LeftShift) || Input.GetKey
(KeyCode.RightShift));
      if (_isRun) {
        if (_moveDirection != Vector3.zero) {
          _moveSpeed += 0.15f;
          _moveSpeed = Mathf.Min(_moveSpeed,runSpeed);
        }
      } else {
        if (_moveDirection != Vector3.zero) {
          _moveSpeed -= 0.15f;
```

```
          _moveSpeed = Mathf.Max(walkSpeed,_moveSpeed);
        } else {
          _moveSpeed = 0.0f;
        }
      }
      if (Input.GetButtonDown ("Jump")) {
        _verticalSpeed = jumpSpeed;
        _isJumping = true;
        _inAirTime = 0.0f;
        _inAirStartTime = Time.time;
      }
    }
  }
} else {
  // TODO – Reset Aiming
}
if (!_isAiming) {
  _movement = (_moveDirection * _moveSpeed) + new Vector3 (0,
_verticalSpeed, 0);
  _movement *= Time.deltaTime;

  if (_movement != Vector3.zero) {
    _characterController.Move(_movement);
  }
  transform.eulerAngles = _currentEulerAngle;
  if (_moveDirection != Vector3.zero) {
    transform.rotation = Quaternion.LookRotation(_moveDirection);
  }
  _currentEulerAngle = transform.eulerAngles;
}
```

5. Next, we go to the `OnAnimatorMove()` function to set up the `MOTION_STATE.AIM` and `IsShot` parameters. Let's replace the code with following highlighted code:

```
// Unity JavaScript user:

function OnAnimatorMove (){
  if (_animator) {
    if (_isJumping) {
      …
    } else {
      if (IsFall) {
        …
      } else {
        if (_isAiming) {
          _motionState = MOTION_STATE.AIM;
        } else {
          _motionState = MOTION_STATE.GROUND;
        }
      }
    }
    _animator.SetFloat ("Speed", _moveSpeed);
    _animator.SetBool ("IsShot", _isShot);
    _animator.SetInteger ("MotionState", parseInt(_motionState));
    }
}
```

```
// C# user:
void OnAnimatorMove (){
  if (_animator) {
    if (_isJumping) {
      …
    } else {
      if (IsFall) {
        …
      } else {
        if (_isAiming) {
          _motionState = MOTION_STATE.AIM;
        } else {
          _motionState = MOTION_STATE.GROUND;
        }
      }
    }
    _animator.SetFloat ("Speed", _moveSpeed);
    _animator.SetBool ("IsShot", _isShot);
    _animator.SetInteger ("MotionState", (int)_motionState);
  }
}
```

6. We have finished setting up the `CharacterControl` script. Now, we will go to the `CameraControl` script. Let's go to the `Chapter5/Scripts/C#` (for C# users) or `Chapter5/Scripts/Javascript` (for Unity JavaScript users) folder in the **Project** view. Double-click on the `CameraControl` script and add the variables as follows:

```
// Unity JavaScript user:
…
var heightSmoothTime : float = 0.1f;
var distanceAimming : float = 1.5f;
var distance : float = 2.5f;
var height : float = 1.0f;
var shootTarget : Transform;
…


// C# user:
…
public float heightSmoothTime = 0.1f;
public float distanceAimming = 1.5f;
public float distance = 2.5f;
public float height = 1.0f;
public Transform shootTarget;
…
```

7. Next, we will go to the `Start()` function to offset the target in the **y** position and replace the old code with the following highlighted code:

```
// Unity JavaScript user:
function Start () {
  var newCenter : Vector3 = _targetTransform.collider.bounds.center +
new Vector3(0,0.45f,0);
  _centerOffset = newCenter - _targetTransform.position;
```

```
}


// C# user:
void Start () {
  Vector3 newCenter = _targetTransform.collider.bounds.center + new
Vector3(0,0.45f,0);
  _centerOffset = newCenter - _targetTransform.position;
}
```

8. Now, we will go to the `LateUpdate()` function because we want to make sure that the camera's position will always be updated after our character has finished moving. Add the following highlighted code to set up the camera's position when our character is aiming:

```
// Unity JavaScript user:

function LateUpdate (){
  var targetCenter : Vector3 = _targetTransform.position +
_centerOffset;

  if (!_characterControl.IsAiming) {
    var originalTargetAngle : float = _targetTransform.eulerAngles.y;
    …
    …
    SetUpRotation(targetCenter);
  } else {
    currentRotation = Quaternion.Euler (0,
_characterControl.GetCharacterEulerAngle.y, 0);
    _cameraTransform.position = targetCenter;
    _cameraTransform.position += currentRotation * Vector3.back *
distanceAimming;
    newCameraPos = _cameraTransform.position;
    newCameraPos.y += height;
    _cameraTransform.position = newCameraPos;
    _cameraTransform.LookAt(shootTarget.position);
  }
}


// C# user:

void LateUpdate () {
  Vector3 targetCenter = _targetTransform.position + _centerOffset;
  if (!_characterControl.IsAiming) {
    float originalTargetAngle = _targetTransform.eulerAngles.y;
    …
    …
    SetUpRotation(targetCenter);
} else {
    Quaternion currentRotation = Quaternion.Euler (0,
_characterControl.GetCharacterEulerAngle.y, 0);
    _cameraTransform.position = targetCenter;
    _cameraTransform.position += currentRotation * Vector3.back *
distanceAimming;
```

```
        Vector3 newCameraPos = _cameraTransform.position;
        newCameraPos.y += height;
        _cameraTransform.position = newCameraPos;
        _cameraTransform.LookAt(shootTarget.position);
    }
}
```

9. Now, we've finished the first coding part. We need to go back to Unity and set up our character and camera's target position when aiming. Let's go to the `Resources/FBX/Robot Artwork` folder in the **Project** view and drag the **robot** prefab to the **Hierarchy** view; then go to the robot **Inspector** view and set it as follows:

Transform	**Position**	**X**: `0`, **Y**: `0.25`, and **Z**: `-6`
Animator	**Controller**	**MyAnimatorController**

10. Next, we will create our `CameraAiming` position object by going to **GameObject | Create Empty**; name it `CameraAimingPosition`. Then, we need to drag this inside our game object of **robot**, as shown in the following screenshot:

11. Then, we click on the **CameraAimingPosition** option and set **Transform** as follows:

Transform	**Position**	**X**: `0.6`,**Y**: `1`, and **Z**: `0.4`

12. After that, we need to add `CameraControl` to our **robot**. Let's go to the robot **Inspector** view and click on the **Add Component** button and then navigate to **Script | CameraControl**.
13. Then, we will see that our robot has **Character Controller**, **Character Control (Script)**, and **Camera Control (Script)** attached to the **Inspector** view. Let's go to the robot **Inspector** view again and set the **Character Controller** and **Camera Control (Script)** parameters as follows:

Character Controller	
Center	**X**: `0`, **Y**: `-0.05`, and **Z**: `0`
Height	`2.4`
Camera Control (Script)	
Shoot Target	**CameraAimingPosition** (drag it here)

We have now finished this step. Let's click on play to see the result. We will see that

we can press the Space bar to jump, *WASD* or arrow keys to walk, and if we are pressing the *Shift* key while the character is walking, this will trigger the **running** state. The last thing to note is, if we hold the *E* key, our character will prepare to shoot, and if we press the left-mouse button, our character will shoot.

Objective complete – mini debriefing

Basically, what we have done here is create the script to control our robot character. We added the new **aiming** and **shooting** animation states to get the scene ready for the next step. In this step, we added the script that will detect whether our character is walking or running by increasing the value of the `_moveSpeed` parameter while we move our character or hold the *Shift* key. On the other hand, we decrease the speed when we aren't moving.

We created the `CameraAimingPosition` object, which is the target position while our character is aiming. We also have the script to switch our camera between aiming and normal movement, as shown in the following screenshot:

However, right now, we won't be able to rotate our character while it is aiming. So, in the next step, we will add the `MouseLook` script and `Laser` target object to make our character able to look around while aiming.

Creating a MouseLook script and laser target scope

From the previous section, we have the basic control setup for our robot to run, walk, aim, or shoot. Now, we need a script to control our character’s rotation using the mouse while the character is aiming. We also add the laser target scope object, which will be used by the player to aim.

Engage thrusters

We will now create the `MouseLook` script to control our character by performing the following steps:

1. We will create the `MouseLook` script that will control our entire menu; go to **Assets** | **Create** | **Javascript** (for Unity JavaScript users) or **Assets** | **Create** | **C#** (for C# users), name it `MouseLook`, double-click on it to launch **MonoDevelop**, and we will start writing some code.
2. Open the `MouseLook` script file and type in the following variables:

```
// Unity JavaScript user:

#pragma strict

enum RotationAxes { MouseXAndY = 0, MouseX = 1, MouseY = 2 }
var axes : RotationAxes = RotationAxes.MouseXAndY;
var sensitivityX : float = 5f;
var sensitivityY : float = 5f;
var minimumX : float = -10f;
var maximumX : float = 70f;
var minimumY : float = -5f;
var maximumY : float = 5f;

private var _characterControl : CharacterControl;
private var _rotationX : float = 0f;
private var _rotationY : float = 0f;


// C# user:


using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {

  public enum RotationAxes { MouseXAndY = 0, MouseX = 1, MouseY = 2 }
  public RotationAxes axes = RotationAxes.MouseXAndY;
  public float sensitivityX = 5f;
  public float sensitivityY = 5f;

  public float minimumX = -10f;
  public float maximumX = 70f;

  public float minimumY = -5f;
  public float maximumY = 5f;

  CharacterControl _characterControl;

  float _rotationX = 0f;
  float _rotationY = 0f;
  …
}
```

Now, we have all the necessary parameters for our script. The `enum` type is to choose whether we need our script to control the `X` and `Y` rotations, only `X` , or only the `Y` rotation.

3. Next, we will start creating the first function, `Awake()`, using the following code to get the `_characterControl` component:

```
// Unity JavaScript user:

function Awake () {
  _characterControl = GetComponent.<CharacterControl>();
}


//C# user:

void Awake () {
  _characterControl = GetComponent<CharacterControl>();
}
```

4. After that, we create the `Start()` function to make sure that our character has a `rigidbody` component attached, even though we don't have it.

Tip

This setting is just for the fact that we can reuse the code for other character's settings in the future.

We will freeze the `rigidbody` rotation as follows:

```
// Unity JavaScript user:

function Start () {
  if (rigidbody) { rigidbody.freezeRotation = true; }
}


//C# user:

void Start () {
  if (rigidbody) { rigidbody.freezeRotation = true; }
}
```

5. Next, we will create the `Update()` function. In this function, we will calculate the rotation of each axis and assign the value to the character to make it rotate when moving the mouse in each direction, as follows:

```
// Unity JavaScript user:

function Update () {
  if (axes == RotationAxes.MouseXAndY) {
    _rotationX += Input.GetAxis("Mouse X") * sensitivityX;
    _rotationX = Mathf.Clamp (_rotationX, minimumX, maximumX);
    _rotationY += Input.GetAxis("Mouse Y") * sensitivityY;
    _rotationY = Mathf.Clamp (_rotationY, minimumY, maximumY);
```

```
    transform.localEulerAngles = new Vector3(-_rotationY, _rotationX +
_characterControl.GetCharacterEulerAngle.y, 0);
  } else if (axes == RotationAxes.MouseX) {
    _rotationX = transform.localEulerAngles.y + Input.GetAxis("Mouse
X") * sensitivityX;
    _rotationX = Mathf.Clamp (_rotationX, minimumX, maximumX);
    transform.Rotate(0, _rotationX +
_characterControl.GetCharacterEulerAngle.y, 0);
  } else {
    _rotationY += Input.GetAxis("Mouse Y") * sensitivityY;
    _rotationY = Mathf.Clamp (_rotationY, minimumY, maximumY);
    transform.localEulerAngles = new Vector3(-_rotationY,
transform.localEulerAngles.y, 0);
  }
}


// C# user:

void Update () {
  if (axes == RotationAxes.MouseXAndY) {
    _rotationX += Input.GetAxis("Mouse X") * sensitivityX;
    _rotationX = Mathf.Clamp (_rotationX, minimumX, maximumX);
    _rotationY += Input.GetAxis("Mouse Y") * sensitivityY;
      _rotationY = Mathf.Clamp (_rotationY, minimumY, maximumY);
    transform.localEulerAngles = new Vector3(-_rotationY, _rotationX +
_characterControl.GetCharacterEulerAngle.y, 0);
  } else if (axes == RotationAxes.MouseX) {
    _rotationX = transform.localEulerAngles.y + Input.GetAxis("Mouse
X") * sensitivityX;
    _rotationX = Mathf.Clamp (_rotationX, minimumX, maximumX);
    transform.Rotate(0, _rotationX +
_characterControl.GetCharacterEulerAngle.y, 0);
  } else {
    _rotationY += Input.GetAxis("Mouse Y") * sensitivityY;
    _rotationY = Mathf.Clamp (_rotationY, minimumY, maximumY);
    transform.localEulerAngles = new Vector3(-_rotationY,
transform.localEulerAngles.y, 0);
  }
}
```

Note

`transform.localEulerAngles`: This is the parameter that we can use to set up the rotation based on the degree of each axis that is related to the parent transform.

For more details on this, visit the following Unity website:

http://docs.unity3d.com/Documentation/ScriptReference/Transform-localEulerAngles.html

6. Now, we need to go back to our `CharacterControl` script and add more variables using the following highlighted code:

```
// Unity JavaScript user:
```

```
enum MOTION_STATE {GROUND,JUMP,FALL,JUMP_HOLD,AIM}

private final var GUN_LASER_DISTANCE : float = 1000f;
var laser : LineRenderer;
private var _mouseLook :  MouseLook;
…


// C# user:

public enum MOTION_STATE {GROUND,JUMP,FALL,JUMP_HOLD,AIM}

const float GUN_LASER_DISTANCE = 1000f;
public LineRenderer laser;
MouseLook _mouseLook;
…
```

7. Go to the `Awake()` and `Start()` functions and set them as in the following highlighted code:

```
// Unity JavaScript user:

function Awake () {
  …
  _mouseLook = GetComponent.<MouseLook>();
}
function Start () {
  …
  _isAiming = false;
  _mouseLook.enabled = false;
  if (laser != null) {
    laser.gameObject.SetActive(false);
  }
}


// C# user:

void Awake () {
  …
  _mouseLook = GetComponent<MouseLook>();
}
void Start () {
  …
  _isAiming = false;
  _mouseLook.enabled = false;
  if (laser != null) {
    laser.gameObject.SetActive(false);
  }
}
```

8. Go inside the `Update()` function to make our robot rotate around while aiming and show the laser target scope. Let's go inside the `if (!_isJumping)` condition and enter the following highlighted code:

```
// Unity JavaScript user:
```

```
function Update () {
  …
  if (!_isJumping) {
    _isAiming = (Input.GetKey(KeyCode.E));
    if (_isAiming) {
       if (laser != null) {
        laser.gameObject.SetActive(true);
        laser.SetPosition(1,new Vector3(GUN_LASER_DISTANCE,0,0));
      }
      _mouseLook.enabled = true;
      _isShot = Input.GetButtonDown("Fire1");
    } else {
      if (laser != null) {
        laser.gameObject.SetActive(false);
      }
      _mouseLook.enabled = false;
      if (_motionState == MOTION_STATE.GROUND) {
        …
      }
    }
  } else {
    // TODO – Reset Aiming
    if (laser != null) {
      laser.gameObject.SetActive(false);
    }
     _mouseLook.enabled = false;
  }
…
}


// C# user:

void Update() {
  …
  if (!_isJumping) {
    _isAiming = (Input.GetKey(KeyCode.E));
    if (_isAiming) {
       if (laser != null) {
        laser.gameObject.SetActive(true);
        laser.SetPosition(1,new Vector3(GUN_LASER_DISTANCE,0,0));
      }
      _mouseLook.enabled = true;
      _isShot = Input.GetButtonDown("Fire1");
    } else {
      if (laser != null) {
        laser.gameObject.SetActive(false);
      }
      _mouseLook.enabled = false;
      if (_motionState == MOTION_STATE.GROUND) {
        …
      }
    }
  } else {
    // TODO – Reset Aiming
```

```
    if (laser != null) {
      laser.gameObject.SetActive(false);
    }
    _mouseLook.enabled = false;

  …
}
```

9. Now, we are finished with the scripting part, so we will go back to the Unity editor. Let's create the laser target scope. First, go to **GameObject** | **Create Empty** to create an empty game object and name it `LaserScope`.
10. Click on the **LaserScope** object in the **Hierarchy** view and then go to **Component** | **Effects** | **Line Renderer** to add the **Line Renderer** component to this object.
11. Drag the **LaserScope** object inside the **gun_model** option in the **robot** object in the **Hierarchy** view, as we can see in the following screenshot:

12. Next, we will click on the **LaserScope** object and go to its **Inspector** view and set it as follows:

Transform			
Position	**X**: `6.5`	**Y**: `0.25`	**Z**: `0`
Rotation	**X**: `0`	**Y**: `0`	**Z**: `0`
Scale	**X**: `1`	**Y**: `1`	**Z**: `1`
Line Renderer			
Cast Shadows	Uncheck		
Receive Shadows	Uncheck		
Materials			
Element 0	**Laser**		
Positions			
Size	`2`		
Element 0	**X**: `0`	**Y**: `0`	**Z**: `0`
Element 1	**X**: `0`	**Y**: `0`	**Z**: `0`

Parameters	
Start Width	`0.02`
End Width	`0.02`
Use World Space	Uncheck

13. Then go back to the **Hierarchy** view and click on the **robot** object in the **Hierarchy** view to go to its **Inspector** view and click on the **Add Component** button; navigate to **Scripts** | **Mouse Look** (or we can drag the `Mouse Look` script here).
14. Go to **Character Control (Script)** in the **Laser** property and then set the **LaserScope** object as follows:

Character Control (script)	
Laser	**LaserScope**

Finally, we can click on play. We will see that the laser will turn on only when our character is aiming. Also, the robot can rotate by moving the mouse. This is shown in the following screenshot:

Isn’t this cool?

Objective complete – mini debriefing

In this section, we created a `MouseLook` script to control the character's rotation while it is aiming. Basically, this script will get the mouse's `x` or `y` axis multiplied by the sensitivity, which will be used as a degree to control the rotation of our character.

Then, we added code in the `CharacterControl` script to toggle the `MouseLook` script to enable or disable the character while the character is in an animation state. We also hid and showed the laser scope.

Finally, we created the `LaserScope` object, which used the **Line Renderer** component to draw a line on the screen.

Classified intel

We already knew that we can use **Gizmos.DrawLine** to draw the line in the debug mode, but drawing the line in the game mode is different. We will need to deal with drawing of the mesh, setting up of vertices, UVW map, and so on, which might be very complicated for a beginner. This is why Unity provides us with **Line Renderer**.

Line Renderer

In this step, we have used the **Line Renderer** component as our **LaserScope**. The **Line Renderer** component is a very convenient way to create a line or/and curve in the 3D world by taking an array of points and drawing them in an order. We can also add materials and apply the start or end color to create more variety. This is shown in the following screenshot:

Here, the faces will always face the camera no matter what position we set. The advantage is that the players always see the object facing toward them. However, when we need to move the camera, the **Line Renderer** component will show all the faces toward the camera, which might sometimes cause an unpredictable shape.

Note

For more details on **Line Renderer**, visit the following link:

http://docs.unity3d.com/Documentation/Components/class-LineRenderer.html

Creating a rocket prefab and particle effects

In the previous section, we got a controllable character with all the animations and mouse rotation, but there is nothing really happening when we click on the left mouse button. So, in this step, we will create the `rocket` prefab object and all of the particle effects for this prefab object, which are **Smoke**, **Fire Trail**, and **Explosion** so that we can use them in the next step.

Engage thrusters

Let's get started. To create a `Rocket` prefab object and particle effects, perform the following steps:

1. First, we will create the `Explosion` prefab by going to **GameObject** | **Create Empty** and name it `Explosion` and reset its **Transform** values as follows:

Transform	
Position	**X**: `0`, **Y**: `0`, and **Z**: `0`
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `1`, **Y**: `1`, and **Z**: `1`

2. Next, we will create the `Flame` particle by going to **GameObject** | **Create Other** | **Particle System** and name it `Flame`. We need to drag it inside our **Explosion** object, as shown in the following screenshot:

3. Then, go to the **Inspector** view of **Flame** and set it as follows:

Transform	
Position	**X**: `0`, **Y**: `0`, and **Z**: `0`
Rotation	**X**: `-90`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `1`, **Y**: `1`, and **Z**: `1`
Particle System	
Duration	`2.00`
Looping	Uncheck
Start Lifetime	`2`
Start Speed	`2`
Start Size	`2.5`
Start Color	**R**: `255`, **G**: `87`, **B**: `38`, **A**: `145`
Max Particle	`70`
Emission	Click to view the drop-down menu

Rate	`30`
Shape	Click to view the drop-down menu
Shape	**Cone**
Angle	`10`
Random Direction	Check
Color over LifeTime	Click to bring the drop-down menu
Color	Set this as shown in the following screenshot:
Renderer	Click to bring the drop-down menu
Material	**Flame**

4. Next, we will add `Light` by going to **GameObject** | **Create Other** | **Point Light**; name it `Light` and drag it inside the **Flame** object, as shown in the following screenshot:

5. Go to the **Inspector** view of **Light** and set its as follows:

Transform	
Position	**X**: `0`, **Y**: `0`, and **Z**: `0.6`
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `1`, **Y**: `1`, and **Z**: `1`
Light	
Range	`15`
Color	**R**: `255`, **G**: `120`, **B**: `47`, **A**: `255`

6. Next, we will add another particle effect, `FireSmoke`, by going to **GameObject** | **Create Other** | **Particle System** and name it `FireSmoke`. Drag it inside our **Explosion** object, as shown in the following screenshot:

7. Go to the **Inspector** view of **FireSmoke** and set it as follows:

Transform	
Position **Rotation** **Scale**	**X**: `0`, **Y**: `0`, and **Z**: `0` **X**: `-90`, **Y**: `0`, and **Z**: `0` **X**: `1`, **Y**: `1`, and **Z**: `1`
Particle System	
Duration	`1.00`
Looping	Uncheck
Start Delay	`1`
Start Lifetime	`2`
Start Speed	`2`
Start Size	`4`
Start Color	**R**: `137`, **G**: `137`, **B**: `137`, **A**: `25`
Simulation Space	**World**
Max Particle	`50`
Emission	Click to bring the drop-down menu
Rate	`3`
Shape	Click to bring the drop-down menu
Shape	**Cone**
Angle	`15`
Random Direction	Check
Color over LifeTime	Click to bring the drop-down menu
	Set it as shown in the following screenshot:

Color	
Renderer	Click to bring the drop-down menu
Material	**Smoke**

8. Next, we will create a script to control our **Explosion** particle. Let's go to **Assets** | **Create** | **Javascript** (for Unity JavaScript users) or **Assets** | **Create** | **C#** (for C# users), name it `AutoDestroyParticle`, double-click on it to launch **MonoDevelop**, and define the variables and functions as follows:

```
// Unity JavaScript user:

#pragma strict

var timeOut : float = 4.0f;
private var _light : Light;
private var _decreaseTime : float;
private var _isRemoveLight : boolean = false;

function Awake () {
  _light = GetComponentInChildren.<Light>();
}

function Start () {
    _decreaseTime = _light.range/timeOut;
  Invoke("KillObject", timeOut);
}

function Update () {
  if (!_isRemoveLight) {
    if (_light.range > 0) {
      _light.range -= _decreaseTime*Time.deltaTime;
    } else {
      _isRemoveLight = true;
      GameObject.Destroy(_light);
    }
  }
}

function KillObject () {
  var fireExplosions : ParticleSystem[] = GetComponentsInChildren.
<ParticleSystem>();
  for (var i : int = 0; i < fireExplosions.Length; ++i) {
    if (fireExplosions[i] != null) {
      fireExplosions[i].Stop();
```

```
      fireExplosions[i].loop = false;
    }
  }
  GameObject.Destroy(gameObject);
}


// C# user:

using UnityEngine;
using System.Collections;

public class AutoDestroyParticle : MonoBehaviour
{
  public float timeOut = 4.0f;
  Light _light;
  float _decreaseTime;
  bool _isRemoveLight = false;

  void Awake () {
     _light = GetComponentInChildren<Light>();
  }
  void Start () {
     _decreaseTime = _light.range/timeOut;
    Invoke("KillObject", timeOut);
  }

  void Update () {
    if (!_isRemoveLight) {
      if (_light.range > 0) {
        _light.range -= _decreaseTime*Time.deltaTime;
      } else {
        _isRemoveLight = true;
        GameObject.Destroy(_light);
      }
    }
  }

  void KillObject () {
    ParticleSystem[] fireExplosions =
GetComponentsInChildren<ParticleSystem>();
    for (int i = 0; i < fireExplosions.Length; ++i) {
      if (fireExplosions[i] != null) {
        fireExplosions[i].Stop();
        fireExplosions[i].loop = false;
      }
    }
    GameObject.Destroy(gameObject);
  }
}
```

9. Then, go back to Unity and add the `AutoDestroyParticle` script to the **Explosion** object in **Hierarchy**. We will also create the prefab of the **Explosion** object by dragging it to the `Resources/Prefabs` folder in the **Project** view. Remove the **Explosion** object from our **Hierarchy** view by right-clicking on this and then

choosing **Delete**. Now, we have got our **Explosion** particle.

10. Now we need to create our `rocket` prefab by going to the `Resources/FBX/Rocket` folder in the **Project** view and drag the `rocket` prefab object to the **Hierarchy** view.
11. Then, we will create `FireTrail`, which gets attached to our rocket. Let's go to **GameObject** | **Create Other** | **Particle System** and name it `FireTrail`. We need to drag it inside our `rocket` object, as shown in the following screenshot:

12. Go to the **Inspector** view of **FireTrail** and set it as follows:

Transform	
Position	**X**: `0`, **Y**: `0`, and **Z**: `-0.25`
Rotation	**X**: `0`, **Y**: `-180`, and **Z**: `0`
Scale	**X**: `1`, **Y**: `1`, and **Z**: `1`
Particle System	
Duration	`4.00`
Looping	Check
Start Lifetime	`2`
Start Speed	`1.5`
Start Size	`0.75`
Start Rotation	`10`
Start Color	**R**: `255`, **G**: `164`, **B**: `66`, **A**: `96`
Simulation Space	**World**
Max Particle	`50`
Emission	Click to bring the drop-down menu
Rate	`15`
Shape	Click to bring the drop-down menu
Shape	**Cone**
Angle	`0`
Radius	`0.05`

Random Direction	Check
Color over LifeTime	Click to view the drop-down menu
Color	Set it as shown in the following screenshot:
Renderer	Click to view the drop-down menu
Material	**FireTrail**

13. Next, we will create the `Rocket` script, which will be used to trigger the **Explosion** particle. Let's go to **Assets** | **Create** | **Javascript** (for Unity JavaScript users) or **Assets** | **Create** | **C#** (for C# users), name it `Rocket`, double-click on it to launch **MonoDevelop**, and define the variables as follows:

```
// Unity JavaScript user:

#pragma strict
@script RequireComponent(ConstantForce)

var timeOut : float = 3.0f;
var explosionParticle : GameObject;

private var _smokeTrail : ParticleSystem;

function Awake () {
  _smokeTrail = GetComponentInChildren.<ParticleSystem>();
}

function Start () {
  Invoke("KillObject", timeOut);
}

function OnCollisionEnter (others : Collision) {
  var contactPoint : ContactPoint = others.contacts[0];
  var rotation : Quaternion = Quaternion.Euler(Vector3.up);
  GameObject.Instantiate(explosionParticle, contactPoint.point,
rotation);

  KillObject();
}

private function KillObject () {
  if (_smokeTrail != null) {
    _smokeTrail.Stop();
```

```
    _smokeTrail.loop = false;
  }
  GameObject.Destroy(gameObject);
}


// C# user:

using UnityEngine;
using System.Collections;

[RequireComponent(typeof(ConstantForce))]
public class Rocket : MonoBehaviour {

  public float timeOut = 3.0f;
  public GameObject explosionParticle;
  ParticleSystem _smokeTrail;

  void Awake () {
    _smokeTrail = GetComponentInChildren<ParticleSystem>();
  }

  void Start () {
    Invoke("KillObject", timeOut);
  }

  void OnCollisionEnter (Collision others) {
    ContactPoint contactPoint = others.contacts[0];
    Quaternion rotation = Quaternion.Euler(Vector3.up);
    GameObject.Instantiate(explosionParticle, contactPoint.point,
rotation);

    KillObject();
  }

  void KillObject () {
    if (_smokeTrail != null) {
      _smokeTrail.Stop();
      _smokeTrail.loop = false;
    }
    GameObject.Destroy(gameObject);
  }

}
```

14. Then, we go back to Unity and add the **Box Collider** component to the `rocket` object by going to **Component** | **Physics** | **Box Collider**.

    ## Tip

    When we add the **Box Collider** component to the new object, **Box Collider** will automatically adjust its size to fit around the object. This is why we don't have to set up the size or the position of **Box Collider**.

15. Now add the `Rocket` script to the **rocket** object in **Hierarchy** and set up the **Inspector** as follows:

Rigidbody	
Use Gravity	Uncheck
Rocket (script)	
Explosion Particle	**Explosion** (drag from the **Project** view that we just created)

16. We need to create the prefab of the **rocket** object by dragging it to the `Resources/Prefabs` folder in the **Project** view and removing the **rocket** object from our **Hierarchy** view by right-clicking and choosing **Delete**. Now, we have got our **rocket** prefab.
17. After that, we will create the `Smoke` particle by going to **GameObject** | **Create Other** | **Particle System**; name it `Smoke`. Drag it inside our **gun_model** object in **Hierarchy**, as shown in the following screenshot:

18. Go to the **Inspector** view of **Smoke** and set it as follows:

Transform	
Position	**X**: `6.9`, **Y**: `0.25`, and **Z**: `0`
Rotation	**X**: `270`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `1`, **Y**: `1`, and **Z**: `1`
Particle System	
Duration	`0.50`
Looping	Uncheck
Start Lifetime	`1`
Start Speed	`1`
Start Size	`0.5`
Start Color	**R**: `120`, **G**: `120`, **B**: `120`, **A**: `80`
Simulation Space	**World**

Play On Awake	Uncheck
Max Particle	20
Emission	Click to view the drop-down menu
Rate	15
Shape	Click to bring the drop-down menu
Shape	**Cone**
Angle	0
Radius	0.5
Random Direction	Check
Color over LifeTime	Click to view the drop-down menu
Color	Set it as shown in the following screenshot:
Renderer	Click to view the drop-down menu
Material	**Smoke**

Now we have finished creating all the particles that we'll use in the next step.

Objective complete – mini debriefing

In this section, we created the `rocket` prefab and particle effects that will appear when the player clicks on **fire**. It seems like we didn't see any progress in this section, but we will definitely see the progress in the last step, so be prepared!

First, we created the `Explosion` particle system, which contains the **Flame**, **Light**, and **FireSmoke** elements. We also created the `AutoDestroyParticle` element to control the time this particle takes to be destroyed from the scene. In this `Rocket` script, we also controlled the intensity of the light related to the particle's appearance. We used the `Invoke()` function to call the `KillObject()` function after the `timeout (4.0 seconds)` parameter.

Next, we created the `Rocket` script, we used `@script RequireComponent(ConstantForce)` in Unity JavaScript and `[RequireComponent(typeof(ConstantForce))]` in C# to tell the script to acquire the `ConstantForce` component for the **rocket** prefab; this will tell Unity to automatically add the `ConstantForce` component when we add this component to the object.

Note

`ConstantForce` is one of the **Physics** components in Unity that will add a constant force to the `RigidBody` object (`ConstantForce` works with the `RigidBody` components, so when we add `ConstantForce` to our object, Unity will automatically add the `RigidBody` object as well), which will contain the properties that we can use to control the rocket's movements. For more detail, have a look at the following website:

http://docs.unity3d.com/Documentation/Components/class-ConstantForce.html

In this script, we also checked whether the rocket collides with other objects in the scene. This will trigger the script to instantiate the **Explosion** particle on the point that it collided by using **ContactPoint**. Then, we created the `FireTrail` particle and attached to the `rocket` object.

Finally, we created the `Smoke` particle and attached it to the gun barrel, which will be triggered when the player shoots and this will be done in the next step.

Classified intel

In this section, we used the `Instantiate()` function to clone a new game object from the `prefab` object in the **Project** view. The `Instantiate()` function takes three parameters, which are the original `Object`, `Position (Vector3)`, and `Rotation (Quaternion)`. The `Position` and `Rotation` objects are the transform parameters at the start position of the object, which will be created in the scene. The `Instantiate()` function can take any kind of object, and the result can also be returned to any kind of object. We can also see more examples and details from the Unity document at the following links:

- http://docs.unity3d.com/Documentation/ScriptReference/Object.Instantiate.html
- http://docs.unity3d.com/Documentation/Manual/InstantiatingPrefabs.html

Next, we will talk about the `Invoke()` function, which we used to call the function after the time we have set in seconds. If some of you have experience with ActionScript or JavaScript, this function is very similar to the `setTimeOut()` function. We can also use `InvokeRepeating()` to call a method similar to the `Invoke()` function, but this function will repeat calling that function every time we set the time in seconds. We can see more details of the `Invoke()` function from the Unity document at the following website:

http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Invoke.html

To know more about the `InvokeRepeating()` function, refer to the following website:

http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.InvokeRepeating

Creating a rocket launcher and RocketUI

In this final step, we will create the `RocketLauncher` and `RocketUI` scripts to make our robot actually shoot the `rocket` object, which we created in the previous step, by adding a script in our `CharacterControl` script to trigger them.

Engage thrusters

Let's get started. To create the `RocketLauncher` and `RocketUI` scripts, perform the following steps:

1. First, in the **Hierarchy** view, we will create the `RocketUI` prefab by going to **GameObject** | **Create Other** | **GUI Texture**; name it `RocketUI` and set it as follows:

Transform	
Position	**X**: `0.9`, **Y**: `0.08`, and **Z**: `0`
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `0`, **Y**: `0`, and **Z**: `1`
GUITexture	
Texture	**rocketUI** (located in the `Resources/UI` folder in the **Project** view)
Pixel Inset	**X**: `-64`, **W**: `128`, **Y**: `-32`, and **H**: `64`

2. Now, we will create the `Text` and `Shadow` objects by going to **GameObject** | **Create Other** | **GUI Text** twice; name the first object `Text` and the second one `Shadow`. Then, in the **Hierarchy** view, we need to drag the **Shadow** object inside the **Text** object and the **Text** object inside the **RocketUI** object, as shown in the following screenshot:

3. Then, we will click on the **Text** object, go to the **Inspector** view, and set it as follows:

Transform	
Position	**X**: `0`, **Y**: `0`, and **Z**: `0`
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `0`, **Y**: `0`, and **Z**: `0`
GUIText	
Text	`20`
Font	**Federation** (located in `Resources/Fonts` in the **Project** view)
Font Size	`25`

Font Style	**Bold**

4. Next, we will click on the **Shadow** object and go to the **Inspector** view and set it as follows:

Transform	
Position	**X**: `0`, **Y**: `0`, and **Z**: `-1`
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `0`, **Y**: `0`, and **Z**: `0`
GUIText	
Text	`20`
Pixel	**X**: `2`, **Y**: `-2`
Font	**Federation** (located in `Resources/Fonts` in the **Project** view)
Font Size	`25`
Font Style	**Bold**
Color	**R**: `0`, **G**: `0`, **B**: `0`, **A**: `255`, (black color)

5. Next, we will create the script to control our **RocketUI** object. Let's go to **Assets** | **Create** | **Javascript** (for Unity JavaScript users) or **Assets** | **Create** | **C#** (for C# users), name it `RocketUI`, double-click on it to launch **MonoDevelop**, and define the variables as follows:

```
// Unity JavaScript user:

#pragma strict

private var _guiTexts : GUIText[];

function Awake () {
  _guiTexts = GetComponentsInChildren.<GUIText>();
}

function UpdateUI ( ammo : int ) {
  for (var i : int = 0; i < _guiTexts.Length; ++i) {
    _guiTexts[i].text = ammo.ToString();
  }
}


// C# user:

using UnityEngine;
using System.Collections;
```

```
public class RocketUI : MonoBehaviour {
  GUIText[] _guiTexts;

  void Awake () {
    _guiTexts = GetComponentsInChildren<GUIText>();
  }

  public void UpdateUI ( int ammo ) {
    for (int i = 0; i < _guiTexts.Length; ++i) {
      _guiTexts[i].text = ammo.ToString();
    }
  }
}
```

6. Then, we go back to Unity and add the `RocketUI` script to the **RocketUI** object in **Hierarchy**.
7. In the **Hierarchy** view, we need to create the `RocketLauncher` object by going to **GameObject | Create Empty**; name it `RocketLauncher` and drag it inside our **gun_model** object, as shown in the following screenshot:

8. Now we will create the script to control our **RocketLauncher** object. Let's go to **Assets | Create | Javascript** (for Unity JavaScript users) or **Assets | Create | C#** (for C# users), name it `RocketLauncher`, double-click on it to launch **MonoDevelop**, and define the variables as follows:

```
// Unity JavaScript user:

#pragma strict

var smoke : ParticleSystem;
var rocket : ConstantForce;
var speed :float = 10f;
var ammoCount : int = 20;

private var lastShot : float = 0.0f;
private var _rocketUI : RocketUI;

function Start () {
  _rocketUI = FindObjectOfType(typeof(RocketUI));
}

public function Fire( reloadTime : float ) {
  if (Time.time > (reloadTime + lastShot) && ammoCount > 0) {
    var rocketPrefab : ConstantForce =
```

```
ConstantForce.Instantiate(rocket, transform.position,
transform.rotation) as ConstantForce;
    rocketPrefab.relativeForce = new Vector3(0, 0, speed);

    smoke.Play();

    //We ignore the collision between rocket and character
    Physics.IgnoreCollision(rocketPrefab.collider,
transform.root.collider);

    //Get the last shot time
    lastShot = Time.time;
    //Decrease the bullet
    ammoCount--;

    _rocketUI.UpdateUI(ammoCount);
  }
}

public function Reload () {
  ammoCount = 20;
  _rocketUI.UpdateUI(ammoCount);
}
```

// C# user:

```
using UnityEngine;
using System.Collections;

public class RocketLauncher : MonoBehaviour {

  public ParticleSystem smoke;
  public ConstantForce rocket;
  public float speed = 10f;
  public int ammoCount = 20;

  float lastShot = 0.0f;
  RocketUI _rocketUI;

  void Start () {
    _rocketUI = GameObject.FindObjectOfType<RocketUI>();
  }

  public void Fire( float reloadTime) {
    if (Time.time > (reloadTime + lastShot) && ammoCount > 0) {
      ConstantForce rocketPrefab = ConstantForce. Instantiate(rocket,
transform.position, transform.rotation)as ConstantForce;
      rocketPrefab.relativeForce = new Vector3(0, 0, speed);

      smoke.Play();

      //We ignore the collision between rocket and character
      Physics.IgnoreCollision(rocketPrefab.collider,
transform.root.collider);
```

```
      //Get the last shot time
      lastShot = Time.time;
      //Decrease the bullet
      ammoCount--;
      _rocketUI.UpdateUI(ammoCount);
    }
  }

  public void Reload () {
    ammoCount = 20;
    _rocketUI.UpdateUI(ammoCount);
  }
}
```

9. Finally, we will go back to Unity and add the `RocketLauncher` script to the **RocketLauncher** object in the **Hierarchy** view. Then, go to its **Inspector** view and set it as follows:

Transform	
Position	**X**: `6.5` **Y**: `0.25` **Z**: `0`
Rotation	**X**: `0` **Y**: `90` **Z**: `0`
Scale	**X**: `1` **Y**: `1` **Z**: `1`
Rocket Launcher (script)	
Smoke	**Smoke** (drag the **Smoke** object located in `robot/gun_model` in the **Hierarchy** view)
Rocket	**rocket** (drag the **rocket** prefab located in the `Resources/Prefabs` folder in the **Project** view)

We are done. Click on play to see the result!

Objective complete – mini debriefing

In this section, we created the `RocketUI` script to show the number of bullets left on the screen. The script will automatically update the number of bullets we have left when we click on **fire** or **reload**.

Next, we created the **RocketLauncher** object and script to fire the rocket, which will show the **Smoke** particle when we click on **fire** by getting the `playbackTime` value from the current animation using `AnimationStateInfo`. We get the time left for the current animation to finish by using `currentState.normalizedTime % 1` to calculate the remaining time and pass it to `BroadcastMessage ("Fire", playbackTime)` to trigger the `Fire` event.

Classified intel

In the `CharacterControl` script, we will see that we have used `BroadcastMessage("Fire", playbackTime)`. We will learn what the `BroadcastMessage()` function actually does.

BroadcastMessage

The `BroadcastMessage()` function basically calls all the functions named `Fire` in this game object or any of its children. This is a great way to make our script and object more organized.

Note

Performance-wise, `BroadcastMessage()` is slower than a function call because it iterates through all the possible target objects, finds matches of the desired function, and executes them. Therefore, it won't cause a huge increase in performance if we don't have a large number of function calls.

We can have different scripts attached to the children of this parent object and trigger them at the same time. For example, `BroadcastMessage("Fire", playbackTime)` will call the `Fire(var f:float)` or `Fire(float f)` function in each component attached to the object (irrespective of whether we're calling it on **Component** or **GameObject**). So, when the user clicks on **fire**, we will have the rocket shot at the same time with the smoke coming out from the launcher without having to code everything in one big script. We can see more details at the following links:

- http://docs.unity3d.com/Documentation/ScriptReference/Component.BroadcastMessa
- http://docs.unity3d.com/Documentation/ScriptReference/GameObject.BroadcastMess

Tip

If we use C#, we can avoid using the `BroadcastMessage()` function by using `delegate` and `event` to trigger all the objects that have an event listener attached. We will talk about `delegate` and `event` in Project 7, *Forge a Destructible and Interactive Virtual World*.

Mission accomplished

In this project, we created the aiming camera and laser, the same as the ones used in *Resident Evil* by adapting the old `CharacterControl` script. We also learned how to set up the camera for a third-person view while the character is aiming, created new `AnimatorController` to control each action, created a rocket launcher, created the `rocket` prefab, and used the Shuriken Particle system to create the smoke from the launcher barrel, smoke trial from the rocket, and the explosion.

Next, we also learned how to use the `Instantiate()` function to clone the game object and display it in the scene. We used `Invoke()` to call the function after the time that we assigned. Lastly, we created a UI to track the number of rockets left by using **GUITexture** and **GUIText**.

Let’s take a look at the following screenshot to see what we have done so far:

The preceding screenshot shows the camera view of the character. The following screenshot shows the character shooting:

20
18
19

Hotshot challenges

Now we know how to create the first person controller, the rocket launcher weapon using particles, and the shadow text by using **GUIText**. Let's make this project more fun by taking up the following challenges:

- Include your own character or even your own type of weapon
- Adjust the particle or use a different particle effect to create the smoke effect or explosion
- Add the ability to change the type of rocket or bullet; you can even have a different type of rocket that is slower or faster than the one created in this project or even add gravity to it and make it a grenade launcher
- Add sound for each action
- Add physics and explosions to our rocket when the rocket hits something
- Add an ammo to pickup

Chapter 6. Make AI Appear Smart

Creating AI can be the most difficult and complex task in the development of a game because we have to create a balance between intelligence with good game play environment restrictions and performance. The game AI is very different from the traditional AI in many ways because traditional AI tries to make it as close to the human brain as possible. On the other hand, the game AI cares more about the player experience. So, the workarounds and cheats are acceptable. In many cases, we will see the computer abilities have toned down to give the human players a sense of fairness. For example, we don't want to have the enemy kill players all the time without a chance to fight back.

Most games need AI for the enemy to be able to react to the player. The AI will run towards and attack the player, or it will jump or walk avoiding the obstacles, and so on. However, we have to be careful with the balance between making the AI smart and the performance speed to get the best moves. This is why the game AI and traditional AI are different. To get the best moves means more calculation, so it might cause a problem with performance slowing down.

Note

For more details on the comparison between the game AI and traditional AI, visit http://www.ai-blog.net/archives/000183.html.

We can use **A* algorithm** for the pathfinder or **Minimax algorithm** to calculate the best move, but these algorithms are very complex for a beginner.

Note

A* algorithm or A Star algorithm is a computer algorithm that is widely used in path finding and graph traversal nodes. Noted for its performance and accuracy, it enjoys widespread use. Peter Hart, Nils Nilsson, and Bertram Raphael first described the algorithm in 1968. It is an extension of Edsger Dijkstra's 1959 algorithm.

The reference is taken from http://en.wikipedia.org/wiki/A*_search_algorithm.

The following links provide some useful information:

- http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
- http://www.policyalmanac.org/games/aStarTutorial.htm

Minimax algorithm is a decision rule used in decision theory, game theory, statistics, and philosophy for minimizing the possible loss while maximizing the potential gain. Alternatively, it can be thought of as maximizing the minimum gain (maximin). Originally formulated for two-player zero-sum game theory, covering both the cases where players take alternate moves and those where they make simultaneous moves, it has also been extended to more complex games and to general decision making in the presence of uncertainty.

The reference is taken from http://en.wikipedia.org/wiki/Minimax.

The following links provide some useful information:

- http://ai-depot.com/articles/minimax-explained/
- http://www.stanford.edu/~msirota/soco/minimax.html

The following link contains useful information for the AI book: http://web.media.mit.edu/~jorkin/aibooks.html.

AI code is a lot to cover and can be written in a whole new book, but we will learn how to create a simple and easy way to make our AI look smart by using a simple method such as the random function instead of using search algorithms to get the possible moves of the enemies. It might not make our AI as smart as using those algorithms, but we will get the basic idea of how to create smart AI.

After the release of Unity 3.5, there is a new feature for AI called **NavMesh** (**Navigation Mesh**), which is a pathfinder system that calculates the walkable area by using the original mesh to create the new one, which is used to calculate the possible path for the AI. Even though the NavMesh feature is available in the free version after Unity 4.2, there is still the **Off Mesh Links** feature. This feature is used to check for the character when it isn't in the mesh area, such as when the character is jumping.

Note

For more information, visit http://docs.unity3d.com/Documentation/Manual/OffMeshLinks.html.

For a quick start tutorial on how to set up NavMesh, visit http://www.youtube.com/watch?v=TD11AzSQ0Ao.

In this chapter, we will reuse the scripts and assets from Project 5, *Build a Rocket Launcher!*, to implement the AI enemy. We will be creating an enemy by implementing the simple AI. However, this AI would be smart enough to detect when to jump, run, walk, stop, or shoot at the player by creating the waypoint for the enemy to walk to each point, run towards the player and then shoot when the player gets closer, and jump when it detects the wall.

Mission briefing

This project will start by setting up the `Waypoint` script. The waypoint is basically a path made from many points that AI will be following from the start to the end point. This script will allow us to control the direction and area where AI can be moved. We will also create the `WaypointsContainer` script and **CustomEditor** for it. This editor script will make it easy for us to add/remove and control how the AI should be moved, such as randomly moving or moving by order to each waypoint.

To make it easier for us to see each waypoint and its moving direction, we will use the `OnDrawGizmos()` function, which will allow us to see the wireframe, raycast line, icons, and the area while we are playing or editing the game in the **Scene** view. This is very powerful for debugging and editing the level in the game.

Next, we will create the AI character and its script, which will be inherited from `CharacterClass`. This class is basically the class that contains all the functions and parameters similar to the `CharacterControl` class that we have used in the Project 5, *Build a Rocket Launcher!*, with some additional methods. The AI script will be able to move the AI character to each waypoint, stop, walk, jump if it hits the wall, and run and shoot if the player is in the range.

Finally, we will add the hit-point UI bar for our character and the AI to show how much damage is caused when the enemy attacks us or when we shoot at the enemy.

Why is it awesome?

When we complete this project, we will be able to create a simple AI behavior that is smart enough to detect the player and respond to the player's reaction. From this project, we will begin to create smart AI for any kind of game. Most of the methods or equations in this project are very straightforward and are easy enough to use to create a simple AI and can be developed to make the AI smarter. For example, we can mix the waypoint script with the NavMesh to make our AI even more efficient.

Note

However, NavMesh is superior to waypoints in certain cases, especially where there are the corners to move around. Having the waypoint implies following only a single path, while the Navmesh gives more freedom to move, thereby appearing more real. For more information on Navmesh in Unity, visit http://docs.unity3d.com/Manual/Navigation.html.

We will also get the basic knowledge to create **CustomEditor**, which will be extremely helpful when we want to design the level in our game. This is the most powerful feature in Unity, which is very useful to create a level editor for our game.

Your Hotshot objectives

We will create the new `AI` script that is derived from the `CharacterClass` script, which has methods and parameters similar to the `CharacterControl` class that we have in Project 5, *Build a Rocket Launcher!*. This script will control the AI movement and behavior by implementing the new waypoint system to limit the movable area of our enemy. Then, we will create the hit-point UI for both player and enemy, as well as the **Restart** button when either one dies. The following are the steps that we will go through in this project:

- Creating the `Waypoint` and `WaypointsContainer` scripts
- Creating a custom editor for the `WaypointsContainer` script
- Creating the enemy movement with the AI script
- Creating a hit-point UI

Mission checklist

Before we start, we will need to get the project folder and assets from Packt's website, http://www.packtpub.com/support?nid=8267, which includes the package from the first project and the assets that we need to use in this project.

Navigate to the preceding URL and download the `Chapter6.zip` package and unzip it. Inside the `Chapter6` folder, there are two Unity packages, which are `Chapter6Package.unitypackage` (we will use this package for this project) and `Chapter6Package_Completed.unitypackage` (this is the completed project package).

Creating the Waypoint and WaypointsContainer scripts

In the first section, we will create the `Waypoint` and `WaypointsContainer` script to place the waypoint for our AI movement direction, which can be edited in the editor.

First, we will create the `Waypoint` script and use the `Gizmos` object to draw an image icon (`Gizmos.DrawIcon()`) and the wire sphere (`Gizmos.DrawWireSphere()`) by putting it in `OnDrawGizmos()`. Both functions are used to draw the image icon and the radius of waypoint, which is used to display the position of waypoint and the radius that the character should start turning to the next waypoint.

Then, we will create `WaypointsContainer` that will contain all necessary functions to get the direction of the current waypoint to the next one. We also use the line color (`Gizmos.DrawLine()`) to create the link of each waypoint in `OnDrawGizmos()`.

Prepare for lift off

We will begin with importing the package, preparing the `Assets` folder, and making sure that we have everything ready to start. Let's create a new project and import the package:

1. Import the assets package by going to **Assets | Import Package | Custom Package…**, choose `Chapter6.unityPackage`, which we downloaded earlier, and then click on the **Import** button in the pop-up window, as shown in the following screenshot:

2. Wait until it's done, and you will see the **Chapter6** and **Resources** folders in the Window view, then go to the **Chapter6** folder and drag the **Gizmos** folder outside the **Chapter6** folder, as shown in the following screenshot:

Why did we move the **Gizmos** folder outside the **Chapter6** folder? Refer to the following Unity document:
https://docs.unity3d.com/Documentation/ScriptReference/Gizmos.DrawIcon.html

We will see that the function `Gizmos.DrawIcon()` takes three parameters, which are `Vector3` for the position that the object will be drawn, `string` for the name of the icon image, and `boolean` to allow scaling. Then, the last sentence said the following:

> *"The image file should be placed in the **Assets/Gizmos** folder"*

This simply means that if we want to have our custom icon image, we basically need to put our image inside the folder mentioned earlier.

We can also show the gizmos icon in the **Scene** view by going to the **Inspector** view and clicking on the **Color Cube** image to choose the type of icon for that game object, as we can see in the following screenshot:

3. Next, go to **Chapter6** | **Scenes**; double-click on the scene (for C# users, double-click on **AI_C#**, and for JavaScript users, double-click on **AI_JS**) as shown in the following screenshot:

4. Then, go to **Chapter6** | **Scenes**; we will see the **C#** and **JavaScript** folder, (for C# users, we will go to **Chapter6** | **Scenes** | **C#** | **Waypoint**, and for JavaScript users, we will go to **Chapter6** | **Scenes** | **JavaScript** | **Waypoint**). Now, we are ready to start.

Engage thrusters

Let's get started:

1. Go to the folder (for C# users, go to **Chapter6** | **Scenes** | **C#** | **Waypoint** and for JavaScript users, go to **Chapter6** | **Scenes** | **JavaScript** | **Waypoint**) in the **Project** view, and then right-click and choose (for C# users) **Create** | **C# Script** or (for JavaScript users) **Create** | **JavaScript** and rename it to `Waypoint`.
2. Double-click on the **Waypoint** script to open it in **MonoDevelop** and start creating the waypoint, which will only have the `OnGizmos()` function to show the icon image and the wire sphere for the radius of the waypoint as follows:

```
// Unity JavaScript user:

#pragma strict

private var _radius : float;
private var _showGizmos : boolean;

function OnDrawGizmos ()
{

  if (transform.parent != null) {
     if (transform.parent.GetComponent.<WaypointsContainer>() != null)
{
       _showGizmos = transform.parent.GetComponent.<WaypointsContainer>
().showPath;
       if (_showGizmos) {
         _radius = transform.parent.GetComponent.<WaypointsContainer>
().radius;
         Gizmos.color = Color.green;
         Gizmos.DrawIcon(transform.position, "wayIcon.psd");
         Gizmos.DrawWireSphere (transform.position, _radius);
       }
     }
  }
}


// C# user:

using UnityEngine;
using System.Collections;

public class Waypoint : MonoBehaviour
{
  float _radius;
  bool _showGizmos;

  void OnDrawGizmos ()
  {
    if (transform.parent != null) {
       if (transform.parent.GetComponent<WaypointsContainer>() != null)
{
```

```
        _showGizmos = transform.parent.GetComponent<WaypointsContainer>
().showPath;
        if (_showGizmos) {
          _radius = transform.parent.GetComponent<WaypointsContainer>
().radius;
          Gizmos.color = Color.green;
          Gizmos.DrawIcon(transform.position, "wayIcon.psd");
          Gizmos.DrawWireSphere (transform.position, _radius);
        }
      }
    }
  }
}
```

Note

`OnDrawGizmos()` lets us draw the gizmos object, which will allow us to see the visual of the empty game object while in the editor. In this case, we use the `Gizmos.DrawIcon()` function to draw the icon image for each waypoint to make it easier to edit. Then, we use `Gizmos.DrawWireSphere()` to draw and calculate the area of each waypoint related to the radius of `WaypointContainer`.

We will see the error on the console that `WaypointsContainer` cannot be found, but don't worry, we will add it in the next step.

3. Next, we will create the script in the same folder and name it `WaypointsContainer`. This script will have the basic function that checks for the direction, distance, and draws the gizmos line between each waypoint. First, let's add the following code:

```
// Unity JavaScript user:

#pragma strict

var showPath : boolean = true;
var isRandom : boolean = false;
var radius : float = 1.0f;
var waypoints : Waypoint[];

private var _lastWaypoint : Waypoint;
private var _nextIndex : int;
private var _wayIndex : int;
private var _wayLength : int;
private var _isHitRadius : boolean;
private var _direction : Vector3;

function Awake ()
{
  showPath = false;
  _isHitRadius = false;
  _wayIndex = 0;
  _nextIndex = _wayIndex + 1;
  _wayLength = waypoints.Length;
  _direction = Vector3.zero;
}
```

```
// C# user:

using UnityEngine;
using System.Collections;

public class WaypointsContainer : MonoBehaviour
{
  public bool showPath = true;
  public bool isRandom = false;
  public float radius = 1.0f;
  public Waypoint[] waypoints;

  Waypoint _lastWaypoint;
  int _nextIndex;
  int _wayIndex;
  int _wayLength;
  bool _isHitRadius;
  Vector3 _direction;

  void Awake ()
  {
    showPath = false;
    _isHitRadius = false;
    _wayIndex = 0;
    _nextIndex = _wayIndex + 1;
    _wayLength = waypoints.Length;
    _direction = Vector3.zero;
  }
}
```

4. Next, we will add another function that basically checks if the enemy is away from the next waypoint or not. We will use this function to make sure that the enemy isn't going too far from the area, which will give our enemy more characteristics. Let's add the following code after the `Awake()` function:

```
// Unity JavaScript user:

function AwayFromWaypoint ( position : Vector3, distance : float ) :
boolean {
  var offset : Vector3 = position -
waypoints[_nextIndex].transform.position;
  var length : float = offset.sqrMagnitude;
  var sqrDistance : float = distance*distance;
  if (length > sqrDistance) {
    return true;
  } else {
    return false;
  }
}


// C# user:

public bool AwayFromWaypoint ( Vector3 position, float distance ) {
```

```
  Vector3 offset = position - waypoints[_nextIndex].transform.position;
  float length = offset.sqrMagnitude;
  float sqrDistance = distance*distance;
  if (length > sqrDistance) {
    return true;
  } else {
    return false;
  }
}
```

5. Then, we need to add another function, which will basically return the direction from our AI to the player. This function is to make our enemy follow the player to make it more aggressive. Let's add the function after the `AwayFromWaypoint()` function as follows:

```
// Unity JavaScript user:

function GetDirectionToPlayer ( enemy : Vector3, player : Vector3 ) :
Vector3 {
  var currentPosition : Vector3 = new Vector3(enemy.x,
waypoints[_wayIndex].transform.position.y, enemy.z);
  var playerPosition : Vector3 = new Vector3(player.x,
waypoints[_wayIndex].transform.position.y, player.z);
  _direction = (playerPosition - currentPosition).normalized;
  return _direction;


}
// C# user:

public Vector3 GetDirectionToPlayer ( Vector3 enemy, Vector3 player ) {
  Vector3 currentPosition = new Vector3(enemy.x,
waypoints[_wayIndex].transform.position.y, enemy.z);
  Vector3 playerPosition = new Vector3(player.x,
waypoints[_wayIndex].transform.position.y, player.z);
  _direction = (playerPosition - currentPosition).normalized;
  return _direction;
}
```

6. Next, we will add the core function, which will calculate and return the enemy's direction related to the player's position. We will also set the way the enemy will be moved through the waypoint either by an order or randomness. So, type the following code:

```
// Unity JavaScript user:

function GetDirection (myTransform : Transform) : Vector3
{
  var offset : Vector3 = myTransform.position -
waypoints[_nextIndex].transform.position;
  var length : float = offset.sqrMagnitude;
  var sqrDistance : float = radius*radius;
  if (length <= sqrDistance) {
    if (!_isHitRadius) {
      _isHitRadius = true;
```

```
      _wayIndex = _nextIndex;
      if (isRandom) {
        var _randomWay : int = Mathf.FloorToInt(Random.value *
_wayLength);
        if (_wayLength > 1) {
          while (_wayIndex == _randomWay) {
            _randomWay = Mathf.FloorToInt(Random.value * _wayLength);
          }
        }
        _nextIndex = _randomWay;
      } else {
        _nextIndex = (_nextIndex + 1) % _wayLength;
      }
    }
  } else {
    _isHitRadius = false;
  }
  var currentPosition : Vector3 = new Vector3 (myTransform.position.x,
waypoints[_nextIndex].transform.position.y, myTransform.position.z);
  _direction = (waypoints[_nextIndex].transform.position -
currentPosition).normalized;
  return _direction;
}
```

// C# user:

```
public Vector3 GetDirection (Transform myTransform)
{
  Vector3 offset = myTransform.position -
waypoints[_nextIndex].transform.position;
  float length = offset.sqrMagnitude;
  float sqrDistance = radius*radius;
  if (length <= sqrDistance) {
    if (!_isHitRadius) {
      _isHitRadius = true;
      _wayIndex = _nextIndex;
      if (isRandom) {
        int _randomWay = Mathf.FloorToInt(Random.value * _wayLength);
        if (_wayLength > 1) {
          while (_wayIndex == _randomWay) {
            _randomWay = Mathf.FloorToInt(Random.value * _wayLength);
          }
        }
        _nextIndex = _randomWay;
      } else {
        _nextIndex = (_nextIndex + 1) % _wayLength;
      }
    }
  } else {
    _isHitRadius = false;
  }
  Vector3 currentPosition = new Vector3 (myTransform.position.x,
waypoints[_nextIndex].transform.position.y, myTransform.position.z);
    _direction = (waypoints[_nextIndex].transform.position -
currentPosition).normalized;
```

```
  return _direction;
}
```

7. The last function of this script is the `OnDrawGizmos()` function, which will only be used in the editor or debugging process, similar to the one we use on the **Waypoint** script. We will use this function to draw the line direction between each waypoint. Let's add it as follows:

```
// Unity JavaScript user:

function OnDrawGizmos ()
{
  if ((waypoints != null) && (waypoints.Length > 1) && (showPath ==
true)) {
    if (isRandom) {
      for (var j : int = 0; j < waypoints.Length; ++j) {
        for (var k : int = j; k < waypoints.Length; ++k) {
          if ((waypoints[j] != null) && (waypoints[k] != null)) {
            Gizmos.color = Color.blue;
            Gizmos.DrawLine(waypoints[j].transform.position,
waypoints[k].transform.position);
          }
        }
      }
    } else {
      for (var point : Waypoint in waypoints) {
        if ((_lastWaypoint != null) && (point != null)) {
          if (point != null) {
            Gizmos.color = Color.blue;
            Gizmos.DrawLine (point.transform.position,
_lastWaypoint.transform.position);
          }
        }
        _lastWaypoint = point;
      }
    }
  }
}
// C# user:

void OnDrawGizmos ()
{
  if ((waypoints != null) && (waypoints.Length > 1) && (showPath ==
true)) {
    if (isRandom) {
      for (int j = 0; j < waypoints.Length; ++j) {
        for (int k = j; k < waypoints.Length; ++k) {
          if ((waypoints[j] != null) && (waypoints[k] != null)) {
            Gizmos.color = Color.blue;
         Gizmos.DrawLine(waypoints[j].transform.position,
waypoints[k].transform.position);
          }
        }
      }
    } else {
```

```
      foreach (Waypoint point in waypoints) {
        if ((_lastWaypoint != null) && (point != null)) {
          if (point != null) {
            Gizmos.color = Color.blue;
            Gizmos.DrawLine (point.transform.position,
_lastWaypoint.transform.position);
          }
        }
        _lastWaypoint = point;
      }
    }
  }
}
```

8. We use `Gizmos.DrawLine()` to draw the line between each waypoint.
9. Now, we are done with the `WaypointsContainer` and `Waypoint` script. Go back to the Unity editor to create the waypoint container game object by going to **GameObject | Create Empty** to create the empty game object and name it `WaypointsContainer`. Then, drag the **WaypointsContainer** script (that we just created) to this **WaypointsContainer** game object, and set the **Transform** properties as follows:

Position	**X**: `0`, **Y**: `0`, and **Z**: `0`
Rotation	**X**:`0`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `1`, **Y**: `1`, and **Z**: `1`

We can click on the little gear in the **Inspector** view and choose **Reset** to reset all to the default positions, as shown in the following screenshot:

10. Next, we need to create a waypoint. Let's create a new empty game object again. Go to **GameObject | Create Empty** to create the empty game object, name it `Waypoint`, and drag the **Waypoint** script to it. Then, we drag this object inside **WaypointsContainer**, which we already have in the scene, and reset its transform position as **X** to `0`, **Y** to `0`, and **Z** to `0`. We will see something similar to the following screenshot:

11. Then, we need to create more **Waypoint** game objects by pressing *Crtl* + *D* (in Windows) or *command* + *D* (on a Mac) five times to duplicate another five **Waypoint** game objects, and set all these objects' transform positions as follows:

Position	**X**: `5`, **Y**: `1.5`, and **Z**: `3`
Position	**X**: `4`, **Y**: `0`, and **Z**: `7`
Position	**X**: `0`, **Y**: `0`, and **Z**: `9`
Position	**X**: `-1`, **Y**: `0`, and **Z**: `6`
Position	**X**: `-1.5`, **Y**: `2`, and **Z**: `3`

If we take a look at the **Hierarchy** view, we will see something similar to the following screenshot:

12. Next, we will click on **WaypointsContainer** and go to its **Inspector** view, set **Size** under the **Waypoints** property to `6`, and drag all the **Waypoint** objects to the **Waypoints** array in the **Inspector** view, as the shown in the following screenshot:

13. Then, if we click on **WaypointsContainer** and go to its **Inspector** view, we will see the **Is Random** property. We can toggle it **On** or **Off** to enable the random movement of the AI, which will also show the result on the editor screen, as we can see in the following screenshot:

14. We can also toggle the **Show Path** parameter to turn the visual gizmos **On** or **Off**. Both results are controlled by the `OnDrawGizmos()` function, which we created in our script.

We have done this step. However, if we have many waypoints, we need to add the **Waypoint** objects to the array one by one. This sounds like a lot of work and sometimes it's inconvenient and takes so much time. So, in the next step, we will solve this problem by creating **CustomEditor** to do this for us.

Objective complete – mini debriefing

What we have done here is created the waypoint system that basically controls the movement of the enemy. We started by creating the **Waypoint** script, which gets the `showPath` and `radius` variable from the **WaypointsContainer** script. Then, we used those variables to show the gizmo objects, which are the icons and wire spheres in the `OnDrawGizmos()` function.

Next, we created the `WaypointsContainer` script that has all the necessary code to control the enemy movement. First, we have the `AwayFromWayPoint()` function that will check the distance between the enemy's current position to the next waypoint. In this function, we've used `myVector3.sqrMagnitude` to check for the distance. If we take a look at the Unity documentation, we can also use `Vector3.Distance()` or `myVector3.magnitude` to check for the distance between two positions. So, why did we use `sqrMagnitude` instead of others?

Let's take a close look at the equation of the `Vector3.Distance()` function:

```
Vector3 vector = new Vector3 (a.x – b.x, a.y – b.y, a.z – b.z);
float distance = Math.Sqrt(vector.x* vector.x+ vector.y* vector.y+
vector.z* vector.z);
```

As we can see, we need to find the difference between two vectors first, use the power of 2 to the result vector, and square root it.

Note

The difference between two vectors is the vector from the tail of the base vector to the head of the reverse of another vector. Then, the root of the square is just to calculate the magnitude of this vector.

Assuming `myVector3` is the difference between vectors, `a` and `b` like the `vector` parameter in the preceding script, then the following is the equation of `myVector.magnitude`:

```
float magnitude = Mathf.Sqrt(myVector.x* myVector.x+ myVector.y*
myVector.y+ myVector.z* myVector.z);
```

The equation of `myVector.sqrMagnitude` is as follows:

```
float sqrMagnitude = myVector.x* myVector.x+ myVector.y* myVector.y+
myVector.z* myVector.z;
```

As we can see, the difference is that `sqrMagnitude` doesn't need to calculate `Math.Sqrt` or the square root, which makes it faster to calculate.

So, if we want to compare the distance, `sqrMagnitude` is often the best choice because using the distance's power of 2 is a lot faster than the square root of the magnitude, as we can see in the following script:

```
If (myVector3.sqrMagnitude < distance*distance) { … }
```

Note

For more details, visit the following websites:

- http://docs.unity3d.com/Documentation/ScriptReference/Vector3-magnitude.html
- http://answers.unity3d.com/questions/384932/best-way-to-find-distance.html

Next, we added the `GetDirectionToPlayer()` function to find the direction between our enemy and the player by using the *y* position of the waypoint to make sure that the direction gets calculated on the XZ plane. Then, we add the `GetDirection()` function, which checks the distance between the enemy position and waypoint position, as shown in the following diagram:

We can see from the preceding diagram that if the distance between the enemy and the waypoint position is smaller than the radius of the waypoint, it will trigger the waypoint to change the next waypoint index, which will also change the movement direction of the enemy.

Next, we used the `OnDrawGizmos()` function to create the visual line link for each **Waypoint** game objects to show in the editor. **Waypoint** is the empty game object, which is sometimes difficult to edit in the editor because we cannot see it in the editors.

Tip

It is better to use gizmo than trying to use camera layers and meshes for the waypoint.

So, using gizmo is the best solution and the best way that Unity provides us to see the visual of an empty game object. We also have the trigger parameter in the **Inspector** view to turn the visual on or off and to tell our enemy to walk randomly or by the order of the waypoint index.

Classified intel

At the `Waypoint` script, we use the following lines in the `OnDrawGizmos()` function to get both values from the `WaypointsContainer` script as follows:

```
// Unity JavaScript user:

_showGizmos = transform.parent.GetComponent.<WaypointsContainer>
().showPath;
_radius = transform.parent.GetComponent.<WaypointsContainer>().radius;


// C# user:

_showGizmos = transform.parent.GetComponent<WaypointsContainer>().showPath;
_radius = transform.parent.GetComponent<WaypointsContainer>().radius;
```

This function is called **generic functions**, which is usually known to the C# user. However, for the JavaScript user, this might be a new thing. This is basically the function that can be called to return the strict type. It means that we don't need to cast the type, as we can see in the following pseudo code:

```
Function FunctionName.<T>() : T;
```

This is very useful if we want to get around the limitations of dynamic tying in JavaScript, as shown in the following code:

```
var obj = GetComponent.<Transform>();
```

From the preceding code, `obj` will be the `Transform` type. However, if we use `GetCompent(Transform)`, the `obj` type will be the `Component` type.

Tip

The good thing about generic functions is that it will return the correct type and the JavaScript code doesn't need to find the correct type when it compiles, which will make our code faster.

For the C# user, generic functions can help us save time to cast it to the correct type, as shown in the following script:

```
Transform t = go.GetComponent<Transform>();
```

We can use a generic function instead of using the following methods:

```
Transform t = (Transform) go.GetComponent(typeof(Transform));
Transform t = go.GetComponent(typeof(Transform)) as Transform;
```

We can see that it's a lot shorter.

Note

For more details, visit http://docs.unity3d.com/Documentation/Manual/GenericFunctions.html.

Gizmos

We've used the `OnDrawGizmos()` function to create the visual viewable for the waypoint game object, which will show only in the editor and we won't see anything during the game play in the **Game** view or in the real game after we build it.

However, if we want to see it while we are playing the game in the **Game** view, we can click on the **Play** button and click on the **Gizmos** button on the top-right to toggle the gizmos on or off, as shown in the following screenshot:

Creating a custom editor for the WaypointsContainer script

In this step, we will create the `WaypointsContainerEditor` script, which will create the custom inspector for our `WaypointsContainer` component. We will have the **Slider** bar to limit the minimum and maximum radius of each the waypoint, the **Add** and **Remove** waypoint buttons that automatically add and remove the waypoint from the scene, and finally, we will update the name of each waypoint automatically when adding or removing from the scene to make it easier for us to edit.

Prepare for lift off

We will begin by removing the old **WaypointsContainer** object in the **Hierarchy** view:

1. Let's right-click on the **WaypointsContainer** game object that we created in the previous step and choose **Delete** to remove it from the scene, as we can see in the following screenshot:

2. Next, we will create the new container, but this time, we will have a bit of a different setup, which will be prepared for the next step. Let's go to **GameObject | Create Empty** to create a new empty game object and name it `Enemy`. Then, we will set its position as **X** to `0.5`, **Y** to `0.25`, and **Z** to `-5`.
3. Then, we create another empty game object; go to **GameObject | Create Empty**, name it `WaypointsContainer`, drag it inside the **Enemy** game object, and set its **Position** as **X** to `0`, **Y** to `0`, and **Z** to `0`, as shown in the following screenshot:

4. Finally, we need to create the **Editor** folder in the **Project** view and create the C# (for C# user) or JavaScript (for JavaScript user) script and name it `WaypointsContainerEditor`, as we can see in the following screenshot:

This will tell Unity that this script is an editor script and will compile after our game script, which allows us to get access to **WaypointsContainer**.

Note

For more information on the script compilation order, visit http://docs.unity3d.com/Documentation/Manual/ScriptCompileOrderFolders.html.

Now, we are ready to begin.

Engage thrusters

Let’s get started:

1. Double click on **WaypointsContainerEditor** to open it and type the following code:

```
// Unity JavaScript user:

#pragma strict
@CustomEditor(WaypointsContainer)

class WaypointsContainerEditor extends Editor {
  private final var OBJECT_NAME : String = "Waypoint";
  private final var s_showPath : String = "showPath";
  private final var s_random : String = "isRandom";
  private final var s_radius : String = "radius";
  private final var s_arraySizePath : String = "waypoints.Array.size";
  private final var s_arrayData : String = "waypoints.Array.data[{0}]";

  private var seo_object : SerializedObject;
  private var sep_radius : SerializedProperty;
  private var sep_showGizmo : SerializedProperty;
  private var sep_random : SerializedProperty;
  private var sep_waypointCount : SerializedProperty;

  private var _wayPointsContainer : WaypointsContainer;

  function OnEnable ()
  {
    seo_object = new SerializedObject (target);
    sep_showGizmo = seo_object.FindProperty (s_showPath);
    sep_random = seo_object.FindProperty (s_random);
    sep_radius = seo_object.FindProperty (s_radius);
    sep_waypointCount = seo_object.FindProperty (s_arraySizePath);
    _wayPointsContainer = seo_object.targetObject as
WaypointsContainer;
  }
}


// C# user:

using UnityEngine;
using UnityEditor;
[CustomEditor(typeof(WaypointsContainer))]
public class WaypointsContainerEditor : Editor
{
  const string OBJECT_NAME = "Waypoint";
  const string s_showPath = "showPath";
  const string s_random = "isRandom";
  const string s_radius = "radius";
  const string s_arraySizePath = "waypoints.Array.size";
  const string s_arrayData = "waypoints.Array.data[{0}]";

  private SerializedObject seo_object;
```

```
  private SerializedProperty sep_radius;
  private SerializedProperty sep_showGizmo;
  private SerializedProperty sep_random;
  private SerializedProperty sep_waypointCount;

  WaypointsContainer _wayPointsContainer;

  void OnEnable ()
  {
    seo_object = new SerializedObject (target);
    sep_showGizmo = seo_object.FindProperty (s_showPath);
    sep_random = seo_object.FindProperty (s_random);
    sep_radius = seo_object.FindProperty (s_radius);
    sep_waypointCount = seo_object.FindProperty (s_arraySizePath);
    _wayPointsContainer = seo_object.targetObject as
WaypointsContainer;
  }
}
```

We have used `SerializeObject` and `SerializeProperty` to access the property from `WaypointsContainer` script and set all properties.

2. Next, we will add the functions to set, get, add, and remove waypoints. Let's add the following code after the `OnEnable()` function:

```
// Unity JavaScript user:

function GetWaypointArray () : Waypoint[]
{
  var waypoints : Waypoint[] =
_wayPointsContainer.GetComponentsInChildren.<Waypoint>();
  return waypoints;
}
function GetWaypointAtIndex (index : int) : Waypoint
{
  return seo_object.FindProperty (String.Format (s_arrayData,
index)).objectReferenceValue as Waypoint;
}
function SetWaypoint (index : int, waypoint : Waypoint)
{
  if (waypoint != null) {
    Undo.RecordObject(waypoint.gameObject,"Update"+index.ToString());
    var nameIndex : String = (index < 9) ? "0"+(index+1).ToString() :
(index+1).ToString();
    waypoint.name = OBJECT_NAME+nameIndex;
    seo_object.FindProperty (String.Format (s_arrayData,
index)).objectReferenceValue = waypoint;
  }
}
function RemoveWaypointAtIndex (index : int)
{
  var arrayCount : int = sep_waypointCount.intValue;
  for (var i : int = index; i < arrayCount - 1; i++) {
    SetWaypoint (i, GetWaypointAtIndex (i + 1));
  }
  if (GetWaypointAtIndex(index) != null) {
```

```
    var go : GameObject = GetWaypointAtIndex(index).gameObject;
    Undo.DestroyObjectImmediate(go);
  }
}
function AddWayPoint ( waypoint : Waypoint)
{
  sep_waypointCount.intValue++;
  SetWaypoint (sep_waypointCount.intValue - 1, waypoint);}
// C# user:

Waypoint[] GetWaypointArray ()
{
  Waypoint[] waypoints =
_wayPointsContainer.GetComponentsInChildren<Waypoint>();
  return waypoints;
}
Waypoint GetWaypointAtIndex (int index)
{
  return seo_object.FindProperty (string.Format (s_arrayData,
index)).objectReferenceValue as Waypoint;
}
void SetWaypoint (int index, Waypoint waypoint)
{
  if (waypoint != null) {
    Undo.RecordObject(waypoint.gameObject,"Update"+index.ToString());
    string nameIndex = (index < 9) ? "0"+(index+1).ToString() :
(index+1).ToString();
    waypoint.name = OBJECT_NAME+nameIndex;
    seo_object.FindProperty (string.Format (s_arrayData,
index)).objectReferenceValue = waypoint;
  }
}
void RemoveWaypointAtIndex (int index)
{
  int arrayCount = sep_waypointCount.intValue;
  for (int i = index; i < arrayCount - 1; i++) {
    SetWaypoint (i, GetWaypointAtIndex (i + 1));
  }
  if (GetWaypointAtIndex(index) != null) {
    GameObject go = GetWaypointAtIndex(index).gameObject;
    Undo.DestroyObjectImmediate(go);
  }
}
void AddWayPoint (Waypoint waypoint)
{
  sep_waypointCount.intValue++;
  SetWaypoint (sep_waypointCount.intValue - 1, waypoint);
}
```

3. Then, we will add the `OnInspectorGUI()` function, which acts in a similar way to the `OnGUI()` function, but this function updates every time we go to the inspector. Let's add the following code after the `AddWayPoint()` function:

```
// Unity JavaScript user:

function OnInspectorGUI ()
```

```
{
  if ((targets != null) && (target != null)) {
    seo_object.Update ();
    sep_showGizmo.boolValue = EditorGUILayout.Toggle ("Show Gizmos",
sep_showGizmo.boolValue);
    if (sep_showGizmo.boolValue) {
      sep_random.boolValue = EditorGUILayout.Toggle ("Random Path",
sep_random.boolValue);
    }
    EditorGUILayout.Slider(sep_radius, 1.0f, 3.0f, "Way Point Radius");
    GUILayout.Label ("Waypoints", EditorStyles.boldLabel);
    var waypoints : Waypoint[] = GetWaypointArray ();
    sep_waypointCount.intValue = waypoints.Length;
    for (var i : int = 0; i < waypoints.Length; i++) {
      GUILayout.BeginHorizontal ();
      var result : Waypoint = EditorGUILayout.ObjectField
(waypoints[i], typeof(Waypoint), true) as Waypoint;
      if (GUI.changed) {
        SetWaypoint (i, result);
      }
      if (GUILayout.Button ("-")) {
        RemoveWaypointAtIndex (i);
      }
      GUILayout.EndHorizontal ();
    }
    if (GUILayout.Button("Add Waypoint")) {
      var indexString : String = (waypoints.Length < 9) ? "0"+
(waypoints.Length+1).ToString() : (waypoints.Length+1).ToString();
      var go : GameObject = new GameObject(OBJECT_NAME+indexString);
      go.transform.parent = _wayPointsContainer.transform;
      go.transform.localPosition = Vector3.zero;
      var waypoint : Waypoint = go.AddComponent.<Waypoint>();
      AddWayPoint (waypoint);
      Undo.RegisterCreatedObjectUndo(go,"AddWaypointButton");
   }
    seo_object.ApplyModifiedProperties ();
  }
}


// C# user:

public override void OnInspectorGUI ()
{
  if ((targets != null) && (target != null)) {
    seo_object.Update ();
    sep_showGizmo.boolValue = EditorGUILayout.Toggle ("Show Gizmos",
sep_showGizmo.boolValue);
    if (sep_showGizmo.boolValue) {
      sep_random.boolValue = EditorGUILayout.Toggle ("Random Path",
sep_random.boolValue);
    }
    EditorGUILayout.Slider(sep_radius, 1.0f, 3.0f, "Way Point Radius");
    GUILayout.Label ("Waypoints", EditorStyles.boldLabel);
    Waypoint[] waypoints = GetWaypointArray ();
    sep_waypointCount.intValue = waypoints.Length;
```

```
    for (int i = 0; i < waypoints.Length; i++) {
      GUILayout.BeginHorizontal ();
      Waypoint result = EditorGUILayout.ObjectField (waypoints[i],
typeof(Waypoint), true) as Waypoint;
      if (GUI.changed) {
        SetWaypoint (i, result);
      }
      if (GUILayout.Button ("-")) {
        RemoveWaypointAtIndex (i);
      }
      GUILayout.EndHorizontal ();
    }
    if (GUILayout.Button("Add Waypoint")) {
      string indexString = (waypoints.Length < 9) ? "0"+
(waypoints.Length+1).ToString() : (waypoints.Length+1).ToString();
      GameObject go = new GameObject(OBJECT_NAME+indexString);
      go.transform.parent = _wayPointsContainer.transform;
      go.transform.localPosition = Vector3.zero;
      Waypoint waypoint = go.AddComponent<Waypoint>();
      AddWayPoint (waypoint);
      Undo.RegisterCreatedObjectUndo(go,"AddWaypointButton");
    }
    seo_object.ApplyModifiedProperties ();
  }
}
```

Here, we have created the `boolean` properties to toggle the show/hide gizmos and enabled/disabled a random waypoint and the `Slider` property to adjust the waypoint radius, and added a button to add a new waypoint to the scene.

4. Before we finish this step, we need to add the one last function that will make the update when we remove the waypoint object or create the object in the **Hierarchy** view. So, let's add the following function after `OnInspectorGUI()`:

```
//Unity JavaScript user:

function UpdateHierarchy ()
{
  if ((targets != null) && (target != null)) {
    seo_object.Update();
    var waypoints : Waypoint[] = GetWaypointArray();
    sep_waypointCount.intValue = waypoints.Length;
    for (var i : int = 0; i < sep_waypointCount.intValue; i++) {
      Undo.RecordObject(waypoints[i].gameObject,"Update"+i.ToString());
      SetWaypoint(i,waypoints[i]);
    }
    seo_object.ApplyModifiedProperties();
  }
}


//C# user:

void UpdateHierarchy ()
{
```

```
  if ((targets != null) && (target != null)) {
    seo_object.Update();
    Waypoint[] waypoints = GetWaypointArray();
    sep_waypointCount.intValue = waypoints.Length;
    for (int i = 0; i < sep_waypointCount.intValue; i++) {
      SetWaypoint(i,waypoints[i]);
    }
    seo_object.ApplyModifiedProperties();
  }
}
```

5. Then, we need to go back to the `OnEnable()` function and add one line of code that will call this function when the **Hierarchy** view has changed. Let's add the highlighted code as follows:

```
// Unity JavaScript user:

function OnEnable ()
{
  …
  EditorApplication.hierarchyWindowChanged = UpdateHierarchy;
}


// C# user:

void OnEnable ()
{
  …
  EditorApplication.hierarchyWindowChanged = UpdateHierarchy;
}
```

6. Now, let's finish our code. We can go back to the Unity editor and add the `WaypointsContainer` script (not the `WaypointsContainerEditor`) to the `WaypointsContainer` object in the **Hierarchy** view. We will see in the following screenshot that the inspector has changed from the way it was:

7. Next, we will click on the **Add Waypoint** button six times. Go to its **Inspector** view

and set each position values as follows:

Position	**X**: `0`, **Y**: `0`, and **Z**: `0`
Position	**X**: `5`, **Y**: `0`, and **Z**: `5`
Position	**X**: `4`, **Y**: `0`, and **Z**: `12`
Position	**X**: `0`, **Y**: `0`, and **Z**: `11`
Position	**X**: `-1.6`, **Y**: `2`, and **Z**: `8`
Position	**X**: `-2`, **Y**: `0`, and **Z**: `3.5`

We will see something like the following screenshot:

Now, we have finished this step. We can also play around by clicking the **–** button to remove the waypoint or delete it from the **Hierarchy** view. This will automatically update the name of the waypoint and reorder each one.

Tip

To create a new waypoint alternatively, we can do it by pressing *Crtl* + *D* (in Windows) or *command* + *D* (on Mac) on the last one in the **Hierarchy** view. This will also update the name and reorder the waypoint; this happens by setting the `EditorApplication.hierarchyWindowChanged()` function.

For more details, visit http://docs.unity3d.com/Documentation/ScriptReference/EditorApplication-hierarchyWindowChanged.html.

We can also change the waypoint radius, show/hide the gizmos, make it random, move the waypoint object around to serve what we need, or even add or remove the waypoint

objects to fit our level.

Tip

The waypoint script will not work properly if we put the waypoint where the enemy can’t walk through, which means that our enemy should be able to walk through and touch each waypoint (hit the green wire sphere area of each waypoint, as shown in the preceding screenshot). Otherwise, the enemy won’t be able to move to the next waypoint.

We can also adjust the radius (you will see the green wire sphere change its size) in the waypoint radius, which will make our enemy start turning to the next waypoint faster or slower. However, we should be careful when adjusting the radius. If we set it too low, the character might not hit it and not turn to the next waypoint. That’s why we have set the minimum radius equal to `1.0` and maximum equal to `3.0` in the `OnInspectorGUI()` function at the line `EditorGUILayout.Slider(sep_radius, 1.0f, 3.0f, "Way Point Radius");`.

In the next step, we will continue by creating the AI script to make our enemy walk through each waypoint.

Objective complete – mini debriefing

First, we use `@CustomEditor(WaypointsContainer)` (in JavaScript) or `[CustomEditor(typeof(WaypointsContainer))]` (in C#) to tell Unity that this custom editor will be based on the `WaypointsContainer` script and we inherited the class from the `Editor` class.

Next, we just added `SerializeObject` and `SerializeProperty`, which allows us to access the property from the `WaypointsContainer` script.

Tip

We can also use `serializedObject` instead of the new `SerializedObject (target)` too. The result is similar; however, in our case, we need to get the serialized object from the target to make sure that when we delete the object in the **Hierarchy** view, it won't return `null`.

Then, we use the `OnEnable()` function to initialize all the properties. `OnEnable()` is the function that is called every time that the game object becomes enabled and active. This function is opposite to `OnDisable()`, which will be called when the game object is disabled and not active.

Note

We can use this to set the parameters or reset it when we enabled or disabled the game object. To toggle the object that is disabled or enabled, we can use `gameObject.SetActive(true/false)` or check or uncheck the box in front of the game object name in the **Inspector** view.

For more details, visit the following links:

- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnEnable.h
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnDisable.l
- http://docs.unity3d.com/Documentation/ScriptReference/GameObject.SetActive.html

Next, we've created the function, which will get the `waypoints` array from the children of this `_waypointsContainer` object. Then, we write the following line to find the waypoint object using the following index:

```
seo_object.FindProperty (string.Format (s_arrayData, index))
```

Note

`FindProperty` allows us to pass the name of the object and return the `SerializedProperty` object. In this case, we need to access the array, so we use the `string.Format()` function and pass the string and index to get the result string such as `waypoints.Array.data[index]`.

For more information on `string.Format()`, visit http://msdn.microsoft.com/en-us/library/system.string.format(v=vs.110).aspx.

At the `SetWaypoint()` function, we've updated the name of waypoint by index, set it to

the `waypoints` array, and used `Undo.RecordObject()` to record the object information for the `undo` command. In the `RemoveWaypointAtIndex()` function, we basically removed the waypoint by index, saved the information of the object for undo, and destroyed the object by using `Undo.DestroyObjectImmediate()`.

Then in the `AddWayPoint()` function, we just increase the size of the array and add the new waypoint object.

Next, we created the `OnInspectorGUI()` function and applied all the properties by using the `EditorGUILayout` object to access each type of the object. This object is similar to the `GUILayout` object, and it allows us to get and set the property.

Note

For more details, visit the following links:

- http://docs.unity3d.com/Documentation/ScriptReference/Editor.OnInspectorGUI.html
- http://docs.unity3d.com/412/Documentation/ScriptReference/EditorGUILayout.html

Here, we have created the `boolean` properties to toggle the show/hide gizmos and enabled/disabled random waypoint and the `Slider` property to adjust the waypoint radius, and added the button to add a new waypoint to the scene.

Finally, we’ve created the `UpdateHierarchy()` function and added the following line in the `OnEnable()` function to tell Unity to call the `UpdateHierarchy()` function every time something changes in the **Hierarchy** view:

```
EditorApplication.hierarchyWindowChanged = UpdateHierarchy;
```

Classified intel

In the Unity editor, there is an old way to access the property from the `Editor` class. We can either use the `target` keyword to access all the properties directly from the editor or the `SerializeProperty` class and the `SerializeObject` keyword to access the property.

One way is to use `target` as follows:

```
function OnInspectorGUI() {
  target.radius = EditorGUILayout.FloatField("Radius:", target.radius);
  if (GUI.changed) {
    EditorUtility.SetDirty(target);
  }
}
```

On the other hand, we use the `SerializedObject` keyword and the `SerializedProperty` class to access the property, as we can see in the following code:

```
SerializedProperty sep_radius;
function OnEnable() {
  sep_radius = serializedObject.FindProperty("radius");
}
function OnInspectorGUI() {
  serializedObject.Update();
  sep_radius.floatValue = EditorGUILayout.FloatField("Radius:",
sep_radius.floatValue);
  serializedObject.ApplyModifiedProperties();
}
```

The `target` keyword can be used to access to the radius property directly by typing `target.radius`. Then, we checked if the GUI has changed or not by using `GUI.changed`. Then, we update the property by using `EditorUtility.SetDirty(target);`.

We use the `SerializedProperty` class and the `serializedObject` keyword to set the serialize property in `OnEnable()`:

Note

To use `SerializedProperty`, we need to put the property between `serializedObject.Update();` and `serializedObject.ApplyModifiedProperties();`. This will make sure that the property is updated and saves it when we have change the value.

Using `SerializedProperty` seems to be more complex than the target, so why are we using it? The advantage of `SerializedProperty` is that it will handle the multi-object editing, undo, and prefab override for us, to which we can attach `@CanEditMultipleObjects` (in JavaScript) or `[CanEditMultipleObjects]` (in C#) to make our script editor multiple objects.

If we copy our current `WaypointsContainer` script and select both the objects, we will see the message in the inspector say **Multi-object editing not supported** as we can see in the following screenshot:

To make it work, we just add the line `@CanEditMultipleObjects` (in JavaScript) or `[CanEditMultipleObjects]` (in C#) in the `WaypointsContainerEditor` script before the class. Let's have a look at the following line:

```
seo_object = new SerializedObject (target);
```

We change this line to the following:

```
seo_object = new SerializedObject (targets);
```

Otherwise, replace the keyword `seo_object` with `serializedObject` and remove the `seo_object` property.

Basically, we were getting the last selection object by using `target`, but if we change it to `targets` or `serializedObject`, it will return all the objects that have been selected.

Tip

To make the multi-object editing work properly with all the properties in **WaypointsContainerEditor**, we might have to change the way the code is structured. For example, we probably need to change the `GetWaypointAtIndex()` function to return `Waypoint[]` for each `WaypointsContainer` scripts.

Creating the enemy movement with the AI script

In the previous section, we have the waypoint set up for our enemy to move, but we don't have the enemy yet. In this section, we will create the script to control it by using a concept that was used in Project 5, *Build a Rocket Launcher!*. This AI script will inherit from **CharacterClass** that comes with the package.

Note

If we open **CharacterClass**, which is located at **Chapter6 | Scenes | C# | Actor** (for C# users) or **Chapter6 | Scenes | JavaScript | Actor** (for JavaScript users), we will see that there is a class that contains the methods and parameters that we have used in Project 5, *Build a Rocket Launcher!*.

Prepare for lift off

We will begin by adding the AI character to the **Hierarchy** view. Let's go to **Resources | Prefabs** in the **Project** view and drag **robotAI_C#** (for C# users) or **robotAI_JS** (for JavaScript users) inside the **Enemy** game object in the **Hierarchy** view, as shown in the following screenshot:

Engage thrusters

Now, we will create the AI script to control our character:

1. Go to the folder (for C# users, **Chapter6** | **Scenes** | **C#** | **Actor** or for JavaScript users, **Chapter6** | **Scenes** | **JavaScript** | **Actor**) in the **Project** view, right-click and choose (for C# user) **Create** | **C# Script** or (for JavaScript users) **Create** | **JavaScript**, and then rename it to `AI`.
2. Double-click on the AI script to open it in MonoDevelop and start by creating the AI class that inherits from `CharacterClass` as follows:

```
// Unity JavaScript user:

#pragma strict

public class AI extends CharacterClass {
  var shotRange : float = 15.0f;
  var playerRange : float = 5.0f;
  var distanceToShot : float = 10.0f;
  var walkingTime : float = 6.0f;
  var thinkingTime : float = 3.0f;
  var waypointsContainer : WaypointsContainer;

  private var _lastTime : float = 0f;
  private var _angle : float;
  private var _angleVelocity : float;
  private var _isThinking : boolean;

  protected override function Start () {
    super.Start();
    if (laser != null) {
      laser.gameObject.SetActive(false);
      laser.SetPosition(1,new Vector3(0f,0f,GUN_LASER_DISTANCE));
    }
  }
}


// C# user:

using UnityEngine;
using System.Collections;

public class AI : CharacterClass {
  public float shotRange = 15.0f;
  public float playerRange = 5.0f;
  public float distanceToShot = 10.0f;
  public float walkingTime = 6.0f;
  public float thinkingTime = 3.0f;
  public CSharp.WaypointsContainer waypointsContainer;

  float _lastTime = 0f;
  float _angle;
  float _angleVelocity;
```

```
  bool _isThinking;

  protected override void Start () {
    base.Start();
    if (laser != null) {
      laser.gameObject.SetActive(false);
      laser.SetPosition(1,new Vector3(0f,0f,GUN_LASER_DISTANCE));
    }
  }
}
```

With the preceding code, we basically created all the necessary parameters for the AI script. We also override the start function and set the laser length on the *y*-axis, which is a bit different to our `CharacterControl` script in Chapter 5, *Build a Rocket Launcher!*.

3. Next, we will create four functions to give our enemy a personality and make it smarter:
   - The first function is the `CanShoot()` function, which will make the enemy shoot when the player is within their shooting range by checking the distance of the player and enemy. We will also use the `Physics.Raycast()` function to see if there is anything blocking the direction of the shot; if there isn't, we just make the enemy shoot by adding the following code:

```
// Unity JavaScript user:

function CanShoot () : boolean {
  var offset : Vector3 = targetLookat.position -
transform.position;
  var length : float = offset.sqrMagnitude;
  var range : float = shotRange * shotRange;
  var hit : RaycastHit;
  if (length <= range) {
    if (Physics.Raycast(transform.position, offset.normalized, hit,
shotRange)) {
      if (hit.transform.CompareTag("Player")) {
        return true;
      }
    }
  }
  return false;
}


// C# user:

public bool CanShoot () {
  Vector3 offset = targetLookat.position - transform.position;
  float length = offset.sqrMagnitude;
  float range = shotRange * shotRange;
  RaycastHit hit ;
  if (length <= range) {
    if (Physics.Raycast(transform.position, offset.normalized, out
hit, shotRange)) {
```

```
      if (hit.transform.CompareTag("Player")) {
        return true;
      }
    }
  }
  return false;
}
```

- Secondly, we will create the `Jump()` function to make our enemy smarter by using the `Physics.Raycast()` function, but this time, we will also check the wall height. If the wall is higher than the limited height, our character will not jump. If not, it will jump over and continue walking towards its direction, as in the following code:

```
// Unity JavaScript user:

function Jump ( direction : Vector3 ) : boolean {
  var hit : RaycastHit;
  var up : Vector3 = Vector3.up * (-
_characterController.height*0.5f);
  var leg : Vector3 = transform.position +
_characterController.center + up;
  var distance : float = _characterController.radius * 2;
  if (Physics.Raycast(leg, direction, hit, distance)) {
    if (hit.transform.CompareTag("Wall")) {
      var height : float = hit.collider.bounds.max.y - hit.point.y;
      if (height <= 2.5f) {
        return true;
      }
    }
  }
  return false;
}


// C# user:

public bool Jump ( Vector3 direction ) {
  RaycastHit hit;
  Vector3 up = Vector3.up * (-_characterController.height*0.5f);
  Vector3 leg = transform.position + _characterController.center +
up;
  float distance = _characterController.radius * 2;
  if (Physics.Raycast(leg, direction, out hit, distance)) {
    if (hit.transform.CompareTag("Wall")) {
      float height = hit.collider.bounds.max.y - hit.point.y;
      if (height <= 2.5f) {
        return true;
      }
    }
  }
  return false;
}
```

- Then, we will check the distance between the player and the enemy to see if the

distance is higher than `shotRange` and lower than `shotRange + playerRange`. So, let's add it as follows:

```
// Unity JavaScript user:

function Run () : boolean {
  var distanceToPlayer : float = (targetLookat.position -
transform.position).sqrMagnitude;
  var runDistance : float = (playerRange+shotRange)*
(playerRange+shotRange);
  var shotDistance : float = shotRange*shotRange;
  if ((distanceToPlayer <= runDistance) && (distanceToPlayer >
shotDistance)) {
    return true;
  }
  return false;
}


// C# user:

public bool Run () {
  float distanceToPlayer = (targetLookat.position -
transform.position).sqrMagnitude;
  float runDistance = (playerRange+shotRange)*
(playerRange+shotRange);
  float shotDistance = shotRange*shotRange;
  if ((distanceToPlayer <= runDistance) && (distanceToPlayer >
shotDistance)) {
    return true;
  }
  return false;
}
```

- In the last function, to control the enemy behavior, we will make our enemy walk and stop for a certain amount of time:

```
// Unity JavaScript user:

function IsThinking() : boolean {
  if (IsAiming) {
    _lastTime = Time.time;
    _isThinking = false;
    return false;
  }
  var time : float;
  if (_isThinking) { time = thinkingTime; }
  else { time = walkingTime; }
  if (Time.time >= (_lastTime + time)) {
    _isThinking = !_isThinking;
    _lastTime = Time.time;
  }
  return _isThinking;
}
```

```
// C# user:

public bool IsThinking() {
  if (IsAiming) {
    _lastTime = Time.time;
    _isThinking = false;
    return false;
  }
  float time;
  if (_isThinking) { time = thinkingTime;}
  else { time = walkingTime; }
  if (Time.time >= (_lastTime + time)) {
    _isThinking = !_isThinking;
    _lastTime = Time.time;
  }
  return _isThinking;
}
```

4. The next step is the override function `GetTargetDirecion()`, which will control all the movement direction of our enemy when it isn't aiming or shooting. So, let's type it as follows:

```
// Unity JavaScript user:

protected override function GetTargetDirection () : Vector3 {
  var targetDirection : Vector3;
  if (IsRun) {
    targetDirection =
waypointsContainer.GetDirectionToPlayer(transform.position,
targetLookat.position);
  } else {
    if ((thinkingTime > 0) && IsThinking()) {
      targetDirection = Vector3.zero;
    } else {
      targetDirection = waypointsContainer.GetDirection(transform);
    }
  }
  return targetDirection;
}


// C# user:

protected override Vector3 GetTargetDirection () {
  Vector3 targetDirection;
  if (IsRun) {
    targetDirection =
waypointsContainer.GetDirectionToPlayer(transform.position,
targetLookat.position);
  } else {
    if ((thinkingTime > 0) && IsThinking()) {
      targetDirection = Vector3.zero;
    } else {
      targetDirection = waypointsContainer.GetDirection(transform);
    }
  }
```

```
  return targetDirection;
}
```

5. Next, we will create the `Update()` function. In this function, we will calculate all the movement and behavior of our enemy. Let's add the following code:

```
// Unity JavaScript user:

function Update () {
  if (!IS_GAMEOVER) {
    ApplyGravity();
    if (!IsJumping) {
      if ((MotionState == MOTION_STATE.GROUND) || (MotionState ==
MOTION_STATE.AIM)) {
        IsAiming = CanShoot();
      } else {
        IsAiming = false;
      }
      if (IsAiming) {
        IsShowLaser(true);
        if (_animator) {
          var currentState : AnimatorStateInfo =
_animator.GetCurrentAnimatorStateInfo(0);
          if (!IsWaitForAiming) {
            var lookat : Vector3 = targetLookat.transform.position;
            lookat.y = transform.position.y;
            transform.LookAt(lookat);

_rocketLauncher.transform.LookAt(targetLookat.transform.position);
            IsShot = true;
            if (IsShot) {
              if (currentState.IsName("Shooting.Aiming")) {
                var playbackTime : float = currentState.length;
                BroadcastMessage("Fire",playbackTime);
                IsWaitForAiming = true;
              }
            }
          } else {
            if (IsShot) {
              if (currentState.IsName("Shooting.Shoot")) {
                IsShot = false;
                playbackTime = currentState.length;
                StartCoroutine(WaitForShot(playbackTime));
              }
            }
          }
        }
      } else {
        IsShowLaser(false);
        IsRun =
(waypointsContainer.AwayFromWaypoint(transform.position,
distanceToShot)) ? false : Run();
        ApplyMoveDirection();
        ApplyMoveSpeed();
        ApplyJumping(Jump(MoveDirection));
      }
```

```
    } else {
      IsShowLaser(false);
    }
    if (!IsAiming) {
      UpdateMovement();
    }
  }
}


// C# user:

void Update () {
  if (!IS_GAMEOVER) {
    ApplyGravity();
    if (!IsJumping) {
      if ((MotionState == MOTION_STATE.GROUND) || (MotionState ==
MOTION_STATE.AIM)) {
        IsAiming = CanShoot();
      } else {
        IsAiming = false;
      }
      if (IsAiming) {
        IsShowLaser(true);
        if (_animator) {
          AnimatorStateInfo currentState =
_animator.GetCurrentAnimatorStateInfo(0);
          if (!IsWaitForAiming) {
            Vector3 lookat = targetLookat.transform.position;
            lookat.y = transform.position.y;
            transform.LookAt(lookat);

_rocketLauncher.transform.LookAt(targetLookat.transform.position);
            IsShot = true;
            if (IsShot) {
              if (currentState.IsName("Shooting.Aiming")) {
                float playbackTime = currentState.length;
                BroadcastMessage("Fire",playbackTime);
                IsWaitForAiming = true;
              }
            }
          } else {
            if (IsShot) {
              if (currentState.IsName("Shooting.Shoot")) {
                IsShot = false;
                float playbackTime = currentState.length;
                StartCoroutine(WaitForShot(playbackTime));
              }
            }
          }
        }
      } else {
        IsShowLaser(false);
        IsRun =
(waypointsContainer.AwayFromWaypoint(transform.position,
distanceToShot)) ? false : Run();
```

```
        ApplyMoveDirection();
        ApplyMoveSpeed();
        ApplyJumping(Jump(MoveDirection));
      }
    } else {
      IsShowLaser(false);
    }
    if (!IsAiming) {
      UpdateMovement();
    }
  }
}
```

6. Now, we have finished our AI script. Go back to the Unity editor and drag the AI script on **robotAI_C#** (for C# users) or **robotAI_JS** (for JS users). Then, we will click on **robotAI_C#** (for C# users) or **robotAI_JS** (for JS users) to bring up its **Inspector** view and drag the game object from the **Hierarchy** view of the parameter, as shown in the following screenshot:

7. Finally, we will click on the **robot** object in the **Hierarchy** view to bring up its **Inspector** view and drag **robotAI_C#** (for C# user) or **robotAI_JS** (for JS user) to the **Target Lookat** slot under **Player (script)**, as we can see in the following screenshot:

Now, we have finished this step. We can click on play to see the result. We can also change the parameter to make our enemy behave differently by changing the value in the AI inspector. In the next step, we will create the hit point for the player and enemy to make it more fun.

Objective complete – mini debriefing

In this section, we just created our AI script, which is derived from `CharacterClass`. Most of the `CharacterClass` script is based on the `CharacterControl` script from Project 5, *Build a Rocket Launcher!*. If we take a look at `CharacterClass`, we will see that we have similar methods and parameters such as the `UpdateAnimator()` function, which get called in `OnAnimatorMove()` to control the animation state of our characters (**Player** and **Enemy**).

We also added the new code section to give our enemy some characteristics and make it smart enough to shoot the player, run towards the player, jump when it hits the wall, and to stop and walk after a certain time.

In the `Run()` function, we used the following code:

```
var distanceToPlayer : float = (targetLookat.position -
transform.position).sqrMagnitude;
var runDistance : float = (playerRange+shotRange)*(playerRange+shotRange);
var shotDistance : float = shotRange*shotRange;
if ((distanceToPlayer <= runDistance) && (distanceToPlayer > shotDistance))
{
  return true;
}
```

We used the following code to check for the distance between the enemy and the player:

```
if ((distanceToPlayer <= runDistance) && (distanceToPlayer > shotDistance))
{ return true; }
```

Have a look at the following diagram:

As we can see from the preceding diagram, the enemy will run towards the player if the distance between the player and the enemy is higher than `shotRange` but lower than or equal to `playerRange`. Also, the enemy will shoot the player if the distance between him and the player is in `shotRange`.

We also use `Physics.Raycast()` to check if there is a wall in front of the enemy or not. If there is, we check the different height of the current object *y* position to the maximum of wall height by using `hit.collider.bounds.max.y - hit.point.y;` and make sure that the height is lower than 2.5. Then, our enemy can jump over.

Classified intel

In this step, we have used both the `Physics.Raycast()` functions to check if there is anything blocking the enemy movement direction or the rocket bullet direction. In the `CanShoot()` function, we basically cast the ray from the position of the enemy's rocket launcher to the player by checking to see if there is anything blocking it, as shown in the following diagram:

Then, we also use the `Physics.Raycast()` function in the `Jump()` function to check if the enemy hits the wall or not. If it hits, we will check for the height for the enemy to jump over or not. By calculating from the raycast hitting position to the maximum *y* position of the wall, we can check the height, as shown in the following diagram:

Creating a hit-point UI

Now, we are at the last step of this project. We will add the hit-point game object for the player and enemy as well as create the `HitPointUI` script.

Engage thrusters

Let's get started:

1. Go to the folder (for C# users, **Chapter6** | **Scenes** | **C#** | **UI**, or for JavaScript users, **Chapter6** | **Scenes** | **JavaScript** | **UI**) in the **Project** view, right-click and choose **Create** | **C# Script** (for C# users) or **Create** | **Javasrcript** (for JavaScript users), rename it to `HitPoint`.
2. Double-click on the `HitPoint` script to open it in MonoDevelop. Start by creating all properties and the `Update()` function, as follows:

```
// Unity JavaScript user:

#pragma strict
var ai : AI;
var player : Player;
var frameTexture : Texture2D;
var hpTexture : Texture2D;
var aiTexture : Texture2D;
var textHpTexture : Texture2D;
var textAiTexture : Texture2D;
function Update() {
  if ((player.HpPercent <= 0.0f) || (ai.HpPercent <= 0.0f)) {
    CharacterClass.IS_GAMEOVER = true;
    StopAllCoroutines();
  }
}


// C# user:

using UnityEngine;
using System.Collections;

public class HitPoint : MonoBehaviour {
  public AI ai;
  public Player player;
  public Texture2D frameTexture;
  public Texture2D hpTexture;
  public Texture2D aiTexture;
  public Texture2D textHpTexture;
  public Texture2D textAiTexture;
  void Update() {
    if ((player.HpPercent <= 0.0f) || (ai.HpPercent <= 0.0f)) {
      CharacterClass.IS_GAMEOVER = true;
      StopAllCoroutines();
    }
  }
}
```

3. Next, we will add the `OnGUI()` function to create the hit-point UI as follows:

```
// Unity JavaScript user:
```

```
function OnGUI() {
  GUI.DrawTexture (new Rect (10,10,46,32), textHpTexture);
  GUI.DrawTexture (new Rect (10,42,95,32), textAiTexture);
  GUI.BeginGroup (new Rect (110,15,156,21));
  GUI.DrawTexture(new Rect (0,0,156,21), frameTexture);
  GUI.BeginGroup (new Rect (0,0,player.HpPercent * 156, 21));
  GUI.DrawTexture (new Rect (0,0,156,21), hpTexture);
  GUI.EndGroup ();
  GUI.EndGroup ();

  GUI.BeginGroup (new Rect (110,47,156,21));
  GUI.DrawTexture(new Rect (0,0,156,21), frameTexture);
  GUI.BeginGroup (new Rect (0,0,ai.HpPercent * 156, 21));
  GUI.DrawTexture (new Rect (0,0,156,21), aiTexture);
  GUI.EndGroup ();
  GUI.EndGroup ();
}


// C# user:

void OnGUI() {
  GUI.DrawTexture (new Rect (10,10,46,32), textHpTexture);
  GUI.DrawTexture (new Rect (10,42,95,32), textAiTexture);
  GUI.BeginGroup (new Rect (110,15,156,21));
  GUI.DrawTexture(new Rect (0,0,156,21), frameTexture);
  GUI.BeginGroup (new Rect (0,0,player.HpPercent * 156, 21));
  GUI.DrawTexture (new Rect (0,0,156,21), hpTexture);
  GUI.EndGroup ();
  GUI.EndGroup ();

  GUI.BeginGroup (new Rect (110,47,156,21));
  GUI.DrawTexture(new Rect (0,0,156,21), frameTexture);
  GUI.BeginGroup (new Rect (0,0,ai.HpPercent * 156, 21));
  GUI.DrawTexture (new Rect (0,0,156,21), aiTexture);
  GUI.EndGroup ();
  GUI.EndGroup ();
}
```

In the preceding function, we just use the `GUI.BeginGroup()` function to draw the mask for the hit-point bar.

4. Now, we will go back to Unity and drag the `HitPoint` script to the UI game object in the **Hierarchy** view. Then, we will go to its **Inspector** view and set the following:

Hit Point UI (Script)		
AI	**robotAI_C# (AI)** or **robotAI_JS (AI)**	Drag the **robotAI_C#** or **robotAI_JS** game object inside the **Enemy** game object to the **Hierarchy** view
Player	**robot (Player)**	Drag the **robot** game object to the **Hierarchy** view here
Frame Texture	**hitPointFrame**	This is located in the **Resources**/**UI** folder

Hp Texture	**hitPointBarHP**	This is located in the **Resources/UI** folder
Ai Texture	**hitPointEnemy**	This is located in the **Resources/UI** folder
Text Hp Texture	**HP**	This is located in the **Resources/UI** folder
Text Ai Texture	**ENEMY**	This is located in the **Resources/UI** folder

Now, we have finished our game, so click on **Play** to see the result. We will see that when the player or enemy gets shot, the hit-point bar will decrease.

Objective complete – mini debriefing

We just created the UI game object and the script, which we use to control the hit-point UI. We also used `GUI.BeginGroup()` to mask the decreasing damage from either the player or enemy hit-points.

Classified intel

In this section, we have used `GUI.BeginGroup()` to mask out the texture to show how many hit points are left. The `GUI.BeginGroup()` function must be close to `GUI.EndGroup()`.

In our code, we basically created the first group to contain the background texture, which is the bar frame. Then, we drew another group on top of the first group, which contains the bar texture as a clip mask. This second group's width will relate to the hit-point value left for the player or enemy, as shown in the following code:

```
//Draw the background group
GUI.BeginGroup (Rect (110,15,156,21));
GUI.DrawTexture(Rect (0,0,156,21), frameTexture);
// Create a second Group which will be clipped
GUI.BeginGroup (Rect (0,0, player.GetHpPercent() * 156, 21));
GUI.DrawTexture (Rect (0,0,156,21), hpTexture);
//End both Groups
GUI.EndGroup ();
GUI.EndGroup ();
```

We can translate the preceding code into the following diagram:

Mission accomplished

In this project, we created `Waypoint`, `WaypointsContainer` and `WaypointsContianerEditor` for the enemy to follow. We also created the enemy AI that can jump, run towards the player, walk, and stop for a certain time by creating the AI script. This script is derived from the `CharacterClass` script that is based on the `CharacterControl` script from Project 5, *Build a Rocket Launcher!*.

We also learned more about the `Gizmos()` function to display the visual of our `Waypoint` empty game object by using `Gizmo.DrawIcon()`, `Gizmo.DrawLine()`, and `Gizmo.DrawWireSphere()`.

We also learned how to use the `CustomEditor` class as well as `SerializedObject` and `SerializedProperty` to create a cool custom editor for our `WaypointsContainer` script.

Finally, we learned how to use the `GUI.BeginGroup()` function to mask and show the hit-point UI object for the player and the enemy.

So, let’s take a look at the following screenshot to see what we have done so far:

Hotshot challenges

Now we have an understanding of the basic concept of creating the enemy AI, but our AI script still needs a lot of improvement to make it smarter. Why don't we do something to spice it up? Let's try the following things:

- Try mixing the `Navmesh` or `Pathfinder` system to our `Waypoints` system
- Add different types of weapons for our enemy
- Try changing or adjusting the parameter of the AI, such as `shotRange` or `playerRange`, to make the enemy react to the player faster
- Add more waypoints for our `WaypointsContainer` game object to make sure our enemy has more choice to walk
- Add multiple enemies in the scene (you will need to adapt the `HitPoint UI` game object to be able to track the hit points for each enemy)
- Try changing the AI code for the enemy to avoid the rocket or make the rocket follow the player's movement
- Add a multi-editor object to `WaypointsContainer` and make it work when editing more than one `WaypointsContainer` object

Chapter 7. Forge a Destructible and Interactive Virtual World

In games, there are many features that help the player get involved and experience the game much more. The most important feature is the game's level. So, what's the meaning of a game's level or just a level in general? First of all, a level marks a gradation in either the difficulty at that stage or the logical progression of the story of the game. It makes our game much more fun to the player. There might be a puzzle to solve, or sometimes it's just a cool graphics environment. Each level will include static and nonstatic objects. The static objects do not respond to physics and may include bridges, houses, and so on. On the other hand, nonstatic objects such as rocks, doors, and switches will interact with the player.

In many cases, we will see that games use interactive objects to make a level much more fun to play by adding events or triggers to the objects and making them interact with the players. For example, the player has to push the switch to open the door, or he gets blocked by a destroyed bridge on moving close to it. We can also add physics to the nonstatic objects to make them behave realistically, such as adding physics to the rocks when they are falling down to the ground.

In this project, we will learn how to create an interactive environment by using the `trigger` event and the `callback` function. We will create a destructible rock that will be triggered when the player gets close to it, and we will also create a destroyable wall that the player and AI can shoot at to destroy it. We will also learn how to set up the `ragdoll` object for the AI, which will add the physics interaction when the player kills it.

Mission briefing

This project will start with setting up the AI's `ragdoll` object, which will be used to replace the AI when it dies. Then, we add the script in our `AI` script to switch the `ragdoll` object with the `AI` object. Next, we will create the destroyable wall from four **cube** objects, which will each have **Rigidbody** attached. Of course, we will create a script to break the object apart by having a script to check for the colliding object.

Then, we will also create another destructible rock from multiple cubes, which will fall when the player gets close to it. We will create a trigger area, `RocksTrigger`, and `Rock` scripts to make the rock fall down when the player enters this trigger area using the `delegate` and `event` functions. However, the `delegate` and `event` functions are only available for C# users. For Unity JavaScript users, we will create the `JSDelegate` class to create our custom `delegate` and `event` functions.

Why is it awesome?

We already know that in Project 1, *Develop a Sprite and Platform Game*, Unity has integrated Box2d as its 2D physics engine. For 3D animations, Unity also has the built-in **NVIDIA PhysX** physics engine, which is very powerful to create realistic physics simulations for our game world. In this project, we will learn how to apply physics to our game by applying ragdoll physics to the character, as well as attaching **Rigidbody** to make the objects or environment react to the player whenever we want. With this technique, we will create more variety in the game play or level, which will make our game very challenging to play.

Your Hotshot objectives

We will start by importing the `chapter 7` package, and then move on to each topic as follows:

- Creating a `ragdoll` object
- Creating a destructible wall
- Creating a rockslide and trigger area
- Creating the `RocksTrigger` and `Rocks` scripts

Mission checklist

Before we start, we will need to get the project folder and assets from http://www.packtpub.com/support?nid=8267, which includes the finished project and the assets we need to use in this project.

Go to the preceding URL and download the `Chapter7.zip` package and unzip it. In the `Chapter7` folder, there are two Unity packages: `Chapter7Package.unitypackage` (we will use this package for this project) and `Chapter7Package_Completed.unitypackage` (this is the complete project package).

Creating a ragdoll object

In the first section, we will set up and create a `ragdoll` object, which will be used to replace the AI game object when it dies.

Prepare for lift off

We will begin by importing a package, preparing the assets folder, and ensuring that we have everything ready in order to start. Let's create a new project and import a package with the following steps:

1. Import the assets package by navigating to **Assets | Import Package | Custom Package…** and choose `Chapter7.unityPackage`, which we downloaded earlier. Then, click on the **Import** button in the pop-up window, as shown in the following screenshot:

2. Wait until the package is imported, and you will see the `Chapter7`, `Editor`, `Gizmos`, `Resources`, and `Standard Assets` folders in the **Project** view. Then, go to the `Chapter7/Scenes` folder and double-click on a scene (C# users can double-click on the **VirtualWorld_C#** scene, and Unity JavaScript users can double-click on the **VirtualWorld_JS** scene), as shown in the following screenshot:

3. Next, go to the `Resources/Prefabs` folder and drag the **robotAI_ragdoll** prefab from the **Project** view to the **Hierarchy** view, as shown in the following screenshot (this screenshot is for the **VirtualWorld_C#** scene):

4. Click on the **robotAI_ragdoll** game object in the **Hierarchy** view, go to the **Scene** view, and press the *F* key to zoom into the **robotAI_ragdoll** game object in the scene, as shown in the following screenshot:

Engage thrusters

Now we can start applying ragdoll physics to the **robotAI_ragdoll** game object as follows:

1. Go to **GameObject | Create Other | Ragdoll…**. You will see the **Create Ragdoll** window pop up, as shown in the following screenshot:

2. Go back to the **Hierarchy** view and click on the triangle in front of the **robotAI_ragdoll** game object to see the **roothandle** child name. Then, drag it to the **Root** option in the **Create Ragdoll** window, as shown in the following screenshot:

3. Next, we will continue dragging other objects to the **Create Ragdoll** window. Let's drag the **upleg_L** object from the **Hierarchy** view to the **Left Hips** option in the **Create Ragdoll** window, and click on the **upleg_L** object to bring its child name **lowleg_L**.
4. Drag the **lowleg_L** object to the **Left Knee** option in the **Create Ragdoll** window. Then, click on the **lowleg_L** object to bring up its child name **heel_L**, and drag it to the **Left Foot** option in the **Create Ragdoll** window.
5. Again, perform this action with the **left bone** object by dragging the **upleg_R** object to the **Right Hips** option, **lowleg_R** to the **Right Knee** option, and **heel_R** to the **Right Foot** option in the **Create Ragdoll** window, as shown in the following screenshot:

It may be time-consuming to expand the object's children hierarchy by clicking on each object. The good thing in Unity is that we can collapse or expand all of the object's children using the *Option* key (for Mac) or the *Alt* key (for Windows).

Tip

We can hold the *Option* (for Mac) or *Alt* (for Windows) key, and click on the triangle in front of the object to expand or collapse the object's children.

6. Now, we are done with the lower body settings. Next, we will continue to set the upper part of the body by applying physics to each bone as follows:

The Create Ragdoll window	The Hierarchy window
Left Arm	**uparm_L**
Left Elbow	**elbow_L**
Right Arm	**uparm_R**
Right Elbow	**elbow_R**
Middle Spine	**spine1**
Head	**head**

We will see the results as shown in the following screenshot:

7. Click on the **Create** button to create the ragdoll for our `AI` object. If we click on **robotAI_ragdoll** in the **Hierarchy** view, we will see that the capsule and box colliders get merged on each joint, as shown in the following screenshot:

However, we are not done yet. From the preceding screenshot, we can see that we still need to adjust the colliders associated with the character's shape.

8. Click on **head** in the **Hierarchy** view, go to the **Inspector** view, and set the following values:

Sphere Collider	
Center	**X**: `-0.04`, **Y**: `-0.06`, and **Z**: `0`
Radius	`0.15`

9. Next, click on **roothandle** in the **Hierarchy** view, go to the **Inspector** view, and set the following:

Box Collider	
Center	**X**: `-0.5`, **Y**: `-0.1`, and **Z**: `-0.175`
Size	**X**: `1.3`, **Y**: `0.5`, and **Z**: `0.35`

10. Click on **spine1** in the **Hierarchy** view, go to the **Inspector** view, and set the following:

Box Collider	
Center	**X**: `-0.32`, **Y**: `0.1`, and **Z**: `-0.175`
Size	**X**: `1.3`, **Y**: `0.5`, and **Z**: `0.35`

11. We just set up the ragdoll for the `AI` game object, but we still need to apply the **collider** and **Rigidbody** components to the **gun** object. So, we will click on the **gun** game object and apply **Rigidbody** to this by navigating to **Component** | **Physics** | **Rigidbody**. Go to the **Inspector** view of this game object and set the following:

Rigidbody	
Mass	`3`

12. Next, click on the triangle in front of the **gun** game object to bring up the **gun_model** object, as shown in the following screenshot:

13. Apply the **Box Collider** component to the **gun_model** object (this will make the gun collide with other objects when it falls down) by going to **Component** | **Physics** | **Box Collider**. We are now done with the creation of the **robotAI_ragdoll** game

object.

14. Next, we will click on the **robotAI_ragdoll** game object in the **Hierarchy** view and go to the **Inspector** view. Then, click on the **Apply** button to update the prefab, as shown in the following screenshot:

15. As we have already updated the **robotAI_ragdoll** prefab in the **Project** view, we don't need to keep the **robotAI_ragdoll** game object in the **Hierarchy** view anymore. So, we just delete this by right-clicking on it and choosing **Delete**.
16. Now, we need to go to the `Chapter7/Scripts/C#/Actor` (for C# users) or `Chapter7/Scripts/Javascript/Actor` (for Unity JavaScript users) folder, and open the `AI` script, which will be used to replace the `robotAI_ragdoll` game object when the AI is dead. Let's open the `AI` script and go to the `UpdateHitPoint()` function and add the highlighted code as follows:

```
// Unity JavaScript user:
protected override function UpdateHitPoint (collision : Collision) {
  if (IS_GAMEOVER == false) {
    if (collision.transform.tag == "Rocket") {
      HitPoint.CURRENT_AI_INDEX = index;
      var rocket : Rocket = collision.gameObject.GetComponent.<Rocket>
();
      HP = Mathf.Clamp(HP-rocket.damage, 0, _maxHP);
      if (HP == 0) {
        var myRagdoll : GameObject =
Instantiate(ragdoll,transform.position,transform.rotation);

        var rigids : Rigidbody[] = myRagdoll.GetComponentsInChildren.
<Rigidbody>();
        var rocketForce : Vector3 = rocket.transform.forward *
rocket.constantForce.relativeForce.magnitude;
        for (var rigid : Rigidbody in rigids) {
          rigid.AddForce(rocketForce,ForceMode.VelocityChange);
        }
        GameObject.Destroy(gameObject);
        onEnemyDie.Invoke();
      }
    }
  }
}


// C# user:

protected override void UpdateHitPoint (Collision collision) {
  if (IS_GAMEOVER == false) {
    if (collision.transform.tag == "Rocket") {
```

```
      HitPoint.CURRENT_AI_INDEX = index;
      Rocket rocket = collision.gameObject.GetComponent<Rocket>();
      HP = Mathf.Clamp(HP-rocket.damage, 0, _maxHP);
      if (HP == 0) {
        GameObject myRagdoll =
(GameObject)Instantiate(ragdoll,transform.position,transform.rotation);
        Rigidbody[] rigids =
myRagdoll.GetComponentsInChildren<Rigidbody>();
        Vector3 rocketForce = rocket.transform.forward *
rocket.constantForce.relativeForce.magnitude;
        foreach (Rigidbody rigid in rigids) {
          rigid.AddForce(rocketForce,ForceMode.VelocityChange);
        }
        GameObject.Destroy(transform.parent.gameObject);
        onEnemyDie();
      }
    }
  }
}
```

Our `AI` class is derived from `CharacterClass`. So, if we open `CharacterClass`, we will see the `UpdateHitPoint()` function getting called from the `OnCollisionEnter()` function. We will also see that there is the `ragdoll GameObject` variable that we use to instantiate a `ragdoll` object. This function will check whether the rocket hits the AI or not. If the AI is hit and its hit point is equal to `0`, we will create a `ragdoll` object for the scene and add the rocket force to the `ragdoll rigidbody` object to make this more realistic.

17. Save the `AI` script and go back to the `Resources/Prefabs` folder in the **Project** view. Click on the **robotAI_C#** (for C# users) or **robotAI_JS** (for Unity JavaScript users) prefab, and go to the **Inspector** view, as shown in the following screenshot:

18. In the **Inspector** view, we will go to the **Ragdoll** property in the **AI (Script)** section, and drag **robotAI_ragdoll** to this **Ragdoll** property, as shown in the following

screenshot:

With this step, we will update all the **robotAI** prefabs in the **Hierarchy** view. Now we have finished this step. We can click on play and see the result—when we kill the enemy, we will see that the `ragdoll` game object gets created at the same position as the **robotAI** game object. Then, the **robotAI** game object gets destroyed, as shown in the following screenshot:

Objective complete – mini debriefing

In this step, we created the `ragdoll` object by assigning bones to it. Then, Unity automatically attaches all colliders, rigidbody, and character joints to match our character to use for the `ragdoll` object. This ragdoll will be added to the game scene when the **robotAI** object is dead using the `Instantiate()` function. We also added the force to **Rigidbody** to make the ragdoll follow the direction of the rocket, which will make it more realistic using the `rigidbody.AddForce()` function.

Classified intel

In this section, we have used the **robotAI_ragdoll** prefab and added it to the scene when our AI gets destroyed. The concept of this section is that we have to separate the ragdoll from the `AI` object. Then, we add the ragdoll and destroy the `AI` object. Of course, this is just one way to create the ragdoll effect for the character. The advantage of separating them is that we don't need to worry about the `ragdoll` object when we need to update the character. We can have the ragdoll already set up and just work on the update of character's animation. This will prevent us from making an error when we change something in the script.

On the other hand, this method isn't that good if we need an accurate collider for each part of the character's body. In this case, we can also have the ragdoll and character animation in the same game object by attaching the **Animator** component to control the animation. Then, we can use **Is Kinematic** in **Rigidbody** to enable or disable the ragdoll physics to use the ragdoll mode or character animation.

When we added the **ragdoll** prefab to the scene, we also added force using the `AddForce()` function to our ragdoll's rigidbody. This makes our `ragdoll` object move following the rocket's direction. We set the force mode to `ForceMode.VelocityChange` to change the velocity as soon as the rocket hits the character by ignoring its mass. This way we can make sure that we have the same velocity applied to all the parts of our `ragdoll` object.

Note

For more information on `ForceMode.VelocityChange`, visit the following URL:

http://docs.unity3d.com/Documentation/ScriptReference/ForceMode.VelocityChange.html

Creating a destructible wall

In this section, we will start by creating a destructible wall with multiple **cube** game objects in the Unity engine, as well as adding some code to the `rocket` script to make this wall breakable when the player shoots the rocket to hit it.

Prepare for lift off

We will begin with creating a new tag for our destructible wall:

1. Let's go to **Edit** | **Project Settings** | **Tags** to bring out the **Inspector** view for **Tags**. In the **Inspector** view, we will click on the triangle in front of the **Tags** element and then on the **Element 4** type **Destructible**, as you can see in the following screenshot:

2. Next, we will go to **GameObject** | **Create Empty** to create an empty game object and name it `Wall` to reset its transform position to **X**: `0`, **Y**: `0`, and **Z**: `0`. Then we are ready to start.

Engage thrusters

Now, we have set the **Destructible** tag and created an empty game object, `Wall`. Next, we will create the four cubes to represent each piece of the broken wall:

1. Go to **GameObject** | **Create Other** | **Cube**, name it `Cube1`, and drag it to the **Wall** game object, which we just created, as shown in the following screenshot:

2. Add the **Rigidbody** component to **Cube1** to make the wall fall realistically when it breaks by navigating to **Component** | **Physics** | **Rigidbody** and adding **Rigidbody**.
3. Next, we will go to the cube's **Inspector** view to set up the parameters as follows:

Transform		
Position	**X**: `3`, **Y**: `3`, and **Z**: `0`	
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `0`	
Scale	**X**: `6`, **Y**: `6`, and **Z**: `1`	
Box Collider		
Material	**Rock** (drag the **Rock** physics material from the `Resources/Physics Materials` folder to the **Project** view)	
Mesh Renderer		
Materials	**Size**	**1**
Element 0	**Rock** (drag the **Rock** material from the `Resources/Materials` folder to the **Project** view)	
Rigidbody		
Mass	`9`	
Is Kinematic	Check	
Tag	**Destructible**	

Now we have finished setting up the first cube. Let's duplicate three more cubes using *Command* + *D* (for Mac) or *Ctrl* + *D* (for Windows), and name all three as `Cube2`, `Cube3`, and `Cube4`, as shown in the following screenshot:

Then, we go to each new cube's **Inspector** view and set up its transform position as follows:

Transform (Cube2)	
Position	**X**: `3`, **Y**: `9`, and **Z**: `0`
Transform (Cube3)	
Position	**X**: `-3`, **Y**: 9, and **Z**: `0`
Transform (Cube4)	
Position	**X**: `-3`, **Y**: `3`, and **Z**: `0`

The wall that is created will look like the following screenshot:

4. Next, we will click on the **Wall** game object in the **Hierarchy** view and go to the **Inspector** view, and set up **Transform** as follows:

Transform	
Position	**X**: `1037.5`, **Y**: `4`, and **Z**: `693`
Rotation	**X**: `0`, **Y**: `36`, and **Z**: `0`
Scale	**X**: `1`, **Y**: `1`, and **Z**: `1`

5. Now, we will add a script that makes this **Wall** object break apart when the character shoots at it. Let's go to the `Chapter7/Scripts/C#/Object` (for C# users) or `Chapter7/Scripts/Javascript/Object` (for Unity JavaScript users) folder, and double-click on the `rocket` script to open this script in the script editor.
6. In the `rocket` script, we will add two parameters at the beginning of the script, as highlighted in the following code:

```
// Unity JavaScript user:
var explosionParticle : GameObject;
var explosionRadius : float = 50;
var explosionForce : float = 1000;


//C# user:

public GameObject explosionParticle;
public float explosionRadius = 50;
public float explosionForce = 1000;
```

7. Next, we will go to the `OnCollisionEnter()` function and add the following highlighted script:

```
// Unity JavaScript user:

function OnCollisionEnter (others : Collision) {
  var contactPoint : ContactPoint = others.contacts[0];
  var rotation : Quaternion = Quaternion.Euler(Vector3.up);
  GameObject.Instantiate(explosionParticle, contactPoint.point,
rotation);
  var position : Vector3 = transform.position;
  var hits : Collider[] = Physics.OverlapSphere(position,
explosionRadius);
  for (var c : Collider in hits) {
    if (c.tag == "Destructible") {
      var r : Rigidbody = c.rigidbody;
      if (r != null) {
        r.isKinematic = false;
        r.AddExplosionForce(explosionForce, position, explosionRadius);
      }
    }
  }
  KillObject();
}


// C# user:
```

```
void OnCollisionEnter (Collision others) {
  ContactPoint contactPoint = others.contacts[0];
  Quaternion rotation = Quaternion.Euler(Vector3.up);
  GameObject.Instantiate(explosionParticle, contactPoint.point,
rotation);
  Vector3 position = transform.position;
  Collider[] hits = Physics.OverlapSphere(position, explosionRadius);
  foreach (Collider c in hits) {
    if (c.tag == "Destructible") {
      Rigidbody r = c.rigidbody;
      if (r != null) {
        r.isKinematic = false;
        r.AddExplosionForce(explosionForce, position, explosionRadius);
      }
    }
  }
  KillObject();
}
```

We have used the `Physics.OverlapSphere()` function to get all the colliders that collide within the explosion radius of the rocket. Then, we will loop through each collider and add the explosion force using the `AddExplosionForce()` function to make the destructible pieces explode.

Tip

The `Physics.OverlapSphere()` function is used to get all the colliders that touch each other or that are inside the given sphere radius. For more details on this function, visit the following URL:

http://docs.unity3d.com/Documentation/ScriptReference/Physics.OverlapSphere.html

The `AddExplosionForce()` function basically applies a force to the **Rigidbody** component, which will simulate the explosion effect. This means that the force will be decrease depending on the distance of **Rigidbody**. For more details on this function, visit the following URL:

http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody.AddExplosionFor

8. Save the script and click on the play button; in the scene, if we go to the path on the right, we will see the **Wall** object that we have just created. We can shoot at it and it will break, as shown in the following screenshot:

We will see that the current wall breaks into four pieces. We can also fracture the mesh in a modeling program and make it much more dynamic and realistic. You might also want to check out the following link for the simple fracture script:

http://forum.unity3d.com/threads/57994-Simple-Fracture-Script-v-1-01

Objective complete – mini debriefing

In this step, we basically created four cubes, and each one has its own collider and **Rigidbody** attached, which will create physical movement when we apply the explosion force. This will create a realistic behavior for the wall when it's breaking apart, as shown in the following diagram:

To make this work, we disabled the **Is Kinematic** component in each cube's **Rigidbody**. The **Is Kinematic** property is basically used to enable or disable the physics calculation of the object that has **Rigidbody** attached. In some cases, we can apply **Rigidbody** on the character's animation and then enable or disable the **Is Kinematic** property when we need to, such as using the ragdoll physics or character animation on the `ragdoll` object as we mentioned in the first step of this project, *Creating a ragdoll object*.

Classified intel

In this step, we have also applied the **Rock** physics material to the **Box Collider** material in the cube. We can apply friction and bounciness values to each object of the **Box Collider** material to get a realistic reaction while calculating the physics.

For the **Rock** physics material, we go to the `Resources/Physics Materials` folder in the **Project** view, and then click on the **Rock** physics material object to bring up its **Inspector** view. We will see to it that we have set the **Dynamic Friction** value to `0.3` and **Static Friction** to `0.3`, which will make each piece slow down when it collides with another. This way, we can make the rock be not too slippery or too hard to move. Also, the rock should not bounce at all. So, we set the **Bounciness** value to `0`, as shown in the following screenshot:

For more details on each property, visit the following URL:

http://docs.unity3d.com/Documentation/Components/class-PhysicMaterial.html

Creating a rockslide and trigger area

In the previous section, we created the destructible wall object, which contains four cubes. Each one has the **Rigidbody** and **Box Collider** components attached, which will make the wall such that we are able to shoot and break it. In this section, we will create a rockslide that will be triggered when the player hits the trigger area, which will have the script attached later in the next step.

Prepare for lift off

We will begin with creating an empty object for our rockslide:

1. Let's go to **GameObject** | **Create Empty** to create the empty game object and name it `Rockslide`. Reset its position to **X**: `0`, **Y**: `0`, and **Z**: `0`.
2. Then, we will create another empty game object by going to **GameObject** | **Create Empty** and name it `Break`. Drag this inside the **Rockslide** game object, as shown in the following screenshot:

3. Next, reset **Transform** of the **Break** game object to default by clicking on the little gear on the right-hand side of the screen and then choose **Reset**.

Engage thrusters

Here, we will create six cube objects that will slide down the hill and two cube objects, which will be static objects, to make it look like part of the rock is still stuck to the terrain. Then, we will create the trigger area to make the rock fall down when the player hits it. Let's get started:

1. Let's go to **GameObject** | **Create Other** | **Cube**, name it `Cube1`, and drag it to the **Break** game object in **Rockslide**, which we just created, as shown in the following screenshot:

2. Next, we will go to the **Inspector** view of **Break** and set its **Transform** position to **X**: `0`, **Y**: `-1.32`, and **Z**: `5.97`.
3. Then, we will add the **Rigidbody** component to the **Cube1** object by going to **Component** | **Physics** | **Rigidbody**.
4. Next, we will go to the cube's **Inspector** view to set up the parameters as follows:

Transform		
Position	**X**: `-1.5`, **Y**: `0`, and **Z**: `-4.1`	
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `0`	
Scale:	**X**: `3`, **Y**: `3`, and **Z**: `5`	
Box Collider		
Material	**Rock** (drag the **Rock** physics material from the `Resources/Physics Materials` folder to the **Project** view)	
Mesh Renderer		
Materials		
Size	`1`	
Element 0		**Rock** (drag the **Rock** material in the `Resources/Materials` folder in the **Project** view)
	Rigidbody	
Mass	`10`	
Is Kinematic	Check	

Tags	**Terrain**	

5. Now, we have finished setting up the first cube. Let's duplicate five more cubes by using the shortcut *Command* + *D* (for Mac) or *Ctrl* + *D* (for Windows), and naming all five cubes to `Cube2`, `Cube3`, `Cube4`, `Cube5`, and `Cube6`, which is similar to what we did for the **Wall** game object in the previous section, *Creating a destructible wall*. Then, we go to each new cube's **Inspector** view and set up its position as follows:

Transform (Cube2)	
Position	**X**: `1.5`, **Y**: `0`, and **Z**: `-4.1`
Transform (Cube3)	
Position	**X**: `-1.5`, **Y**: `0`, and **Z**: `0.92`
Transform (Cube4)	
Position	**X**: `1.5`, **Y**: `0`, and **Z**: `0.92`
Transform (Cube5)	
Position	**X**: `-1.5`, **Y**: `0`, and **Z**: `5.92`
Transform (Cube6)	
Position	**X**: `1.5`, **Y**: `0`, and **Z**: `5.92`

The six cubes will form the following structure:

6. Now, we need two static rocks that won't fall down. Let's duplicate the **Cube1** object that we just created by using *Command* + *D* (for Mac) or *Ctrl* + *D* (for Windows). Name it `CubeBase1`, and drag it outside the **Break** game object but inside the

Rockslide game object, as shown in the following screenshot:

7. Go to the **Inspector** view of **CubeBase1** and set the parameters as follows:

Transform	
Position	**X**: `0`, **Y**: `-1.32`, and **Z**: `-3.15`
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `6`, **Y**: `3`, and **Z**: `5`
Rigidbody	Right-click and choose **Remove Component**

8. Duplicate this object to another side using *Command* + *D* (for Mac) or *Ctrl* + *D* (for Windows), name it `CubeBase2`, and set **Transform** as follows:

Transform	
Position	**X**: `0`, **Y**: `-1.32`, and **Z**: `16.9`
Rotation	**X**: `0`, **Y**: `0`, and **Z**: `0`
Scale	**X**: `6`, **Y**: `3`, and **Z**: `5`

9. We are almost done creating this object. The last thing to create the trigger area to make the rock fall down when the player hits this area. So, we go to **GameObject** | **Create Empty**, name it `TriggerArea`, and drag it in the **Rockslide** game object.
10. Add the **Box Collider** component to this by going to **Component** | **Physics** | **Box Collider**. Then, set the parameters as follows:

Transform	
Position	**X**: `-35`, **Y**: `-7`, and **Z**: `-3`
Rotation	**X**: `0`, **Y**: `335`, and **Z**: `0`
Scale	**X**: `1`, **Y**: `1`, and **Z**: `1`
Box Collider	
Is Trigger	Check

Size	**X**: `12`, **Y**: `36`, and **Z**: `24`

You will see that the object will look something similar to the following screenshot:

11. Finally, we will click on the **Rockslide** game object, go to its **Inspector** view, and set its **Transform** component as follows:

Transform	
Position	**X**: `1068`, **Y**: `24`, and **Z**: `677`
Rotation	**X**: `0`, **Y**: `140`, and **Z**: `0`
Scale	**X**: `1`, **Y**: `1`, and **Z**: `1`

Now, we have finished this step. In the next section, we will add a script to make our rocks slide down the hill when the player hits the trigger area. The following screenshot shows the **Rockslide** game object and trigger area:

TRIGGER AREA
ROCKSLIDE

Objective complete – mini debriefing

In this section, we set up the rock slider, which will be used in the next step. We've set up rocks from multiple cubes similar to the way we set up the destructible wall. However, in this step, we also created the trigger area, which will be used to trigger the rock to slide down when the character hits this area.

Creating the RocksTrigger and Rocks scripts

Now, we are at the last step of this project. We will create the `RocksTrigger` and `Rocks` scripts to make the rock fall down.

Engage thrusters

Let's create the `RocksTrigger` script first by performing the following steps:

1. Go to the `Scripts` folder in the **Project** view to create the script by right-clicking and navigating to **Create** | **C#** (for C# users) or **Create** | **Javascript** (for JavaScript users) and rename it `RocksTrigger`.
2. Double-click on the **RocksTrigger** script to open it in MonoDevelop and start adding the script as follows:

```
// Unity JavaScript user:

#pragma strict
private var _isTrigger : boolean = false;
static var onTrigger : JSDelegate = new JSDelegate();

function OnTriggerEnter(collider : Collider) : void {
  if ((collider.transform.tag == "Player") && (_isTrigger == false)) {
    _isTrigger = true;
    onTrigger.Invoke();
  }
}


// C# user:

using UnityEngine;
using System.Collections;

public class RocksTrigger : MonoBehaviour {
  bool _isTrigger = false;
  public delegate void OnRocksTrigger ();
  public static event OnRocksTrigger onTrigger;

  public void OnTriggerEnter(Collider collider) {
    if ((collider.transform.tag == "Player") && (_isTrigger == false))
{
      _isTrigger = true;
      onTrigger();
    }
  }
}
```

   Here, we've used the `delegate` and `event` methods (available only in C#), which basically call the `onTrigger()` function on any object that has the event listener. However, in JavaScript, we don't have the `delegate` and `event` methods, so we use the custom class called `JSDelegate`, which is already present in with the project's package. We can go to the `Chapter7/Scripts/Javascript/Utils` folder in the **Project** view and open `JSDelegate` to check out this class. We will get more details on this in the *Classified intel* section.

3. We already have the `trigger` script. Next, we need the `listener` script to listen to the trigger and make the rocks fall down. We will create a new script by going to the

`Scripts` folder in the **Project** view to create a script by right-clicking and navigating to **Create | C#** (for C# users) or **Create | Javascript** (for JavaScript user) and rename it `Rocks`.

4. Double-click on the **Rocks** script to open it in MonoDevelop and start adding the script as follows:

```
// Unity JavaScript user:

#pragma strict
@Range(-100, 0)
var downForce : int = -80;
private var _rigidbodys : Rigidbody[];

function OnEnable () : void
{
  RocksTrigger.onTrigger.Add(OnTrigger);
}

function OnDisble () : void
{
  RocksTrigger.onTrigger.Remove(OnTrigger);
}

function OnTrigger () : void
{
  EnabledRigidbody();
}

function Awake () : void {
  _rigidbodys = gameObject.GetComponentsInChildren.<Rigidbody>();
}

function Start () : void {
  DisabledRigidBody();
}

function EnabledRigidbody () : void {
  for (var r : Rigidbody in _rigidbodys) {
    r.useGravity = true;
    r.isKinematic = false;
    r.AddForce(new Vector3(0,downForce,0),ForceMode.VelocityChange);
  }
}

function DisabledRigidBody() : void {
  for (var r : Rigidbody in _rigidbodys) {
    r.useGravity = false;
    r.isKinematic = true;
  }
}


// C# user:

using UnityEngine;
```

```
using System.Collections;

public class Rocks : MonoBehaviour {
  [Range(-100, 0)]
  public int downForce = -80;

  Rigidbody[] _rigidbodys;

  void OnEnable ()
  {
    RocksTrigger.onTrigger += OnTrigger;
  }

  void OnDisble ()
  {
    RocksTrigger.onTrigger -= OnTrigger;
  }

  void OnTrigger ()
  {
    EnabledRigidbody();
  }

  void Awake ()  {
    _rigidbodys = gameObject.GetComponentsInChildren<Rigidbody>();
  }

  void Start ()  {
    DisabledRigidBody();
  }

  public void EnabledRigidbody () {
    foreach (Rigidbody r in _rigidbodys) {
      r.useGravity = true;
      r.isKinematic = false;
      r.AddForce(new Vector3(0,downForce,0),ForceMode.VelocityChange);
    }
  }

  public void DisabledRigidBody() {
    foreach (Rigidbody r in _rigidbodys) {
      r.useGravity = false;
      r.isKinematic = true;
    }
  }
}
```

In the preceding function, we create the event listener and then apply a down force to make the rock fall down faster by using the `OnEnable()` and `OnDisable()` functions to add and remove the event listener. `OnEnable()` gets called every time the game object is enabled or active. On the other hand, `OnDisable()` will be called every time the game object is disabled or inactive.

Tip

We also use the `[Range(Min,Max)]` function in C# or `@Range(Min,Max)` followed by the `int`, `float`, or `double` variables to create the limit-value slider in the editor.

This is a very convenient way to create the slider without creating a custom inspector as we did in Project 6, *Make AI Appear Smart*. We can see the result in the following screenshot:

5. Now we are finished with the script. We will go back to Unity, go to the **Hierarchy** view, and click on the **TriggerArea** game object. Then, we will drag the **RocksTrigger** script on this game object, as shown in the following screenshot:

6. Still in the **Hierarchy** view, we will click on the **Break** game object and then drag the **Rocks** script on it, as shown in the following screenshot:

Now that we are done with the project, we can click on play to see the result. We will see that if we are entering the trigger area, the rocks will start falling down, as shown in the following screenshot:

TRIGGER
PLAYER
HP
× 20

Objective complete – mini debriefing

We just created the `RocksTrigger` and `Rocks` scripts, which are used to trigger the `rockslide` event when the player hits the trigger area. First, we created the `Rocks` script and used `OnTriggerEnter()` to check whether the player has entered the trigger area or not. Then, we call the `onTrigger()` event function to send the event to the listener, which is `RocksTrigger`, using the `delegate` and `event` functions in C#.

However, there are no `delegate` and `event` functions in Unity JavaScript. So, we have to use our custom script, which is the `JSDelegate` script available in the `Chapter7/Scripts/Javascript` folder in the **Project** view, which already comes with the project's package.

Next, we listen to the trigger by adding the following script:

```
// Unity JavaScript user:

function OnEnable () : void
{
  RocksTrigger.onTrigger.Add(OnTrigger);
}

function OnDisble () : void
{
  RocksTrigger.onTrigger.Remove(OnTrigger);
}

function OnTrigger () : void
{
  EnabledRigidbody();
}


// C# user:

void OnEnable ()
{
  RocksTrigger.onTrigger += OnTrigger;
}

void OnDisble ()
{
  RocksTrigger.onTrigger -= OnTrigger;
}

void OnTrigger ()
{
  EnabledRigidbody();
}
```

We used the `OnEnable()` function to add a function that will be called when the event is triggered. We use the `OnDisable()` function to remove a function when the game object is inactive. Then, the `OnTrigger()` function is the function that will be called when the event

is triggered, which we call the `EnabledRigidbody()` function, to make the rock slide.

Tip

The `delegate` and `event` methods are better than the `BroadcastMessage` method that we've used in Project 5, *Build a Rocket Launcher!*, because we don't have to set up the listener as a child of the event trigger. Also, the `BroadcastMessage` method is slower than the `delegate` and `event` methods because it needs to iterate to all the children in this game object to be able to call the function.

Classified intel

In this section, we have used the `delegate` and `event` functions to set the event and event listener. The basic idea of the `delegate` and `event` functions in C# is that we can call the functions from other scripts by having it to listen to the event function. We do this by first creating the `delegate` function as follows:

```
public delegate void OnDelegateFunction();
```

Then, we create a `static` event of the `delegate` function that we just created:

```
public static event OnDelegateFunction myEvent;
```

We use `static` because we want to have the same event for every listener. This way, we can have multiple listeners listen to the same event. Next, we trigger this event by calling `myEvent();`.

Then, we use the plus (+) sign to add the function to the event and use the negative (-) sign to remove it from the event that is calling.

For a Unity JavaScript user, if we go to the `Chapter7/Scripts/Javascript` folder in the **Project** view, we will see the `JSDelegate` script there. Double-click on this script to open it and you will see the following code:

```
import System.Collections.Generic;

class JSDelegate {
  private var _callbacks : List.<Function> = new List.<Function>();
  function Add (callback : Function) : void {
    _callbacks.Add(callback);
  }
  function Remove (callback : Function) : void {
    _callbacks.Remove(callback);
  }
  function Clear () : void {
    _callbacks.Clear();
  }
  function Invoke () : void {
    for (var callback in _callbacks) {
      callback();
    }
  }
}
```

If we check out this script, we will see that this script uses `List<T>`, which is available in the C# generic library, so we need to use `import System.Collections.Generic;`.

Note

`List<T>` is basically similar to JavaScript's the `Array` or `ArrayList` type, but we need to specify the `<T>` type. This also performs significantly faster than JavaScript's `Array` and `ArrayList`. A good link to understand which type of array to use in Unity is as follows:

http://wiki.unity3d.com/index.php?

title=Which_Kind_Of_Array_Or_Collection_Should_I_Use?

We will see that we have `List` to store the `Function` type. Then, we have `Add`, `Remove`, `Clear`, and `Invoke` functions. The `Invoke()` function is basically to call all the functions in `List`. This is similar to the `delegate` and `event` functions in C#.

Mission accomplished

In this project, we first created the `ragdoll` object and applied this to our `AI` character. Then, we created the destructible `Wall` object and destroyed it when we shot at it by adding some script to the `rocket` script. We also created the `Rockslide` game object, the `Rocks` script to enable and disable **Rigidbody** of the rocks, and the `TriggerArea` game object and `RocksTrigger` script to make the rocks fall down when the player hits the trigger area. The following screenshot shows these stages:

Hotshot challenges

Now, we have understood the concept of creating destructible objects, but the objects that we created are on the cube from the Unity engine. We can make it more interesting with something like the following:

- Create your own object in any 3D software instead of the cube to make it much more realistic and attach the `Rocks` script to it and see how it works
- Add some script that will make the rock damage the player and AIs when they get hit while the rock is falling down
- Add the smoke particle to the rocks when they are falling down
- Make a ragdoll match the last AI animation post by creating a ragdoll with the `AI` game object and using the **is Kinematic** method to enable or disable ragdoll physics instead of replacing the new object
- Create a random rock that will fall every time the player walks by the lake
- Create a trigger area such as door or switch that can trigger multiple events at the same time by using the `delegate` and `event` functions

Chapter 8. Let the World See the Carnage – Saving and Loading High Scores

The high score is the highest logged point value, which is used to show how well the players do in the game. Many a times, a game will have a list of several high scores called the high score table, which shows a list of scores that each player gets when playing the game. During the era of arcade games or some current puzzle games that involve endless cycles of continuous gameplay, scores had a much greater relevance in giving the player the replayable value and the feeling of achievement.

Why do we need to save the high score? The advantage of the high score is to keep a record of the players and how well they progress each time they play the game. It also creates a challenge for the players to beat their record and keep playing the game again. There are some games such as *Journey* that don't need the high scores. However, for most online games, the high score is very important to let the players see their progress and compare it with that of their friends or other players.

We can save the high scores on the player's local machine or use the database and keep it on the web server. In most case, we will use the database server to keep track of the high-score table, which is more secure. In this project, we will show you how to save the high score on the local machine using the **PlayerPref** and **System.Serializable** techniques. Then, we will also save and load the high score from the server database using **WWWFrom**, **XML**, and so on.

Mission briefing

This project will start with a creation of the simple high score table so that the players can submit their names and scores locally as well as post it to the server database. We will continue the project from Project 7, *Forge a Destructible and Interactive Virtual World*.

First, we will create the high score menu UI using `OnGUI()`, which will be visible when the game is over. The menu will include the final score, a text input area for the player's name, a submit button, a local hi-score button, a server hi-score button, and a restart button.

Then, we will have the `UserData` script that contains the name and score for each user. We will also create the `LocalHiscore` script, which we will use to save and load the local hi-score using `PlayerPref`, `BinaryFormatter`, and `MemoryStream` to serialize and deserialize the data.

Tip

The serialize and deserialize techniques will make it easy for us to save complex data as well as preventing the user from changing the data outside the game.

Next, we will create the `XMLParser` script to get the XML value from the provided database server and the `ServerHiscore` script to save and load the user's data to the server using the `www` object. We also encrypt our data for security purposes using the **MD5** encryption class written by Matthew Wegner, which we will download from http://wiki.unity3d.com/index.php?title=MD5.

Tip

The MD5 encryption script will allow us to encrypt the hash key, which will prevent the submission of fake high scores.

Why is it awesome?

What we will get from this project is a way to use `PlayerPrefs` combined with `BinaryFormatter` and `MemoryStream` to serialize and deserialize objects to save and load complex data. From this project, we will be able to adapt the technique to save more complex in-game data, such as the location of the player, current stage, or current hit points. We will also learn how to set up a basic database server using MySQL and PHP scripts to return high score data in the XML format and parse it to the game using `XMLDocument` from the .NET library. This is very useful when reading the XML file. Finally, we will get to know how to prevent the submission of fake data using the MD5 script to encrypt user data before sending it to the server database.

Your Hotshot objectives

We will start by importing the `chapter 8` package, and then we will go through each of the following topics:

- Creating the `UserData` and `Hiscore` scripts
- Saving and loading the local high score
- Creating an `XMLParser` script
- Saving and loading the server high score

Mission checklist

Before we start, we will need to get the project folder and assets from Packt's website, http://www.packtpub.com/support?nid=8267, which includes the first finished project and the assets that we need to use in this project.

Go to the preceding URL and download the `Chapter8.zip` package and unzip it. Inside the `Chapter8` folder, there are two unity packages, which are `Chapter8Package.unitypackage` (we will use this package for this project) and `Chapter8Package_Completed.unitypackage` (this is the completed project package).

Creating the UserData and Hiscore scripts

In the first section, we will create the `UserData` script, which will collect the username and score. This script will inherit from the `IComparable<T>` interface, which is a custom comparison method in the .NET library that a value type or class implements to create a type-specific comparison method to order instances. This class will have the `CompareTo()` function, which will be used to sort users by their score. Then, we will add the `UserData` object in the `UIHiscore` script and set up the input text field for the player's name.

Note

For more information about the `IComparable<T>` interface, visit the following links:

- http://msdn.microsoft.com/en-us/library/4d7sx9hd(v=vs.110).aspx
- http://www.dotnetperls.com/icomparable

Then, we will create the `Hiscore` script, which will be the super class of both the `LocalHiscore` and `ServerHiscore` scripts. The `Hiscore` script will include the necessary variables such as the array of the `UserData`, `Save` and `Load` functions, `Sort` function, and so on.

Prepare for lift off

We will begin with importing the package, preparing the assets folder, and making sure that we have everything ready to start. Let's create a new project and import the package:

1. Import the assets package by navigating to **Assets | Import Package | Custom Package…**, choose the `Chapter8.unityPackage` file, which we downloaded earlier, and then click on the **Import** button in the pop-up window, as shown in the following screenshot:

2. Wait until it's done and we will see the **Chapter8**, **Editor**, **Gizmos**, **Resources**, and **Standard Assets** folders in the **Project** view. Then, we will navigate to **Chapter8 | Scenes** and double-click on either scene (for C# users, double-click on the **Hiscore_C#** scene; for Unity JavaScript users, double-click on the **Hiscore_JS** scene), as shown in the following screenshot:

3. Next, we will navigate to **Chapter8** | **Scripts/C#/Hiscore** (for C# users) or **Chapter8** | **Scripts** | **Javascript** | **Hiscore** (for Unity JavaScript users), right-click and navigate to **Create** | **C# Script** (for C# users) or **Create** | **Javascript**, and name it `UserData`, as shown in the following screenshot:

Engage thrusters

Now, we can start creating the `UserData` script by performing the following steps:

1. Double-click on the `UserData` script that we just created and include the code as follows:

```
// Unity JavaScript user:

#pragma strict
import System;
import System.Collections.Generic;

public class UserData implements IComparable.<UserData> {
  var name : String;
  var score : int;

  public function CompareTo ( compare : UserData ) : int
  {
    // Null value means that this object is greater
    if (compare == null) return 1;
    else {
      return this.score.CompareTo(compare.score);
    }
  }
}


// C# user:

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

[System.Serializable]
public class UserData : IComparable<UserData> {
  public string name;
  public int score;

  public int CompareTo (UserData compare )
  {
    // Null value means that this object is greater
    if (compare == null) return 1;
    else {
      return this.score.CompareTo(compare.score);
    }
  }
}
```

In this step in C# script, we added `[System.Serializable]` to enable the ability to serialize the `UserData` object. This process will translate the object data structure to a series of bits, which can easily be stored in a file or memory buffer. On the other hand, we don’t need to include any script for Unity JavaScript because all Unity

JavaScript classes are automatically serialized. This script is used to contain the information for each user. This will be used with `BinaryFormatter` and `MemoryStream` to save multiple users' data to the local machine using `PlayerPrefs` in the *Saving and loading the local high score* section.

Tip

In C#, sometimes, we add `[System.Serializable]` in our code to embed a class with subproperties in the inspector. This is similar to how `Vector3` looks in the inspector. However, we don't need to add anything for Unity JavaScript because the `Serializable` attribute is implicit and not necessary.

More details can be found at http://docs.unity3d.com/Documentation/ScriptReference/Serializable.html.

2. Next, we will navigate to **Chapter8** | **Scripts** | **C#** | **UI** (for C# users) or **Chapter8** | **Scripts** | **Javascript** | **UI** (for Unity JavaScript users); double-click on **UIHiscore** to open it, as shown in the following screenshot:

3. Next, we will go inside the `UIHiscore` script to add the script. Let's add the variables, as shown in the following highlighted code:

```
// Unity JavaScript user:

…
var customSkin : GUISkin;
@Range(1,50) var maxUser : int = 10;
private var _user : UserData;


// C# user:

…
public GUISkin customSkin;
[Range(1,50)] public int maxUser = 10;
```

```
UserData _user;
```

4. Then, we will go to the `Start()` function and add the highlighted code as follows to create the current `UserData` script and set the default value for it:

```
// Unity JavaScript user:

…
function Start () {
  _page = PAGE.GAMEOVER;
  _clickRestart = false;
  _clickSubmit = false;
  _user = new UserData();
  _user.name = "PLAYER 1";
  _user.score = 0;
}


// C# user:

…
void Start () {
  _page = PAGE.GAMEOVER;
  _clickRestart = false;
  _clickSubmit = false;
  _user = new UserData();
  _user.name = "PLAYER 1";
  _user.score = 0;
}
```

5. Now go to the `Gameover()` function. Inside the `if (_clickSubmit == false) {` line, we will add the highlighted code as follows to create the input text field for the username:

```
// Unity JavaScript user:
…
private function Gameover ()
{
  …
  if (_clickSubmit == false) {
   GUI.Label(new Rect((Screen.width - 300)*0.5f, (Screen.height*0.1f) +
80, 300, 25), "Enter Your Name", GUI.skin.GetStyle("CustomText3"));
   //Creating the input text field to get the player name
     _user.name = GUI.TextField(new Rect((Screen.width - 240)*0.5f,
(Screen.height*0.1f) + 120, 240, 40), _user.name, 8);
     //Submit button
     if (GUI.Button(_buttonRect1, "SUBMIT")) {
       _clickSubmit = true;
       _user.score = HitPoint.CURRENT_SCORE;
       //Submitting both local and server high score here
     }
   }
  …
}
```

```
// C# user:
…
void Gameover ()
{
  …
  if (_clickSubmit == false) {
   GUI.Label(new Rect((Screen.width - 300)*0.5f, (Screen.height*0.1f) +
80, 300, 25), "Enter Your Name", GUI.skin.GetStyle("CustomText3"));
   //Creating the input text field to get the player name
     _user.name = GUI.TextField(new Rect((Screen.width - 240)*0.5f,
(Screen.height*0.1f) + 120, 240, 40), _user.name, 8);
     //Submit button
     if (GUI.Button(_buttonRect1, "SUBMIT")) {
       _clickSubmit = true;
       _user.score = HitPoint.CURRENT_SCORE;
       //Submitting both local and server high score here
     }
   }
  …
}
```

6. Now, we have finished adding `UserData` in the `UIHiscore` script. If we click on **Play** and let our character die or kill all enemies, we will see that there is the **GAMEOVER** UI similar to to the following screenshot (to make it faster to test, we can set the `CharacterClass.IS_GAMEOVER` to `true` and click on **Play** to see the result too):

The **SUBMIT**, **LOCAL HI-SCORE**, and **SERVER HI-SCORE** buttons don't do anything right now. We will add the functionality for each of the other buttons later.

7. Next, we will create the `Hiscore` script, which will be the base case of both the `LocalHiscore` and `ServerHiscore` scripts. So, let's navigate to **Chapter8** | **Scripts** |

C# | **Hiscore** (for C# users) or **Chapter8** | **Scripts** | **Javascript** | **Hiscore** (for Unity JavaScript users), right-click and navigate to **Create** | **C# Script** (for C# users) or **Create** | **Javascript** (for Unity JavaScript users), and name it `Hiscore`.

8. Double-click on the `Hiscore` script and add the following code:

```
// Unity JavaScript user:

#pragma strict
import System.Collections.Generic;

public class Hiscore extends MonoBehaviour {
  protected var _hashKey : String = "UNITYGAMEDEVELOPMENTHOTSHOT";
  protected var _privateMaxUser : int;
  protected var _users : List.<UserData>;

  function get userLength() : int {
    return (_users.Count < _privateMaxUser) ? _users.Count :
_privateMaxUser;
  }

  function Initialize ( maxUser : int  ) {
    _privateMaxUser = maxUser;
  }

  function LoadUserData () {
    _users = new List.<UserData>();
  }

  function SaveUserData ( user : UserData ) { }

  function GetUserDataAt ( index : int  ) : UserData {
    return _users[index];
  }

  function SortByScore () {
    _users.Sort();
    _users.Reverse();
  }
}


// C# user:

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

public class Hiscore : MonoBehaviour {
  protected string _hashKey = "UNITYGAMEDEVELOPMENTHOTSHOT";
  protected int _privateMaxUser;
  protected List<UserData> _users;

  public int userLength {
    get {
      return (_users.Count < _privateMaxUser) ? _users.Count :
```

```
_privateMaxUser;
    }
  }

  public virtual void Initialize ( int maxUser ) {
    _privateMaxUser = maxUser;
  }

  public virtual void LoadUserData () {
    _users = new List<UserData>();
  }

  public virtual void SaveUserData ( UserData user ) { }

  public UserData GetUserDataAt ( int index ) {
    return _users[index];
  }

  public void SortByScore () {
    _users.Sort();
    _users.Reverse();
  }
}
```

Now we have finished this step. In the next step, we will create the `LocalHiscore` script to save and load the user data from the local machine.

Objective complete – mini debriefing

In this step, we created the `UserData` script that implements from the `IComparable<T>` generic interface to create a custom sort for the `UserData` object. We also added `System.Serialiable` to the `UserData` script to enable the ability to serialize this object and save it in a later step. Then, we added the `UserData` object in the `UIHiscore` script and created the username input text field. Finally, we created the `Hiscore` class that will be the base class for the `LocalHiscore` and `ServerHiscore` scripts in the next step. In this `Hiscore` class, we created `List<UserData>` to store all the users' data. Then, we sorted them using `_users.Sort()` to sort from the lowest to the highest score. However, we need to show the score from the highest, so we also use `users.Reverse()` to reverse the sorting object.

Classified intel

In this section, we used the `IComparable<T>` generic interface to create a custom sorting method to sort the generic `List<T>` objects. We know that the generic `List<T>` objects are similar to `ArrayList`, except with the specific type; in this case, `UserData`.

To be able to order `List<T>`, we will call the `List<T>.Sort()` method. This method will automatically call the `CompareTo()` function, which is the method in `IComparable<T>`. This method is used to compare the value between two objects, which will return only three values (less than 0 or -1, 0, greater than 0 or 1). The value less than 0 means the first object is less than other object. 0 means both objects are equal. Then the value greater than 0 means the first object is greater than other object.

In the `Hiscore` script in `SortByScore()`, we will see that we used the `Sort()` function for `_users`, which is the `List<UserData>()` array. The `Sort()` method uses the default comparer, `Comparer<T>.Default`, for type `T` to determine the order of the list elements. Then, the `Comparer<T>.Default` property will check whether or not the `IComparable<T>` generic interface is implemented, which we already implemented in our `UserData` script.

The result from the sorting method will return `_users` from the lowest score to the highest score. So, we use the `Reverse()` function to reverse the result, which will give `_users` order from the highest to the lowest score.

Note

`List<T>` is in the .NET framework class library. We might want to check out the following MSDN Microsoft library links:

- For more information about the `List<T>` method, visit http://msdn.microsoft.com/en-us/library/s6hkc2c4(v=vs.110).aspx.
- For more information about the `Sort()` method, visit http://msdn.microsoft.com/en-us/library/b0zbh7b6(v=vs.110).aspx.
- For more information about the `Comparer<T>` class and example, visit http://msdn.microsoft.com/en-us/library/azhsac5f(v=vs.110).aspx.
- For more information about the `IComparable<T>` interface and the `CompareTo()` method, visit the following links: http://msdn.microsoft.com/en-us/library/4d7sx9hd(v=vs.110).aspx and http://msdn.microsoft.com/en-us/library/43hc6wht(v=vs.110).aspx respectively
- For more information about the `List<T>.Reverse()` method, visit http://msdn.microsoft.com/en-us/library/b0axc2h2(v=vs.110).aspx.

We can also overload the `Sort()` function to sort using the name of the user instead of the score by adding the following code in the `Hiscore` script:

```
// Unity JavaScript user:

function SortByName () {
  _users.Sort(function (user1 : UserData, user2 : UserData)
  {
    if (user1.name == null && user2.name == null) return 0;
```

```
    else if (user1.name == null) return -1;
    else if (user2.name == null) return 1;
    else return user1.name.CompareTo(user2.name);
  });
}


// C# user:
public void SortByName () {
  _users.Sort(delegate (UserData user1, UserData user2)
  {
    if (user1.name == null && user2.name == null) return 0;
    else if (user1.name == null) return -1;
    else if (user2.name == null) return 1;
    else return user1.name.CompareTo(user2.name);
  });
}
```

From the method, we will see that we still use the `Sort()` function, but this time, we compare using the name. The result will order the usernames from A to Z.

Saving and loading the local high score

In this section, we will be creating the `LocalHiscore` script, which we will use to save and load the users' data. We will use `PlayerPrefs`, `BinaryFormatter`, and `MemoryStream` to save and load the user data. Then, we will add the ability to save and load the local high score to our `UIHiscore` script.

Engage thrusters

Now we can start creating the `LocalHiscore` script using the following steps:

1. Navigate to **Chapter8** | **Scripts** | **C#/Hiscore** (for C# users) or **Chapter8** | **Scripts** | **Javascript** | **Hiscore** (for Unity JavaScript users), right-click and navigate to **Create** | **C# Script** (for C# users) or **Create** | **Javascript**, and name it `LocalHiscore`.
2. First, we will create a random name for the default user and the `SaveUserData()` function to save the user data. Let's double-click on the `LocalHiscore` script that we just created and use the following code:

```
// Unity JavaScript user:

#pragma strict
import System;
import System.Runtime.Serialization.Formatters.Binary;
import System.IO;
import System.Collections.Generic;

class LocalHiscore extends Hiscore {
  private var RANDOM_NAMES : String[] = ["Antony", "John", "Will",
"Kate", "Jill"];

  override function SaveUserData ( user : UserData ) {
    _users.Add(user);
    var binary : BinaryFormatter = new BinaryFormatter();
    var memory : MemoryStream = new MemoryStream();
    binary.Serialize(memory, _users);

PlayerPrefs.SetString(_hashKey+"Highscore",Convert.ToBase64String(memor
y.GetBuffer()));
  }
}


// C# user:

using UnityEngine;
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
using System.Collections;
using System.Collections.Generic;

public class LocalHiscore : Hiscore {
  private string[] RANDOM_NAMES = new string[] {"Antony", "John",
"Will", "Kate", "Jill"};

  public override void SaveUserData ( UserData user ) {
    _users.Add(user);
    BinaryFormatter binary = new BinaryFormatter();
    MemoryStream memory = new MemoryStream();
    binary.Serialize(memory, _users);
```

```
PlayerPrefs.SetString(_hashKey+"Highscore",Convert.ToBase64String(memor
y.GetBuffer()));
  }
}
```

We created the save function by first adding the new user to the list, and then we created `BinaryFormatter` to serialize all users data to `MemoryStream`. Finally, we saved the users data by converting it to a string and saving it in `PlayerPrefs`.

3. Next, add two more functions—`Initialize()` and `LoadUserData()`—to set and load the user data. Let's add both the methods as follows:

```
// Unity JavaScript user:

…
override function Initialize ( maxUser : int ) {
  super.Initialize(maxUser);
  LoadUserData();
}

override function LoadUserData () {
  super.LoadUserData();
  var data : String PlayerPrefs.GetString(_hashKey+"Highscore");

  if (data != "") {
    var binary : BinaryFormatter = new BinaryFormatter();
    var memory : MemoryStream = new
MemoryStream(Convert.FromBase64String(data));
    _users = binary.Deserialize(memory) as List.<UserData>;
  } else {
    for (var i : int = 0 ; i < RANDOM_NAMES.Length; ++i) {
      var user : UserData = new UserData();
      user.name = RANDOM_NAMES[i];
      user.score = Mathf.FloorToInt(UnityEngine.Random.value * 1000);
      _users.Add(user);
    }
  }
  this.SortByScore();
}
…


// C# user:

…
public override void Initialize ( int maxUser ) {
  base.Initialize(maxUser);
  LoadUserData();
}
public override void LoadUserData () {
  base.LoadUserData();
  string data = PlayerPrefs.GetString(_hashKey+"Highscore");
  if (data != "") {
    BinaryFormatter binary = new BinaryFormatter();
    MemoryStream memory = new
MemoryStream(Convert.FromBase64String(data));
```

```
    _users = binary.Deserialize(memory) as List<UserData>;
  } else {
    for (int i = 0 ; i < RANDOM_NAMES.Length; ++i) {
      UserData user = new UserData();
      user.name = RANDOM_NAMES[i];
      user.score = Mathf.FloorToInt(UnityEngine.Random.value * 1000);
      _users.Add(user);
    }
  }
  this.SortByScore();
}
```

We just created the function to load the string data using `PlayerPrefs.GetString(key,"")` from the key. We also checked that it was not an empty string. Then, we deserialized the data by first converting it from string to the byte array using `MemoryStream`, and then we used `BinaryFormatter` to deserialize the data to `List<UserData>`.

4. Next, we will go to the `UIHiscore` script to add the script to initialize the local hi-score table, as shown in the following highlighted code:

```
// Unity JavaScript user:
…
private var _user : UserData;
private var _localHiscore : LocalHiscore;
private var _scrollLocal : Vector2 = Vector2.zero;
…
function Awake () {
  _localHiscore = GetComponent.<LocalHiscore>();
  if (_localHiscore == null) {
    _localHiscore = gameObject.AddComponent.<LocalHiscore>();
  }
}
function Start () {
  …
  _localHiscore.Initialize(maxUser);
}


// C# user:
…
UserData _user;
LocalHiscore _localHiscore;
Vector2 _scrollLocal = Vector2.zero;
…
void Awake () {
  _localHiscore = GetComponent<LocalHiscore>();
  if (_localHiscore == null) {
    _localHiscore = gameObject.AddComponent<LocalHiscore>();
  }
}
void Start () {
   …
   _localHiscore.Initialize(maxUser);
}
```

5. Next, we will go to the `Gameover()` function to add the load and save data functions when the user clicks the button, as shown in the following highlighted code:

```
// Unity JavaScript user:
…
private function Gameover ()
{
  …
  if (_clickSubmit == false) {
    …
    if (GUI.Button(_buttonRect1, "SUBMIT")) {
       …
       _user.score = HitPoint.CURRENT_SCORE;
       _localHiscore.SaveUserData(_user);
    }
  }
  if (GUI.Button(_buttonRect2, "LOCAL HI-SCORE")) {
  // Load
    _localHiscore.LoadUserData();
    _page = PAGE.LOCALSCORE;
  }
…
}


// C# user:
…
void Gameover ()
{
   …
  if (_clickSubmit == false) {
    …
    if (GUI.Button(_buttonRect1, "SUBMIT")) {
       …
       _user.score = HitPoint.CURRENT_SCORE;
       _localHiscore.SaveUserData(_user);
    }
  }
  if (GUI.Button(_buttonRect2, "LOCAL HI-SCORE")) {
  // Load
    _localHiscore.LoadUserData();
    _page = PAGE.LOCALSCORE;
  }
…
}
```

6. Finally, we will go to the `LocalHiscore()` function to add the code to show the high-score table data, as shown in the following highlighted code:

```
// Unity JavaScript user:
…
private function LocalHiscore ()
{
  CreateBackgroundBox("LOCAL HI-SCORE");
  if (_localHiscore.userLength > 0) {
    _scrollLocal = GUI.BeginScrollView (new Rect ((Screen.width -
```

```
320)*0.5f, (Screen.height*0.1f) + 80, 320, 180), _scrollLocal, new Rect
(0, 0, 300, 30*_localHiscore.userLength));
    for (var i : int = 0; i < _localHiscore.userLength; i++) {
      GUILayout.BeginHorizontal(GUILayout.Width(300));
      GUILayout.Label((i+1).ToString() + ". " +
_localHiscore.GetUserDataAt(i).name, GUI.skin.GetStyle("Name"));
      GUILayout.Label(_localHiscore.GetUserDataAt(i).score.AddCommas(),
GUI.skin.GetStyle("Score"));
      GUILayout.EndHorizontal();
    }
    GUI.EndScrollView();
  }
…
}


// C# user:
…
void LocalHiscore ()
{
  CreateBackgroundBox("LOCAL HI-SCORE");
  if (_localHiscore.userLength > 0) {
    _scrollLocal = GUI.BeginScrollView (new Rect ((Screen.width -
320)*0.5f, (Screen.height*0.1f) + 80, 320, 180), _scrollLocal, new Rect
(0, 0, 300, 30*_localHiscore.userLength));
    for (int i = 0; i < _localHiscore.userLength; i++) {
      GUILayout.BeginHorizontal(GUILayout.Width(300));
      GUILayout.Label((i+1).ToString() + ". " +
_localHiscore.GetUserDataAt(i).name, GUI.skin.GetStyle("Name"));
      GUILayout.Label(_localHiscore.GetUserDataAt(i).score.AddCommas(),
GUI.skin.GetStyle("Score"));
      GUILayout.EndHorizontal();
    }
    GUI.EndScrollView();
  }
   …
}
```

We just created the scroll view for our local high-score table. Now, we have finished this step, we can click on the **Play** button to see the result. Have a look at the following screenshot:

GAMEOVER
Final Score: 2,340
Enter Your Name
PLAYER 1
SUBMIT
LOCAL HI-SCORE
SERVER HI-SCORE
RESTART
LOCAL HI-SCORE
1. Antony 947
2. Will 832
3. John 670
4. Kate 368
5. Jill 123
BACK
RESTART
GAMEOVER
Final Score: 2,340
LOCAL HI-SCORE
SERVER HI-SCORE
RESTART
LOCAL HI-SCORE
1. PLAYER 1 2,340
2. Antony 947
3. Will 832
4. John 670
5. Kate 368
6. Jill 123
BACK
RESTART

Objective complete – mini debriefing

In this step, we basically created `LocalHiscore` to save and load the user's local high-score data and displayed it on the **LOCAL HI-SCORE** page. The `LocalHiscore` class is derived from the `Hiscore` class, which we already created. In this class, we added the `Initialize()`, `LoadUserData()`, and `SaveUserData()` methods, which is overridden from the `Hiscore` base class. Using the `virtual` and `override` keywords, we will be able to share the method and variables from the `Hiscore` class.

In the `SaveUserData()` function, we first passed the user data that we'd like to save. Then, we added the user data to the list of `UserData`, which is `_users.Add(user);`. Next, we created `BinaryFormatter` and `MemoryStream` to serialize the list of `UserData` to the series of bits to the memory buffer using `binary.Serialize(memory, _users);`. Next, we used `PlayerPrefs` to save the data by first converting the string using `Convert.ToBase64String(memory.GetBuffer());`. Then, we used `PlayerPrefs.SetString()` to set the data by passing the key, which is `_hashkey + "Highscore"`, and the converting string data.

Note

We need to convert our data to a string first. This is because `PlayerPrefs` can only save `string`, `int`, or `float` and not the bytes array. We can check the following link for more details: http://docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.html.

Next, we created `Initialize()` and `LoadUserData()` to set up and load the default user data if there is no local user data. In the first line, we called `base.Initialize()` (for C# users) and `super.Initialize()` (for Unity JavaScript users). This basically calls the `Initialize()` method from the base class, which is the `Hiscore` class to set the maximum users that should be shown on the high-score table.

In the `LoadUserData()` method, we first checked whether the local user data exists or not using `string data = PlayerPrefs.GetString(_hashkey+"Highscore", "");`. Then, we checked to see whether the return string isn't equal to an empty string. We converted it back to the memory buffer and deserialized it to the list of `UserData`. On the other hand, if the data was an empty string, we added the random user to the list of `UserData`. Then, we sorted the list based on the score of each user.

Finally, we added the script to the `UIHiscore` class to save, load, and show the local high score from the list of `UserData`.

Classified intel

In Unity C#, we used `[System.Serializable]` to enable the ability to serialize this object. **Serialization** is the process of translating the data structure or object state in the format that can easily be stored (for example, a file/memory buffer, or transmitted data across a network connection link) and reconstructed later in the same or another computer environment.

Note

More explanation about serialization can be found from the following links:

- http://stackoverflow.com/questions/3042665/what-is-the-meaning-of-serialization-concept-in-programming-languages
- http://en.wikipedia.org/wiki/Serialization

Whether you add `@script System.Serializable` in the class or not, we will be able to serialize them.

The serialization technique can be stored as a file and saved on your local machine too. This is a very efficient way to save the game data, such as a player's hit point, player's location, and enemies' hit point. To store the data to the file, we can add something similar to the following code:

```
// Unity JavaScript user:

function SaveData () {
  var binary : BinaryFormatter = new BinaryFormatter();
  var file : FileStream = new
FileStream(Application.persistentDataPath+"/highscores.dat",
FileMode.Append);
  binary.Serialize(file, _users);
}
function LoadData () {
  try {
    var binary : BinaryFormatter = new BinaryFormatter();
    var file : FileStream = new
FileStream(Application.persistentDataPath+"/highscoresJS.dat",
FileMode.Open);
    _users = binary.Deserialize(file) as List.<UserData>;
  }  catch (error)  {
    Debug.Log("File not found");
  }
}


// C# user:

public void SaveData () {
  BinaryFormatter binary = new BinaryFormatter();
  FileStream file = new
FileStream(Application.persistentDataPath+"/highscores.dat",
FileMode.Append);
```

```
  binary.Serialize(file, _users);
}
public void LoadData () {
  try {
    BinaryFormatter binary = new BinaryFormatter();
    FileStream file = new
FileStream(Application.persistentDataPath+"/highscores.dat",
FileMode.Open);
    _users = binary.Deserialize(file) as List<UserData>;
  } catch (System.IO.FileNotFoundException) {
    Debug.Log("File not found");
  }
}
```

Next, we will talk about `PlayerPrefs`, which is basically used to save and load the data using the key string to identify each piece of data. The values that we can set or get are `string`, `float`, and `int`. We will use `PlayerPrefs.SetString(Key,Value)`, `PlayerPrefs.SetFloat(Key,Value)`, and `PlayerPrefs.SetInt(Key,Value)` to store the data. These methods will set each value based on the key string and save them to the local machine when the user quits the application. However, we might want to write the data after setting the value. In this case, we will use the `PlayerPrefs.Save()` function to write the data at the time we call this method.

Tip

It's not recommend that you call `PlayerPrefs.Save()` during the actual gameplay, because it will write the data to the disk and cause small hiccups. More details can be found at http://docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.Save.html.

We use `PlayerPrefs.GetString(Key,DefaultValue="")`, `PlayerPrefs.GetFloat(Key,DefaultValue=0.0f)`, and `PlayerPrefs.GetInt(Key,DefaultValue=0)` to load the data. These methods will get the data value from the given key string. If the key cannot be found, it will return the default value depending on the individual function. We can also use `PlayerPrefs.HasKey(Key)` to check if the key exists or not. At last, if we want to remove the key data, we can use `PlayerPrefs.Delete(Key)` to remove the specific key. Also, if we want to remove all the keys, we can use `PlayerPrefs.DeleteAll()`.

Note

For more details on `PlayerPrefs`, visit http://docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.html.

Creating an XMLParser script

In this section, we will create the `XMLParser` script that will parse the XML data from the server to use for the *Saving and loading server high score* section. We will use `XmlDocument` from the .NET framework in C#.

Engage thrusters

Now, we can start creating the `XMLParser` script using the following steps:

1. We will navigate to **Chapter8** | **Scripts** | **C#** | **Hiscore** (for C# users) or **Chapter8** | **Scripts** | **Javascript** | **Hiscore** (for Unity JavaScript users), and right-click and go to **Create** | **C# Script** (for C# users) or **Create** | **Javascript**. We will name it `XMLParser` and double-click on it to open the `XMLParser` file.
2. First, we will start coding at the beginning of the `XMLParser` script, as shown in the following code:

```
// Unity JavaScript user:

#pragma strict
import System.Xml;


// C# user:

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

   We added `using System.Xml`, and `import System.Xml` allows us to access the `System.Xml` library in the .NET framework.

3. Next, we will create the `XMLParser` class as the static class, which allows us to access the method of this class directly without creating a new `XMLParser` object and all the necessary variables. Let's add the following code:

```
// Unity JavaScript user:

public static class XMLParser {
  private var _doc : XmlDocument;
  private var _root : XmlNode;
  private var _users : UserData[];
  private var _userLength : int;

  public function get users() : UserData[] {
    return _users;
  }
  public function get usersLength () : int {
    return _userLength;
  }
}


// C# user:

public static class XMLParser {
  static XmlDocument _doc;
  static XmlNode _root;
  static UserData[] _users;
  static int _usersLength;
```

```
  public static UserData[] users {
    get { return _users; }
  }
  public static int usersLength {
    get { return _usersLength; }
  }
}
```

4. Then, we will add the `Parse()` function, which will be used to parse the XML string to the array of `UserData`. Let's add this method inside the `XMLParser` class, as shown in the following highlighted code:

```
// Unity JavaScript user:

public static class XMLParser {
  …
  public function get usersLength () : int {
    return _userLength;
  }

public function Parse(  xml : String ) : void {
    _doc = new XmlDocument();
    _doc.LoadXml(xml); // Loading from String
    //Using doc.Load("HiScore.xml"); When load from an xml file
    //Using Last Child to Skip the <?xml version="1.0" encoding="UTF-
8"?>
    //If we load from the xml file we will use the FirstChild instead
    _root = _doc.LastChild;
    if (_root.HasChildNodes ) {
      _userLength = _root.ChildNodes.Count;
      _users = new UserData[_userLength];
      for (var i : int = 0; i < _userLength; i++) {
        if (_root.ChildNodes[i].InnerText.Contains("No entries yet.")
== false) {
          var nameAtt : XmlAttribute =
_root.ChildNodes[i].Attributes["name"];
          var scoreAtt : XmlAttribute =
_root.ChildNodes[i].Attributes["score"];
          var user : UserData = new UserData();
          user.name = nameAtt.Value;
          user.score = parseInt(scoreAtt.Value);
          _users[i] = user;
        } else {
          break;
        }
      }
    }
  }
}
// C# user:

public static class XMLParser {
  …
  public static int usersLength {
    get { return _usersLength; }
```

```
  }

  public static void Parse( string xml) {
    _doc = new XmlDocument();
    _doc.LoadXml(xml); // Loading from String
    //Using doc.Load("HiScore.xml"); When load from an xml file
    //Using Last Child to Skip the <?xml version="1.0" encoding="UTF-
8"?>
    //If we load from the xml file we will use the FirstChild instead
    _root = _doc.LastChild;
    if (_root.HasChildNodes ) {
      _usersLength = _root.ChildNodes.Count;
      _users = new UserData[_usersLength];
      for (int i = 0; i < _usersLength; i++) {
        if (_root.ChildNodes[i].InnerText.Contains("No entries yet.")
== false) {
          XmlAttribute nameAtt =
_root.ChildNodes[i].Attributes["name"];
          XmlAttribute scoreAtt =
_root.ChildNodes[i].Attributes["score"];
          UserData user = new UserData();
          user.name = nameAtt.Value;
          user.score = Convert.ToInt32(scoreAtt.Value);
          _users[i] = user;
        } else {
          break;
        }
      }
    }
  }
}
```

Objective complete – mini debriefing

In this section, we basically just created the `XMLParser` script to parse the XML string that we loaded from the server, and then we stored the user's data in this class to use it at a later stage.

First, we used the `static` keyword for this class because we wanted it to be accessible from the entire project. Then, we created the `XmlDocument` and `XmlNode` parameters to hold the XML data that we want to parse. Then, we had the array of `UserData` to contain all the user data. The last parameter is to store the length of the users that we have got from the XML data.

Next, we created the `Parse(string xml)` function. In this function, we created `XmlDocument` and then loaded the XML string data of this document using `_doc.LoadXml(xml)` to load the string XML that we pass from the server. Then, we get `XmlNode` from the last child of `XmlDocument` as follows:

```
_root = _doc.LastChild;
```

We used `LastChild()` because we wanted to skip the first node, which is the headlineof the `<?xml version="1.0" encoding="UTF-8"?>` XML file. Next, we checked if the `_root` node has a child node or not. If it has, we get the length of this child node. Then, we created the array of `UserData` to store username and score data.

Next, we looped through the child node, got the name and score from the XML attribute, set it to the `UserData` object, and then we added the data to the array of `UserData`.

Classified intel

From the previous section, if we go to the `UIHiscore` script at the `LocalHiscore()` function inside the `for` loop, we will see that we have shown the score with the commas as the following script:

```
_localHiscore.GetUserDataAt(i).score.AddCommas()
```

As we can see from the preceding script, `_localHiscore.GetUserDataAt(i).score` is the `int` type, which doesn't have the `AddCommas()` method to call. So, why doesn't it work? Let's go to the **Project** view in Unity and go to the **Standard Assets** folder; we will see the **ExtensionMethods** C# script, as shown in the following screenshot:

We can double-click on the file to open it. We will see that there are four methods here. All these are the extension methods. The extension method is the custom method for the class, struct, or variable that we don't have the permission or access to modify, such as `float`, `int`, `Transform`, and `Vector3`. To create the extension method, we need to create a static class and the static method, which need to have the first parameter set to `this` followed by its class type and the class name, as shown in the following code:

```
using UnityEngine;
using System.Collections;
public static class ExtensionMethods {
  public static void ResetTransform ( this Transform t )
  {
    t.position = Vector3.zero;
    t.rotation = Quaternion.identity;
    t.localScale = Vector3.one;
  }
}
```

To use it, we can just call the `transform.ResetTransform()` function from the game object.

This extension method can only be created in C#. So, how can we use it if we use Unity JavaScript? We can use the extension method by reordering the script complier to compile the `ExtensionMethods` script before other scripts. This is because all the Unity JavaScript

will get executed before the C# script. To solve this problem, we can either put the `ExtensionMethods` script in the **Standard Assets** folder or navigate to **Edit | Project Settings | Script Execution Order** and add the script that we want to reorder.

Note

For more details, visit the following links:

- http://docs.unity3d.com/Documentation/Manual/ScriptCompileOrderFolders.html
- http://docs.unity3d.com/Documentation/Components/class-ScriptExecution.html

Saving and loading server high score

In this section, we will create the `ServerHiScore` script to post and load the high score data from the server, which also inherits from the `Hiscore` class. We will use the `WWWForm` class to communicate with the PHP file on the website, which I already set up on my website. We will also use the hash key and encrypt it with MD5 encryption to protect and check for the user before posting the score to the database.

Prepare for lift off

Before we create the `ServerHiScore` script, we will take a look at this implementation of MD5 encryption, which is written by Matthew Wegner (http://www.unifycommunity.com/wiki/index.php?title=MD5).

If we go to the **Project** view, navigate to **Chapter8** | **Scripts** | **C#** | **Utils** (for C# users) or **Chapter8** | **Scripts** | **Javascript** | **Utils** (for Unity JavaScript users), and double-click on it to open the MD5 file. We will see that there is `Md5Sum(string)` to encrypt the string data.

Engage thrusters

Let's create the `ServerHiScore` script first using the following steps:

1. Navigate to **Chapter8** | **Scripts** | **C#** | **Hiscore** (for C# users) or **Chapter8** | **Scripts** | **Javascript** | **Hiscore** (for Unity JavaScript users) and right-click and go to **Create** | **C# Script** (for C# users) or **Create** | **Javascript**. Name it `ServerHiscore` and double-click to open the `ServerHiscore` file.
2. First, we will start coding at the beginning of the `ServerHiscore` script, as shown in the following code:

```
//Unity JavaScript user:

#pragma strict
class ServerHiscore extends Hiscore
{
  private var _wwwForm : WWWForm;
  private var _isLoaded : boolean;
  private var _phpUrl : String;

  function get isLoaded () : boolean {
    return _isLoaded;
  }
  function Initialize ( maxUser : int , phpUrl : String  ) {
    super.Initialize(maxUser);
    _phpUrl = phpUrl;
  }
}
// C# user:

using UnityEngine;
using System.Collections;
public class ServerHiscore : Hiscore
{
  public delegate void CallBackFunction ( string data );
  WWWForm _wwwForm;
  bool _isLoaded;
  string _phpUrl;

  public bool isLoaded {
    get { return _isLoaded; }
  }
  public void Initialize ( int maxUser, string phpUrl) {
    base.Initialize(maxUser);
    _phpUrl = phpUrl;
  }
}
```

   From the preceding code, we created `WWWForm` to send and load the data to the server. We also created the `Initialize()` function, which will set the maximum number of users that can be displayed on the UI. Also, we set the PHP URL that will be used for positing and loading data from the server.

3. Next, we will create the `WaitForServerResponse()` function, which will wait for the

server to finish sending the data. In this function, we will use the coroutine to wait for the server, and then, if there is no error, we will call the callback function. Let's add the `WaitForServerResponse()` function to this class as follows:

```
// Unity JavaScript user:

private function WaitForServerResponse ( www : WWW, callback : Function
) : IEnumerator
{
  yield www;
  if (www.error == null) {
    Debug.Log("Successful.");
    if (callback != null) {
      callback(www.text);
      callback = null;
    }
  } else {
    Debug.LogError("Failed.");
  }
  //Clear Data
  www.Dispose();
}


// C# user:

IEnumerator WaitForServerResponse ( WWW www, CallBackFunction callback)
{
  yield return www;
  if (www.error == null) {
    Debug.Log("Successful.");
    if (callback != null) {
      callback(www.text);
      callback = null;
    }
  } else {
    Debug.LogError("Failed.");
  }
  //Clear Data
  www.Dispose();
}
```

4. Then, we will add the `SaveUserData()` method, which will create `WWWForm`, set the data, and send it to the server using the `WWW` object. On the last line, we will use `StartCoroutine` to wait for the server to finish reading the data, as shown in the following code:

```
// Unity JavaScript user:

override function SaveUserData ( user : UserData ) {
  _wwwForm = new WWWForm();
  // Calling PHP for posting score action
  _wwwForm.AddField("action","PostScore");
  // Encrypt with MD5
  _wwwForm.AddField("hash",MD5.Md5Sum(user.name + "-" +
```

```
user.score.ToString() + "-" + _hashKey));
  _wwwForm.AddField("score",user.score);
  _wwwForm.AddField("name",user.name);
  StartCoroutine(WaitForServerResponse(new
WWW(_phpUrl,_wwwForm),null));
}


// C# user:

public override void SaveUserData ( UserData user ) {
  _wwwForm = new WWWForm();
  // Calling PHP for posting score action
  _wwwForm.AddField("action","PostScore");
  // Encrypt with MD5
  _wwwForm.AddField("hash",MD5.Md5Sum(user.name + "-" +
user.score.ToString() + "-" + _hashKey));
  _wwwForm.AddField("score",user.score);
  _wwwForm.AddField("name",user.name);
  StartCoroutine(WaitForServerResponse(new
WWW(_phpUrl,_wwwForm),null));
}
```

5. From the preceding code, we have the function to send data. Now, we will create the function to load data. Before creating the function to load data, we need to create the `ParseData()` function, which will be used to parse the XML data that is sent from the server. Let's add the following function:

```
// Unity JavaScript user:

private function ParseXMLData ( xmlText  : String)
{
  Debug.Log("XML Text = " + xmlText);
  XMLParser.Parse(xmlText);
  if (XMLParser.users != null) {
    _users.AddRange(XMLParser.users);
  }
  _isLoaded = true;
}


// C# user:

void ParseXMLData ( string xmlText )
{
  Debug.Log("XML Text = " + xmlText);
  XMLParser.Parse(xmlText);
  if (XMLParser.users != null) {
    _users.AddRange(XMLParser.users);
  }
  _isLoaded = true;
}
```

6. Then, we will add the last method, `LoadUserData()`, to load the user data from the server, wait until it is finished, and then call `ParseXMLData()` to parse the data to

_users. Let's add the following code:

```
// Unity JavaScript user:

override function LoadUserData () {
  super.LoadUserData();
  _isLoaded = false;
  _wwwForm = new WWWForm();
  // Calling PHP for getting score action
  _wwwForm.AddField("action","GetScore");
  StartCoroutine(WaitForServerResponse(new
WWW(_phpUrl,_wwwForm),ParseXMLData));
}


// C# user:

public override void LoadUserData () {
  base.LoadUserData();
  _isLoaded = false;
  _wwwForm = new WWWForm();
  // Calling PHP for getting score action
  _wwwForm.AddField("action","GetScore");
  StartCoroutine(WaitForServerResponse(new
WWW(_phpUrl,_wwwForm),ParseXMLData));
}
```

7. Now, we finished creating the `ServerHiscore` script. Next, we will go to the `UIHiscore` script to add the code that will show all users' scores from the server. First, we will add the script to initialize the server hi-score table, as shown in the following highlighted code:

```
// Unity JavaScript user:
…
…
private var _scrollLocal : Vector2 = Vector2.zero;
var phpUrl : String = "http://www.jatewit.com/Packt/HiScore.php";
private var _scrollServer : Vector2 = Vector2.zero;
private var _serverHiscore : ServerHiscore;
…
function Awake () {
  _localHiscore = GetComponent.<LocalHiscore>();
  if (_localHiscore == null) {
    _localHiscore = gameObject.AddComponent.<LocalHiscore>();
  }
  _serverHiscore = GetComponent.<ServerHiscore>();
  if (_serverHiscore == null) {
    _serverHiscore = gameObject.AddComponent.<ServerHiscore>();
  }
}
function Start () {
  …
  _localHiscore.Initialize(maxUser);
  _serverHiscore.Initialize(maxUser,phpUrl);
}
```

```
// C# user:
…
…
Vector2 _scrollLocal = Vector2.zero;
public string phpUrl = "http://www.jatewit.com/Packt/HiScore.php";
Vector2 _scrollServer = Vector2.zero;
ServerHiscore _serverHiscore;
…
void Awake () {
  _localHiscore = GetComponent<LocalHiscore>();
  if (_localHiscore == null) {
    _localHiscore = gameObject.AddComponent<LocalHiscore>();
  }
  _serverHiscore = GetComponent<ServerHiscore>();
  if (_serverHiscore == null) {
    _serverHiscore = gameObject.AddComponent<ServerHiscore>();
  }
}
void Start () {
   …
   _localHiscore.Initialize(maxUser);
   _serverHiscore.Initialize(maxUser,phpUrl);
}
```

8. We are still in the `UIHiscore` class. We will go to the `Gameover()` function to add the load and save data functions when the user clicks on the button, as shown in the following highlighted code:

```
// Unity JavaScript user:
…
private function Gameover ()
{
  …
  if (_clickSubmit == false) {
    …
    if (GUI.Button(_buttonRect1, "SUBMIT")) {
       …
       _localHiscore.SaveUserData(_user);
       _serverHiscore.SaveUserData(_user);
    }
  }
  if (GUI.Button(_buttonRect2, "LOCAL HI-SCORE")) {
  …
  }
  if (GUI.Button(_buttonRect3, "SERVER HI-SCORE")) {
     _serverHiscore.LoadUserData();
     _page = PAGE.SERVERSCORE;
  }
}


// C# user:
…
void Gameover ()
```

```
{
   …
  if (_clickSubmit == false) {
    …
    if (GUI.Button(_buttonRect1, "SUBMIT")) {
       …
       _localHiscore.SaveUserData(_user);
       _serverHiscore.SaveUserData(_user);
    }
  }
  if (GUI.Button(_buttonRect2, "LOCAL HI-SCORE")) {
  …
  }
  if (GUI.Button(_buttonRect3, "SERVER HI-SCORE")) {
     _serverHiscore.LoadUserData();
     _page = PAGE.SERVERSCORE;
  }
}
```

9. At last, we will go to the `ServerHiscore()` function to add the code to show the high score table data, as shown in the following highlighted code:

```
// Unity JavaScript user:
…
private function ServerHiscore ()
{
  CreateBackgroundBox("SERVER HI-SCORE");
  if (_serverHiscore.isLoaded && (_serverHiscore.userLength > 0)) {
    _scrollServer = GUI.BeginScrollView (new Rect ((Screen.width -
320)*0.5f, (Screen.height*0.1f) + 80, 320, 180), _scrollServer, new
Rect (0, 0, 300, 30*_serverHiscore.userLength));
    for (var i : int = 0; i < _serverHiscore.userLength; i++) {
      GUILayout.BeginHorizontal(GUILayout.Width(300));
      GUILayout.Label((i+1).ToString() + ". " +
_serverHiscore.GetUserDataAt(i).name, GUI.skin.GetStyle("Name"));

GUILayout.Label(_serverHiscore.GetUserDataAt(i).score.AddCommas(),
GUI.skin.GetStyle("Score"));
      GUILayout.EndHorizontal();
    }
    GUI.EndScrollView();
  } else {
    GUI.Label(new Rect((Screen.width-150)*0.5f,
(Screen.height*0.1f)+120, 150, 50), "LOADING…",
GUI.skin.GetStyle("CustomText"));
  }
  if (GUI.Button(_buttonRect3, "BACK")) {
    _page = PAGE.GAMEOVER;
  }
}


// C# user:
…
void ServerHiscore ()
{
```

```
  CreateBackgroundBox("SERVER HI-SCORE");
  if (_serverHiscore.isLoaded && (_serverHiscore.userLength > 0)) {
    _scrollServer = GUI.BeginScrollView (new Rect ((Screen.width -
320)*0.5f, (Screen.height*0.1f) + 80, 320, 180), _scrollServer, new
Rect (0, 0, 300, 30*_serverHiscore.userLength));
    for (int i = 0; i < _serverHiscore.userLength; i++) {
      GUILayout.BeginHorizontal(GUILayout.Width(300));
      GUILayout.Label((i+1).ToString() + ". " +
_serverHiscore.GetUserDataAt(i).name, GUI.skin.GetStyle("Name"));

GUILayout.Label(_serverHiscore.GetUserDataAt(i).score.AddCommas(),
GUI.skin.GetStyle("Score"));
      GUILayout.EndHorizontal();
    }
    GUI.EndScrollView();
  } else {
    GUI.Label(new Rect((Screen.width-150)*0.5f,
(Screen.height*0.1f)+120, 150, 50), "LOADING…",
GUI.skin.GetStyle("CustomText"));
  }
  if (GUI.Button(_buttonRect3, "BACK")) {
    _page = PAGE.GAMEOVER;
  }
}
```

We just created the scroll view for our server high score table. Now, we have finished this project. We can click on the **Play** button to see the result.

Objective complete – mini debriefing

In this section, we learned how to use the `WWWForm` and `WWW` objects to post and load the high score from/to the server. We also used the MD5 encryption to encrypt the key before posting the data to protect it from unwanted users. Then, we used the `StartCoroutine()` function to wait for the response from the server.

First, we created the `ServerHiScore` script to send and receive the user data from the server database. In the `Initialize()` function, we set the maximum number of users that will be shown in the UI. We also set the PHP URL for communication with the server database.

Next, we have the `WaitingForResponse()` function, which will wait for the response from the server and check if the sending request succeeds. Then, we checked if there is any callback function to call. If there is a callback function, we will call it after the communication has been finished. Finally, we just cleared all data using `www.Dispose()`.

Then, in the `SaveUserData()` function, we first created `WWWForm`. Then, we used `AddField("action", "PostScore");`, which will tell PHP that we want to send the score by setting `action` to `PostScore`. The `action` parameter and `PostScore` value are set in the PHP code, which you can see in `HiScore.php` by navigating to **Chapter8** | **PHP** in the **Project** view included with the package. Then, we added `hash` filed with the MD5 encryption value of the combination of `user.name`, `user.score`, and `hashKey`. We also added the `score` and `name` fields and set their value to the `WWWForm` object. In the last line, we used the `StartCoroutine(WaitingForResponse(new WWW(_phpUrl, _wwwForm), null));` function to wait for the response from the server. The `StartCoroutine()` function basically takes `IEnumerator`, which is the `WaitingForResponse(new WWW(_phpUrl, _wwwForm), null))` function here. This function basically creates the `WWW` object that sends the `WWWForm` object to the specific `_phpUrl` variable. It also takes the function to callback when it is finished.

Note

For more information on `WWWForm` and `WWW`, visit the following links:

- http://docs.unity3d.com/Documentation/ScriptReference/WWWForm.html
- http://docs.unity3d.com/Documentation/ScriptReference/WWW.html

Then, we created the `ParseXMLData()` function, which will call the `XMLParser. Parse()` function to parse the XML string data that returns from the server, and then, we stored it in the `XMLParser` class as the `UserData` array. Then, we set the `UserData` array to the list of `UserData` using `_users.AddRange(XMLParser.users);`. At last, we set `_isLoaded` to `true`, which means that the data has been loaded.

Next, we created the `LoadUserData()` function, which is very similar to the `SaveUserData()` function, except that we only send one parameter to PHP, which is the `action` field to tell PHP that we want `GetScore`. Also, in the `StartCoroutine()` function, we put the callback function in the `WaitingForResponse()` function, which is `ParseXMLData`. This function will be called after the loading is completed.

Then, we go back to the `UIHiScore` script and add the `ServerHiScore` object to show the server hi-score data in the UI.

Classified intel

In this step, we use `AddField("fieldname", "value");` in `WWWForm` to add the value and pass it to the server. In this function, `fieldname` depends on the PHP script that the programmer has set up.

We can open the `HiScore.php` file in **Chapter8 | PHP** that is attached with the project package, and then we can take a look at the following line:

```
$action = $_POST[ 'action' ]; //Get request action from Unity
```

The word `action` is the same keyword that we assigned in the `AddField()` function at the beginning of the `SaveUserData()` function:

```
_wwwForm.AddField("action", "PostScore");
```

This keyword is the value that we will use to communicate between the Unity script and PHP script on the web server side.

Note

In this PHP file, I've set up the database using MySQL. So, if you have your own database set up on MySQL, you can adjust this PHP to point to your database and put the file to your own web hosting.

To create your custom high score database, you can go to the following link to get free MySQL hosting: http://www.freemysqlhosting.net.

After you finished registration and got the database name ready, you will see something like the following screenshot:

Now, you can open the `HiScore.php` file, and then you can change the first and second lines of this code to match your database information as follows:

```
$link = mysql_connect("your host","your username","your password") or die(
mysql_error());
mysql_select_db("your database name") or die(mysql_error());
```

The following list explains the variables being used in the preceding code:

- **your host:** This is the information in the **Database Host** in the preceding screenshot
- **your username**: This is the information in the **Database Username** in the preceding screenshot

- **your password**: This is the **Database Password**, which will be in your e-mail
- **your database name**: This is the information in the **Database Name** in the preceding screenshot

Then, you can log in to your database (www.phpmyadmin.co) and click on your database's name at the right-hand side of the web page. Then, you can create the table and name it `scores`. Next, you need to put each value as shown in the following screenshot:

At last, you can put the `HiScore.php` file on your web hosting service. Then, you will need to change `phpUrl` in the `UIHiscore` script to your web hosting.

Note

For more information on how to set up MySQL database on your website, you can go to the following link and download the file: http://www.webwisesage.com/addons/free_ebook.html.

There is also a video tutorial of how to set up the MySQL database, PHP, and Flash by Lee Brimelow; just check out the MySQL and PHP parts. You can find it at the following link: http://www.gotoandlearn.com/play.php?id=20.

Mission accomplished

In this project, we have created the scripts that help us save, load, and post the high score locally and to the server database. We also used the serialization technique to save the list of users to the bytes array, convert it to the string, and then use `PlayerPrefs` to save it to the local machine. Then, we created the script to parse the XML string format to the value.

Finally, I would like to thank all of you for reading this book. I hope that you got some useful information or some new techniques from this book. Thank you very much!

Hotshot challenges

We have learned many things from this project, such as saving and loading the value locally using `PlayerPrefs` and `Serialization`, using `WWWForm` to post and load the high score from the server, encrypting the key code with MD5, and loading the XML string using `XmlDocument`. However, these aren't the only things that we can do. Let's try something out and see how much we learned from this project:

- Create the save game position for our game using the `PlayerPrefs` and `Serialization` techniques to save the current position of our character in the game and load it as well
- Save the game to the file using the `PlayerPrefs` and `Serialization` techniques
- Try to adapt the `XMLParser` script to load the XML file using `xml.Load(filename.xml)` to load the XML file to your game
- Create your database and PHP on your website using `HiScore.php` and changing `phpUrl` on your website; you can also change the hash key to the one that you prefer
- Make the game prompt the user to enter their name only if they actually qualify for the new high score
- Post to the server the high score data using the serialization technique and the MD5 encryption

Appendix A. Important Functions

The purpose of this appendix is to explain the meaning of some important methods used in Unity, referenced from the Unity scripting documentation. The following is a basic Unity MonoBehaviour diagram:

Awake()

The `Awake()` function is called when the script instance is being loaded.

`Awake` is used to initialize any variable or a game state before the game starts. It is called only once during the lifetime of the script instance. It is also called after all the objects are initialized, so you can safely speak to other objects or query them using, for example, `GameObject.FindWithTag`. Each `Awake()` function of a GameObject is called in a random order between objects. As a result of this, you should use `Awake()` to set up references between scripts and use `Start()` to pass any information back and forth. `Awake()` is always called before any `Start()` function. This allows you to order the initialization of scripts.

Tip

For C# and Boo users, use `Awake()` instead of the constructor for initialization, as the serialized state of the component is undefined at construction time. `Awake()` is called once, just like the constructor.

`Awake()` cannot be a coroutine (we will cover more about coroutines in Appendix B, *Coroutines and Yield*).

An example of `Awake()` is as follows:

```
// JavaScript user:

private var myTarget : GameObject;
function Awake() {
    myTarget = GameObject.FindWithTag("Target");
}


// C# user:

GameObject myTarget;
void Awake() {
    myTarget = GameObject.FindWithTag("Target");
}
```

Start()

The `Start()` function is called just before any of the `Update()` methods is called.

`Start()` is only called once in the lifetime of the behavior. The difference between `Awake()` and `Start()` is that `Start()` is only called if the script instance is enabled. This allows you to delay any initialization of code until it is really needed.

The `Start()` function is called after all `Awake()` functions on all script instances have been called. An example of the `Start()` function is as follows:

```
// JavaScript user:

private var myLife : int;
function Start() {
    myLife = 5;
}


// C# user:

int myLife;
void Start() {
    myLife = 5;
}
```

Update()

The `Update()` function is called at every frame, if `MonoBehaviour` is enabled.

Note

`MonoBehaviour` is the base class for every script derived from Unity. We can use `SetActive(true)` to enable or `SetActive(false)` to disable the script. For more details, visit the following link:

https://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html

`Update()` is the most commonly used function to implement any kind of game behavior. An example of the `Update()` function is as follows:

```
// JavaScript user:

// Moves the object forward 1 meter per second
function Update () {
  transform.Translate(0, 0, Time.deltaTime*1);
}


//C# user:

// Moves the object forward 1 meter per second
void Update () {
  transform.Translate(0f, 0f, Time.deltaTime*1f);
}
```

FixedUpdate()

The `FixedUpdate()` function is called every fixed framerate frame (or every constant time), if the `MonoBehaviour` class is enabled.

Note

The difference between `FixedUpdate()` and `Update()` is that the `Update()` function is called every frame. On the other hand, `FixedUpdate()` is called every fixed framerate frame. This depends on the **Fixed Timestep** value that we have set up. This will result in stable physics calculations for all machines. For more information, visit the following link:

https://docs.unity3d.com/Documentation/Components/class-TimeManager.html

`FixedUpdate()` should be used instead of `Update()` when dealing with physics calculations. For example, when adding a force to a rigidbody, you have to apply the force to every fixed frame inside `FixedUpdate()` instead of every frame inside `Update()`, because the physics simulation is carried out in discrete timesteps. The `FixedUpdate()` function is called immediately before each step. An example of `FixedUpdate()` is as follows:

```
// JavaScript user:

// Apply an upward force to the rigidbody every frame
function FixedUpdate () {
  rigidBody.AddForce(Vector3.up);
}


// C# user:

// Apply an upward force to the rigidbody every frame
void FixedUpdate () {
  rigidBody.AddForce(Vector3.up);
}
```

LateUpdate()

The `LateUpdate()` function is called every frame after all the `Update()` functions have been called, if the `MonoBehaviour` class is enabled.

`LateUpdate()` is called after all `Update()` functions have been called. This is useful to order script execution. For example, `follow camera` should always be implemented in `LateUpdate()`, because this tracks objects that may have moved inside `Update()`.

An example of `LateUpdate()` is as follows:

```
// JavaScript user:

// Moves the object forward 1 meter per second
function LateUpdate () {
  transform.Translate(0, 0, Time.deltaTime*1);
}


// C# user:

// Moves the object forward 1 meter per second
void LateUpdate () {
  transform.Translate(0f, 0f, Time.deltaTime*1f);
}
```

OnEnable()

The `OnEnable()` function is called when an object is enabled and active.

This means that the `OnEnable()` function will be called every time the object is enabled and active. This is different from the `Start()` function, which is only called once for the object's lifetime.

An example of `OnEnable()` is as follows:

```
// JavaScript user:

// Reset object position every time the object is active
function OnEnable () {
  transform.position = Vector3.zero;
}


// C# user:

// Reset object position every time the object is active
void OnEnable () {
  transform.position = Vector3.zero;
}
```

OnDisable()

The `OnDisable()` function is called when an object/behavior is disabled or inactive.

This is also called when an object is destroyed and can be used for any clean-up code. When scripts are reloaded after compilation has finished, `OnDisable()` will be called, followed by an `OnEnable()` function after the script has been loaded.

An example of `OnDisable()` is as follows:

```
// JavaScript user:

function OnDisable () {
  Debug.Log("script becomes inactive");
}


// C# user:

void OnDisable () {
  Debug.Log("script becomes inactive");
}
```

OnGUI()

The `OnGUI()` function is called to render and handle GUI events, such as `GUI.Button`, `GUI.Label`, and `GUI.Box`.

This means that your `OnGUI()` implementation might be called several times per frame (one call per event). If the `MonoBehaviour` object is enabled and is set to `false`, `OnGUI()` will not be called.

An example of `OnGUI()` is as follows:

```
// JavaScript user:

//Draw the Button (width=150,height=50) at the position x = 10, y = 10.
function OnGUI () {
  if (GUI.Button(Rect(10, 10, 150, 50), "My Button")) {
      Debug.Log("Hello World");
  }
}


// C# user:

//Draw the Button (width=150,height=50) at the position x = 10, y = 10.
void OnGUI () {
  if (GUI.Button(new Rect(10, 10, 150, 50), "My Button")) {
      Debug.Log("Hello World");
  }
}
```

OnDrawGizmos()

Implement the `OnDrawGizmos()` function if you want to draw gizmos that are also pickable and always drawn. This allows you to quickly pick important objects in your scene. You can also use `OnDrawGizmos()` to draw a line or different type of gizmos, such as `Gizmos.DrawRay`, `Gizmos.DrawLine`, and `Gizmos.DrawWireSphere`, which will make it easier for you to debug the code.

Tip

`OnDrawGizmos()` will use a mouse position that is relative to the **Scene** view. This function only works for debugging in the editor.

An example of `OnDrawGizmos()` is as follows:

```
// JavaScript user:

var target : Transform;
// Draw the blue line from this object to the target
function OnDrawGizmos () {
    if (target != null) {
      Gizmos.color = Color.Blue;
       Gizmos.DrawLine(transform.position, target.position);
    }
}


// C# user:

Transform target;
// Draw the blue line from this object to the target
void OnDrawGizmos () {
    if (target != null) {
       Gizmos.color = Color.Blue;
       Gizmos.DrawLine(transform.position, target.position);
  }
}
```

References

For more details and references, check out the following Unity scripting document links:

- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Awake.html
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Start.html
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.Update.htm
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.FixedUpdat
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.LateUpdate
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnGUI.htm
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnDrawGiz
- http://docs.unity3d.com/Documentation/ScriptReference/Gizmos.DrawLine.html
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnEnable.h
- https://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.OnDisable

Appendix B. Coroutines and Yield

This appendix presents a brief review of `coroutines` and `yield`, from the Unity scripting references.

Coroutines

`StartCoroutine` returns a `Coroutine`. Instances of this class are only used to reference these `coroutines` and do not hold any exposed properties or functions.

A `coroutine` is a function that can suspend its `yield` execution until the given `YieldInstruction` class finishes calling.

An example of `coroutine` is as follows:

```
// JavaScript user:

function Start() {
  // Starting = 0.0
  Debug.Log ("Starting = " + Time.time);
  // Start function WaitAndPrint as a Coroutine
  yield WaitAndPrint();
  // Done WaitAndPrint = 5.0
  Debug.Log ("Done WaitAndPrint = " + Time.time);
}

function WaitAndPrint() {
  //Suspend execution for 5 seconds
  yield WaitForSeconds(5);
  // WaitAndPrint = 5.0
  Debug.Log ("WaitAndPrint = " + Time.time);
}


// C# user:

IEnumerator Start() {
  // Starting = 0.0
  Debug.Log ("Starting = " + Time.time);
  // Start function WaitAndPrint as a Coroutine
  yield return WaitAndPrint();
  // Done WaitAndPrint = 5.0
  Debug.Log ("Done WaitAndPrint = " + Time.time);
}

IEnumerator WaitAndPrint() {
  //Suspend execution for 5 seconds
  yield return new WaitForSeconds(5f);
  // WaitAndPrint = 5.0
  Debug.Log ("WaitAndPrint = " + Time.time);
}
```

YieldInstruction

When writing a game code, one often ends up needing to script a sequence of events. This could result in code like the following:

```
// JavaScript user:

private var state : int = 0;
function Update() {
  if (state == 0) {
       // do step 0
       Debug.Log("Do step 0");
       state = 1;
       return;
  }
  if (state == 1) {
       // do step 1
       Debug.Log("Do step 1");
       state = 0;
       return;
  }
}


// C# user:

int state = 0;
void Update() {
  if (state == 0) {
       // do step 0
       Debug.Log("Do step 0");
       state = 1;
       return;
  }
  if (state == 1) {
       // do step 1
       Debug.Log("Do step 1");
       state = 0;
       return;
  }
}
```

The preceding code basically executes `step 0` and `step 1` and then goes back to `step 0` (as a loop), but if there are more events, execution will happen after `step 1` and so on. Too many `if` statements can make the code look ugly in the long run. In this case, it's more convenient to use the `yield` statement. The `yield` statement is a special kind of `return` that ensures that the function will continue from the line after the `yield` statement is called the next time. The result could be something like the following code:

```
// JavaScript user:

function Start() {
    while (true) { //Use this line instead of Update()
      //do step 0
```

```
      Debug.Log("Do step 0");
      yield;  //wait for one frame
      //do step 1
      Debug.Log("Do step 1");
      yield;  //wait for one frame
    }
}


// C# user

IEnumerator Start() {
    while (true) { //Use this line instead of Update()
      //do step 0
      Debug.Log("Do step 0");
      yield return null;  //wait for one frame
      //do step 1
      Debug.Log("Do step 1");
      yield return null;  //wait for one frame
    }
}
```

The preceding code will have a similar result without having a new variable and an extra `if` statement to check for each step event.

You can also pass special values to the `yield` statement to delay the execution of the `Update()` function until a certain event has occurred, such as `WaitForSeconds`, `WaitForFixedUpdate`, `Coroutine`, and `StartCoroutine`.

Tip

You cannot use `yield` from within `Update()` or `FixedUpdate()`, but you can use `StartCoroutine` to start a function that can use `yield`.

WaitForSeconds

`WaitForSeconds` suspends the `coroutine` execution for the given amount of seconds.

`WaitForSeconds` can only be used with a `yield` statement in `coroutines`. An example of `WaitForSeconds` is as follows:

```
// JavaScript user:

function Start() {
  // Prints 0
  Debug.Log (Time.time);
  // Waits 5 seconds
  yield WaitForSeconds (5);
  // Prints 5.0
  Debug.Log (Time.time);
}


// C# user:

IEnumerator Start() {
  // Prints 0
  Debug.Log (Time.time);
  // Waits 5 seconds
  yield return new WaitForSeconds (5f);
  // Prints 5.0
  Debug.Log (Time.time);
}
```

You can both stack and chain `coroutines`.

The following example will execute `Do()`. We will see that it will call the first line in `Do()`. Then, it will call the last line in `Start()` immediately, while waiting for `5` seconds to call the last script in `Do()`.

```
// JavaScript user:

function Start() {
  // StartCoroutine(Do()) (In JavaScript, you can also use DO() whichwill
get the same result.
  StartCoroutine(Do());
  Debug.Log ("This is printed immediately");
}

function Do() {
    Debug.Log ("Do now");
    yield WaitForSeconds (5); //Wait for 5 seconds
    Debug.Log ("Do 5 seconds later");
}


// C# user:

void Start() {
```

```
  StartCoroutine(Do());
  Debug.Log ("This is printed immediately");
}

IEnumerator Do() {
    Debug.Log ("Do now");
    yield return new WaitForSeconds (5f); //Wait for 5 seconds
    Debug.Log ("Do 5 seconds later");
}
```

The following example will execute `Do()` and wait until `Do()` has finished waiting for `5` seconds before continuing its own execution:

```
// JavaScript user:

//Chain Coroutine
function Start() {
  yield StartCoroutine(Do());
  Debug.Log ("This is printed after 5 seconds");
  Debug.Log ("This is after the Do coroutine has finished execution");
}

function Do() {
    Debug.Log ("Do now");
    yield WaitForSeconds (5); //Wait for 5 seconds
    Debug.Log ("Do 5 seconds later");
}


// C# user:

//Chain Coroutine
IEnumerator Start() {
  yield return StartCoroutine(Do());
  Debug.Log ("This is printed after 5 seconds");
  Debug.Log ("This is after the Do coroutine has finished execution");
}

IEnumerator Do() {
    Debug.Log ("Do now");
    yield return new WaitForSeconds (5f); //Wait for 5 seconds
    Debug.Log ("Do 5 seconds later");
}
```

WaitForFixedUpdate

`WaitForFixedUpdate` waits until the next frame rate of the `FixedUpdate()` function. (For more details on `FixedUpdate()`, refer to Appendix A, *Important Functions*.)

`WaitForFixedUpdate` can only be used with a `yield` statement in `coroutines`.

An example of `WaitForFixedUpdate` is as follows:

```
// JavaScript user:

function Start() {
  // Wait for FixedUpdate to finished
  yield new WaitForFixedUpdate();
  // Call After FixedUpdate
  Debug.Log ("Call after FixedUpdate");
}

function FixedUpdate() {
    Debug.Log ("FixedUpdate");
}


// C# user

IEnumerator Start() {
  // Wait for FixedUpdate to finished
  yield return new WaitForFixedUpdate();
  // Call After FixedUpdate
  Debug.Log ("Call after FixedUpdate");
}

function FixedUpdate() {
    Debug.Log ("FixedUpdate");
}
```

StartCoroutine

`StartCoroutine` starts a `coroutine`.

The execution of `coroutine` can be paused at any point using the `yield` statement. The `yield return` value specifies when `coroutine` is resumed. `Coroutines` are excellent when modeling behavior over several frames. `Coroutines` have virtually no performance overhead. The `StartCoroutine()` function always returns a value immediately; therefore, you can yield the result. This will wait until `coroutine` has finished execution.

Tip

When using JavaScript, it is not necessary to use `StartCoroutine`; the compiler will do this for you. However, when writing C# code, you must call `StartCoroutine`. (For more details, refer to Appendix C, *Major Differences Between C# and Unity JavaScript*.)

In the following example, we will show how to invoke a `coroutine` and continue executing the function in parallel:

```
// JavaScript user:

function Start() {
  // Starting = 0.0
  Debug.Log ("Starting = " + Time.time);
  // StartCoroutine WaitAndPrint (In JavaScript, you can also
useWaitAndPrint(5.0) which will get the same result.
  StartCoroutine(WaitAndPrint(5.0));
  // Before WaitAndPrint = 5.0
  Debug.Log ("Before WaitAndPrint = " + Time.time);
}

function WaitAndPrint(waitTime : float) {
    //Suspend execution for 5 seconds
    yield WaitForSeconds(waitTime);
    // WaitAndPrint = 5.0
    Debug.Log ("WaitAndPrint = " + Time.time);
}


// C# user:

void Start() {
  // Starting = 0.0
  Debug.Log ("Starting = " + Time.time);
  StartCoroutine(WaitAndPrint(5.0f));
  // Before WaitAndPrint = 5.0
  Debug.Log ("Before WaitAndPrint = " + Time.time);
}

IEnumerator WaitAndPrint(float  waitTime) {
    //Suspend execution for 5 seconds
    yield return new WaitForSeconds(waitTime);
    // WaitAndPrint = 5.0
    Debug.Log ("WaitAndPrint = " + Time.time);
```

```
}
```

The following example will wait until the `WaitAndPrint()` function has finished its execution and then continue executing the rest of the code in the `Start()` function:

```
// JavaScript user:

function Start() {
  // Starting = 0.0
  Debug.Log ("Starting = " + Time.time);
  // StartCoroutine WaitAndPrint (In JavaScript, you can also useyield
WaitAndPrint(5.0) which will get the same result.
  yield StartCoroutine(WaitAndPrint(5.0));
  // Done WaitAndPrint = 5.0
  Debug.Log ("Done WaitAndPrint = " + Time.time);
}

function WaitAndPrint(waitTime : float) {
    //Suspend execution for 5 seconds
    yield WaitForSeconds(waitTime);
    // WaitAndPrint = 5.0
    Debug.Log ("WaitAndPrint = " + Time.time);
}


// C# user:

IEnumerator Start() {
  // Starting = 0.0
  Debug.Log ("Starting = " + Time.time);
  yield return StartCoroutine(WaitAndPrint(5.0f));
  // Done WaitAndPrint = 5.0
  Debug.Log ("Done WaitAndPrint = " + Time.time);
}

IEnumerator WaitAndPrint(float  waitTime) {
    //Suspend execution for 5 seconds
    yield return new WaitForSeconds(waitTime);
    // WaitAndPrint = 5.0
    Debug.Log ("WaitAndPrint = " + Time.time);
}
```

Using StartCoroutine with method name (string)

In most cases, you want to use the `StartCoroutine` variation at the start of a code. However, `StartCoroutine` using a string method name allows you to use `StopCoroutine` with a specific method name.

Tip

The downside is that the string version has a higher runtime overhead to start `coroutine`, and you can pass only one parameter.

In the following example, we will see how to invoke `coroutine` using a string name and stop it:

```
// JavaScript user:

function Start() {
  // Start Coroutine DoSomething
  StartCoroutine("DoSomething", 5.0);
  //  Wait for 2 seconds
  yield WaitForSeconds(2.0);
  // Stop Coroutine DoSomething
  StopCoroutine("DoSomething");
}

function DoSomething (someParameter : float) {
  while (true) {
        // DoSomething Loop
        Debug.Log ("DoSomething Loop = " + Time.time);
     // Yield execution of this coroutine and return to the main loop until
next frame
           yield;
  }
}


// C# user:

IEnumerator Start() {
  // Start Coroutine DoSomething
  StartCoroutine("DoSomething", 5.0f);
  //  Wait for 2 seconds
  yield return new WaitForSeconds(2.0f);
  // Stop Coroutine DoSomething
  StopCoroutine("DoSomething");
}

IEnumerator DoSomething (float  someParameter) {
  while (true) {
      // DoSomething Loop
      Debug.Log ("DoSomething Loop = " + Time.time);
  // Yield execution of this coroutine and return to the main
  loop until next frame
      yield return null;
  }
```

}

StopCoroutine

`StopCoroutine` stops all `coroutines` for the specific method name running on this behavior.

Tip

Note that only `StartCoroutine` using a string method name can be stopped using `StopCoroutine`.

An example of `StopCoroutine` is as follows:

```
// JavaScript user:

function Start() {
  // Start Coroutine DoSomething
  StartCoroutine("DoSomething", 5.0);
  //  Wait for 2 seconds
  yield WaitForSeconds(2.0);
  // Stop Coroutine DoSomething
  StopCoroutine("DoSomething");
}

function DoSomething (someParameter : float) {
  while (true) {
      // DoSomething Loop
      Debug.Log ("DoSomething Loop = " + Time.time);
  // Yield execution of this coroutine and return to the main loop until
next frame
      yield;
  }
}


// C# user:

IEnumerator Start() {
  // Start Coroutine DoSomething
  StartCoroutine("DoSomething", 5.0f);
  //  Wait for 2 seconds
  yield return new WaitForSeconds(2.0f);
  // Stop Coroutine DoSomething
  StopCoroutine("DoSomething");
}

IEnumerator DoSomething (float someParameter) {
  while (true) {
        // DoSomething Loop
           Debug.Log ("DoSomething Loop = " + Time.time);
    // Yield execution of this coroutine and return to the main loop until
next frame
          yield return null;
  }
}
```

StopAllCoroutines

`StopAllCoroutines` stops all coroutines running on this behavior.

An example of `StopAllCoroutines` is as follows:

```
// JavaScript user:

function Start() {
  // Start Coroutine DoSomething
  StartCoroutine("DoSomething", 5.0);
  //  Wait for 1 seconds
  yield WaitForSeconds(1.0);
  // Stop All Coroutine
  StopAllCoroutines();
}

function DoSomething (someParameter : float) {
  while (true) {
    // DoSomething Loop
          Debug.Log ("DoSomething Loop = " + Time.time);
    // Yield execution of this coroutine and return to the main loop until
next frame
          yield;
  }
}


// C# user:

IEnumerator Start() {
  // Start Coroutine DoSomething
  StartCoroutine("DoSomething", 5.0f);
  //  Wait for 1 seconds
  yield return new WaitForSeconds(1.0f);
  // Stop All Coroutine
  StopAllCoroutines();
}

IEnumerator DoSomething (float  someParameter) {
  while (true) {
    // DoSomething Loop
          Debug.Log ("DoSomething Loop = " + Time.time);
    // Yield execution of this coroutine and return to the main loop until
next frame
          yield return null;
  }
}
```

References

For more details and references, please check out the following links from Unity scripting documents:

- http://docs.unity3d.com/Documentation/ScriptReference/YieldInstruction.html
- http://docs.unity3d.com/Documentation/ScriptReference/WaitForSeconds.html
- http://docs.unity3d.com/Documentation/ScriptReference/WaitForFixedUpdate.html
- http://docs.unity3d.com/Documentation/ScriptReference/Coroutine.html
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.StartCorout
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.StopCorout
- http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.StopAllCor

Appendix C. Major Differences Between C# and Unity JavaScript

This appendix will provide a brief reference of the syntactical differences between C# and JavaScript in Unity. This section references the Unity answer forum:

http://answers.unity3d.com/questions/12911/what-are-the-syntax-differences-in-c-and-javascript.html

Unity script directives

Unity has a number of script directives, such as `RequireComponent`; we can find them at http://docs.unity3d.com/412/Documentation/ScriptReference/20_class_hierarchy.Attributes and http://docs.unity3d.com/Manual/Attributes.html. The following code illustrates the use of `RequireComponent`:

```
// JavaScript user:

@script RequireComponent(RigidBody)


// C# user:

[RequireComponent(typeof(Rigidbody))]
```

Type names

A couple of the basic types are spelled differently in pure Unity C#. In JavaScript, we use `Boolean` and `String`, but in pure Unity C#, we use `bool` and `string`, as shown in the following code:

```
// JavaScript user:

var isHit : Boolean;  //The space between ':' isn't necessary.
var myName : String;


// C# user:

bool isHit;
string myName;
```

However, there is an exception. If you include `System` in your C# script, you will be able to use .NET's `String` and `Boolean` classes (uppercase) as shown in the following script:

```
// C# user:

using System;

Boolean isHit;
String myName;
```

Variable declaration

Variable declaration is different and includes access and type specification, which are explained as follows:

- **A JavaScript user**: For this, a type specification is not necessary:

  ```
  var playerLife = 1;       //**public** access is default
  private var playerLife = 2;    //a private var
  var myObj : GameObject;  //a type is specified (no   value assigned)
  ```

- **A C# user**: For this, a type is always stated when declaring a variable:

  ```
  public var playerLife = 1;     //a public var
  int playerLife = 2;      //**private** access is default
  public GameObject myObj;  a type is specified (no value assigned)
  ```

Variables with dynamically typed resolution

Only in JavaScript, variables can have an unspecified type. This only occurs if you don't assign a value or specify a type while declaring the variable:

```
// JavaScript user: The type specification is not necessary.
var playerLife : int;       //statically typed (because
                                type specified)
var playerLife = 2;      //statically typed (because
                                type is inferred from value
                                assigned)
var playerLife;        //dynamically typed (because
                                   neither a type or value is
                                   specified)
```

The dynamically typed variables will cause slower performance, and you can run into casting problems. You can use `#pragma strict` by including this at the top of a script to tell Unity to disable the dynamic typing in the script and report compile errors when there is a dynamic type in the script.

On the other hand, `var` in C# means that the type of variable being declared on the right-hand side of the code will be inferred by the compiler, which will be similar to a static type declaration. However, `var` cannot be used for the dynamic type:

```
// C# user:

var playerLife = 2;      //This statement is equivalent
int playerLife = 2;      //to this statement
```

Multidimensional array declaration

The multidimensional array for a C# and Unity JavaScript user is very similar to the following script:

```
// JavaScript user:

var myArray = new int[8,8];    //8x8 2d int array


// C# user:

int[,] myArray = new int[8,8];     //8x8 2d int array
```

Character literals not supported

Unity's JavaScript seems to be missing the syntax to declare character literals. This means you need to get them implicitly by referencing a character index from a string as follows:

```
// JavaScript user:

var myChar = "a"[0];  //implicitly retrieves the first  character of the
string "a"


// C# user:

char myChar = 'a';          //character 'a'
```

Class declarations

You can define classes in JavaScript in a way that is similar to C#. The following example is a class named `MyClass` that inherits from the `MonoBehaviour` class:

```
// JavaScript user:

class MyClass extends MonoBehaviour {
  var myVar = 1;
  function Start() {
    Debug.Log("Hello World!");
  }
}


// C# user:

class MyClass : MonoBehaviour {
  public int myVar = 1;
  void Start() {
    Debug.Log("Hello World!");
  }
}
```

However, in JavaScript, if you're inheriting from `MonoBehaviour`, you don't need to write a class body at all. You can also write following the script in JavaScript, which will fetch a similar result as the preceding JavaScript example:

```
var myVar = 1;
function Start() {
  Debug.Log("Hello World!");
}
```

Unity will automatically implement an explicit class body for you. You can also write classes that do not inherit from anything; however, you cannot place these scripts in the game objects, but you have to instantiate them with the `new` keyword as follows:

```
// JavaScript user:

class MyClass {
  var myVar = 1;
  function MyClass() {
    Debug.Log("Hello World!");
  }
}


// C# user:

class MyClass {
  public int myVar = 1;
  void MyClass() {
    Debug.Log("Hello World!");
  }
```

```
}
```

Note

If you are inheriting from `MonoBehaviour`, you should not use constructors or destructors. Instead, use the event handler functions such as `Start()`, `Awake()`, and `OnEnabled()`.

Limited interface support

While Unity's JavaScript does support inheritance and interfaces, it has a very limiting caveat: you can either inherit your class from an existing class or declare one interface:

```
// JavaScript user: (Only on allowed)

class MyClass extends MyObject implements IMyObject {…}


// C# user:

class MyClass : MonoBehaviour, IMyObject, IMyItem {…}
```

Generics

The C# syntax supports generics that allow you to use classes and methods, which do not specifically declare a type. Instead, the type is passed as a parameter when calling the method or instantiating the class at runtime.

.Net comes with some useful generic classes such as `List` and `Dictionary`, and Unity's own API has some generic functions. They remove the need for some of the verbose casting, which would otherwise be necessary in C#, as follows:

```
// JavaScript user:

//Automatically cast the correct type
var someScript : MyScript = GetComponent(MyScript);

//or using the Generic version in JavaScript
var someScript : MyScript = GetComponent.<MyScript>();


// C# user:

//with out Generic
MyScript someScript = (MyScript)GetComponent(typeof(MyScript));
//or using the Generic version in C#
MyScript someScript = GetComponent<MyScript>();
```

The foreach keyword

C# iterators use `foreach` instead of `for`. Also notice the variable declaration within the `for/foreach` statement in the following code. C# requires the type of the item contained in the list to be explicitly declared:

```
// JavaScript user:

for (var item in itemList) {
  item.DoSomething();
}


// C# user:

foreach (ItemType item in itemList) {
  item.DoSomething();
}
```

The new keyword

In JavaScript, you can create a new instance of an object or struct without using the `new` keyword. In C#, using `new` is mandatory:

```
// JavaScript user:

var myPosition = Vector3(0,0,0);
var myInstance = MyClass();
//We can also use new keyword in JavaScript
var myInstance = new MyClass();


// C# user:

Vector3 myPosition = new Vector3(0,0,0);
MyClass myInstance = new MyClass();
```

The yield instruction and coroutine

There are differences syntaxes between C# and JavaScript; they are as follows:

```
// JavaScript user:

yield WaitForSeconds(3);        //pauses for 3 seconds
yield WaitForMyFunction();        //start coroutine

function WaitForMyFunction() {…}      //coroutine function


// C# user:

yield return new WaitForSeconds(3);    //pauses for 3 seconds
yield return WaitForMyFunction();    //start coroutine

IEnumerator WaitForMyFunction() {…}    //coroutine function
```

Tip

JavaScript will automatically generate the return type to `IEnumerator` if you put the `yield` instruction inside the function. On the other hand, in C#, you will need to specify the return type to `IEnumerator`.

However, if we want to wait for the user input in C#, which may be over several frames, you will have to use `StartCoroutine`. In JavaScript, the compilers will automatically do it for you:

```
// JavaScript user:

yield WaitForMyFunction(5);
//This is similar to
yield StartCoroutine(WaitForMyFunction(5));

function WaitForMyFunction(waitTime : float) {…}
//coroutine function


// C# user:

//Need to put StartCoroutine
yield return StartCoroutine(WaitForMyFunction(5));

IEnumerator WaitForMyFunction(waitTime : float) {…}
//coroutine function
```

Casting

JavaScript automatically casts from one type to another, where possible. For example, the `Instantiate` command returns a type of `Object`, as shown in the following code:

```
// JavaScript user:

//There's no need to cast the result of "Instantiate" provided the
variable's type is declared.
var newObject : GameObject = Instantiate(sourceObject);


// C# user:

// in C#, both the variable and the result of instantiate must be declared.
// C# first version
GameObject foo = (GameObject) Instantiate(sourceObject);
// C# second version
GameObject foo = Instantiate(sourceObject) as GameObject;
```

Tip

There are two different ways of casting in C#. For the first line in the preceding code, if the object can’t be instantiated, it will throw an exception. You will need to use a `try/catch` statement to handle this exception properly. The second line, if it fails, will set `foo` to `null` and not throw an exception. Then, you would just need to test if the returned object is `null` or not.

Properties with getters/setters

It is possible to define special functions that can be accessed as if they are variables. For instance, we could say `foo.someVar = "testing";`, and under the hood, there are `get` and `set` functions that process the `testing` argument and store it internally. However, they can also do any other processing on it, for instance, capitalizing the first letter before storing it. So, you’re not just doing a variable assignment, but you’re calling a function that sets the variable, and it can do whatever other functions can do:

```
// JavaScript user:

private var foo = 8;  //"backing store"
function get Foo () : int {
  return foo;
}
function set Foo (value) {
  foo = value;
}


// C# user:

public class MyClass {
private int foo = 8;  //"backing store"
public int Foo {
    get {
       return foo;
    }
    set {
      foo = value;
    }
  }
}
```

Changing struct properties by value versus by reference

Structures are passed by values in C#, so you cannot change the x, y, or z value of a Vector3. You need to create a new Vector3 and assign it to the Vector3 that you want. However, in JavaScript, you can write it as follows:

```
// JavaScript user:

transform.position.x = 1;


// C# user:

transform.position = new Vector3(1, transform.position.y,
transform.position.z);
```

Function/method definitions

First of all, terminology in JavaScript uses the term `Function`, while C# calls these `Methods`. They mean the same thing, and most C# coders understand the term `Function`.

JavaScript functions are declared with the `function` keyword before the function name. C# method declarations just use the return type and the method name. The return type is often `void` for common Unity events. JavaScript functions are `public` by default, and you can specify them as `private`, if required. C# methods are `private` by default, and you can specify that they should be `public`, if required.

In JavaScript, you can omit the parameter types and the return type from the declaration, but it's also possible to explicitly specify these (which is sometimes necessary if you run into type ambiguity or problems):

```
// JavaScript user:

// a common Unity MonoBehaviour event handler:
function Start () { ...function body here… }

// a private function:
private function TakeDamage (amount) {
  energy -= amount;
}

// a public function with a return type.
// the parameter type is "Transform", and the return type is "int"

function GetHitPoint (hp : int) : int {
   return (maxHp – hp);
}


// C# user:

// a common Unity monoBehaviour event handler:
void Start() { ...function body here… }

// a private function:
void TakeDamage(int amount) {
  energy -= amount;
}

// a public function with a return type.
// the parameter type is "Transform", and the return type is "int"

public int GetHitPoint (int hp) {
  return (maxHp – hp);
}
```

References

For more details and references, please check out the following links:

- http://answers.unity3d.com/questions/12911/what-are-the-syntax-differences-in-c-and-javascript.html
- http://www.unifycommunity.com/wiki/index.php?title=Csharp_Differences_from_JS
- http://docs.unity3d.com/Documentation/Manual/CreatingAndUsingScripts.html

Appendix D. Shaders and Cg/HLSL Programming

This appendix presents a brief overview of the structure of surface shaders and Cg/HLSL programming.

Shaders in Unity can be written in one of the following three different ways:

- **Surface shaders**: These will probably be the best option if your shader needs to be affected by the lights and shadows. These shaders also make it easy to write complex shaders in a compact way—it's a higher level of abstraction for interaction with Unity's lighting pipeline. Most surface shaders automatically support forward and deferred lighting (the exception is some very custom lighting models), which allows your shader to efficiently interact with many real-time lights. You write surface shaders in a couple of lines of Cg/HLSL, and a lot more code gets autogenerated from that.

  Tip

  Do not use the surface shaders if the shaders have anything to do with lights, such as image effects or special-effects shaders (glowing effect and so on), because surface shaders will do the lighting calculations for no reason.

- **Vertex and fragment shaders**: These are your best option to write the image effect or special-effect shaders. They will be required if your shader doesn't need to interact with lighting or you need some very exotic effects that the surface shaders can't handle. Shader programs written in this way are the most flexible way to create the effect you need (even surface shaders are automatically converted to a bunch of vertex and fragment shaders), but that comes with more work: you have to write more code, and it's difficult to make it interact with lighting. These shaders are written in Cg/HLSL as well.
- **Fixed function shaders**: These need to be written for old hardware that doesn't support programmable shaders. You will probably want to write fixed function shaders as an *n*th fallback to your fancy fragment or surface shaders to make sure that your game still renders something sensible when it is run on an old hardware or simpler mobile platforms. Fixed function shaders are entirely written in a language called **ShaderLab**, which is similar to Microsoft's FX files or NVIDIA's CgFX.

  Tip

  This is also a good option if your shader doesn't need a fancy effect such as a 2D game on mobile. This could save your time so that you can write a shader that supports old and new hardware.

Regardless of which type you choose, the actual code of the shader code will always be wrapped in ShaderLab, which is used to organize the shader structure. It looks like the following code:

```
Shader "MyShader" {
    Properties {
      // All properties go here
       _MyTexture ("My Texture", 2D) = "white" { }
  }
  SubShader {
       //  Choose your written style
       //  - surface shader or
       //  - vertex and fragment shader or
       //  - fixed function shader
  }
  SubShader {
  // Optional - A simpler version of the SubShader above that can run   on
older graphics cards
  }
}
```

However, we will only talk about the surface shaders that we used in Project 3, *Shade Your Hero/Heroine*, of this book.

Note

For more information about other shader types, visit the following URLs:

- **ShaderLab**: http://docs.unity3d.com/Documentation/Components/SL-Shader.html
- **Vertex and fragment shaders**: http://docs.unity3d.com/Documentation/Components/SL-ShaderPrograms.html

ShaderLab properties

From the preceding example, in the `Properties` block, we can define the type of properties, as shown in the following table:

Type	Description
`name ("display name", Range (min, max)) = number`	This defines a float property, represented as a slider from `min` to `max` in the **Inspector** view
`name ("display name", Color) = (number,number,number,number)`	This defines a color property
`name ("display name", 2D) = "name" { options }`	This defines a 2D texture property
`name ("display name", Rect) = "name" { options }`	This defines a rectangle (nonpower of 2) texture property
`name ("display name", Cube) = "name" { options }`	This defines a cubemap texture property
`name ("display name", Float) = number`	This defines a float property
`name ("display name", Vector) = (number,number,number,number)`	This defines a four-component vector property

Each property inside the shader is referenced by a name (in Unity, it's common to start shader property names with an underscore). The property will show up in the **Material** inspector as **Display name**. For each property, a default value is given after the equals sign as follows:

- **For range and float properties**: Its default value is just a single number
- **For color and vector properties**: Its default value is four numbers in parentheses
- **For texture (2D, rect, and cube)**: Its default value is either an empty string or one of the built-in default textures such as `white`, `black`, `gray`, or `bump`

An example of the `shaderLab` properties is as follows:

```
Properties {
  _MainTex ("Texture ", 2D) = "white" {} // textures
  // color
  _SpecColor ("Specular color", Color) = (0.30,0.85,0.90,1.0)
  _Gloss ("Shininess", Range (1.0,512)) = 80.0 // sliders
}
```

Surface shaders

To use the surface shaders, you need to define a surface function (`void surf(Input IN, inout SurfaceOutput o)`) that takes any UVs or data you need as an input and fills in the output structure, `SurfaceOutput`. `SurfaceOutput`, which basically describes the properties of the surface (its albedo color, normal, emission, specularity, and so on). Then, you write this code in Cg/HLSL.

The surface shaders' compiler then figures out what inputs are needed, what outputs are filled, and so on and generates actual vertex and pixel shaders as well as rendering passes to handle forward and deferred rendering.

The surface shaders placed inside the `CGPROGRAM…ENDCG` block are to be placed inside the `SubShader` block, and it uses the `#pragma surface…` directive to indicate that it's a surface shader. You will see that the surface shaders are placed inside the `CGPROGRAM` and `ENDCG` blocks in the following example:

```
Shader "My Lambert" {
    Properties {
      _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader {
      Tags { "RenderType"="Opaque" }
      LOD 200 //Optional that allows the script to turned the shaderon or
off when the player's hardware didn't support your shader.
      CGPROGRAM
      #pragma surface surf Lambert
      sampler2D _MainTex;

      struct Input {
        float2 uv_MainTex;
      };

      void surf (Input IN, inout SurfaceOutput o) {
        fixed4 c = tex2D (_MainTex, IN.uv_MainTex);
        o.Albedo = c.rgb;
        o.Alpha = c.a;
      }
      ENDCG
  }
    FallBack "Diffuse"
}
```

#pragma surface

The `#pragma surface` directive is used as follows:

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

The required parameters to use this directive are as follows:

- `surfaceFunction`: This is used to define which Cg function has the surface shader code. The function should have the form of `void surf (Input IN, inout SurfaceOutput o)`, where `Input` is a structure you have defined. `Input` should contain any texture coordinates and extra automatic variables needed by surface function.
- `lightModel`: This is used to define a lighting model to be used. The built-in models are **Lambert** (diffuse) and **BlinnPhong** (specular). You can also write your own lighting model using the following custom lighting models:
  - `half4 LightingName (SurfaceOutput s, half3 lightDir, half atten);`: This is used in forward rendering the path for light models that are not view-direction dependent (for example, diffuse)
  - `half4 LightingName (SurfaceOutput s, half3 lightDir, half3 viewDir, half atten);`: This is used in forward rendering path for light models that are view-direction dependent
  - `half4 LightingName_PrePass (SurfaceOutput s, half4 light);`: This is used in the deferred lighting path.

Tip

Note that you don't need to declare all functions. A lighting model either uses view direction or it does not. Similarly, if the lighting model does not work in deferred lighting, you just do not declare the `_PrePass` function, and all shaders that use it will compile to forward rendering only, such as the shader that we created in Project 3, *Shade Your Hero/Heroine*. We don't need the `_PrePass` function because our shader needs the view direction (`viewDir`) and the light direction (`lightDir`) for our custom lighting function to calculate the ramp effect for the cartoon style shader (toon shader / cel shader), which is only available in forward rendering.

Optional parameters `[optionalparams]` to use `#pragma surface` are listed in the following table:

Type	Description
`alpha`	This is the alpha blending mode. This is used for semitransparent shaders.
`alphatest:VariableName`	This is the alpha testing mode. This is used for transparent-cutout shaders. The cut-off value is in the `float` variable with `VariableName`.
`vertex:VertexFunction`	This is the custom vertex modification function. See tree bark shader, for example.
`exclude_path:prepass`	

or `exclude_path:forward`	This does not generate passes for the given rendering path.
`addshadow`	This adds shadow caster and collector passes. This is commonly used with custom vertex modification so that shadow casting also gets a procedural vertex animation.
`dualforward`	This uses dual lightmaps in the forward path.
`fullforwardshadows`	This supports all shadow types in the forward rendering path.
`decal:add`	This is an additive decal shader (for example, `terrain AddPass`).
`decal:blend`	This is a semitransparent decal shader.
`softvegetation`	This makes the surface shader render only when soft vegetation is on.
`Noambient`	This does not apply any ambient lighting or spherical harmonic lights.
`novertexlights`	This does not apply any spherical harmonics or per-vertex lights in forward rendering.
`nolightmap`	This disables lightmap support in this shader (makes a shader smaller).
`Noforwardadd`	This disables forward rendering of an additive pass. This makes the shader support one full directional light, with all other lights computed per vertex/SH. This makes shaders smaller as well.
`approxview`	This computes normalized view direction per vertex instead of per pixel for shaders that need it. This is faster, but view direction is not entirely correct when the camera gets close to the surface.
`halfasview`	This passes a half-direction vector into the lighting function instead of view direction. Half direction will be computed and normalized per vertex. This is faster, but not entirely correct.

Additionally, you can write `#pragma debug` inside the `CGPROGRAM` block, and then the surface compiler will spit out a lot of comments of the generated code. You can view them using an open compiled shader in the **Shader** inspector.

Surface shaders input structure

The input structure, `Input`, generally has any texture coordinates needed by the shader. texture coordinates and must be named `uv` followed by a texture name (or start it with `uv2` to use the second texture coordinate set).

An example of surface shader input structure is as follows:

```
Properties {
    _MainTex ("Texture", 2D) = "white" {}
}
......
    sampler2D _MainTex;
......
    struct Input {
         float2 uv_MainTex;
    };
```

We can also have additional values that can be put into the `Input` structure, as mentioned in the following table:

Type	Description
`float3 viewDir`	This will contain the view direction to compute Parallax effects, rim lighting, and so on.
`float4 with COLOR semantic`	This will contain an interpolated per-vertex color.
`float4 screenPos`	This will contain the screen space position for reflection effects. This is used by the WetStreet shader in Dark Unity, for example.
`float3 worldPos`	This will contain the world space position.
`float3 worldRefl`	This will contain the world reflection vector if the surface shader does not write to `o.Normal`. See the Reflect-Diffuse shader, for example.
`float3 worldNormal`	This will contain the world normal vector if the surface shader does not write to `o.Normal`.
`float3 worldRefl; INTERNAL_DATA`	This will contain the world reflection vector if the surface shader writes to `o.Normal`. To get the reflection vector based on per-pixel normal map, use `WorldReflectionVector (IN, o.Normal)`. See the Reflect-Bumped shader for example.
`float3 worldNormal; INTERNAL_DATA`	This will contain the world normal vector if the surface shader writes to `o.Normal`. To get the normal vector based on per-pixel normal map, use `WorldNormalVector (IN, o.Normal)`.

The SurfaceOutput structure

A standard output structure of surface shaders is as follows:

```
struct SurfaceOutput {
    fixed3 Albedo;
    fixed3 Normal;
    fixed3 Emission;
    half Specular;
    fixed Gloss;
    fixed Alpha;
};
```

You can also find it in the `Lighting.cginc` file inside Unity (`unity install path}/Data/CGIncludes/Lighting.cginc` in Windows and `/Applications/Unity/Unity.app/Contents/CGIncludes/Lighting.cginc` in Mac).

Cg/HLSL Programming

This section presents a brief idea of how to access the shader properties in the Cg/HLSL programming and what are the data type and common methods used in Cg/HLSL programming.

Accessing shader properties in Cg/HLSL

A shader can be declared in its properties in a `Properties` block. If you want to access some of those properties in Cg/HLSL shader programming, you need to declare a Cg/HLSL variable with the same name and a matching type.

Consider the following example:

```
Properties {
    _MainTex ("Texture", 2D) = "white" {}
}

SubShader {
    ……
    CGPROGRAM
    sampler2D _MainTex;
    …
```

The properties that map to the Cg/HLSL variables are as follows:

- Color and vector properties map to `float4` variables.
- Range and float properties map to `float` variables.
- Texture properties map to `sampler2D` variables for regular (2D) textures. The `CUBE` and `RECT` textures map to `samplerCUBE` and `samplerRECT` variables, respectively.

Data types

Cg/HLSL has six basic data types. Some of them are the same as in C, while others are especially added for GPU programming. These types are listed in the following table:

Data type	Description
`float`	This is a 32-bit floating point number (a high-precision floating point is generally 32 bits, just like the `float` type in regular programming languages)
`half`	This is a 16-bit floating point number (a medium-precision floating point is generally 16 bits, with a range of -60,000 to +60,000 and 3.3 decimal digits of precision)
`int`	This is a 32-bit integer
`fixed`	This is a 12-bit fixed point number (a low-precision fixed point is generally 11 bits, with a range of -2.0 to +2.0 and 1/256th precision)
`bool`	This is a boolean variable (`FALSE` = `0` and `TRUE` = `1`)
`sampler*`	This represents a texture object (`sampler1D`, `sampler2D`, `sampler3D`, `samplerCUBE`, or `samplerRECT`)

Cg/HLSL also features vector and matrix data types that are based on the basic data types, such as `float3` and `float4x4`. Such data types are quite common when dealing with 3D graphics programming. Cg/HLSL also has the `struct` and `array` data types, which work in a way that is similar to their C equivalents.

Common methods to create shaders

Method	Description
`dot(a, b)`	This gives the dot product of two vectors.
`cross(A, B)`	This gives the cross product of vectors `A` and `B`. `A` and `B` must be three-component vectors.
`max(a, b)`	This gives the maximum of `a` and `b`.
`min(a, b)`	This gives the minimum of `a` and `b`.
`floor(x)`	This gets the largest integer not greater than `x`.
`round(x)`	This gets the closest integer to `x`.
`ceil(x)`	This gets the smallest integer not less than `x`.
`pow(x, y)`	This computes `x` raised to the power `y`.
`normalize(v)`	This returns a vector of length `1` that points in the same direction as vector `v`.
`saturate(x)`	This clamps `x` to the `[0, 1]` range.
`tex2D(sampler, x)`	This is a 2D texture lookup (the sampled data at the location indicated by the texture coordinate set in the sampler object).

The methods mentioned in the preceding table are the common methods that you can use to create your shader with Cg/HLSL. There are a lot of methods that you can also use in Cg/HLSL.

For more details, you can visit the following site:

http://http.developer.nvidia.com/CgTutorial/cg_tutorial_appendix_e.html

Tip

Note that `UnpackNormal(x)` is the method that is provided by Unity to unpack the normal or bump texture, which you can find in the `UnityCG.cginc` file inside Unity (`unity install path}/Data/CGIncludes/UnityCG.cginc` in Windows and `/Applications/Unity/Unity.app/Contents/CGIncludes/UnityCG.cginc` in Mac).

References

For more details and references, please check out the following links:

- http://docs.unity3d.com/Documentation/Manual/Shaders.html
- http://docs.unity3d.com/Documentation/Components/SL-SurfaceShaders.html
- http://docs.unity3d.com/Documentation/Components/SL-PropertiesInPrograms.html
- https://docs.unity3d.com/Documentation/Components/SL-Properties.html
- http://docs.unity3d.com/Documentation/Components/SL-ShaderPerformance.html
- http://en.wikipedia.org/wiki/Cg_%28programming_language%29
- http://http.developer.nvidia.com/CgTutorial/cg_tutorial_frontmatter.html

Index

A

- AddExplosionForce() function / Engage thrusters
- AddForce() function / Classified intel
- addshadow parameter / #pragma surface
- AddWayPoint() function / Objective complete – mini debriefing
- AI behavior
  - features / Why is it awesome?
  - hotshot objectives / Your Hotshot objectives
  - mission checklist / Mission checklist
  - Waypoint script, creating / Creating the Waypoint and WaypointsContainer scripts, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - WaypointsContainer script, creating / Creating the Waypoint and WaypointsContainer scripts, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - custom editor, creating for WaypointsContainer script / Creating a custom editor for the WaypointsContainer script, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - enemy movement, creating with AI script / Creating the enemy movement with the AI script, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - hit-point UI, creating / Creating a hit-point UI, Engage thrusters, Objective complete – mini debriefing
  - mission accomplished / Mission accomplished
  - hotshot challenges / Hotshot challenges
- AI character
  - creating / Mission briefing
- AI script
  - enemy movement, creating with / Creating the enemy movement with the AI script, Engage thrusters, Objective complete – mini debriefing, Classified intel
- Albedo
  - about / Objective complete – mini debriefing
  - URL / Objective complete – mini debriefing
- alpha parameter / #pragma surface
- alphatest$VariableName parameter / #pragma surface
- Ambient light
  - adding / Shader programming – Ambient and Specular light, Engage thrusters, Objective complete – mini debriefing, Classified intel
- Animations inspector
  - about / The Animations inspector
  - URL, for information / The Animations inspector

- animation splitting
  - URL, for information / Engage thrusters
- animation transitions
  - URL, for information / Classified intel
- animator controller
  - creating / Creating an animator controller, Engage thrusters, Objective complete – mini debriefing
  - setting up / Setting up a character animation and animator controller, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
- approxview parameter / #pragma surface
- Awake() function
  - about / Engage thrusters, Awake()
  - example / Awake()
- AwayFromWayPoint() function / Objective complete – mini debriefing

B

- Blend Tree
  - about / Blend Tree
  - URL / Blend Tree
- Blinn-Phong
  - about / Shader programming – Diffuse and Bump (normal) maps
- bool data type
  - about / Data types
- BroadcastMessage() function
  - about / BroadcastMessage
- BroadcastMessage method / Objective complete – mini debriefing
- Bump map
  - creating / Shader programming – Diffuse and Bump (normal) maps, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel

C

- C#
  - about / Controlling the character with the PlayerController_2D class
- C# user
  - about / Variable declaration
- CameraControl script
  - features, adding to / Adding new features to the CharacterControl and CameraControl scripts, Engage thrusters, Objective complete – mini debriefing
- CanShoot() function / Engage thrusters
  - about / Classified intel
- casting
  - about / Casting
- ceil(x) method / Common methods to create shaders
- Cg/HLSL programming
  - about / Cg/HLSL Programming
  - shader properties, accessing / Accessing shader properties in Cg/HLSL
  - data types / Data types
  - methods used, to create shaders / Common methods to create shaders
- Cg/HLSL variables
  - properties, mapping to / Accessing shader properties in Cg/HLSL
- Cg parameter
  - about / Objective complete – mini debriefing
- character
  - controlling, with PlayerController_2D class / Controlling the character with the PlayerController_2D class, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
- character animation
  - setting up / Setting up character animation and level, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Setting up a character animation and animator controller, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
- character control
  - third-person camera, creating for / Creating a third-person camera to follow our character, Engage thrusters, Classified intel
- character control project
  - overview / Mission briefing
  - features / Why is it awesome?
  - Hotshot objectives / Your Hotshot objectives
  - requisites / Mission checklist
  - character animation, setting up / Setting up character animation and level, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
  - animator controller, creating / Creating an animator controller, Engage thrusters, Objective complete – mini debriefing

    - character control script, creating / Creating a character control script, Engage thrusters, Objective complete – mini debriefing
    - mission accomplished / Mission accomplished
    - Hotshot challenges / Hotshot challenges
- CharacterControl script
    - features, adding to / Adding new features to the CharacterControl and CameraControl scripts, Engage thrusters, Objective complete – mini debriefing
- character control script
    - creating / Creating a character control script, Engage thrusters, Objective complete – mini debriefing
- character sprite
    - creating / Engage thrusters, Objective complete – mini debriefing
- CheckMax() function / Engage thrusters, Objective complete – mini debriefing
- class declarations
    - about / Class declarations
- collider
    - setting up / Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
- Comparer<T> class
    - URL / Classified intel
- CompareTo() function / Creating the UserData and Hiscore scripts
- CompareTo() method
    - URL / Classified intel
- components, code snippet
    - about / Objective complete – mini debriefing
    - name / Objective complete – mini debriefing
    - display / Objective complete – mini debriefing
    - property type / Objective complete – mini debriefing
    - default / Objective complete – mini debriefing
- ConstantForce component
    - about / Objective complete – mini debriefing
- const keyword / Objective complete – mini debriefing
- coroutine
    - about / Coroutines, The yield instruction and coroutine
    - example / Coroutines
- Coroutines
    - about / Coroutines
- cross(A, B) method / Common methods to create shaders
- custom editor
    - creating, for WaypointsContainer script / Creating a custom editor for the WaypointsContainer script, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
- custom lighting model
    - about / Shader programming – Ambient and Specular light

    - URL / Objective complete – mini debriefing
- Custom Style feature
    - about / Objective complete – mini debriefing
- Custom Styles property / Engage thrusters

D

- 2D character
  - setting up, from sprite sheet / Prepare for lift off
- 2D level
  - setting up / Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
- 2D platform game project
  - overview / Mission briefing
  - features / Why is it awesome?
  - Hotshot objectives / Your Hotshot objectives
  - requisites / Mission checklist
  - 2D level, setting up / Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
  - collider, setting up / Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
  - 2D character, creating from sprite sheet / Prepare for lift off
  - character sprite, creating / Engage thrusters, Objective complete – mini debriefing
  - character, controlling with PlayerController_2D class / Controlling the character with the PlayerController_2D class, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - trigger event, creating for key game object / Creating a key, door, and replay button, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
  - trigger event, creating for door game object / Creating a key, door, and replay button, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
  - trigger event, creating for replay button / Creating a key, door, and replay button, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
  - mission accomplished / Mission accomplished
  - Hotshot challenges / Hotshot challenges
- 3D Studio Max
  - tips, for users / Tip for 3Ds Max users
- decal$add parameter / #pragma surface
- decal$blend parameter / #pragma surface
- Deferred Lighting
  - about / Objective complete – mini debriefing
- delegate function / Classified intel
- destructible wall
  - creating / Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
- Diffuse map
  - creating / Shader programming – Diffuse and Bump (normal) maps, Prepare for

lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
- dot(a, b) method / Common methods to create shaders
- Drag
  - about / Engage thrusters
- dualforward parameter / #pragma surface

E

- EditorApplication.hierarchyWindowChanged() function / Engage thrusters
- EditorGUILayout object / Objective complete – mini debriefing
- enemy movement
  - creating, with AI script / Creating the enemy movement with the AI script, Engage thrusters, Objective complete – mini debriefing, Classified intel
- enum type / Engage thrusters
- EQUIPMENT tab
  - about / Mission briefing
  - creating / Creating the EQUIPMENT tab, Engage thrusters, Objective complete – mini debriefing, Classified intel
- EquipWindow() function / Engage thrusters
- Event class
  - about / Engage thrusters
- event function / Classified intel
- exclude_path$forward parameter / #pragma surface
- exclude_path$prepass parameter / #pragma surface
- Exit Time parameter / Classified intel

F

- FBX import
  - URL, for information / Setting up character animation and level
- features
  - adding, to CharacterControl script / Adding new features to the CharacterControl and CameraControl scripts, Engage thrusters, Objective complete – mini debriefing
  - adding, to CameraControl script / Adding new features to the CharacterControl and CameraControl scripts, Engage thrusters, Objective complete – mini debriefing
- features, 2D platform game project / Why is it awesome?
- features, character control project / Why is it awesome?
- final
  - about / Engage thrusters
- final keyword / Objective complete – mini debriefing
- FindProperty
  - about / Objective complete – mini debriefing
- Fixed Angle
  - about / Objective complete – mini debriefing
- fixed data type
  - about / Data types
- FixedUpdate() function
  - about / Engage thrusters, Classified intel, FixedUpdate()
  - and Update() function, difference between / FixedUpdate()
  - example / FixedUpdate()
- F key / Prepare for lift off
- float data type
  - about / Data types
- floor(x) method / Common methods to create shaders
- foreach keyword
  - about / The foreach keyword
- Forward
  - about / Objective complete – mini debriefing
- fullforwardshadows parameter / #pragma surface
- function
  - defining / Function/method definitions
- function keyword
  - about / Function/method definitions

G

- generic functions
  - about / Classified intel
- generics
  - about / Generics
- GetDirection() function / Objective complete – mini debriefing
- GetDirectionToPlayer() function / Objective complete – mini debriefing
- get keyword / Engage thrusters
- getters/setters
  - properties with / Properties with getters/setters
- GetWaypointAtIndex() function / Classified intel
- Gizmos
  - about / Gizmos
- Gizmos() function / Mission accomplished
- Gizmos.DrawIcon() function
  - about / Engage thrusters
- Gizmos.DrawLine()
  - about / Engage thrusters
- Gizmos.DrawWireSphere() function
  - about / Engage thrusters
- Gizmos object
  - about / Creating the Waypoint and WaypointsContainer scripts
- GUI.BeginGroup() function / Engage thrusters, Classified intel, Mission accomplished
- GUI.BeginScrollView() function / Classified intel
- GUI.DragWindow() function / Classified intel
- GUI.SelectionGrid
  - URL / Classified intel
- GUI.SelectionGrid parameter / Objective complete – mini debriefing
- GUI.tooltip parameter / GUI.tooltip
- GUI class
  - URL / Classified intel
- GUILayout class / Classified intel
  - URL / Classified intel
- GUI object
  - about / Mission briefing
- GUI skin
  - customizing, with GUISkin feature / Customizing skin with GUISkin, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
- GUISkin feature
  - GUI skin, customizing with / Customizing skin with GUISkin, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel

H

- halfasview parameter / #pragma surface
- half data type
  - about / Data types
- Half Lambert
  - adding / Shader programming – Half Lambert, Rim Light, and Toon Ramp, Engage thrusters, Objective complete – mini debriefing
- Half Lambert lighting
  - about / Objective complete – mini debriefing
- high score
  - UserData script, creating / Creating the UserData and Hiscore scripts
  - Hiscore scripts, creating / Creating the UserData and Hiscore scripts
  - LocalHiscore script, creating / Saving and loading the local high score
  - XMLParser script, creating / Creating an XMLParser script
  - ServerHiScore script, creating / Saving and loading server high score
  - mission accomplished / Mission accomplished
  - challenges / Hotshot challenges
- Hiscore scripts
  - creating / Creating the UserData and Hiscore scripts, Prepare for lift off
- hit-point UI
  - creating / Creating a hit-point UI, Engage thrusters, Objective complete – mini debriefing

I

- input structure
  - additional values / Surface shaders input structure
- input structure, surface shaders / Surface shaders input structure
- Instantiate() function
  - about / Classified intel, Mission accomplished / Objective complete – mini debriefing
- Instantiate command
  - about / Casting
- int data type
  - about / Data types
- interactive environment
  - features / Why is it awesome?
  - challenges / Hotshot challenges
- interface support
  - about / Limited interface support
- Interpolate mode
  - about / Objective complete – mini debriefing
- INVENTORY tab
  - about / Mission briefing
  - creating / Creating the INVENTORY tab, Engage thrusters, Objective complete – mini debriefing
  - GUI.tooltip parameter / GUI.tooltip
- Invoke() function / Objective complete – mini debriefing
  - about / Classified intel, Mission accomplished
- InvokeRepeating()
  - about / Classified intel
- Is Kinematic property / Objective complete – mini debriefing
- item game object
  - creating / Mission briefing
- items.SetupScrollBar() / Engage thrusters
- ItemWindow() function / Objective complete – mini debriefing

J

- JavaScript user
  - about / Variable declaration
- Jump() function / Engage thrusters, Classified intel

L

- Lambert
  - about / Shader programming – Diffuse and Bump (normal) maps
- laser target scope
  - creating / Creating a MouseLook script and laser target scope, Engage thrusters, Objective complete – mini debriefing
- LateUpdate() function / Engage thrusters
  - about / Engage thrusters, Classified intel, LateUpdate()
  - example / LateUpdate()
- layers
  - about / Engage thrusters
  - URL, for information / Engage thrusters
- lighting model
  - about / #pragma surface
- Linear Drag
  - about / Engage thrusters
- Line Renderer component
  - using / Line Renderer
  - URL / Line Renderer
- List<T>.Reverse() method
  - URL / Classified intel
- List<T> method
  - URL / Classified intel
- LoadUserData() method / Engage thrusters
- LocalHiscore script
  - creating / Engage thrusters, Objective complete – mini debriefing, Classified intel
- LOD (Level of Detail)
  - about / Objective complete – mini debriefing
  - URL / Objective complete – mini debriefing

M

- Mass
  - about / Engage thrusters
- Mathf.Abs(n) function
  - about / Engage thrusters
- Mathf.Repeat(t,l) function
  - about / Engage thrusters
- Mathf.SmoothDamp() function / Engage thrusters
- Mathf.SmoothDampAngle() function / Engage thrusters
- max(a, b) method / Common methods to create shaders
- MD5 encryption / Mission briefing
- Mecanim animation system
  - about / Objective complete – mini debriefing
  - URL, for information / Objective complete – mini debriefing
- menu creation, RPG
  - hotshot objectives / Your Hotshot objectives
  - checklist / Mission checklist
  - GUI skin customization, with GUISkin feature / Customizing skin with GUISkin, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - menu game object, creating / Creating a menu object, Engage thrusters, Classified intel
  - STATUS tab, creating / Creating the STATUS tab, Engage thrusters, Classified intel
  - INVENTORY tab, creating / Creating the INVENTORY tab, Engage thrusters, Objective complete – mini debriefing
  - EQUIPMENT tab, creating / Creating the EQUIPMENT tab, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - mission accomplished / Mission accomplished
  - hotshot challenges / Hotshot challenges
- menu game object
  - creating / Mission briefing, Creating a menu object, Engage thrusters, Objective complete – mini debriefing, Classified intel
- method
  - defining / Function/method definitions
- method name (string)
  - StartCoroutine, used with / Using StartCoroutine with method name (string)
- min(a, b) method / Common methods to create shaders
- Mocap
  - about / Hotshot challenges
- MonoBehaviour
  - about / Update()
- MonoDevelop

    - about / Mission checklist, Controlling the character with the PlayerController_2D class
- MouseLook script
    - creating / Creating a MouseLook script and laser target scope, Engage thrusters, Objective complete – mini debriefing
- Move() function / Mission briefing
- multidimensional array declaration
    - about / Multidimensional array declaration

N

- new keyword
  - about / Class declarations, The new keyword
- Noambient parameter / #pragma surface
- Noforwardadd parameter / #pragma surface
- nolightmap parameter / #pragma surface
- normal
  - about / Engage thrusters
- normalize(v) method / Common methods to create shaders
- novertexlights parameter / #pragma surface
- NVIDIA CG
  - URL / Mission accomplished
- NVIDIA PhysX physics engine / Why is it awesome?

O

- o.Albedo parameter / Objective complete – mini debriefing
- OnAnimatorMove() function / Mission briefing, Engage thrusters
  - about / Engage thrusters
  - URL, for information / Engage thrusters
- OnDisable() function / Objective complete – mini debriefing
  - about / OnDisable()
  - example / OnDisable()
- OnDrawGizmos() function / Engage thrusters, Objective complete – mini debriefing
  - about / Mission briefing, Engage thrusters, Gizmos, OnDrawGizmos()
  - example / OnDrawGizmos()
- OnEnable() function / Objective complete – mini debriefing, Objective complete – mini debriefing
  - about / OnEnable()
  - example / OnEnable()
- OnGizmos() function
  - about / Engage thrusters
- OnGUI() function
  - about / OnGUI()
  - example / OnGUI()
- OnGUI function
  - about / Mission briefing, Engage thrusters
- OnTrigger() function / Objective complete – mini debriefing
- Option key / Engage thrusters
- Orthographic Projection / Classified intel
- output structure, surface shaders / The SurfaceOutput structure

P

- #pragma strict
  - about / Engage thrusters
- #pragma surface directive
  - about / #pragma surface
  - surfaceFunction parameter / #pragma surface
  - lightModel parameter / #pragma surface
  - optional parameters / #pragma surface
  - alpha parameter / #pragma surface
  - alphatest$VariableName parameter / #pragma surface
  - vertex$VertexFunction parameter / #pragma surface
  - exclude_path$prepass parameter / #pragma surface
  - exclude_path$forward parameter / #pragma surface
  - addshadow parameter / #pragma surface
  - dualforward parameter / #pragma surface
  - fullforwardshadows parameter / #pragma surface
  - decal$add parameter / #pragma surface
  - decal$blend parameter / #pragma surface
  - softvegetation parameter / #pragma surface
  - Noambient parameter / #pragma surface
  - novertexlights parameter / #pragma surface
  - nolightmap parameter / #pragma surface
  - Noforwardadd parameter / #pragma surface
  - approxview parameter / #pragma surface
  - halfasview parameter / #pragma surface
- ParseData() function / Engage thrusters
- ParseXMLData() function
  - URL / Classified intel
- particle effects
  - creating / Creating a rocket prefab and particle effects, Engage thrusters, Objective complete – mini debriefing, Classified intel
- Perspective Projection / Classified intel
- Phong reflection model
  - about / Shader programming – Diffuse and Bump (normal) maps
- Physics.OverlapSphere() function / Engage thrusters
- Physics.RayCast() function
  - about / Classified intel
- Physics.Raycast() function / Engage thrusters
  - about / Objective complete – mini debriefing
- Physics2D.Raycast / Physics2D.Raycast
- Physics2D Material
  - URL, for information / Classified intel
- physics 2D material object

    - creating / Classified intel
- PlayerController_2D class
    - character, controlling with / Controlling the character with the PlayerController_2D class, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
- PlayerPref
    - using / Saving and loading the local high score, Engage thrusters
    - URL / Classified intel
- Polygon Physics 2D
    - URL / Engage thrusters
- pow(x, y) method / Common methods to create shaders

R

- radius variable
  - about / Objective complete – mini debriefing
- ragdoll object
  - creating / Prepare for lift off, Engage thrusters
  - adding, to AI object / Classified intel
- readonly
  - about / Engage thrusters
- Rect object / Classified intel, Classified intel
- Rect parameter / Classified intel
- references
  - about / References
- references, Cg/HLSL programming
  - about / References
- RemoveWaypointAtIndex() function / Objective complete – mini debriefing
- rendering path
  - URL / Objective complete – mini debriefing
- RequireComponent
  - about / Engage thrusters
  - URL, for information / Engage thrusters
- Restart button / The Restart button
- Reverse() function / Classified intel
- Rigidbody
  - about / Engage thrusters
- rigidbody.AddForce() function / Objective complete – mini debriefing
- Rigidbody2D
  - about / Engage thrusters
- Rim Light
  - adding / Shader programming – Half Lambert, Rim Light, and Toon Ramp, Engage thrusters, Objective complete – mini debriefing
- rocket launcher
  - creating / Mission briefing, Creating a rocket launcher and RocketUI, Engage thrusters, Objective complete – mini debriefing
- rocket launcher creation
  - features / Why is it awesome?
  - hotshot objectives / Your Hotshot objectives
  - mission checklist / Mission checklist
  - character animation, setting up / Setting up a character animation and animator controller, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing
  - animator controller, setting up / Setting up a character animation and animator controller, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing

    - features, adding to CharacterControl script / Adding new features to the CharacterControl and CameraControl scripts, Engage thrusters, Objective complete – mini debriefing
    - features, adding to CameraControl script / Adding new features to the CharacterControl and CameraControl scripts, Engage thrusters, Objective complete – mini debriefing
    - MouseLook script, creating / Creating a MouseLook script and laser target scope, Engage thrusters, Objective complete – mini debriefing
    - laser target scope, creating / Creating a MouseLook script and laser target scope, Engage thrusters, Objective complete – mini debriefing
    - particle effects, creating / Creating a rocket prefab and particle effects, Engage thrusters, Objective complete – mini debriefing, Classified intel
    - rocket prefab, creating / Engage thrusters, Objective complete – mini debriefing, Classified intel
    - rocket launcher, creating / Creating a rocket launcher and RocketUI, Engage thrusters, Objective complete – mini debriefing
    - RocketUI, creating / Creating a rocket launcher and RocketUI, Engage thrusters, Objective complete – mini debriefing
    - mission accomplished / Mission accomplished
    - hotshot challenges / Hotshot challenges
- rocket prefab
    - creating / Creating a rocket prefab and particle effects, Engage thrusters, Objective complete – mini debriefing, Classified intel
- RocketUI
    - creating / Creating a rocket launcher and RocketUI, Engage thrusters, Objective complete – mini debriefing
- rockslide
    - creating / Prepare for lift off, Engage thrusters
- Rocks scripts
    - creating / Engage thrusters, Objective complete – mini debriefing, Classified intel
- RocksTrigger script
    - creating / Engage thrusters, Objective complete – mini debriefing, Classified intel
- rotation of imported model, fixing
    - URL / Tip for 3Ds Max users
- round(x) method / Common methods to create shaders

S

- sampler* data type
  - about / Data types
- sampler2
  - about / Objective complete – mini debriefing
- saturate(x) method / Common methods to create shaders
- SaveUserData() method / Engage thrusters
- script compilation order
  - URL / Prepare for lift off
- serialization / Classified intel
- serializedObject keyword / Classified intel
- SerializedObject keyword / Classified intel
- SerializedProperty class / Classified intel
  - advantage / Classified intel
- ServerHiScore script
  - about / Saving and loading server high score
  - creating, prerequisite / Prepare for lift off
  - creating / Engage thrusters
- ServerHiScore script variables
  - your host / Classified intel
  - your username / Classified intel
  - your password / Classified intel
- SetupScrollBar() function / Objective complete – mini debriefing, Objective complete – mini debriefing
- shader
  - creating / Mission briefing
- shader creation
  - features / Why is it awesome?
  - hotshot objectives / Your Hotshot objectives
  - mission checklist / Mission checklist
  - Diffuse map, creating / Shader programming – Diffuse and Bump (normal) maps, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - Bump map, creating / Shader programming – Diffuse and Bump (normal) maps, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - Ambient light, adding / Shader programming – Ambient and Specular light, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - Specular light, adding / Shader programming – Ambient and Specular light, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - Half Lambert, adding / Shader programming – Half Lambert, Rim Light, and Toon Ramp, Engage thrusters, Objective complete – mini debriefing
  - Rim Light, adding / Shader programming – Half Lambert, Rim Light, and Toon

Ramp, Engage thrusters, Objective complete – mini debriefing
    - Toon Ramp, adding / Shader programming – Half Lambert, Rim Light, and Toon Ramp, Engage thrusters, Objective complete – mini debriefing
    - mission accomplished / Mission accomplished
    - hotshot challenges / Hotshot challenges
- ShaderLab properties
    - about / ShaderLab properties
    - types / ShaderLab properties
    - example / ShaderLab properties
- shader programming
    - starting / Prepare for lift off
- shader properties
    - accessing, in Cg/HLSL / Accessing shader properties in Cg/HLSL
- showPath variable
    - about / Objective complete – mini debriefing
- Shuriken Particle
    - about / Mission briefing
- simple menu
    - creating / Mission briefing
- softvegetation parameter / #pragma surface
- Sort() method
    - URL / Classified intel
- Sorting Layer
    - about / The Sprite Renderer and Sorting Layer
    - URL, for information / The Sprite Renderer and Sorting Layer
- Specular light
    - adding / Shader programming – Ambient and Specular light, Engage thrusters, Objective complete – mini debriefing, Classified intel
- Sprite Renderer
    - about / The Sprite Renderer and Sorting Layer
    - URL, for information / The Sprite Renderer and Sorting Layer
- sprite sheet
    - URL, for information / Mission briefing
    - 2D character, setting up from / Prepare for lift off
- Start() function / Objective complete – mini debriefing, Objective complete – mini debriefing, Engage thrusters
    - about / Start()
    - example / Start()
- StartCoroutine
    - about / StartCoroutine
    - used, with method name (string) / Using StartCoroutine with method name (string)
- StartCoroutine function
    - about / StartCoroutine

- STATUS tab
  - about / Mission briefing
  - creating / Creating the STATUS tab, Engage thrusters, Classified intel
- StatusWindow() function / Engage thrusters, Objective complete – mini debriefing
- StopAllCoroutines
  - about / StopAllCoroutines
  - example / StopAllCoroutines
- StopCoroutine
  - about / StopCoroutine
  - example / StopCoroutine
- string.Format() function
  - about / Objective complete – mini debriefing
  - URL / Objective complete – mini debriefing
- struct properties
  - changing by value, versus by reference / Changing struct properties by value versus by reference
- Sub-State Machine
  - URL / Objective complete – mini debriefing
- surf() function
  - about / Shader programming – Ambient and Specular light, Engage thrusters
- surfaceFunction parameter / #pragma surface
- surface shaders
  - about / Surface shaders
  - #pragma surface directive / #pragma surface
  - input structure / Surface shaders input structure
  - output structure / The SurfaceOutput structure
- switch-case statement / Engage thrusters

T

- tags
  - about / Engage thrusters
  - URL, for information / Engage thrusters
- target keyword / Classified intel
- tex2D() function / Objective complete – mini debriefing
- tex2D(sampler, x) method / Common methods to create shaders
- TexturePacker
  - about / Mission briefing
  - URL / Mission briefing
- third-person camera
  - creating, for character control / Creating a third-person camera to follow our character, Engage thrusters, Classified intel
- Toon Ramp
  - adding / Shader programming – Half Lambert, Rim Light, and Toon Ramp, Engage thrusters, Objective complete – mini debriefing
- transform.localEulerAngles parameter / Engage thrusters
- trigger area
  - creating / Engage thrusters
- type names
  - about / Type names

U

- Undo.RecordObject() / Objective complete – mini debriefing
- Unify Community
  - URL / Mission accomplished
- Unity
  - URL, for downloading latest version / Mission checklist
- Unity 3D
  - URL, for documentation / Engage thrusters
  - URL, for functions / Engage thrusters
- Unity Asset Store
  - URL, for downloading Mecanim Locomotion Starter Kit package / Hotshot challenges
- Unity document
  - URL / Prepare for lift off
- Unity JavaScript
  - about / Controlling the character with the PlayerController_2D class
- Unity MonoBehaviour diagram
  - Awake() function / Awake()
  - Start() function / Start()
  - Update() function / Update()
  - FixedUpdate() function / FixedUpdate()
  - LateUpdate() function / LateUpdate()
  - OnEnable() function / OnEnable()
  - OnDisable() function / OnDisable()
  - OnGUI() function / OnGUI()
  - OnDrawGizmos() function / OnDrawGizmos()
- Unity script directives
  - about / Unity script directives
- Unity scripting document links
  - references / References
- Unity scripting documents
  - references / References
- Unity ShaderLab forum
  - URL / Mission accomplished
- Unity Shader Reference
  - URL / Mission accomplished
- UnpackNormal() function / Objective complete – mini debriefing
- UnpackNormal(x) method / Common methods to create shaders
- unsupported character literals
  - about / Character literals not supported
- Update() function / Engage thrusters
  - about / Engage thrusters, Classified intel, Engage thrusters, Update()
  - example / Update()

    - FixedUpdate() function, difference between / FixedUpdate()
- UpdateAnimator() function
    - about / Objective complete – mini debriefing
- UserData script
    - creating / Creating the UserData and Hiscore scripts, Prepare for lift off, Engage thrusters
- uv_BumpMap parameter / Objective complete – mini debriefing

V

- variable declaration
  - about / Variable declaration
- variables
  - with dynamically typed resolution / Variables with dynamically typed resolution
- Vector3.Slerp() function
  - about / Engage thrusters
  - URL, for information / Engage thrusters
- vertex$VertexFunction parameter / #pragma surface
- Vertex Lit
  - about / Objective complete – mini debriefing

W

- WaitForFixedUpdate
  - about / WaitForFixedUpdate
  - example / WaitForFixedUpdate
- WaitForSeconds
  - about / WaitForSeconds
  - example / WaitForSeconds
- WaitForServerResponse() function / Engage thrusters
- WaitingForResponse() function / Objective complete – mini debriefing
- WaypointsContainer script
  - creating / Creating the Waypoint and WaypointsContainer scripts, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
  - custom editor, creating for / Creating a custom editor for the WaypointsContainer script, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel
- Waypoint script
  - creating / Creating the Waypoint and WaypointsContainer scripts, Prepare for lift off, Engage thrusters, Objective complete – mini debriefing, Classified intel

X

- XMLParser script
  - creating / Engage thrusters, Objective complete – mini debriefing, Classified intel

Y

- yield command
  - about / Coroutines
- YieldInstruction
  - about / YieldInstruction
- yield instruction
  - about / The yield instruction and coroutine
- yield return value
  - about / StartCoroutine
- yield statement
  - about / YieldInstruction