The background of the cover features a complex, abstract geometric pattern composed of numerous blue and purple facets, creating a sense of depth and perspective.

Community Experience Distilled

Learning Unity Android Game Development

Learn to create stunning Android games using Unity

Thomas Finnegan

www.allitebooks.com

[PACKT]
PUBLISHING

Learning Unity Android Game Development

Learn to create stunning Android games using Unity

Thomas Finnegan



BIRMINGHAM - MUMBAI

Learning Unity Android Game Development

Copyright © 2015 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: April 2015

Production reference: 1240415

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-469-1

www.packtpub.com

Credits

Author

Thomas Finnegan

Project Coordinator

Harshal Ved

Reviewer

Ryan Watkins

Proofreaders

Safis Editing

Lauren E. Harkins

Commissioning Editor

Ashwin Nair

Indexer

Hemangini Bari

Acquisition Editor

Harsha Bharwani

Production Coordinator

Aparna Bhagat

Content Development Editor

Ajinkya Paranjape

Cover Work

Aparna Bhagat

Technical Editor

Faisal Siddiqui

Copy Editors

Dipti Kapadia

Jasmine Nadar

About the Author

Thomas Finnegan graduated from Brown College in 2010, and he now works as a freelance game developer. Since 2010, he has worked on everything from mobile platforms to web development, and he has even worked with experimental devices. His past clients include Carmichael Lynch, Coleco, and Subaru. His most recent project is Battle Box 3D, a virtual tabletop. Currently, he teaches game development at the Minneapolis Media Institute in Minnesota.

I would like to thank my fiancée for kicking me back into gear whenever I become distracted. I would also like to thank my parents for always supporting me. Without my friends and family, my life would be dismal.

About the Reviewer

Ryan Watkins sometimes enjoys Black Forest cake. You can find Ryan on LinkedIn at www.linkedin.com/in/ryanswatkins.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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



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

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

Why subscribe?

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

Free access for Packt account holders

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

Table of Contents

Preface	vii
Chapter 1: Saying Hello to Unity and Android	1
Understanding what makes Unity great	2
Unity – the best among the rest	3
Understanding what makes Android great	4
Understanding how Unity and Android work together	4
Differences between the Pro and Basic versions of Unity	5
An overview of license comparison	6
NavMeshes, pathfinding, and crowd simulation	6
LOD support	6
The audio filter	6
Video playback and streaming	7
Fully-fledged streaming with asset bundles	7
The 100,000 dollar turnover	7
Mecanim – IK Rigs	7
Mecanim – sync layers and additional curves	8
The custom splash screen	8
Real-time spot/point and soft shadows	8
HDR and tone mapping	9
Light probes	9
Lightmapping with global illumination and area lights	9
Static batching	10
Render-to-texture effects	10
Fullscreen post-processing effects	10
Occlusion culling	10
Deferred rendering	10
Stencil buffer access	11
GPU skinning	11
Navmesh – dynamic obstacles and priority	11
Native code plugins' support	11
Profiler and GPU profiling	11
Script access to the asset pipeline	11
Dark skin	12

Table of Contents

Setting up the development environment	12
Installing the JDK	12
Installing the Android SDK	13
Installing Unity 3D	16
The optional code editor	19
Connecting to a device	19
A simple device connection	19
Unity Remote	22
Building a simple application	23
Hello World	23
Summary	31
Chapter 2: Looking Good – The Graphical Interface	33
Creating a Tic-tac-toe game	34
The game board	34
Creating the board	36
Game squares	39
Controlling the game	40
Messing with fonts	45
Rotating devices	48
Menus and victory	51
Setting up the elements	52
Adding the code	54
Putting them together	58
A better way to build for a device	59
Summary	61
Chapter 3: The Backbone of Any Game – Meshes, Materials, and Animations	63
Setting up	64
Importing the meshes	65
Tank import settings	66
Meshes	68
Normals & Tangents	69
Materials	70
The Revert and Apply buttons	71
Setting up the tank	72
The tank	72
Keeping score	75
Repeat buttons	76
Controlling the chassis	78
Controlling the turret	80
Putting the pieces together	82

Table of Contents

Creating materials	86
The city	86
Main Maps	88
Secondary Maps	89
Moving treads	93
Animations in Unity	94
The target's animations	100
State machines to control animations in Unity	101
Target state machine	101
Scripting the target	109
Creating the prefab	112
Ray tracing to shooting	114
Summary	117
Chapter 4: Setting the Stage – Camera Effects and Lighting	119
Camera effects	120
Skyboxes and distance fog	120
Target indicator	122
Creating the pointer	122
Controlling the indicator	124
Working with a second camera	126
Turbo boost	129
Lights	132
Adding more lights	133
Lightmaps	136
Cookies	141
Blob shadows	143
Summary	146
Chapter 5: Getting Around – Pathfinding and AI	147
Understanding AI and pathfinding	148
The NavMesh	148
The NavMeshAgent component	154
Making the enemy chase the player	157
Revealing the player's location	157
Chasing the player	158
Being attacked by the enemy	161
Attacking the enemy	166
Spawning enemy tanks	168
Summary	171

Table of Contents

Chapter 6: Specialities of the Mobile Device – Touch and Tilt	173
Setting up the development environment	174
A basic environment	175
Controlling with tilt	176
Following with the camera	179
Adding the monkey	181
Keeping the monkey on the board	184
Winning and losing the game	186
Putting together the complex environment	190
Adding bananas	193
Collecting bananas with touch	196
Summary	200
Chapter 7: Throwing Your Weight Around – Physics and a 2D Camera	201
2D games in a 3D world	202
Setting up the development environment	202
Physics	205
Building blocks	205
Physics materials	210
Characters	210
Creating the enemy	211
Creating the ally	216
Controls	217
Attacking with a slingshot	218
Watching with the camera	227
Creating the parallax background	232
Adding more birds	235
The yellow bird	235
The blue bird	236
The black bird	238
Level selection	241
Summary	243
Chapter 8: Special Effects – Sound and Particles	245
Understanding audio	246
Import settings	246
Audio Listener	248
Audio Source	248
Adding background music	251
Poking bananas	253

Table of Contents

Understanding particle systems	255
Particle system settings	255
Creating dust trails	265
Putting it together	270
Exploding bananas	270
Summary	275
Chapter 9: Optimization	277
Minimizing the application footprint	277
Editor log	278
Asset compression	279
Models	279
The Model tab	280
The Rig tab	282
The Animations tab	283
Textures	284
Audio	286
Player settings	288
Rendering	288
Optimization	290
Tracking performance	291
Editor statistics	292
The Profiler	294
Tracking script performance	296
Minimizing lag	303
Occlusion	304
Tips for minimizing lag	308
Summary	309
Index	311

Preface

In this book, we explore the ever-expanding world of mobile game development. Using Unity 3D and Android SDK, we learn how to create every aspect of a mobile game while leveraging the new features of Unity 5.0 and Android L. Every chapter explores a new piece of the development puzzle. By exploring the special features of development with mobile platforms, every game in the book is designed to increase your understanding of these features. We will finish the book with a total of four complete games and all of the tools that you need to create many more.

The first game that we will make is Tic-Tac-Toe. This game functions just like the classic paper version. Two players take turns filling a grid with their symbols and the first to make a line of the same three symbols wins. This is the perfect game for us to explore the graphical interface options that we have in Unity. By learning how to add buttons, text, and pictures to the screen, you will have all the understanding and the tools that are needed to add any interface that you might want to any game.

The next game that we will create is the Tank Battle game. In this game, the player takes control of a tank to drive around a small city and shoot targets and enemies. This game spans three chapters, allowing us to explore the many key points of creating games for the Android platform. We start it by creating a city and making the player's tank move around by using the controls about which we learned when we made the Tic-Tac-Toe game. We also create and animate the targets at which a player will shoot. In the second part of this game, we add some lighting and special camera effects. By the end of the chapter, the environment looks great. In the third part of the game's creation, we create some enemies. Using the power of Unity, these enemies chase the player around the city and attack them when they are close.

The third game is a simple clone of a popular mobile game. Using the power of Unity's physics system, we are able to create structures and throw birds at them. Knock down the structures to gain points and destroy the target pigs to win the level. We also explore some of the specific features of a 2D game and Unity's 2D pipeline, such as a parallax scrolling background and the use of sprites. We complete the chapter and the game with the creation of a level-selection menu and the saving of high scores.

Finally, we will create a Monkey Ball-style game. This game involves using the special inputs of a mobile device to control the movement of the ball and the player's interaction with the world. When a player's device is tilted, they will be able to guide the monkey around the level. When they touch the screen, they can do damage and eventually collect bananas that are scattered throughout the game. This game also shows you how to include the special effects that are necessary to complete the look of every game. We create explosions when bananas are collected and dust trails when our monkey moves around. We also add in sound effects for touching and exploding.

We wrap up the book by taking a look at optimization. We explore all the great features of Unity and even create a few of our own to make our game run as best it can. We also take a little bit of time to understand some things that we can do to minimize the file size of our assets while maximizing their look and effect in the game. At this point, our journey ends, but we will have four great games that are just about ready for the market.

What this book covers

Chapter 1, Saying Hello to Unity and Android, explores the feature lists of the Android platform and the Unity 3D game engine and explains why they are great choices for development. We also cover the setting up of the development environment and create a simple Hello World application for your device and emulators.

Chapter 2, Looking Good – The Graphical Interface, takes a detailed look at the graphical user interface. By creating a Tic-Tac-Toe game, you learn about the user interface while you make it pleasing to look at.

Chapter 3, The Backbone of Any Game – Meshes, Materials, and Animations, explores how you can utilize meshes, materials, and animations in Unity. Through the creation of a Tank Battle game, we cover the core of what players will see when they play the game.

Chapter 4, Setting the Stage – Camera Effects and Lighting, explains the camera effects and lighting options that are available in Unity. With the addition of shadows, lightmaps, distance fog, and a skybox, our Tank Battle environment becomes more dynamic. By utilizing special camera effects, we create extra feedback for players.

Chapter 5, Getting Around – Pathfinding and AI, shows the creation of bile enemies in our Tank Battle game. We explore pathfinding and AI to give players a target that is more meaningful than a stationary dummy.

Chapter 6, Specialities of the Mobile Device – Touch and Tilt, covers the features that make the modern mobile device special. We create a Monkey Ball-style game to understand the touch interface and tilt controls.

Chapter 7, Throwing Your Weight Around – Physics and a 2D Camera, shows you how to create a clone of the Angry Birds game while taking a short break from the Monkey Ball game. Physics and Unity's 2D pipeline are also explored here.

Chapter 8, Special Effects – Sound and Particles, returns us to the Monkey Ball game to add special effects. The inclusion of sound effects and particles allows us to create a more complete game experience.

Chapter 9, Optimization, covers optimization in Unity 3D. We cover the benefits and costs of making our Tank Battle and Monkey Ball games as efficient as possible.

What you need for this book

Throughout this book, we will be working with both the Unity 3D game engine and the Android platform. As you have seen in the previous section, we will cover both the acquisition and installation of Unity and Android SDK in the first chapter. To get the most out of this book, you will need access to an Android-powered device; either a phone or tablet that will work well. Some sections of the book cover features that are only available in the Pro version of Unity. For simplicity's sake, we will assume that you are working on a Windows-powered computer. In addition, the code throughout the book is written in C#, though JavaScript versions of each chapter's project are available for reference. To fully utilize the models provided for each chapter's projects, you will need Blender, which is a free modeling program that is available at <http://www.blender.org>. You will also need a photo editing program; both Photoshop and Gimp are excellent choices. You will need both a modeling program, such as Blender, and an image editing program, such as Photoshop or Gimp, to create and work with your own content. We also recommend that you obtain a source by which to create or acquire audio files. All of the audio files provided by this book can be found at <http://www.freesound.org>.

Who this book is for

This book will be optimal for readers who are new to game development and mobile development using Unity 5.0 and Android L. Readers who learn best with real-world examples rather than dry documentation will find every chapter useful. Even if you have little or no programming skill, this book will enable you to jump in and learn some concepts and standards for programming and game development.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `CheckVictory` function runs through the possible combinations for victory in the game."

A block of code is set as follows:

```
public void NewGame() {  
    xTurn = true;  
    board = new SquareState[9];  
    turnIndicatorLandscape.text = "X's Turn";  
}
```

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

```
adb kill-server  
adb start-server  
adb devices
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Follow that up by clicking on the **Download the SDK Tools for Windows** button."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

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

Customer support

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

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. 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 with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/LearningUnityAndroidGameDevelopment_Graphics.pdf.

Errata

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

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

Piracy

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

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

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

Questions

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

1

Saying Hello to Unity and Android

Welcome to the wonderful world of mobile game development. Whether you are still looking for the right development kit or have already chosen one, this chapter will be very important. In this chapter, we explore the various features that come with choosing **Unity** as your development environment and **Android** as the target platform. Through comparison with major competitors, it is discovered why Unity and Android stand at the top of the pile. Following this, we will examine how Unity and Android work together. Finally, the development environment will be set up and we will create a simple Hello World application to test whether everything is set up correctly. For the purposes of this book, it is assumed that you are working in a Windows-based environment.

In this chapter, we will cover the following topics:

- Major Unity features
- Major Android features
- Unity licensing options
- Installing the JDK
- Installing the Android software development kit (SDK)
- Installing Unity 3D
- Installing Unity Remote

Understanding what makes Unity great

Perhaps the greatest feature of Unity is how open-ended it is. Nearly all game engines currently on the market are limited in what one can build with them. It makes perfect sense but it can limit the capabilities of a team. The average game engine has been highly optimized for creating a specific game type. This is great if all you plan on making is the same game again and again. It can be quite frustrating when one is struck with inspiration for the next great hit, only to find that the game engine can't handle it and everyone has to retrain in a new engine or double the development time to make the game engine capable. Unity does not suffer from this problem. The developers of Unity have worked very hard to optimize every aspect of the engine, without limiting what types of games can be made using it. Everything ranging from simple 2D platformers to massive online role-playing games are possible in Unity. A development team that just finished an ultrarealistic first-person shooter can turn right around and make 2D fighting games without having to learn an entirely new system.

Being so open-ended does, however, bring a drawback. There are no default tools that are optimized for building the perfect game. To combat this, Unity grants the ability to create any tool one can imagine, using the same scripting that creates the game. On top of that, there is a strong community of users that have supplied a wide selection of tools and pieces, both free and paid, that can be quickly plugged in and used. This results in a large selection of available content that is ready to jump-start you on your way to the next great game.

When many prospective users look at Unity, they think that, because it is so cheap, it is not as good as an expensive AAA game engine. This is simply not true. Throwing more money at the game engine is not going to make a game any better. Unity supports all of the fancy shaders, normal maps, and particle effects that you could want. The best part is that nearly all of the fancy features that you could want are included in the free version of Unity, and 90 percent of the time beyond that, you do not even need to use the Pro-only features.

One of the greatest concerns when selecting a game engine, especially for the mobile market, is how much girth it will add to the final build size. Most game engines are quite hefty. With Unity's code stripping, the final build size of the project becomes quite small. Code stripping is the process by which Unity removes every extra little bit of code from the compiled libraries. A blank project compiled for Android that utilizes full code stripping ends up being around 7 megabytes.

Perhaps one of the coolest features of Unity is its multi-platform compatibility. With a single project, one can build for several different platforms. This includes the ability to simultaneously target mobiles, PCs, and consoles. This allows you to focus on real issues, such as handling inputs, resolution, and performance.

In the past, if a company desired to deploy their product on more than one platform, they had to nearly double the development costs in order to essentially reprogram the game. Every platform did, and still does, run by its own logic and language. Thanks to Unity, game development has never been simpler. We can develop games using simple and fast scripting, letting Unity handle the complex translation to each platform.

Unity – the best among the rest

There are of course several other options for game engines. Two major ones that come to mind are **cocos2d** and **Unreal Engine**. While both are excellent choices, you will find them to be a little lacking in certain respects.

The engine of Angry Birds, cocos2d, could be a great choice for your next mobile hit. However, as the name suggests, it is pretty much limited to 2D games. A game can look great in it, but if you ever want that third dimension, it can be tricky to add it to cocos2d; you may need to select a new game engine. A second major problem with cocos2d is how bare bones it is. Any tool for building or importing assets needs to be created from scratch, or it needs to be found. Unless you have the time and experience, this can seriously slow down development.

Then there is the staple of major game development, Unreal Engine. This game engine has been used successfully by developers for many years, bringing great games to the world Unreal Tournament and Gears of War not the least among them. These are both, however, console and computer games, which is the fundamental problem with the engine. Unreal is a very large and powerful engine. Only so much optimization can be done on it for mobile platforms. It has always had the same problem; it adds a lot of girth to a project and its final build. The other major issue with Unreal is its rigidity in being a first-person shooter engine. While it is technically possible to create other types of games in it, such tasks are long and complex. A strong working knowledge of the underlying system is a must before achieving such a feat.

All in all, Unity definitely stands strong amidst game engines. Perhaps, you have already discovered this and that is why you are reading this book. But these are still great reasons for choosing Unity for game development. Unity projects can look just as great as AAA titles. The overhead and girth in the final build are small and this is very important when working on mobile platforms. The system's potential is open enough to allow you to create any type of game that you might want, where other engines tend to be limited to a single type of game. In addition, should your needs change at any point in the project's life cycle, it is very easy to add, remove, or change your choice of target platforms.

Understanding what makes Android great

With over 30 million devices in the hands of users, why would you not choose the Android platform for your next mobile hit? Apple may have been the first one out of the gate with their iPhone sensation, but Android is definitely a step ahead when it comes to smartphone technology. One of its best features is its blatant ability to be opened up so that you can take a look at how the phone works, both physically and technically. One can swap out the battery and upgrade the micro SD card on nearly all Android devices, should the need arise. Plugging the phone into a computer does not have to be a huge ordeal; it can simply function as a removable storage media.

From the point of view of the cost of development as well, the Android market is superior. Other mobile app stores require an annual registration fee of about 100 dollars. Some also have a limit on the number of devices that can be registered for development at one time. The Google Play market has a one-time registration fee of 25 dollars, and there is no concern about how many Android devices or what type of Android devices you are using for development.

One of the drawbacks of some of the other mobile development kits is that you have to pay an annual registration fee before you have access to the SDK. With some, registration and payment are required before you can view their documentation. Android is much more open and accessible. Anybody can download the Android SDK for free. The documentation and forums are completely viewable without having to pay any fee. This means development for Android can start earlier, with device testing being a part of it from the very beginning.

Understanding how Unity and Android work together

As Unity handles projects and assets in a generic way, there is no need to create multiple projects for multiple target platforms. This means that you could easily start development with the free version of Unity and target personal computers. Then, at a later date, you can switch targets to the Android platform with the click of a button. Perhaps, shortly after your game is launched, it takes the market by storm and there is a great call to bring it to other mobile platforms. With just another click of the button, you can easily target iOS without changing anything in your project.

Most systems require a long and complex set of steps to get your project running on a device. For the first application in this book, we will be going through that process because it is important to know about it. However, once your device is set up and recognized by the Android SDK, a single button click will allow Unity to build your application, push it to a device, and start running it. There is nothing that has caused more headaches for some developers than trying to get an application on a device. Unity makes this simple.

With the addition of a free Android application, **Unity Remote**, it is simple and easy to test mobile inputs without going through the whole build process. While developing, there is nothing more annoying than waiting for 5 minutes for a build every time you need to test a minor tweak, especially in the controls and interface. After the first dozen little tweaks, the build time starts to add up. Unity Remote makes it simple and easy to test everything without ever having to hit the **Build** button.

These are the big three reasons why Unity works well with Android:

- Generic projects
- A one-click build process
- Unity Remote

We could, of course, come up with several more great ways in which Unity and Android can work together. However, these three are the major time and money savers. You could have the greatest game in the world, but if it takes 10 times longer to build and test, what is the point?

Differences between the Pro and Basic versions of Unity

Unity comes with two licensing options, Pro and Basic, which can be found at <https://store.unity3d.com>. In order to follow the bulk of this book, Unity Basic is all that is required. However, real-time shadows in *Chapter 4, Setting the Stage – Camera Effects and Lighting*, and some of the optimization features discussed in *Chapter 9, Optimization*, will require Unity Pro. If you are not quite ready to spend the 3,000 dollars that is required to purchase a full Unity Pro license with the Android add-on, there are other options. Unity Basic is free and comes with a 30-day free trial of Unity Pro. This trial is full and complete, as if you have purchased Unity Pro, the only downside being a watermark in the bottom-right corner of your game stating **Demo Use Only**. It is also possible to upgrade your license at a later date. Where Unity Basic comes with mobile options for free, Unity Pro requires the purchase of Pro add-ons for each of the mobile platforms.

An overview of license comparison

License comparisons can be found at <http://unity3d.com/unity/licenses>. This section will cover the specific differences between Unity Android Pro and Unity Android Basic. We will explore what the features are and how useful each one is in the following points:

NavMeshes, pathfinding, and crowd simulation

This feature is Unity's built-in pathfinding system. It allows characters to find their way from a point to another around your game. Just bake your navigation data in the editor and let Unity take over at runtime. Until recently, this was a Unity Pro only feature. Now the only part of it that is limited in Unity Basic is the use of off-mesh links. The only time you are going to need them is when you want your AI characters to be able to jump across and otherwise navigate around gaps.

LOD support

LOD (short for **level of detail**) lets you control how complex a mesh is, based on its distance from the camera. When the camera is close to an object, you can render a complex mesh with a bunch of detail in it. When the camera is far from that object, you can render a simple mesh because all that detail is not going to be seen anyway. Unity Pro provides a built-in system to manage this. However, this is another system that could be created in Unity Basic. Whether or not you are using the Pro version, this is an important feature for game efficiency. By rendering less complex meshes at a distance, everything can be rendered faster, leaving more room for awesome gameplay.

The audio filter

Audio filters allow you to add effects to audio clips at runtime. Perhaps you created gravel脚步 sounds for your character. Your character is running and we can hear the footsteps just fine, when suddenly they enter a tunnel and a solar flare hits, causing a time warp and slowing everything down. Audio filters would allow us to warp the gravel脚步 sounds to sound as if they were coming from within a tunnel and were slowed by a time warp. Of course, you could also just have the audio guy create a new set of tunnel gravel footsteps in the time warp sounds, although this might double the amount of audio in your game and limit how dynamic we can be with it at runtime. We either are or are not playing the time warp footsteps. Audio filters would allow us to control how much time warp is affecting our sounds.

Video playback and streaming

When dealing with complex or high-definition cut scenes, being able to play videos becomes very important. Including them in a build, especially with a mobile target, can require a lot of space. This is where the streaming part of this feature comes in. This feature not only lets us play videos but also lets us stream a video from the Internet. There is, however, a drawback to this feature. On mobile platforms, the video has to go through the device's built-in video-playing system. This means that the video can only be played in fullscreen and cannot be used as a texture for effects such as moving pictures on a TV model. Theoretically, you could break your video into individual pictures for each frame and flip through them at runtime, but this is not recommended for build size and video quality reasons.

Fully-fledged streaming with asset bundles

Asset bundles are a great feature provided by Unity Pro. They allow you to create extra content and stream it to users without ever requiring an update to the game. You could add new characters, levels, or just about any other content you can think of. Their only drawback is that you cannot add more code. The functionality cannot change, but the content can. This is one of the best features of Unity Pro.

The 100,000 dollar turnover

This one isn't so much a feature as it is a guideline. According to Unity's End User License Agreement, the basic version of Unity cannot be licensed by any group or individual who made \$100,000 in the previous fiscal year. This basically means that if you make a bunch of money, you have to buy Unity Pro. Of course, if you are making that much money, you can probably afford it without an issue. This is the view of Unity at least and the reason why there is a 100,000 dollar turnover.

Mecanim – IK Rigs

Unity's new animation system, **Mecanim**, supports many exciting new features, one of which is **IK** (short form for **Inverse Kinematics**). If you are unfamiliar with the term, IK allows one to define the target point of an animation and let the system figure out how to get there. Imagine you have a cup sitting on a table and a character that wants to pick it up. You could animate the character to bend over and pick it up; but, what if the character is slightly to the side? Or any number of other slight offsets that a player could cause, completely throwing off your animation? It is simply impractical to animate for every possibility. With IK, it hardly matters that the character is slightly off.

We just define the goal point for the hand and leave the animation of the arm to the IK system. It calculates how the arm needs to move in order to get the hand to the cup. Another fun use is making characters look at interesting things as they walk around a room: a guard could track the nearest person, the player's character could look at things that they can interact with, or a tentacle monster could lash out at the player without all the complex animation. This will be an exciting one to play with.

Mecanim – sync layers and additional curves

Sync layers, inside Mecanim, allow us to keep multiple sets of animation states in time with each other. Say you have a soldier that you want to animate differently based on how much health he has. When he is at full health, he walks around briskly. After a little damage to his health, the walk becomes more of a trudge. If his health is below half, a limp is introduced into his walk, and when he is almost dead he crawls along the ground. With sync layers, we can create one animation state machine and duplicate it to multiple layers. By changing the animations and syncing the layers, we can easily transition between the different animations while maintaining the state machine.

The additional curves feature is simply the ability to add curves to your animation. This means we can control various values with the animation. For example, in the game world, when a character picks up its feet for a jump, gravity will pull them down almost immediately. By adding an extra curve to that animation, in Unity, we can control how much gravity is affecting the character, allowing them to actually be in the air when jumping. This is a useful feature for controlling such values alongside the animations, but you could just as easily create a script that holds and controls the curves.

The custom splash screen

Though pretty self-explanatory, it is perhaps not immediately evident why this feature is specified, unless you have worked with Unity before. When an application that is built in Unity initializes on any platform, it displays a splash screen. In Unity Basic, this will always be the Unity logo. By purchasing Unity Pro, you can substitute for the Unity logo with any image you want.

Real-time spot/point and soft shadows

Lights and shadows add a lot to the mood of a scene. This feature allows us to go beyond blob shadows and use realistic-looking shadows. This is all well and good if you have the processing space for it. However, most mobile devices do not. This feature should also never be used for static scenery; instead, use static lightmaps, which is what they are for.

However, if you can find a good balance between simple needs and quality, this could be the feature that creates the difference between an alright and an awesome game. If you absolutely must have real-time shadows, the directional light supports them and is the fastest of the lights to calculate. It is also the only type of light available to Unity Basic that supports real-time shadows.

HDR and tone mapping

HDR (short for **high dynamic range**) and tone mapping allow us to create more realistic lighting effects. Standard rendering uses values from zero to one to represent how much of each color in a pixel is on. This does not allow for a full spectrum of lighting options to be explored. HDR lets the system use values beyond this range and processes them using tone mapping to create better effects, such as a bright morning room or the bloom from a car window reflecting the sun. The downside of this feature is in the processor. The device can still only handle values between zero and one, so converting them takes time. Additionally, the more complex the effect, the more time it takes to render it. It would be surprising to see this used well on handheld devices, even in a simple game. Maybe the modern tablets could handle it.

Light probes

Light probes are an interesting little feature. When placed in the world, light probes figure out how an object should be lit. Then, as a character walks around, they tell it how to be shaded. The character is, of course, lit by the lights in the scene, but there are limits on how many lights can shade an object at once. Light probes do all the complex calculations beforehand, allowing for better shading at runtime. Again, however, there are concerns about processing power. Too little power and you won't get a good effect; too much and there will be no processing power left for playing the game.

Lightmapping with global illumination and area lights

All versions of Unity support lightmaps, allowing for the baking of complex static shadows and lighting effects. With the addition of global illumination and area lights, you can add another touch of realism to your scenes. However, every version of Unity also lets you import your own lightmaps. This means that you could use some other program to render the lightmaps and import them separately.

Static batching

This feature speeds up the rendering process. Instead of spending time grouping objects for faster rendering on each frame , this allows the system to save the groups generated beforehand. Reducing the number of draw calls is a powerful step towards making a game run faster. That is exactly what this feature does.

Render-to-texture effects

This is a fun feature, but of limited use. It allows you to use the output from a camera in your game as a texture. This texture could then, in its most simple form, be put onto a mesh and act as a surveillance camera. You could also do some custom post processing, such as removing the color from the world as the player loses their health. However, this option could become very processor-intensive.

Fullscreen post-processing effects

This is another processor-intensive feature that probably will not make it into your mobile game. However, you can add some very cool effects to your scene, such as adding motion blur when the player is moving really fast or a vortex effect to warp the scene as the ship passes through a warped section of space. One of the best effects is using the bloom effect to give things a neon-like glow.

Occlusion culling

This is another great optimization feature. The standard camera system renders everything that is within the camera's view frustum, the view space. Occlusion culling lets us set up volumes in the space our camera can enter. These volumes are used to calculate what the camera can actually see from those locations. If there is a wall in the way, what is the point of rendering everything behind it? Occlusion culling calculates this and stops the camera from rendering anything behind that wall.

Deferred rendering

If you desire the best looking game possible, with highly detailed lighting and shadows, this is a feature of interest for you. Deferred rendering is a multi-pass process for calculating your game's light and shadow detail. This is, however, an expensive process and requires a decent graphics card to fully maximize its use. Unfortunately, this makes it a little outside of our use for mobile games.

Stencil buffer access

Custom shaders can use the stencil buffer to create special effects by selectively rendering over specific pixels. It is similar to how one might use an alpha channel to selectively render parts of a texture.

GPU skinning

This is a processing and rendering method by which the calculations for how a character or object appears, when using a skeleton rig, is given to the graphics card rather than getting it done by the central processor. It is significantly faster to render objects in this way. However, this is only supported on DirectX 11 and OpenGL ES 3.0, leaving it a bit out of reach for our mobile games.

Navmesh – dynamic obstacles and priority

This feature works in conjunction with the pathfinding system. In scripts, we can dynamically set obstacles, and characters will find their way around them. Being able to set priorities means that different types of characters can take different types of objects into consideration when finding their way around. For example, a soldier must go around the barricades to reach his target. The tank, however, could just crash through, should the player desire.

Native code plugins' support

If you have a custom set of code in the form of a **Dynamic Link Library (DLL)**, this is the Unity Pro feature you need access to. Otherwise, the native plugins cannot be accessed by Unity for use with your game.

Profiler and GPU profiling

This is a very useful feature. The profiler provides tons of information about how much load your game puts on the processor. With this information, we can get right down into the nitty-gritties and determine exactly how long a script takes to process. Towards the end of the book, though, we will also create a tool to determine how long specific parts of your code take to process.

Script access to the asset pipeline

This is an alright feature. With full access to the pipeline, there is a lot of custom processing that can be done on assets and builds. The full range of possibilities is beyond the scope of this book. However, you can think of it as something that can make tint all of the imported textures slightly blue.

Dark skin

This is entirely a cosmetic feature. Its point and purpose are questionable. However, if a smooth, dark-skinned look is what you desire, this is the feature that you want. There is an option in the editor to change it to the color scheme used in Unity Basic. For this feature, whatever floats your boat goes.

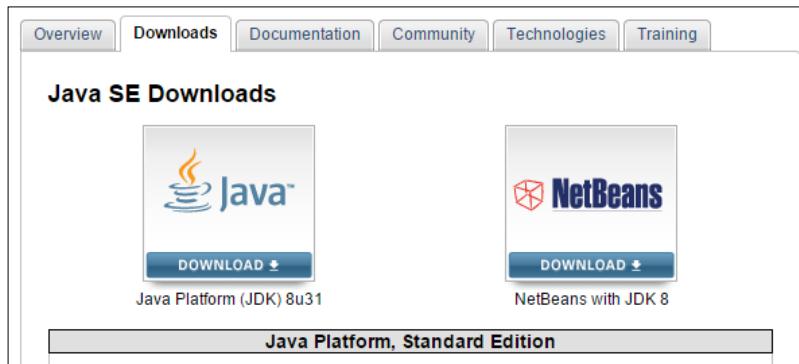
Setting up the development environment

Before we can create the next great game for Android, we need to install a few programs. In order to make the Android SDK work, we will first install the **Java Development Kit (JDK)**. Then we will install the Android SDK. After that, we will install Unity. We then have to install an optional code editor. To make sure everything is set up correctly, we will connect to our devices and take a look at some special strategies if the device is a tricky one. Finally, we will install Unity Remote, a program that will become invaluable in your mobile development.

Installing the JDK

Android's development language of choice is Java; so, to develop for it, we need a copy of the **Java SE Development Kit** on our computer. The process of installing the JDK is given in the following steps:

1. The latest version of the JDK can be downloaded from <http://www.oracle.com/technetwork/java/javase/downloads/index.html>. So open the site in a web browser, and you will be able to see the screen showed in the following screenshot:



2. Select **Java Platform (JDK)** from the available versions and you will be brought to a page that contains the license agreement and allows you to select the type of file you wish to download.
3. Accept the license agreement and select your appropriate Windows version from the list at the bottom. If you are unsure about which version to choose, then **Windows x86** is usually a safe choice.
4. Once the download is completed, run the new installer.
5. After a system scan, click on **Next** two times, the JDK will initialize, and then click on the **Next** button one more time to install the JDK to the default location. It is as good there as anywhere else, so once it is installed, hit the **Close** button.

We have just finished installing the **JDK**. We need this so that our Android development kit will work. Luckily, the installation process for this keystone is short and sweet.

Installing the Android SDK

In order to actually develop and connect to our devices, we need to have installed the Android SDK. Having the SDK installed fulfills two primary requirements. First, it makes sure that we have the bulk of the latest drivers for recognizing devices. Second, we are able to use the **Android Debug Bridge (ADB)**. ADB is the system used for actually connecting to and interacting with a device. The process of installing the Android SDK is given in the following steps:

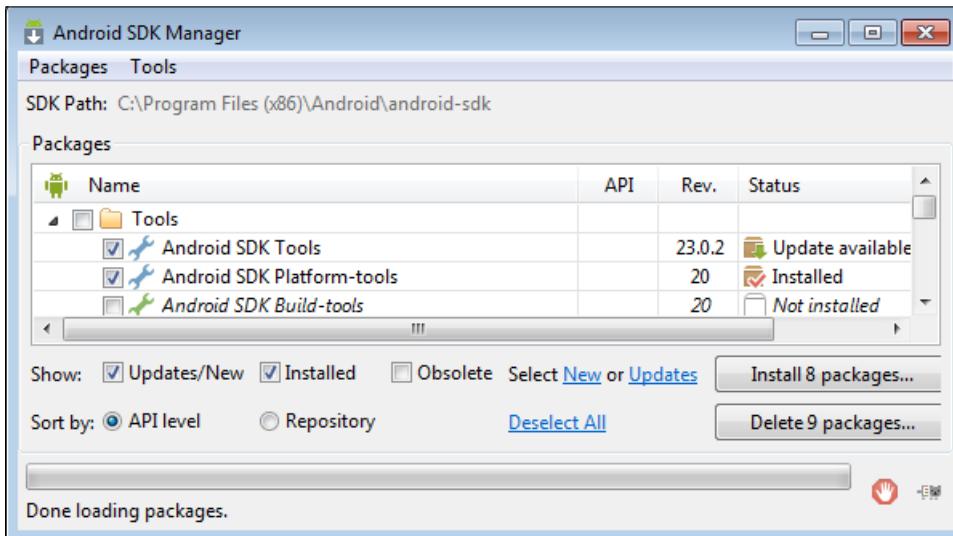
1. The latest version of the Android SDK can be found at <http://developer.android.com/sdk/index.html>, so open a web browser and go to the given site.
2. Once there, scroll to the bottom and find the **SDK Tools Only** section. This is where we can get just the SDK, which we need to make Android games with Unity, without dealing with the fancy fluff of the Android Studio.

3. We need to select the .exe package with **(Recommended)** underneath it (as shown in the following screenshot):

Other Download Options			
SDK Tools Only			
If you prefer to use a different IDE or run the tools from the command line or with build scripts, you can instead download the stand-alone Android SDK Tools. These packages provide the basic SDK tools for app development, without an IDE. Also see the SDK tools release notes .			
Platform	Package	Size	SHA-1 Checksum
	installer_r24.0.2-windows.exe (Recommended)	91428280 bytes	edac14e1541e97d68821fa3a709b4ea8c659e676
Mac OS X	android-sdk_r24.0.2-macosx.zip	139473113 bytes	51269c8336f936fc9b9538f9b9ca236b78fb4e4b
	android-sdk_r24.0.2-linux.tgz	87262823 bytes	3ab5e0ab0db5e7c45de9da7ff525dee6cfa97455
Linux	android-sdk_r24.0.2-linux.tgz	140097024 bytes	b6fd75e8b06b0028c2427e6da7d8a09d8f956a86

4. You will then be sent to a **Terms and Conditions** page. Read it if you prefer, but agree to it to continue. Then hit the **Download** button to start downloading the installer.
5. Once it has finished downloading, start it up.
6. Hit the first **Next** button and the installer will try to find an appropriate version of the JDK. You will come to a page that will notify you about not finding the JDK if you do not have it installed.
7. If you skipped ahead and do not have the JDK installed, hit the **Visit java.oracle.com** button in the middle of the page and go back to the previous section for guidance on installing it. If you do have it, continue with the process.
8. Hitting **Next** again will bring you to a page that will ask you about the person for whom you are installing the SDK .
9. Select **Install for anyone using this computer** because the default install location is easier to get to for later purposes.
10. Hit **Next** twice, followed by **Install** to install the SDK to the default location.
11. Once this is done, hit **Next** and **Finish** to complete the installation of the Android SDK Manager.

12. If Android SDK Manager does not start right away, start it up. Either way, give it a moment to initialize. The SDK Manager makes sure that we have the latest drivers, systems, and tools for developing with the Android platform. However, we have to actually install them first (which can be done from the following screen):



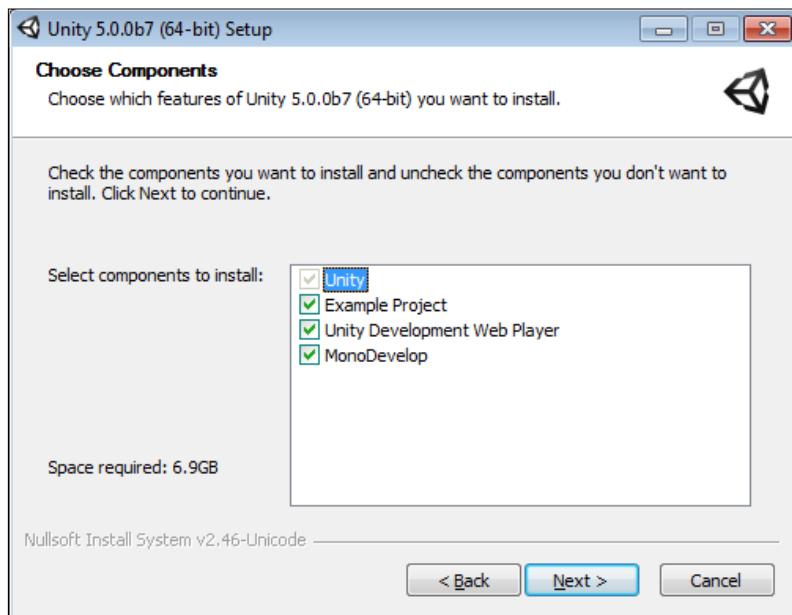
13. By default, the SDK manager should select a number of options to install. If not, select the latest Android API (Android L (API 20) as of the time of writing this book), **Android Support Library** and **Google USB Driver** found in **Extras**. Be absolutely sure that **Android SDK Platform-tools** is selected. This will be very important later. It actually includes the tools that we need to connect to our device.
14. Once everything is selected, hit **Install packages** at the bottom-right corner.
15. The next screen is another set of license agreements. Every time a component is installed or updated through the SDK manager, you have to agree to the license terms before it gets installed. Accept all of the licenses and hit **Install** to start the process.
16. You can now sit back and relax. It takes a while for the components to be downloaded and installed. Once this is all done, you can close it out. We have completed the process, but you should occasionally come back to it. Periodically checking the SDK manager for updates will make sure that you are using the latest tools and APIs.

The installation of the **Android SDK** is now finished. Without it, we would be completely unable to do anything on the Android platform. Aside from the long wait to download and install components, this was a pretty easy installation.

Installing Unity 3D

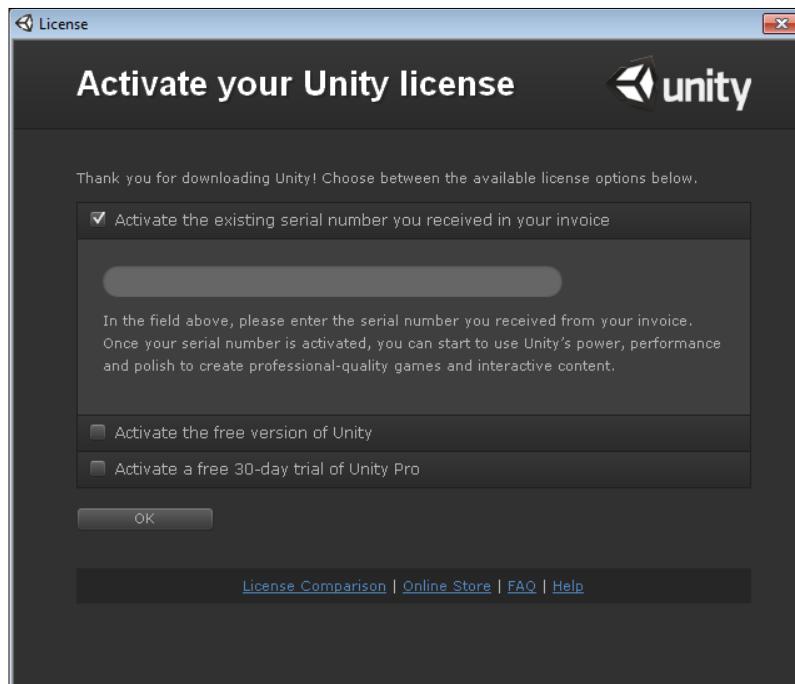
Perhaps the most important part of this whole book, without which none of the rest has meaning, is installing Unity. Perform the following steps to install Unity:

1. The latest version of Unity can be found at <http://www.unity3d.com/unity/download>. As of the time of writing this book, the current version is 5.0.
2. Once it is downloaded, launch the installer and click on **Next** until you reach the **Choose Components** page, as shown in the following screenshot:



3. Here, we are able to select the features of Unity installation. None of these options are actually necessary for following the rest of this book, but they warrant a look since Unity will ask for the components you wish to install every time you update or reinstall it:
 - **Example Project:** This is the current project built by Unity to show off some of its latest features. If you want to jump in early and take a look at what a complete Unity game can look like, leave this checked.

- **Unity Development Web Player:** This is required if you plan on developing browser applications with Unity. As this book is focused on Android development, it is entirely optional. It is, however, a good one to check. You never know when you may need a web demo and since it is entirely free to develop for the web using Unity, there is no harm in having it.
 - **MonoDevelop:** It is a wise choice to leave this option unchecked. There is more detail in the next section, but it will suffice for now to say that it just adds an extra program for script editing that is not nearly as useful as it should be.
4. Once you have selected or deselected your desired options, hit **Next**. If you wish to follow the book exactly, note that we will uncheck **MonoDevelop** and leave the rest checked.
 5. Next is the location of installation. The default location works well, so hit **Install** and wait. This will take a couple of minutes, so sit back, relax, and enjoy your favorite beverage.
 6. Once the installation is complete, the option to run Unity will be displayed. Leave it checked and hit **Finish**. If you have never installed Unity before, you will be presented with a license activation page (as shown in the following screenshot):



7. While Unity does provide a feature-rich, free version, in order to follow the entirety of this book, one is required to make use of some of the Unity Pro features. At <https://store.unity3d.com>, you have the ability to buy a variety of licenses. To follow the whole book, you will at least need to purchase Unity Pro and Android Pro licenses. Once they are purchased, you will receive an e-mail containing your new license key. Enter that in the provided text field.
8. If you are not ready to make a purchase, you have two alternatives. We will go over how to reset your license in the *Building a simple application* section later in the chapter. The alternatives are as follows:
 - The first alternative is that you can check the **Activate the free version of Unity** checkbox. This will allow you to use the free version of Unity. As discussed earlier, there are many reasons to choose this option. The most notable at the moment is cost.
 - Alternatively, you can select the **Activate a free 30-day trial of Unity Pro** option. Unity offers a fully functional, one-time installation and a free 30-day trial of Unity Pro. This trial also includes the Android Pro add-on. Anything produced during the 30 days is completely yours, as if you had purchased a full Unity Pro license. They want you to see how great it is, so you will come back and make a purchase. The downside is that the **Trial Version** watermark will be constantly displayed at the corner of the game. After the 30 days, Unity will revert to the free version. This is a great option, should you choose to wait before making a purchase.
9. Whatever your choice is, hit **OK** once you have made it.
10. The next page simply asks you to log in with your Unity account. This will be the same account that you used to make your purchase. Just fill out the fields and hit **OK**.
11. If you have not yet made a purchase, you can hit **Create Account** and have it ready for when you do make a purchase.
12. The next page is a short survey on your development interests. Fill it out and hit **OK** or scroll straight to the bottom and hit **Not right now**.
13. Finally, there is a thank you page. Hit **Start using Unity**.
14. After a short initialization, the project wizard will open and we can start creating the next great game. However, there is still a bunch of work to do to connect the development device. So for now, hit the **X** button in the top-right corner to close the project wizard. We will cover how to create a new project in the *Building a simple application* section later on.

We just completed installing Unity 3D. The whole book relies on this step. We also had to make a choice about licenses. If you chose to purchase the Pro version, you will be able to follow everything in this book without problems. The alternatives, though, will have a few shortcomings. You will either not have full access to all of the features or be limited to the length of the trial period while making due with a watermark in your games.

The optional code editor

Now a choice has to be made about code editors. Unity comes with a system called **MonoDevelop**. It is similar in many respects to **Visual Studio**. And like Visual Studio, it adds many extra files and much girth to a project, all of which it needs to operate. All this extra girth makes it take an annoying amount of time to start up, before one can actually get to the code.

Technically, you can get away with a plain text editor, as Unity doesn't really care. This book recommends using Notepad++, which is found at <http://notepad-plus-plus.org/download>. It is free to use and it is essentially Notepad with code highlighting. There are several fancy widgets and add-ons for Notepad++ that add even greater functionality to it, but they are not necessary for following this book. If you choose this alternative, installing Notepad++ to the default location will work just fine.

Connecting to a device

Perhaps the most annoying step in working with Android devices is setting up the connection to your computer. Since there are so many different kinds of devices, it can get a little tricky at times just to have the device recognized by your computer.

A simple device connection

The simple device connection method involves changing a few settings and a little work in the command prompt. It may seem a little scary, but if all goes well you will be connected to your device shortly:

1. The first thing you need to do is turn on the phone's **Developer options**. In the latest version of Android, these have been hidden. Go to your phone's settings page and find the **About phone** page.

2. Next, you need to find the **Build number** information slot and tap it several times. At first, it will appear to do nothing, but it will shortly display that you need to press the button a few more times to activate the **Developer options**. The Android team did this so that the average user does not accidentally make changes.
3. Now go back to your settings page and there should be a new **Developer options** page; select it now. This page controls all of the settings you might need to change while developing your applications.
4. The only checkbox we are really concerned with checking right now is **USB debugging**. This allows us to actually detect our device from the development environment.
5. If you are using Kindle, be sure to go into **Security** and turn on **Enable ADB** as well.



There are several warning pop-ups that are associated with turning on these various options. They essentially amount to the same malicious software warnings associated with your computer. Applications with immoral intentions can mess with your system and get to your private information. All these settings need to be turned on if your device is only going to be used for development. However, as the warnings suggest, if malicious applications are a concern, turn them off when you are not developing.

6. Next, open a command prompt on your computer. This can be done most easily by hitting your Windows key, typing cmd.exe, and then hitting *Enter*.
7. We now need to navigate to the ADB commands. If you did not install the SDK to the default location, replace the path in the following commands with the path where you installed it.

If you are running a 32-bit version of Windows and installed the SDK to the default location, type the following in the command prompt:

```
cd c:\program files\android\android-sdk\platform-tools
```

If you are running a 64-bit version, type the following in the command prompt:

```
cd c:\program files (x86)\android\android-sdk\platform-tools
```

8. Now, connect your device to your computer, preferably using the USB cable that came with it.
9. Wait for your computer to finish recognizing the device. There should be a **Device drivers installed** type of message pop-up when it is done.

10. The following command lets us see which devices are currently connected and recognized by the ADB system. Emulated devices will show up as well.

Type the following in the command prompt:

```
adb devices
```

11. After a short pause for processing, the command prompt will display a list of attached devices along with the unique IDs of all the attached devices. If this list now contains your device, congratulations! You have a developer-friendly device. If it is not completely developer-friendly, there is one more thing that you can try before things get tricky.
12. Go to the top of your device and open your system notifications. There should be one that looks like the USB symbol. Selecting it will open the connection settings. There are a few options here and by default Android selects to connect the Android device as a **Media Device**.
13. We need to connect our device as a **Camera**. The reason is the connection method used. Usually, this will allow your computer to connect.

We have completed our first attempt at connecting to our Android devices. For most, this should be all that you need to connect to your device. For some, this process is not quite enough. The next little section covers solutions to resolve the issue for connecting trickier devices.

For trickier devices, there are a few general things that we can try; if these steps fail to connect your device, you may need to do some special research.

1. Start by typing the following commands. These will restart the connection system and display the list of devices again:

```
adb kill-server  
adb start-server  
adb devices
```

2. If you are still not having any luck, try the following commands. These commands force an update and restart the connection system:

```
cd ../tools  
android update adb  
cd ../platform-tools  
adb kill-server  
adb start-server  
adb devices
```

3. If your device is still not showing up, you have one of the most annoying and tricky devices. Check the manufacturer's website for data syncing and management programs. If you have had your device for quite some time, you have probably been prompted to install this more than once. If you have not already done so, install the latest version even if you never plan on using it. The point is to obtain the latest drivers for your device, and this is the easiest way.
4. Restart the connection system again using the first set of commands and cross your fingers!
5. If you are still unable to connect, the best, professional recommendation that can be made is to google for the solution to your problem. Conducting a search for your device's brand with adb at the end should turn up a step-by-step tutorial that is specific to your device in the first couple of results. Another excellent resource for finding out all about the nitty-gritties of Android devices can be found at <http://www.xda-developers.com/>.

Some of the devices that you will encounter while developing will not connect easily. We just covered some quick steps and managed to connect these devices. If we could have covered the processes for every device, we would have. However, the variety of devices is just too large and the manufacturers keep making more.

Unity Remote

Unity Remote is a great application created by the Unity team. It allows developers to connect their Android-powered devices to the Unity Editor and provide mobile inputs for testing. This is a definite must for any aspiring Unity and Android developer. If you are using a non-Amazon device, acquiring Unity Remote is quite easy. At the time of writing this book, it could be found on Google Play at <https://play.google.com/store/apps/details?id=com.unity3d.genericremote>. It is free and does nothing but connects your Android device to the Unity Editor, so the app permissions are negligible. In fact, there are currently two versions of Unity Remote. To connect to Unity 4.5 and later versions, we must use Unity Remote 4.

If, however, you like the ever-growing Amazon market or seek to target Amazon's line of Android devices, adding Unity Remote will become a little trickier. First, you need to download a special Unity Package from the Unity Asset Store. It can be found at <https://www.assetstore.unity3d.com/en/#!/content/18106>. You will need to import the package into a fresh project and build it from there. Import the package by going to the top of Unity, navigate to **Assets | Import Package | Custom Package**, and then navigate to where you saved it. In the next section, we will build a simple application and put it on our device. After you have imported the package, follow along from the step where we open the **Build Settings** window, replacing the simple application with the created APK.

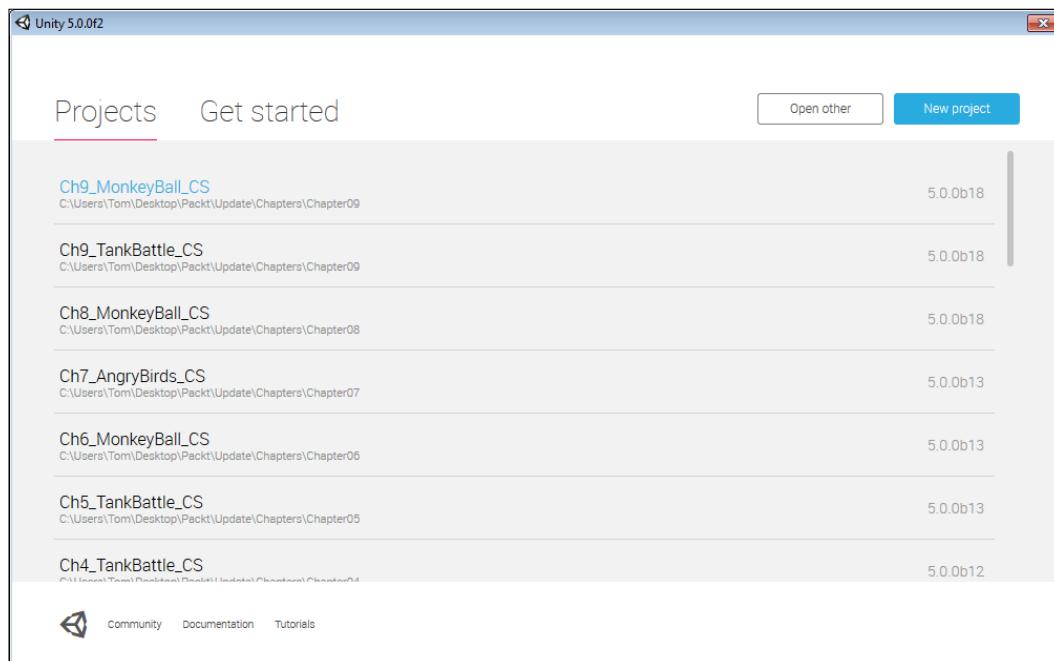
Building a simple application

We are now going to create a simple Hello World application. This will familiarize you with the Unity interface and how to actually put an application on your device.

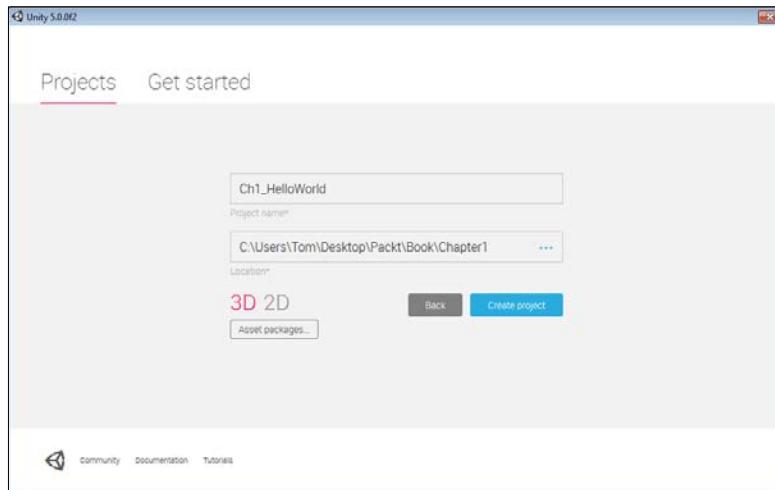
Hello World

To make sure everything is set up properly, we need a simple application to test with and what better to do that with than a Hello World application? To build the application, perform the following steps:

1. The first step is pretty straightforward and simple: start Unity.
2. If you have been following along so far, once this is done you should see a screen resembling the next screenshot. As the tab might suggest, this is the screen through which we open our various projects. Right now, though, we are interested in creating one; so, select **New Project** from the top-right corner and we will do just that:



3. Use the **Project name*** field to give your project a name; Ch1_HelloWorld fits well for a project name. Then use the three dots to the right of the **Location*** field to choose a place on your computer to put the new project. Unity will create a new folder in this location, based on the project name, to store your project and all of its related files:



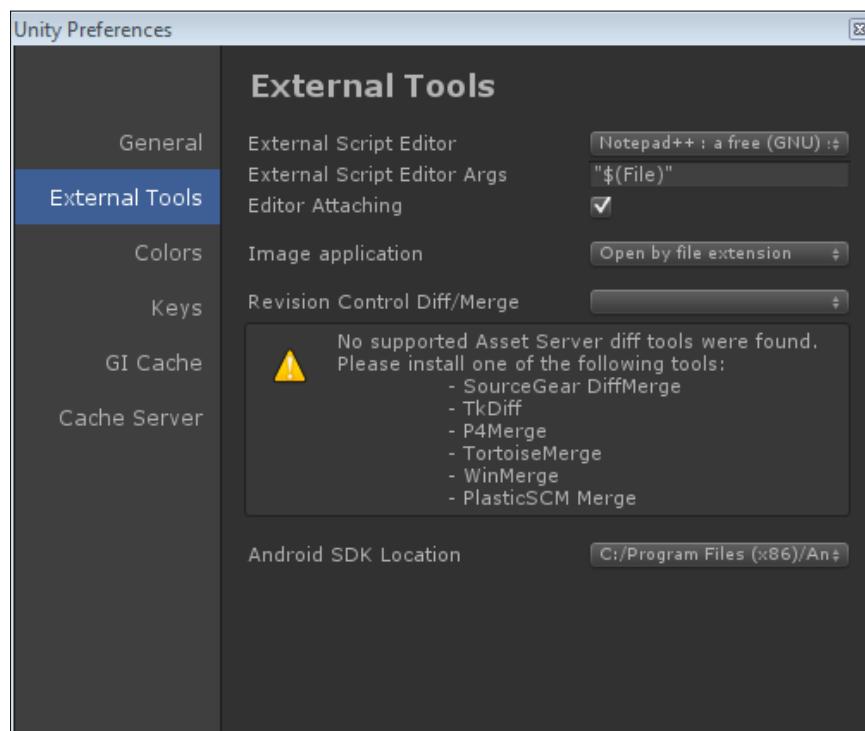
4. For now, we can ignore the **3D** and **2D** buttons. These let us determine the defaults that Unity will use when creating a new scene and importing new assets. We can also ignore the **Asset packages** button. This lets you select from the bits of assets and functionality that is provided by Unity. They are free for you to use in your projects.
5. Hit the **Create Project** button, and Unity will create a brand-new project for us.

The following screenshot shows the windows of the Unity Editor:



6. The default layout of Unity contains a decent spread of windows that are needed to create a game:
 - Starting from the left-hand side, **Hierarchy** contains a list of all the objects that currently exist in our scene. They are organized alphabetically and are grouped under parent objects.
 - Next to this is the **Scene** view. This window allows us to edit and arrange objects in the 3D space. In the top left-hand side, there are two groups of buttons. These affect how you can interact with the **Scene** view.
 - The button on the far left that looks like a hand lets you pan around when you click and drag with the mouse.
 - The next button, the crossed arrows, lets you move objects around. Its behavior and the gizmo it provides will be familiar if you have made use of any modeling programs.
 - The third button changes the gizmo to rotation. It allows you to rotate objects.
 - The fourth button is for scale. It changes the gizmo as well.
 - The fifth button lets you adjust the position and the scale based on the bounding box of the object and its orientation relative to how you are viewing it.
 - The second to last button toggles between **Pivot** and **Center**. This will change the position of the gizmo used by the last three buttons to be either at the pivot point of the selected object, or at the average position point of all the selected objects.
 - The last button toggles between **Local** and **Global**. This changes whether the gizmo is orientated parallel with the world origin or rotated with the selected object.
 - Underneath the **Scene** view is the **Game** view. This is what is currently being rendered by any cameras in the scene. This is what the player will see when playing the game and is used for testing your game. There are three buttons that control the playback of the **Game** view in the upper-middle section of the window.
 - The first is the **Play** button. It toggles the running of the game. If you want to test your game, press this button.
 - The second is the **Pause** button. While playing, pressing this button will pause the whole game, allowing you to take a look at the game's current state.

- The third is the **Step** button. When paused, this button will let you progress through your game one frame at a time.
 - On the right-hand side is the **Inspector** window. This displays information about any object that is currently selected.
 - In the bottom left-hand side is the **Project** window. This displays all of the assets that are currently stored in the project.
 - Behind this is **Console**. It will display debug messages, compile errors, warnings, and runtime errors.
7. At the top, underneath **Help**, is an option called **Manage License....** By selecting this, we are given options to control the license. The button descriptions cover what they do pretty well, so we will not cover them in more detail at this point.
8. The next thing we need to do is connect our optional code editor. At the top, go to **Edit** and then click on **Preferences...**, which will open the following window:



9. By selecting **External Tools** on the left-hand side, we can select other software to manage asset editing.

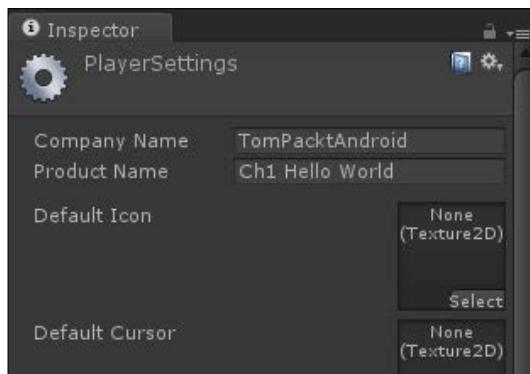
10. If you do not want to use MonoDevelop, select the drop-down list to the right of **External Script Editor** and navigate to the executable of **Notepad++**, or any other code editor of your choice.
11. Your **Image application** option can also be changed here to **Adobe Photoshop** or any other image-editing program that you prefer, in the same way as the script editor.
12. If you installed the Android SDK to the default location, do not worry about it. Otherwise, click on **Browse...** and find the android-sdk folder.
13. Now, for the actual creation of this application, right-click inside your **Project** window.
14. From the new window that pops up, select **Create** and **C# Script** from the menu.
15. Type in a name for the new script (`HelloWorld` will work well) and hit *Enter* twice: once to confirm the name and once to open it.

 Since this is the first chapter, this will be a simple Hello World application. Unity supports C#, JavaScript, and Boo as scripting languages. For consistency, this book will be using C#. If you, instead, wish to use JavaScript for your scripts, copies of all of the projects can be found with the other resources for this book, under a `_JS` suffix for JavaScript.

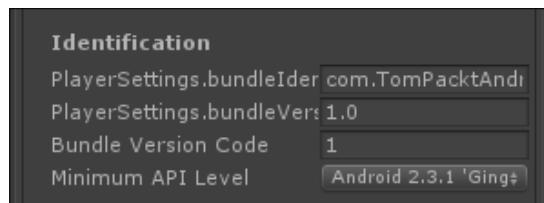
16. Every script that is going to attach to an object extends the functionality of the `MonoBehaviour` class. JavaScript does this automatically, but C# scripts must define it explicitly. However, as you can see from the default code in the script, we do not have to worry about setting this up initially; it is done automatically. Extending the `MonoBehaviour` class lets our scripts access various values of the game object, such as the position, and lets the system automatically call certain functions during specific events in the game, such as the Update cycle and the GUI rendering.
17. For now, we will delete the `Start` and `Update` functions that Unity insists on including in every new script. Replace them with a bit of code that simply renders the words **Hello World** in the top-left corner of the screen; you can now close the script and return to Unity:

```
public void OnGUI() {
    GUILayout.Label("Hello World");
}
```

18. Drag the `HelloWorld` script from the **Project** window and drop it on the **Main Camera** object in the **Hierarchy** window. Congratulations! You have just added your first bit of functionality to an object in Unity.
19. If you select **Main Camera** in **Hierarchy**, then **Inspector** will display all of the components attached to it. At the bottom of the list is your brand-new `HelloWorld` script.
20. Before we can test it, we need to save the scene. To do this, go to **File** at the top and select **Save Scene**. Give it the name `HelloWorld` and hit **Save**. A new icon will appear in your **Project** window, indicating that you have saved the scene.
21. You are now free to hit the **Play** button in the upper-middle section of the editor and witness the magic of Hello World.
22. We now get to build the application. At the top, select **File** and then click on **Build Settings....**
23. By default, the target platform is **PC**. Under **Platform**, select **Android** and hit **Switch Platform** in the bottom-left corner of the **Build Settings** window.
24. Underneath the **Scenes In Build** box, there is a button labeled **Add Current**. Click on it to add our currently opened scene to the build. Only scenes that are in this list and checked will be added to the final build of your game. The scene with the number zero next to it will be the first scene that is loaded when the game starts.
25. There is one last group of things to change before we can hit the **Build** button. Select **Player Settings...** at the bottom of the **Build Settings** window.
26. The **Inspector** window will open **Player Settings** (shown in the following screenshot) for the application. From here, we can change the splash screen, icon, screen orientation, and a handful of other technical options:



27. At the moment, there are only a few options that we care about. At the top, **Company Name** is the name that will appear under the information about the application. **Product Name** is the name that will appear underneath the icon on your Android device. You can largely set these to anything you want, but they do need to be set immediately.
28. The important setting is **Bundle Identifier**, underneath **Other Settings** and **Identification**. This is the unique identifier that singles out your application from all other applications on the device. The format is `com.CompanyName.ProductName`, and it is a good practice to use the same company name across all of your products. For this book, we will be using `com.TomPacktAndBegin.Ch1.HelloWorld` for **Bundle Identifier** and opt to use an extra dot (period) for the organization.



29. Go to **File** and then click on **Save** again.
30. Now you can hit the **Build** button in the **Build Settings** window.
31. Pick a location to save the file, and a file name (`Ch1_HelloWorld.apk` works well). Be sure to remember where it is and hit **Save**.
32. If during the build process Unity complains about where the Android SDK is, select the `android-sdk` folder inside the location where it was installed. The default would be `C:\Program Files\Android\android-sdk` for a 32-bit Windows system and `C:\Program Files (x86)\Android\android-sdk` for a 64-bit Windows system.
33. Once loading is done, which should not be very long, your APK will have been made and we are ready to continue.
34. We are through with Unity for this chapter. You can close it down and open a command prompt.
35. Just as we did when we were connecting our devices, we need to navigate to the `platform-tools` folder in order to connect to our device. If you installed the SDK to the default location, use:
 - For a 32-bit Windows system:

```
cd c:\program files\android\android-sdk\platform-tools
```

- For a 64-bit Windows system:

```
cd c:\program files (x86)\android\android-sdk\platform-tools
```

36. Double-check to make sure that the device is connected and recognized by using the following command:

```
adb devices
```

37. Now we will install the application. This command tells the system to install an application on the connected device. The -r indicates that it should override if an application is found with the same **Bundle Identifier** as the application we are trying to install. This way you can just update your game as you develop, rather than uninstalling before installing the new version each time you need to make an update. The path to the .apk file that you wish to install is shown in quotes as follows:

```
adb install -r "c:\users\tom\desktop\packt\book\ch1_helloworld.apk"
```

38. Replace it with the path to your APK file; capital letters do not matter, but be sure to have all the correct spacing and punctuations.
39. If all goes well, the console will display an upload speed when it has finished pushing your application to the device and a success message when it has finished the installation. The most common causes for errors at this stage are not being in the platform-tools folder when issuing commands and not having the correct path to the .apk file, surrounded by quotes.
40. Once you have received your success message, find the application on your phone and start it up.
41. Now, gaze in wonder at your ability to create Android applications with the power of Unity.

We have created our very first Unity and Android application. Admittedly, it was just a simple Hello World application, but that is how it always starts. This served very well for double-checking the device connection and for learning about the build process without all the clutter from a game.

If you are looking for a further challenge, try changing the icon for the application. It is a fairly simple procedure that you will undoubtedly want to perform as your game develops. How to do this was mentioned earlier in this section, but, as a reminder, take a look at **Player Settings**. Also, you will need to import an image. Take a look under **Assets**, in the menu bar, to know how to do this.

Summary

There were a lot of technical things in this chapter. First, we discussed the benefits and possibilities when using Unity and Android. That was followed by a whole lot of installation; the JDK, the Android SDK, Unity 3D, and Unity Remote. We then figured out how to connect to our devices through the command prompt. Our first application was quick and simple to make. We built it and put it on a device.

In the next chapter, we will create a game, Tic-tac-toe, that is significantly more interactive. We will explore the wonderful world of graphical user interfaces. So not only will we make the game, but we will make it look good too.

2

Looking Good – The Graphical Interface

In the previous chapter, we covered the features of Unity and Android. We also discussed the benefits of using them together. After we finished installing a bunch of software and setting up our devices, we created a simple Hello World application to confirm that everything was connected correctly.

This chapter is all about **Graphical User Interface (GUI)**. We will start by creating a simple Tic-tac-toe game, using the basic pieces of GUI that Unity provides. Following this, we will discuss how we can change the styles of our GUI controls to improve the look of our game. We will also explore some tips and tricks to handle the many different screen sizes of Android devices. Finally, we will learn about a much quicker way, compared to the one covered in the previous chapter, to put our games on the device. With all that said, let's jump in.

In this chapter, we will cover the following topics:

- User preferences
- Buttons, text, and images
- Dynamic GUI positioning
- Build and run

In this chapter, we will be creating a new project in Unity. The first section here will walk you through its creation and setup.

Creating a Tic-tac-toe game

The project for this chapter is a simple Tic-tac-toe style game, similar to what any of us might play on paper. As with anything else, there are several ways in which you can make this game. We are going to use Unity's uGUI system in order to better understand how to create a GUI for any of our other games.

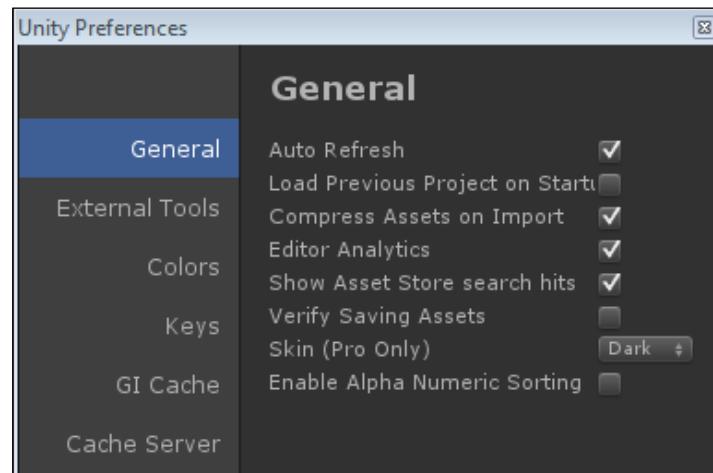
The game board

The basic Tic-tac-toe game involves two players and a 3×3 grid. The players take turns filling squares with Xs and Os. The player who first fills a line of three squares with his or her letter wins the game. If all squares are filled without a player achieving a line of three, the game is a tie. Let's start with the following steps to create our game board:

1. The first thing to do is to create a project for this chapter. So, start up Unity and we will do just that.

If you have been following along so far, Unity should boot up into the last project that was open. This isn't a bad feature, but it can become extremely annoying. Think of it like this: you have been working on a project for a while and it has grown large. Now you need to quickly open something else, but Unity defaults to your huge project. If you wait for it to open before you can work on anything else, it can consume a lot of time.

To change this feature, go to the top of the Unity window and click on **Edit**, followed by **Preferences**. This is the same place where we changed our script editor's preferences. This time, though, we are going to change settings in the **General** tab. The following screenshot shows the options that are present under the **General** tab:



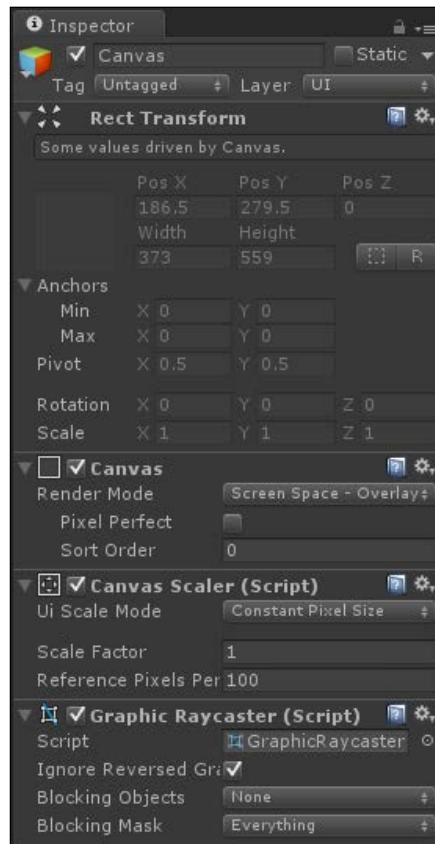
2. At this moment, our primary concern is the **Load Previous Project on Startup** option; however, we will still cover all the options in turn. All the options under the **General** tab are explained in detail as follows:
 - **Auto Refresh:** This is one of the best features of Unity. Because an asset is changed outside of Unity, this option lets Unity automatically detect the change and refresh the asset inside your project.
 - **Load Previous Project on Startup:** This is a great option, and you should make sure that this is unchecked whenever installing Unity. When checked, Unity will immediately open the last project you worked on rather than **Project Wizard**.
 - **Compress Assets on Import:** This is the checkbox for automatically compressing your game assets when they are first imported into Unity.
 - **Editor Analytics:** This checkbox is for Unity's anonymous usage statistics. Leave it checked and the Unity Editor will send information occasionally to the Unity source. It doesn't hurt anything to leave it on and helps the Unity team to make the Unity Editor better; however, it comes down to personal preference.
 - **Show Asset Store search hits:** This setting is only relevant if you plan to use **Asset Store**. The asset store can be a great source of assets and tools for any game; however, since we are not going to use it, its relevance to this book is rather limited. It does what the name suggests. When you search the asset store for something within the Unity Editor, the number of results is displayed based on this checkbox.
 - **Verify Saving Assets:** This is a good one to leave unchecked. If this is on, every time you click on **Save** in Unity, a dialog box will pop up so that you can make sure to save any and all of the assets that have changed since your last save. This option is not so much about your models and textures, but is concerned with Unity's internal files, materials, and prefabs. It's best to leave it off for now.
 - **Skin (Pro Only):** This option only applies to Unity Pro users. It gives the option to switch between the light and dark versions of the Unity Editor. It is purely cosmetic, so go with your gut for this one.
3. With your preferences set, now go to **File** and then select **New Project**.
4. Click on the **Browse...** button to pick a location and name for the new project.
5. We will not be using any of the included packages, so click on **Create** and we can get on with it.

By changing a few simple options, we can save ourselves a lot of trouble later. This may not seem like that big of a deal now for simple projects from this book, but for large and complex projects, not choosing the correct options can cause a lot of hassle for you even if you just want to make a quick switch between projects.

Creating the board

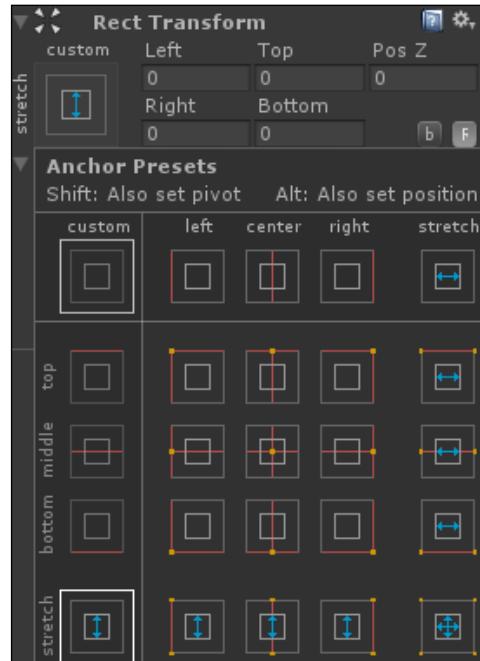
With the new project created, we have a clean slate to create our game. Before we can create the core functionality, we need to set up some structure in our scene for our game to work and our players to interact with:

1. Once Unity finishes initializing the new project, we need to create a new **canvas**. We can do this by navigating to **GameObject | UI | Canvas**. The whole of Unity's uGUI system requires a canvas in order to draw anything on the screen. It has a few key components, as you can see in the following **Inspector** window, which allow it and everything else in your interface to work:



- **Rect Transform:** This is a special type of the normal transform component that you will find on nearly every other object you will use in your games. It keeps track of the object's position on screen, its size, its rotation, the pivot point around which it will rotate, and how it will behave when the screen size changes. By default, the Rect Transform for a canvas is locked to include the whole screen's size.
 - **Canvas:** This component controls how it and the interface elements it controls interact with the camera and your scene. You can change this by adjusting **Render Mode**. The default mode, **Screen Space - Overlay**, means that everything will be drawn on screen and over the top of everything else in the scene. The **Screen Space - Camera** mode will draw everything a specific distance away from the camera. This allows your interface to be affected by the perspective nature of the camera, but any models that might be closer to the camera will appear in front of it. The **World Space** mode ensures that the canvas and elements it controls are drawn in the world just like any of the models in your scene.
 - **Graphics Raycaster:** This is the component that lets you actually interact with and click on your various interface elements.
2. When you added the canvas, an extra object called **EventSystem** was also created. This is what allows our buttons and other interface elements to interact with our scripts. If you ever accidentally delete it, you can recreate it by going to the top of Unity and navigating to **GameObject | UI | EventSystem**.
 3. Next, we need to adjust the way the Unity Editor will display our game so that we can easily make our game board. To do this, switch to the **Game** view by clicking on its tab at the top of the **Scene** view.
 4. Then, click on the button that says **Free Aspect** and select the option near the bottom: **3 : 2 Landscape (3 : 2)**. Most of the mobile devices your games will be played on will use a screen that approximates this ratio. The rest will not see any distortion in your game.
 5. To allow our game to adjust to the various resolutions, we need to add a new component to our canvas object. With it selected in the **Hierarchy** panel, click on **Add Component** in the **Inspector** panel and navigate to **Layout | Canvas Scaler**. The selected component allows us to work from a base screen resolution, letting it automatically scale our GUI as the devices change.
 6. To select a base resolution, select **Scale With Screen Size** from the **Ui Scale Mode** drop-down list.

7. Next, let's put 960 for X and 640 for Y. It is better to work from a larger resolution than a smaller one. If your resolution is too small, all your GUI elements will look fuzzy when they are scaled up for high-resolution devices.
8. To keep things organized, we need to create three empty GameObjects. Go back to the top of Unity and select **Create Empty** three times under **GameObject**.
9. In the **Hierarchy** tab, click and drag them to our canvas to make them the canvas's children.
10. To make each of them usable for organizing our GUI elements, we need to add the Rect Transform component. Find it by navigating to **Add Component | Layout | Rect Transform** in **Inspector** for each.
11. To rename them, click on their name at the top of the **Inspector** and type in a new name. Name one **Board**, another **Buttons**, and the last one **Squares**.
12. Next, make **Buttons** and **Squares** children of **Board**. The **Buttons** element will hold all the pieces of our game board that are clickable while **Squares** will hold the squares that have already been selected.
13. To keep the **Board** element at the same place as the devices change, we need to change the way it anchors to its parent. Click on the box with a red cross and a yellow dot in the center at the top right of **Rect Transform** to expand the **Anchor Presets** menu:



14. Each of these options affects which corner of the parent the element will stick to as the screen changes size. We want to select the bottom-right option with four arrows, one in each direction. This will make it stretch with the parent element.
15. Make the same change to Buttons and squares, as well.
16. Set **Left**, **Top**, **Right**, and **Bottom** of each of these objects to 0. Also, make sure that **Rotation** is set to 0 and **Scale** is set to 1. Otherwise, our interface may be scaled oddly when we work or play on it.
17. Next, we need to change the anchor point of the board. If **Anchor** is not expanded, click on the little triangle on the left-hand side to expand it. Either way, the **Max X** value needs to be set to 0.667 so that our board will be a square and cover the left two-thirds of our screen.

This game board is the base around which the rest of our project will be created. Without it, the game won't be playable. The game squares use it to draw themselves on screen and anchor themselves to relevant places. Later, when we create menus, is needed to make sure that a player only sees what we need he or she to be interacting with at that moment.

Game squares

Now that we have our base game board in place, we need the actual game squares. Without them, it is going to be kind of hard to play the game. We need to create nine buttons for the player to click on, nine images for the background of the selected squares, and nine texts to display which person controls the squares. To create and set them up, perform these steps:

1. Navigate to **Game Object** | **UI** just like we did for the canvas, but this time select **Button**, **Image**, and **Text** to create everything we need.
2. Each of the image objects needs one of the text objects as a child. Then, all the images must be children of the **Squares** object and the buttons must be children of the **Buttons** object.
3. All the buttons and images need a number in their name so that we can keep them organized. Name the buttons `Button0` through `Button8` and the images `Square0` through `Square8`.
4. The next step is to lay out our board so that we can keep things organized and in sync with our programming. We need to set each numbered set specifically. But first, pick the crossed arrows from the bottom-right corner of **Anchor Presets** for all of them and ensure that their **Left**, **Top**, **Right**, and **Bottom** values are set to 0.

5. To set each of our buttons and squares at the right place, just match the numbers to the following table. The result will be that all the squares will be in order, starting at the top left and ending at the bottom right:

Square	Min X	Min Y	Max X	Max Y
0	0	0.67	0.33	1
1	0.33	0.67	0.67	1
2	0.67	0.67	1	1
3	0	0.33	0.33	0.67
4	0.33	0.33	0.67	0.67
5	0.67	0.33	1	0.67
6	0	0	0.33	0.33
7	0.33	0	0.67	0.33
8	0.67	0	1	0.33

6. The last thing we need to add is an indicator to show whose turn it is. Create another **Text** object just like we did before and rename it `Turn Indicator`.
7. After you make sure that the **Left**, **Top**, **Right**, and **Bottom** values are set to 0 again, set **Anchor Point Preset** to the blue arrows once more.
8. Finally, set **Min X** under **Anchor** to 0.67.
9. We now have everything we need to play the basic game of Tic-tac-toe. To check it out, select the **Squares** object and uncheck the box in the top-right corner to turn it off. When you hit play now, you should be able to see your whole game board and click on the buttons. You can even use Unity Remote to test it with the touch settings. If you have not already done so, it would be a good idea to save the scene before continuing.

The game squares are the last piece to set up our initial game. It almost looks like a playable game now. We just need to add a few scripts and we will be able to play all the games of Tic-tac-toe we could ever desire.

Controlling the game

Having a game board is one of the most important parts of creating any game. However, it doesn't do us any good if we can't control what happens when its various buttons are pressed. Let's create some scripts and write some code to fix this now:

1. Create two new scripts in the **Project** panel, just as we did for the *Hello World* project in the previous chapter. Name the new scripts `TicTacToeControl` and `SquareState`. Open them and clear out the default functions, just as we did in *Chapter 1, Saying Hello to Unity and Android*.

2. The `squareState` script will hold the possible states of each square of our game board. To do this, clear absolutely everything out of the script, including the `using UnityEngine` line and the `public class SquareState` line, so that we can replace them with a simple enumeration. An enumeration is just a list of potential values. This one is concerned with the player who controls the square. It will allow us to keep track of whether X is controlling it, O is controlling it, or if it is clear. The `clear` statement becomes the first and therefore, the default state:

```
public enum SquareState {  
    Clear,  
    Xcontrol,  
    Ocontrol  
}
```

3. In our other script, `TicTacToeControl`, we need to start by adding an extra line at the very beginning, right under `using UnityEngine`. This line lets our code interact with the various GUI elements, and most importantly, with this game, allowing us to change the text of who controls a square and whose turn it is:

```
using UnityEngine.UI;
```

4. Next, we need two variables that will largely control the flow of the game. They need to be added in place of the two default functions. The first defines our game board. It is an array of nine squares to keep track of who owns what. The second keeps track of whose turn it is. When the Boolean is `true`, the X player gets a turn. When the Boolean is `false`, the O player gets a turn:

```
public SquareState[] board = new SquareState[9];  
public bool xTurn = true;
```

5. The next variable will let us change the text on screen for whose turn it is:

```
public Text turnIndicatorLandscape;
```

6. These three variables will give us access to all the GUI objects we set up in the last section, allowing us to change the image and text based on who owns the square. We can also turn the buttons and squares on and off as they are clicked. All of them are marked with **Landscape** so that we will be able to keep them straight later, when we have a second board for the **Portrait** orientation of devices:

```
public GameObject[] buttonsLandscape;  
public Image[] squaresLandscape;  
public Text[] squareTextsPortrait;
```

7. The last two variables, for now, will give us access to the images we need to change the backgrounds of:

```
public Sprite oImage;  
public Sprite xImage;
```

8. Our first function for this script will be called every time a button is clicked. It receives the number of buttons clicked, and the first thing it does is turn the button off and the square on:

```
public void ButtonClick(int squareIndex) {  
    buttonsLandscape[squareIndex].SetActive(false);  
    squaresLandscape[squareIndex].gameObject.SetActive(true);
```

9. Next, the function checks the Boolean we created earlier to see whose turn it is. If it is the X player's turn, the square is set to use the appropriate image and text, indicating that their control is set. It then marks on the script's internal board that controls the square before finally switching to the O player's turn:

```
if(xTurn) {  
    squaresLandscape[squareIndex].sprite = xImage;  
    squareTextsLandscape[squareIndex].text = "X";  
  
    board[squareIndex] = SquareState.XControl;  
    xTurn = false;  
    turnIndicatorLandscape.text = "O's Turn";  
}
```

10. This next block of code does the same thing as the previous one, except it marks control for the O player and changes the turn to the X player:

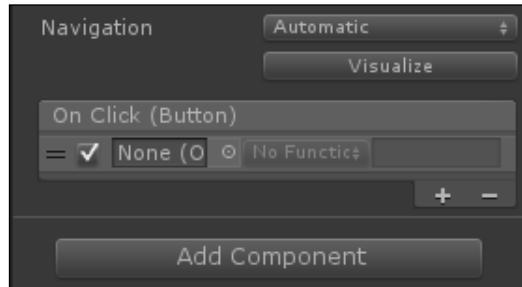
```
else {  
    squaresLandscape[squareIndex].sprite = oImage;  
    squareTextsLandscape[squareIndex].text = "O";  
  
    board[squareIndex] = SquareState.OControl;  
    xTurn = true;  
    turnIndicatorLandscape.text = "X's Turn";  
}
```

11. That is it for the code right now. Next, we need to return to the Unity Editor and set up our new script in the scene. You can do this by creating another empty GameObject and renaming it **GameControl**.

12. Add our **TicTacToeControl** script to it by dragging the script from the **Project** panel and dropping it in the **Inspector** panel when the object is selected.

13. We now need to attach all the object references our script needs in order to actually work. We don't need to touch the **Board** or **XTurn** slots in the **Inspector** panel, but the **Turn Indicator** object does need to be dragged from the **Hierarchy** tab to the **Turn Indicator Landscape** slot in the **Inspector** panel.
14. Next, expand the **Buttons Landscape**, **Squares Landscape**, and **Square Texts Landscape** settings and set each **Size** slot to 9.
15. To each of the new slots, we need to drag the relevant object from the **Hierarchy** tab. The **Element 0** object under **Buttons Landscape** gets **Button0**, **Element 1** gets **Button1**, and so on. Do this for all the buttons, images, and texts. Ensure that you put them in the right order or else our script will appear confusing as it changes things when the player is playing.
16. Next, we need a few images. If you have not already done so, import the starting assets for this chapter by going to the top of Unity, by navigating to **Assets | Import New Asset**, and selecting the files to import them. You will need to navigate to and select each one at a time. We have **Onormal** and **Xnormal** for indicating control of the square. The **ButtonNormal** image is used when the button is just sitting there and **ButtonActive** is used when the player touches the button. The **Title** field is going to be used for our main menu a little bit later.
17. In order to use any of these images in our game, we need to change their import settings. Select each of them in turn and find the **Texture Type** dropdown in the **Inspector** panel. We need to change them from **Texture** to **Sprite (2D \ uGUI)**. We can leave the rest of the settings at their defaults. The **Sprite Mode** option is used if we have a sprite sheet with multiple elements in one image. The **Packing Tag** option is used for grouping and finding sprites in the sheet. The **Pixels To Units** option affects the size of the sprite when it is rendered in world space. The **Pivot** option simply changes the point around which the image will rotate.
18. For the four square images, we can click on **Sprite Editor** to change how the border appears when they are rendered. When clicked, a new window opens that shows our image with some green lines at the edges and some information about it in the lower-right. We can drag these green lines to change the **Border** property. Anything outside the green lines will not be stretched with the image as it fills spaces that are larger than it. A setting around 13 for each side will keep our whole border from stretching.
19. Once you make any changes, ensure that you hit the **Apply** button to commit them.
20. Next, select the **GameControl** object once more and drag the **ONormal** image to the **OImage** slot and the **XNormal** image to the **XImage** slot.

21. Each of the buttons need to be connected to the script. To do this, select each of them from **Hierarchy** in turn and click on the plus sign at the bottom-right corner of their **Inspector**:



22. We then need to click on that little circle to the left of **No Function** and select **GameControl** from the list in the new window.
23. Now, navigate to **No Function** | **TicTacToeControl** | **ButtonClick (int)** to connect the function in our code to the button.
24. Finally, for each of the buttons, put the number of the button in the number slot to the right of the function list.
25. To keep everything organized, rename your **Canvas** object **GameBoard_Landscape**.
26. Before we can test it out, be sure that the **Squares** object is turned on by checking the box in the top-left corner of its **Inspector**. Also, uncheck the box of each of its image children.



This may not look like the best game in the world, but it is playable. We have buttons that call functions in our scripts. The turn indicator changes as we play. Also, each square indicates who controls it after they are selected. With a little more work, this game could look and work great.

Messing with fonts

Now that we have a basic working game, we need to make it look a little better. We are going to add our button images and pick some new font sizes and colors to make everything more readable:

1. Let's start with the buttons. Select one of the Button elements and you will see in the **Inspector** that it is made of an **Image (Script)** component and a **Button (Script)** component. The first component controls how the GUI element will appear when it just sits there. The second controls how it changes when a player interacts with it and what bit of functionality this triggers:
 - **Source Image:** This is the base image that is displayed when the element just sits there and is untouched by the player.
 - **Color:** This controls the tinting and fading of the image that is being used.
 - **Material:** This lets you use a texture or shader that might otherwise be used on 3D models.
 - **Image Type:** This determines how the image will be stretched to fill the available space. Usually, it will be set to **Sliced**, which is for images that use a border and can be optionally filled with a color based on the **Fill Center** checkbox. Otherwise, it will often be set to **Simple**, for example, when you are using a normal image and can prevent the **Preserve Aspect** box from being stretched by odd sized Rect Transforms.
 - **Interactable:** This simply toggles whether or not the player is able to click on the button and trigger functionality.
 - **Transition:** This changes how the button will react as the player interacts with it. **ColorTint** causes the button to change color as it is interacted with. **SpriteSwap** will change the image when it is interacted with. **Animation** will let you define more complex animation sequences for the transitions between states.
 - The **Target Graphic** is a reference to the base image used for drawing the button on screen.

- The **Normal** slot, **Highlighted** slot, **Pressed** slot, and **Disabled** slot define the effects or images to use when the button is not being interacted with or is moused over, or the player clicks on it and the button has been turned off.
2. For each of our buttons, we need to drag our **ButtonNormal** image from our **Project** panel to the **Source Image** slot.
 3. Next, click on the white box to the right of the **Color** slot to open the color picker. To stop our buttons from being faded, we need to move the **A** slider all the way to the right or set the box to 255.
 4. We want to change images when our buttons are pressed, so change the **Transition** to **SpriteSwap**.
 5. Mobile devices have almost no way of hovering over GUI elements, so we do not need to worry about the **Highlighted** state. However, we do want to add our **ButtonActive** image to the **Pressed Sprite** slot so that it will switch when the player touches the button.
 6. The button squares should be blank until someone clicks on them, so we need to get rid of the text element. The easiest way to do this is to select each one under the button and delete it.
 7. Next, we need to change the **Text** child of each of the image elements. It is the **Text (Script)** component that allows us to control how text is drawn on screen.
 - **Text:** This is the area where we can change text that will be drawn on screen.
 - **Font:** This allows us to pick any font file that is in our project to use for the text.
 - **Font Style:** This will let you adjust the bold and italic nature of the text.
 - **Font Size:** This is the size of the text. This is just like picking a font size in your favorite word processor.
 - **Line Spacing:** This is the distance between each line of text.
 - **Rich Text:** This will let you use a few special HTML style tags to affect only part of the text with a color, italics, and so on.
 - **Alignment:** This changes the location where the text will be centered in the box. The first three boxes adjust the horizontal position. The second three change the vertical position.

- **Horizontal Overflow / Vertical Overflow:** These adjust whether the text can be drawn outside the box, wrapped to a new line, or clipped off.
 - **Best Fit:** This will automatically adjust the size of the text to fit a dynamically size-changing element, within a **Min** and **Max** value.
 - **Color/Material:** These change the color and texture of the text as and when it is drawn.
 - **Shadow (Script):** This component adds a drop shadow to the text, just like what you might add in Photoshop.
8. For each of our text elements, we need to use a **Font Size** of 120 and the **Alignment** should be centered.
 9. For the **Turn Indicator** text element, we also need to use a **Font Size** of 120 and it also needs to be centered.
 10. The last thing to do is to change the **Color** of the text elements to a dark gray so that we can easily see it against the color of our buttons:



Now, our board works and looks good, too. Try taking a stab at adding your own images for the buttons. You will need two images, one for when the button sits there and one for when the button is pressed. Also, the default Arial font is boring. Find a new font to use for your game; you can import it just like any other game asset.

Rotating devices

If you have been testing your game so far, you have probably noticed that the game only looks good when we hold the device in the landscape mode. When it is held in the portrait mode, everything becomes squished as the squares and turn indicator try to share the little amount of horizontal space that is available. As we have already set up our game board in one layout mode, it becomes a fairly simple matter to duplicate it for the other mode. However, it does require duplicating a good portion of our code to make it all work properly:

1. To make a copy of our game board, right-click on it and select **Duplicate** from the new menu. Rename the duplicate game board **GameBoard_Portrait**. This will be the board used when our player's device is in the portrait mode. To see our changes while we are making them, turn off the landscape game board and select **3:2 Portrait (2:3)** from the drop-down list at the top left of the **Game** window.
2. Select the **Board** object that is a child of **GameBoard_Portrait**. In its **Inspector** panel, we need to change the anchors to use the top two-thirds of the screen rather than the left two-thirds. The values of **0** for **Min X**, **0 . 33** for **Min Y**, and **1** for both **Max X** and **Max Y** will make this happen.
3. Next, **Turn Indicator** needs to be selected and moved to the bottom third of the screen. Values of **0** for **Min X** and **Min Y**, **1** for **Max X**, and **0 . 33** for **Max Y** will work well here.
4. Now that we have our second board set up, we need to make a place for it in our code. So, open the **TicTacToeControl1** script and scroll to the top so that we can start with some new variables.
5. The first variable we are going to add will give us access to the turn indicator for the portrait mode of our screen:

```
public Text turnIndicatorPortrait;
```

6. The next three variables will keep track of the buttons, square images, and owner text information. These are just like the three lists we created earlier to keep track of the board while it is in the landscape mode:

```
public GameObject[] buttonsPortrait;
public Image[] squaresPortrait;
public Text[] squareTextsPortrait;
```

7. The last two variables we are going to add to the top of our script here are for keeping track of the two canvas objects that actually draw our game boards. We need these so that we can switch between them as the user turns their device around:

```
public GameObject gameBoardGroupLandscape;  
public GameObject gameBoardGroupPortrait;
```

8. Next, we need to update a few of our functions so that they make changes to both boards and not just the landscape board. These first two lines turn the portrait board's buttons off and the squares on when the player clicks on them. They need to go at the beginning of our `ButtonClick` function. Put them right after the two lines where we use `SetActive` on the buttons and squares for the landscape set:

```
buttonsPortrait [squareIndex].SetActive (false);  
squaresPortrait [squareIndex].gameObject.SetActive (true);
```

9. These two lines change the image and text for the controlling square in favor of the X player for the **Portrait** set. They go inside the `if` statement of our `ButtonClick` function, right after the two lines that do the same thing for the landscape set:

```
squaresPortrait [squareIndex].sprite = xImage;  
squareTextsPortrait [squareIndex].text = "X";
```

10. This line goes at the end of that same `if` statement and changes the **Portrait** set's turn indicator text:

```
turnIndicatorPortrait.text = "O's Turn";
```

11. The next two lines change image and text in favor of the O player. They go after the same lines for the **Landscape** set, inside of the `else` statement of our `ButtonClick` function:

```
squaresPortrait [squareIndex].sprite = oImage;  
squareTextsPortrait [squareIndex].text = "O";
```

12. This is the last line we need to add to our `ButtonClick` function; it needs to be put at the end of the `else` statement. It simply changes the text indicating whose turn it is:

```
turnIndicatorPortrait.text = "X's Turn";
```

13. Next, we need to create a new function to control the changing of our game boards when the player changes the orientation of their device. We will start by defining the `Update` function. This is a special function called by Unity for every single frame. It will allow us to check for a change in orientation for every frame:

```
public void Update() {
```

14. The function begins with an `if` statement that uses `Input.deviceOrientation` to find out how the player's device is currently being held. It compares the finding to the `LandscapeLeft` orientation to see whether the device is begin held sideways, with the home button on the left side. If the result is true, the **Portrait** set of GUI elements are turned off while the **Landscape** set is turned on:

```
if (Input.deviceOrientation == DeviceOrientation.LandscapeLeft) {  
    gameBoardGroupPortrait.SetActive(false);  
    gameBoardGroupLandscape.SetActive(true);  
}
```

15. The next `else if` statement checks for a `Portrait` orientation if the home button is down. It turns **Portrait** on and the **Landscape** set off, if `true`:

```
else if (Input.deviceOrientation == DeviceOrientation.Portrait) {  
    gameBoardGroupPortrait.SetActive(true);  
    gameBoardGroupLandscape.SetActive(false);  
}
```

16. This `else if` statement is checking `LandscapeRight` when the home button is on the right side:

```
else if (Input.deviceOrientation == DeviceOrientation.  
LandscapeRight) {  
    gameBoardGroupPortrait.SetActive(false);  
    gameBoardGroupLandscape.SetActive(true);  
}
```

17. Finally, we check the `PortraitUpsideDown` orientation, which is when the home button is at the top of the device. Don't forget the extra bracket to close off and end the function:

```
else if (Input.deviceOrientation == DeviceOrientation.  
PortraitUpsideDown) {  
    gameBoardGroupPortrait.SetActive(true);  
    gameBoardGroupLandscape.SetActive(false);  
}  
}
```

18. We now need to return to Unity and select our **GameControl** object so that we can set up our new **Inspector** properties.

19. Drag and drop the various pieces from the portrait game board in **Hierarchy** to the relevant slot in **Inspector**, **Turn Indicator** to the **Turn Indicator Portrait** slot, the buttons to the **Buttons Portrait** list in order, the squares to **Squares Portrait**, and their text children to the **Square Texts Portrait**.
20. Finally, drop the **GameBoard_Portrait** object in the **Game Board Group Portrait** slot.



We should now be able to play our game and see the board switch when we change the orientation of our device. You will have to either build your project on your device or connect using Unity Remote because the Editor itself and your computer simply don't have a device orientation like your mobile device. Be sure to set the display mode of your **Game** window to **Remote** in the top-left corner so that it will update along with your device while using Unity Remote.

Menus and victory

Our game is nearly complete. The last things we need are as follows:

- An opening menu where players can start a new game
- A bit of code for checking whether anybody has won the game
- A game over menu for displaying who won the game

Setting up the elements

Our two new menus will be quite simple when compared to the game board. The opening menu will consist of our game's title graphic and a single button, while the game over menu will have a text element to display the victory message and a button to go back to the main menu. Let's perform the following steps to set up the elements:

1. Let's start with the opening menu by creating a new **Canvas**, just like we did before, and rename it `OpeningMenu`. This will allow us to keep it separate from the other screens we have created.
2. Next, the menu needs an **Image** element and a **Button** element as children.
3. To make everything easier to work with, turn off the game boards with the checkbox at the top of their **Inspector** windows.
4. For our image object, we can drag our **Title** image to the **Source Image** slot.
5. For the image's **Rect Transform**, we need to set the **Pos X** and **Pos Y** values to 0.
6. We also need to adjust the **Width** and **Height**. We are going to match the dimensions of the original image so that it will not be stretched. Put a value of 320 for **Width** and 160 for **Height**.
7. To move the image to the top half of the screen, put a 0 in the **Pivot Y** slot. This changes where the position is based for the image.
8. For the button's **Rect Transform**, we again need the value of 0 for both **Pos X** and **Pos Y**.
9. We again need a value of 320 for the **Width**, but this time we want a value of 100 for the **Height**.
10. To move it to the bottom half of the screen, we need a value of 1 in the **Pivot Y** slot.
11. Next up is to set the images for the button, just like we did earlier for the game board. Put the `ButtonNormal` image in the **Source Image** slot. Change **Transition** to **SpriteSwap** and put the `ButtonActive` image in the **Pressed Sprite** slot. Do not forget to change **Color** to have an **A** value of 255 in the color picker so that our button is not partially faded.
12. Finally, for this menu to change the button text, expand **Button** in the **Hierarchy** and select the **Text** child object.

13. Right underneath **Text** in the **Inspector** panel for this object is a text field where we can change the text displayed on the button. A value of **New Game** here will work well. Also, change **Font Size** to 45 so that we can actually read it.



14. Next, we need to create the game over menu. So, turn off our opening menu and create a new canvas for our game over menu. Rename it **GameOverMenu** so that we can continue to be organized.
15. For this menu, we need a **Text** element and a **Button** element as its children.
16. We will set this one up in an almost identical way to the previous one. Both the text and button need values of 0 for the **Pos X** and **Pos Y** slots, with a value of 320 for **Width**.
17. The text will use a **Height** of 160 and a **Pivot Y** of 0. We also need to set its **Font Size** to 80. You can change the default text, but it will be overwritten by our code anyway.
18. To center our text in the menu, select the middle buttons from the two sets next to the **Alignment** property.
19. The button will use a **Height** of 100 and a **Pivot Y** of 1.
20. Also, be sure you set the **Source Image**, **Color**, **Transition**, and **Pressed Sprite** to the proper images and settings.

21. The last thing to set is the button's text child. Set the default text to **Main Menu** and give it a **Font Size** of 45.



That is it for setting up our menus. We have all the screens we need to allow the player to interact with our game. The only problem is we don't have any of the functionality to make them actually do anything.

Adding the code

To make our game board buttons work, we had to create a function in our script they could reference and call when they are touched. The main menu's button will start a new game, while the game over menu's button will change screens to the main menu. We will also need to create a little bit of code to clear out and reset the game board when a new game starts. If we don't, it will be impossible for the player to play more than one round before being required to restart the whole app if they want to play again.

1. Open the `TicTacToeControl` script so that we can make some more changes to it.
2. We will start with the addition of three variables at the top of the script. The first two will keep track of the two new menus, allowing us to turn them on and off as per our need. The third is for the text object in the game over screen that will give us the ability to put up a message based on the result of the game.
3. Next, we need to create a new function. The `NewGame` function will be called by the button in the main menu. Its purpose is to reset the board so that we can continue to play without having to reset the whole application:

```
public void NewGame() {
```

4. The function starts by setting the game to start on the X player's turn. It then creates a new array of `SquareStates`, which effectively wipes out the old game board. It then sets the turn indicators for both the **Landscape** and **Portrait** sets of controls:

```
xTurn = true;
board = new SquareState[9];
turnIndicatorLandscape.text = "X's Turn";
turnIndicatorPortrait.text = "X's Turn";
```

5. We next loop through the nine buttons and squares for both the **Portrait** and **Landscape** controls. All the buttons are turned on and the squares are turned off using `SetActive`, which is the same as clicking on the little checkbox at the top-left corner of the **Inspector** panel:

```
for(int i=0;i<9;i++) {
    buttonsPortrait[i].SetActive(true);
    squaresPortrait[i].gameObject.SetActive(false);

    buttonsLandscape[i].SetActive(true);
    squaresLandscape[i].gameObject.SetActive(false);
}
```

6. The last three lines of code control which screens are visible when we change over to the game board. By default, it chooses to turn on the **Landscape** board and makes sure that the **Portrait** board is turned off. It then turns off the main menu. Don't forget the last curly bracket to close off the function:

```
gameBoardGroupPortrait.SetActive(false);
gameBoardGroupLandscape.SetActive(true);
mainMenuGroup.SetActive(false);
}
```

7. Next, we need to add a single line of code to the end of the `ButtonClick` function. It is a simple call to check whether anyone has won the game after the buttons and squares have been dealt with:

```
CheckVictory();
```

8. The `CheckVictory` function runs through the possible combinations for victory in the game. If it finds a run of three matching squares, the `SetWinner` function will be called and the current game will end:

```
public void CheckVictory() {
```

9. A victory in this game is a run of three matching squares. We start by checking the column that is marked by our loop. If the first square is not `Clear`, compare it to the square below; if they match, check it against the square below that. Our board is stored as a list but drawn as a grid, so we have to add three to go down a square. The `else if` statement follows with checks of each row. By multiplying our loop value by three, we will skip down a row of each loop. We'll again compare the square to `SquareState.Clear`, then to the square to its right, and finally, with the two squares to its right. If either set of conditions is correct, we'll send the first square in the set to another function to change our game screen:

```
for(int i=0;i<3;i++) {  
    if(board[i] != SquareState.Clear && board[i] == board[i + 3] &&  
    board[i] == board[i + 6]) {  
        SetWinner(board[i]);  
        return;  
    }  
    else if(board[i * 3] != SquareState.Clear && board[i * 3] ==  
    board[(i * 3) + 1] && board[i * 3] == board[(i * 3) + 2]) {  
        SetWinner(board[i * 3]);  
        return;  
    }  
}
```

10. The following code snippet is largely the same as the `if` statements we just saw. However, these lines of code check the diagonals. If the conditions are `true`, again send out to the other function to change the game screen. You have probably also noticed the returns after the function calls. If we have found a winner at any point, there is no need to check any more of the board. So, we'll exit the `CheckVictory` function early:

```
if(board[0] != SquareState.Clear && board[0] == board[4] &&  
board[0] == board[8]) {  
    SetWinner(board[0]);  
    return;  
}  
else if(board[2] != SquareState.Clear && board[2] == board[4] &&  
board[2] == board[6]) {  
    SetWinner(board[2]);  
    return;  
}
```

11. This is the last little bit for our `CheckVictory` function. If no one has won the game, as determined by the previous parts of this function, we have to check for a tie. This is done by checking all the squares of the game board. If any one of them is `Clear`, the game has yet to finish and we exit the function. But, if we make it through the entire loop without finding a `Clear` square, we set the winner by declaring a tie:

```
for(int i=0;i<board.Length;i++) {  
    if(board[i] == SquareState.Clear)  
        return;  
}  
SetWinner(SquareState.Clear);  
}
```

12. Next, we create the `SetWinner` function that is called repeatedly in our `CheckVictory` function. This function passes who has won the game, and it initially turns on the game over screen and turns off the game board:

```
public void SetWinner(SquareState toWin) {  
    gameOverGroup.SetActive(true);  
    gameBoardGroupPortrait.SetActive(false);  
    gameBoardGroupLandscape.SetActive(false);
```

13. The function then checks to see who won and picks an appropriate message for the `victorText` object:

```
if(toWin == SquareState.Clear) {  
    victorText.text = "Tie!";  
}  
else if(toWin == SquareState.XControl) {  
    victorText.text = "X Wins!";  
}  
else {  
    victorText.text = "O Wins!";  
}
```

14. Finally, we have the `BackToMainMenu` function. This is short and sweet; it is simply called by the button on the game over screen to switch back to the main menu:

```
public void BackToMainMenu() {  
    gameOverGroup.SetActive(false);  
    mainMenuGroup.SetActive(true);  
}
```

That is all the code we have in our game. We have all the visual pieces that make up our game and now, we also have all the functional pieces. The last step is to put them together and finish the game.

Putting them together

We have our code and our menus. Once we connect them together, our game will be complete. To put it all together, perform the following steps:

1. Go back to the Unity Editor and select the **GameControl** object from the **Hierarchy** panel.
2. The three new properties in its **Inspector** window need to be filled in. Drag the **OpeningMenu** canvas to the **Main Menu Group** slot and **GameOverMenu** to the **Game Over Group** slot.
3. Also, find the text object child of **GameOverMenu** and drag it to the **Victor Text** slot.
4. Next, we need to connect the button functionality for each of our menus. Let's start by selecting the button object child of our **OpeningMenu** canvas.
5. Click on the little plus sign at the bottom right of its **Button (Script)** component to add a new functionality slot.
6. Click on the circle in the center of the new slot and select **GameControl** from the new pop-up window, just like we did for each of our game board buttons.
7. The drop-down list that currently says **No Function** is our next target. Click on it and navigate to **TicTacToeControl | NewGame ()**.
8. Repeat these few steps to add the functionality to the Button child of **GameOverMenu**. Except, select **BackToMainMenu()** from the list.
9. The very last thing to do is to turn off both the game boards and the game over menu, using the checkbox in the top left of the **Inspector**. Leave only the opening menu on so that our game will start there when we play it.

Congratulations! This is our game. All our buttons are set, we have multiple menus, and we even created a game board that changes based on the orientation of the player's device. The last thing to do is to build it for our devices and go show it off.

A better way to build for a device

Now, for the part of the build process that everyone itches to learn. There is a quicker and easier way to have your game built and play it on your Android device. The long and complicated way is still very good to know. Should this shorter method fail, and it will at some point, it is helpful to know the long method so that you can debug any errors. Also, the short path is only good for building for a single device. If you have multiple devices and a large project, it will take significantly more time to load them all with the short build process. Follow these steps:

1. Start by opening the **Build Settings** window. Remember, it can be found under **File** at the top of the Unity Editor.
If you have not already done so, save your scene. The option to save your scene is also found under **File** at the top of the Unity Editor.
2. Click on the **Add Current** button to add our current scene, also the only scene, to the **Scenes In Build** list. If this list is empty, there is no game.
3. Be sure to change your **Platform** to **Android** if you haven't already done so. It is, after all, still the point of this book.
4. Do not forget to set the **Player Settings**. Click on the **Player Settings** button to open them up in the **Inspector** window. You might remember this from *Chapter 1, Saying Hello to Unity and Android*.
5. At the top, set the **Company Name** and **Product Name** fields. Values of `TomPacktAndroid` and `Ch2 TicTacToe`, respectively, for these fields will match the included completed project. Remember, these fields will be seen by the people playing your game.
6. The **Bundle Identifier** field under **Other Settings** needs to be set, as well. The format is still `com.CompanyName.ProductName`, so `com.TomPacktAndroid.Ch2.TicTacToe` will work well. In order to see our cool dynamic GUI in action on a device, there is one other setting that should be changed. Click on **Resolution** and **Presentation** to expand the options.
7. We are interested in **Default Orientation**. The default is **Portrait**, but this option means that the game will be fixed in the portrait display mode. Click on the drop-down menu and select **Auto Rotation**. This option tells Unity to automatically adjust the game to be upright irrespective of the orientation in which it is being held.

The new set of options that popped up when **Auto Rotation** was selected allows the limiting of the orientations that are supported. Perhaps you are making a game that needs to be wider and held in landscape orientation. By unchecking **Portrait** and **Portrait Upside Down**, Unity will still adjust (but only for the remaining orientations).

[ On your Android device, the controls are along one of the shorter sides; these usually are the home, menu, and back or recent apps buttons. This side is generally recognized as the bottom of the device and it is the position of these buttons that dictates what each orientation is. The **Portrait** mode is when these buttons are down relative to the screen. The **Landscape Right** mode is when they are to the right. The pattern begins to become clear, does it not?]

8. For now, leave all the orientation options checked and we will go back to **Build Settings**.
9. The next step (and this very important) is to connect your device to your computer and give it a moment to be recognized. If your device is not the first one connected to your computer, this shorter build path will fail.
10. In the bottom-right corner of the **Build Settings** window, click on the **Build And Run** button. You will be asked to give the application file, the APK, a relevant name, and save it to an appropriate location. A name such as Ch2_TicTacToe.apk will be fine, and it is suitable enough to save the file to the desktop.
11. Click on **Save** and sit back to watch the wonderful loading bar that is provided. If you paid attention to the loading bar we built in the *Hello World* project in *Chapter 1, Saying Hello to Unity and Android*, you will notice we took an extra step this time around. After the application is built, there is a pushing to device step. This means that the build was successful and Unity is now putting the application on your device and installing it. Once this is done, the game will start on the device and the loading will be done.

We just learned about the **Build And Run** button provided by the **Build Settings** window. This is quick, easy, and free from the pain of using the command prompt; isn't the short build path wonderful? However, if the build process fails for any reason, including being unable to find the device, the application file will not be saved. You will have to go through the entire build process again, if you want to try installing again. This isn't so bad for our simple Tic-tac-toe game, but it might consume a lot of time for a larger project. Also, you can only have one Android device connected to your computer while building. Any more devices and the build process is a guaranteed failure. Unity also doesn't check for multiple devices until after it has gone through the rest of the potentially long build process.

Other than these words of caution, the **Build And Run** option is really quite nice. Let Unity handle the hard part of getting the game to your device. This gives us much more time to focus on testing and making a great game.

If you are up for a challenge, this is a tough one: creating a single player mode. You will have to start by adding an extra button to the opening screen for selecting the second game mode. Any logic for the computer player should go in the `Update` function. Also, take a look at `Random.Range` for randomly selecting a square to take control. Otherwise, you could do a little more work and make the computer search for a square where it can win or create a line of two matches.

Summary

At this point, you should be familiar with Unity's new uGUI system, including how to position the GUI elements, styling them to meet your needs, and adding functionality to them.

In this chapter, we learned all about the GUI by creating a Tic-tac-toe game. We first became familiar with creating buttons and other objects to be drawn on the game's GUI Canvas. After delving into improving the look of our game, we continued to improve it when we added dynamic orientation to the game board. We created an opening and closing screen to round out the game experience. Finally, we explored an alternative build method for putting our game onto devices.

In the next chapter, we will be starting a new and more complex game. The tank battle game we will create will be used to gain an understanding of the basic building blocks of any game: meshes, materials, and animations. When everything is done, we will be able to drive a tank around a colorful city and shoot animated targets.

3

The Backbone of Any Game – Meshes, Materials, and Animations

In the previous chapter, we learned about the GUI. We started by creating a simple Tic-tac-toe game to learn about the basic pieces of the game. We followed this by changing the look of the game and making the board handle multiple screen orientations. We completed with a few menus.

This chapter is about the core of any game: meshes, materials, and animations. Without these blocks, there is generally nothing to show to players. You could, of course, just use flat images in the GUI. But, where is the fun in that? If you are going to choose a 3D game engine, you might as well make full use of its capabilities.

To understand meshes, materials, and animations, we will be creating a tank battle game. This project will be used in a few other chapters. By the end of the book, it will be one of the two robust games that we will have created. For this chapter, the player will get to drive a tank around a small city, they will be able to shoot at animated targets, and we will also add a counter to track the scores.

This chapter covers the following topics:

- Importing meshes
- Creating the materials
- Animations
- Creating the prefabs
- Ray tracing

We will be starting a new project in this chapter, so follow the first section to get it started.

Setting up

Though this project will eventually grow to become much larger than the previous ones, the actual setup is similar to the previous projects and is not overly complex. You will need a number of starting assets for this project; they will be described during the setup process. Due to the complexity and specific nature of these assets, it is recommended to use the ones provided with the code bundle of this book for now.

As we did in the previous two chapters, we will need to create a new project so that we can create our next game. Obviously, the first thing to do is to start a new Unity project. For organizational purposes, name it Ch3_TankBattle. The following points are the prerequisites that are required for this project to kick-start:

1. This project will also grow to become much larger than our previous projects, so we should create some folders to keep things organized. For starters, create six folders. The top-level folders will be the `Models`, `Scripts`, and `Prefabs` folders. Inside `Models`, create `Environment`, `Tanks`, and `Targets`. Having these folders makes the project significantly more manageable. Any complete model can consist of a mesh file, one or more textures, a material for every texture, and potentially dozens of animation files.
2. Before we continue, it is a good idea to change your target platform to Android, if you haven't already done so. Every time the target platform is changed, all of the assets in the project need to be reimported. This is an automatic step carried out by Unity, but it will take an increasing amount of time as our project grows. By setting our target platform before there is anything in the project, we save lots of time later.
3. We will also make use of a very powerful part of Unity: prefabs. These are special objects that make the process of creating a game significantly easier. The name means prefabricated – created beforehand and replicated. What this means for us is that we can completely set up a target for our tank to shoot at and turn it into a prefab. Then, we can place instances of the prefab throughout the game world. If we ever need to make a change to the targets, all we need to do is modify the original prefab. Any change made to a prefab is also made on any instance of that prefab. Don't worry; it makes more sense when it is used.
4. We will need to create some meshes and textures for this project. To start with, we will need a tank (it is kind of hard to have a battle of tanks without any tanks). The tank that is provided with this code bundle has a turret and cannon, which are separate pieces. We will also use a trick to make the tank's treads look like they are moving, so each of them are a separate piece and uses a separate texture.

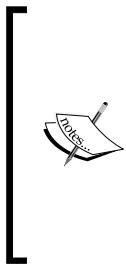
5. Finally, we will need an animated target. The one that is provided with the code bundle of this book is rigged up like the human arm with a bull's eye for the hand. It has four animations. The first starts in a curled position and goes to an extended position. The second is the reverse of the first one, going from the extended position to the curled position. The third starts in the extended position and is flung back, as if it is hit in the front, and returns to the curled position. The last is just like the third one, but it goes forward as if it is hit from behind. These are fairly simple animations, but they will serve us well in learning about Unity's animation system.

Very little happened here; we simply created the project and added some folders. There was also a little discussion about the assets that we would be using for this chapter's project.

Importing the meshes

There are several ways to import assets to Unity. We will be going through perhaps the simplest (and certainly the best) ways to import groups of assets. Let's get started:

1. Inside the Unity Editor, start by right-clicking on your `Tanks` folder and select **Show in Explorer** from the menu.
2. This opens the folder that contains the asset that was selected. In this case, the `Models` folder opens in the Windows' folder browser. We just need to put our tank and its textures into the `Tanks` folder.



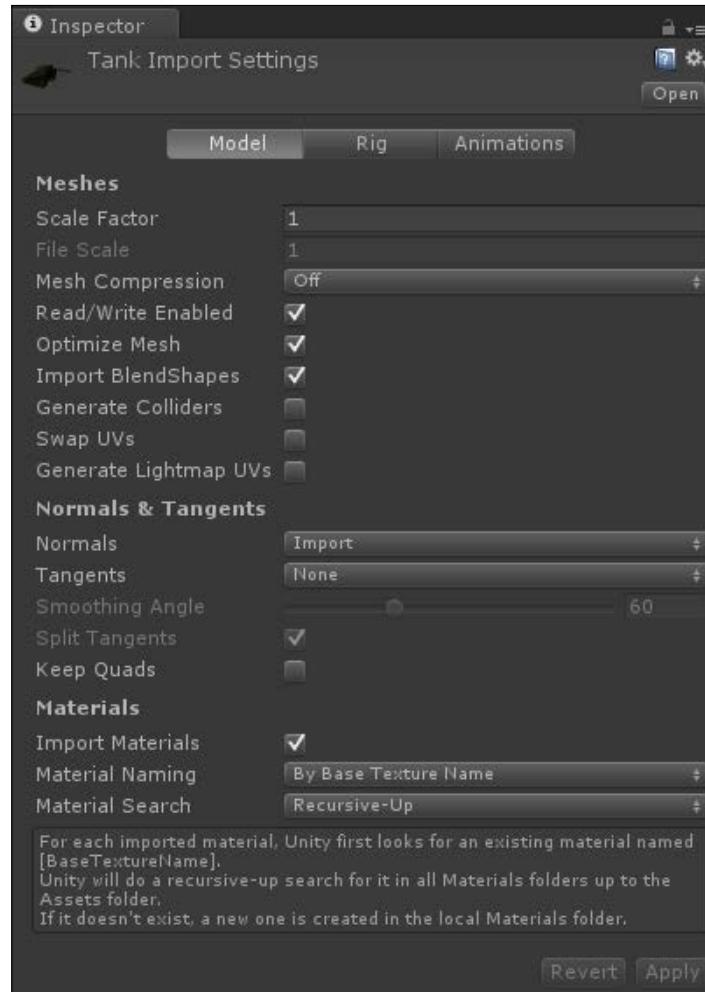
The files provided for this chapter are `Tank.blend`, `Tanks_Type01.png`, and `TankTread.png`. In addition, utilizing the `.blend` files in Unity requires Blender to be installed in your system. Blender is a free modeling program that is available at <http://www.blender.org>. Unity makes use of it in order to convert the previously mentioned files into ones that it can fully utilize.

3. When we return to Unity, the fact that we added files will be detected, and they will automatically be imported. This is one of the best things about Unity. There is no need to explicitly tell Unity to import. If there are changes within the project's assets, it just updates the assets automatically.
4. You may also notice that an extra folder and some files were created when Unity imported our tank. Whenever a new mesh is imported, by default Unity will try to pair it with the materials. We will go into more detail about what a material is in Unity in the next section. For now, it is an object that keeps track of how to display a texture on a mesh. Based on the information in the mesh, Unity looks in the project for a material with the correct name. If one cannot be found, a `Materials` folder is created next to the mesh and the missing materials are created inside it. When creating these materials, Unity also searches for the right textures. This is why it is important to add textures to the folder at the same time as the mesh, so that they can all be imported together. If you did not add the textures at the same time as the tank, the section about creating materials will describe how to add textures to materials.

We have imported our tank into Unity. It is really quite simple. Changes made to any of the assets or folders of the project are automatically detected by Unity, and anything that is needed is accordingly imported.

Tank import settings

Importing any asset into Unity is done by using a default group of settings. Any of these settings can be changed from the **Inspector** window. With your new tank selected, we will go over the import settings for a model here:



We can see in the preceding screenshot that the top of the **Inspector** window has three tabs: **Model**, **Rig**, and **Animations**. The **Model** page handles the mesh itself, while **Rig** and **Animations** are for importing animations. For now, we only care about the **Model** page, so select it if it is not already selected. Each section of the **Model** page is broken down here.

Meshes

The **Meshes** section of the previous screenshot has the following options:

- The **Meshes** section of the **Import Settings** window starts with the **Scale Factor** attribute. This is a value that tells Unity how big the mesh is by default. One generic unit or one meter from your modeling program translates to one unit in Unity. This tank was made in generic units, so the tank's scale factor is one. If you were working in centimeters when making the tank, the scale factor would be 0.01 because a centimeter is a hundredth of a meter.
- The **File Scale** option is the scale used in the modeling program when the model was originally created. It is primarily informational. If you need to adjust the size of the imported model, adjust the **Scale Factor**.
- The next option, **Mesh Compression**, will become important in the final chapter when we go over the optimization of our games. The higher the compression is set to, the smaller will be the size of the file in the game. However, this will start to introduce weirdness in your mesh as Unity works to make it smaller. For now, leave it as **Off**.
- The **Read/Write Enabled** option is useful if you want to make changes to the mesh while the game is playing. This allows you to do some really cool things, such as destructible environments, where your scripts break the meshes into pieces based on where they are being shot. However, it also means that Unity has to keep a copy of the mesh in memory, which could really start to lag a system if it is complex. This is outside the scope of this book, so unchecking this option is a good idea.
- The **Optimize Mesh** option is a good one to leave on, unless you are doing something specific and fancy with the mesh. With this on, Unity does some special 'behind the scenes' magic. In computer graphics and especially Unity, every mesh is ultimately a series of triangles that are drawn on a screen. This option allows Unity to reorder the triangles in the file so that the whole mesh can be drawn faster and more easily.

- The **Import BlendShapes** option allows Unity to make sense of any BlendShapes that might be part of the model. These are animated positions of the vertexes of the model. Usually, they are used for facial animations. The next option, **Generate Colliders**, is a useful one if you're doing complex things with physics. Unity has a set of simple collider shapes that should be used whenever possible because they are easier to process. However, there are situations where they won't quite get the job done; for example, a rubble or a half-pipe where the collision shape is too complex to be made with a series of simple shapes. That is why Unity has a **Mesh Collider** component. With this option checked, a **Mesh Collider** component is added to every mesh in our model. We will be sticking with simple colliders in this chapter, so leave the **Generate Colliders** option off.
- The **Swap UVs** and **Generate Lightmap UVs** options are primarily used when working with lighting, especially lightmaps. Unity can handle two sets of UV coordinates on a model. Normally, the first is used for the texture and the second for the lightmap or shadow texture. If they are in the wrong order, **Swap UVs** will change them so that the second set comes first. If you need an unwrap for a lightmap but did not create one, **Generate Lightmap UVs** will create one for you. We are not working with lightmaps in this project, so both of these can remain off.

Normals & Tangents

The **Normals & Tangents** section of the earlier screenshot has the following options:

- The next section of options, **Normals & Tangents**, begins with the **Normals** option. This defines how Unity will hold the normals of your mesh. By default, they are imported from the file; however, there is also the option to make Unity calculate them based on the way the mesh is defined. Otherwise, if we set this option to **None**, Unity will not import the normals. Normals are needed if we want our mesh to be affected by real-time lighting or make use of normal maps. We will be making use of real-time lighting in this project, so leave it set to **Import**.

- The **Tangents**, **Smoothing Angle**, and **Split Tangents** options are used if your mesh has a normal map. Tangents are needed to determine how lighting interacts with a normal mapped surface. By default, Unity will calculate these for you. Importing tangents is only possible from a few file types. The smoothing angle, based on the angle between the two faces, dictates whether shading across an edge would be smooth or sharp. The **Split Tangents** option is there to handle a few specific lighting quirks. If lighting is broken by seams, enabling this option will fix it. Normal maps are great for making a low-resolution game look like a high-resolution one. However, because of all the extra files and information needed to use them, they are not ideal for a mobile game. Therefore, we will not be using them in this book and so all of these options can be turned off to save memory.
- The **Keep Quads** option will allow your models to take advantage of DirectX 11's new tessellation techniques for creating high-detail models from low-detail models and a special displacement map. Unfortunately, it will be a while before mobile devices can support such detail, and even longer before they become commonplace.

Materials

The **Materials** section of the previous screenshot has the following options:

- The last section, **Materials**, defines how Unity should look for materials. The first option, **Import Materials**, allows you to decide whether or not a material should be imported. If it is turned off, a default white material will be applied. This material will not show up anywhere in your project; it is a hidden default. For models that will not have any textures, such as collision meshes, this can be turned off. For our tank and nearly every other case, this should be left on.
- The last two options, **Material Naming** and **Material Search**, work together to name and find the materials for the mesh. Directly below them, there is a text box that describes how Unity will go about searching for the material.
 - The name of the material being searched for can be the name of the texture used in the modeling program, the name of the material created in the modeling program, or the name of the model and the material. If a texture name cannot be found, the material name will be used.

- By default, Unity does a recursive-up search. This means that we start the search by looking in the `Materials` folder, followed by a search for any materials that are in the same folder. We then check the parent folder for matching materials, followed by the folder above that. This continues until we find either the material that has the correct name or we reach the root assets folder.
- Alternatively, we have the options of checking the entire project or only looking in the `Materials` folder that is next to our model. The defaults for these options are just fine. In general, they do not need to be changed. They can be easily dealt with, especially for a large project, using the Unity Editor scripting, which will not be covered in this book.

The Revert and Apply buttons

Next, the screenshot has the **Revert** and **Apply** buttons, which are explained here:

- Whenever changes are made to the import settings, one of the two buttons, **Revert** or **Apply**, must be chosen. The **Revert** button cancels the changes and switches the import settings back to what they were before changes were made. The **Apply** button confirms the changes and reimports the model with the new settings if these buttons are not selected; Unity will complain with a pop up and force you to make a choice before letting you mess with anything else.



- Finally, we have two types of previews as we can see in the previous screenshot. The **Imported Object** section is a preview of what the object will look like in the **Inspector** window, if we added the object to the **Scene** view and selected it. The **Preview** window, the section where we can see the tank model, is what the model will look like in the **Scene** view. You can click and drag the object in this window to rotate it and look at it from different angles. In addition, there is a little blue button in this window. By clicking on this button, you will be able to add labels to the object. Then, these labels will also be searchable in the **Project** window.

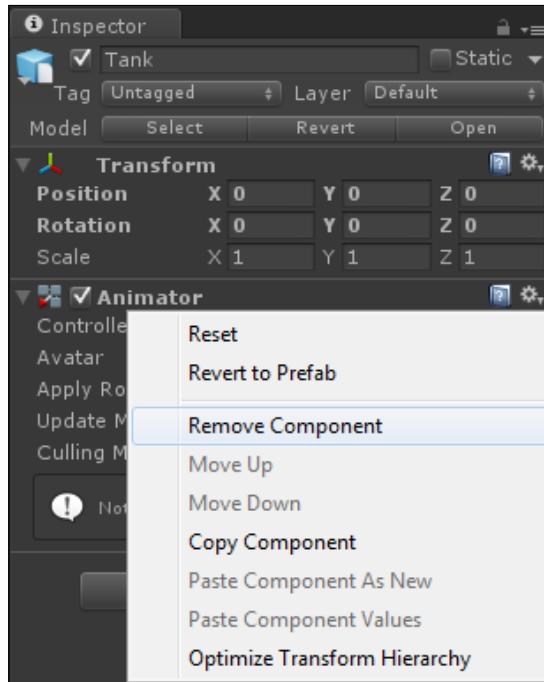
Setting up the tank

Now that we have imported the tank, we need to set it up. We will be adjusting the arrangement of the tank as well as creating a few scripts.

The tank

At this point, the creation of our tank will primarily consist of the creation and arrangement of the tank's components. Using the following steps, we can set up our tank:

1. Start by dragging the tank from the **Project** window to the **Hierarchy** window. You will notice that the name of the tank appears in blue color in the **Hierarchy** window. This is because it is a prefab instance. Any model in your project largely acts like a prefab. However, we want our tank to do more than just sit there; so, being a prefab of a static mesh is not helpful. Therefore, select your tank in the **Hierarchy** window and we will start to make it useful by removing the **Animator** component. To do this, select the gear to the right of the **Animator** component in the **Inspector** window. From the new drop-down list, select **Remove Component**, as seen in the following screenshot, and it will be removed:

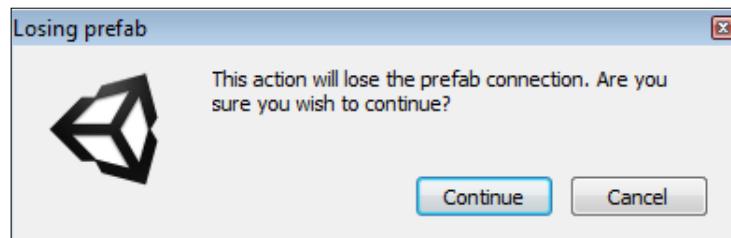


2. If you are using the tank that is provided by default, selecting the different parts of it will reveal that all the pivot points are at the base. This will not be useful for making our turret and cannon pivot properly. The easiest way to solve this is by adding new empty **GameObjects** to act as pivot points.

 Any object in the scene is a **GameObject**. Any empty **GameObject** is one that only has a **Transform** component.

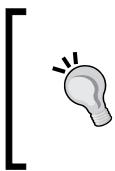
3. At the top of the Unity Editor, **Create Empty** is the first option under the **GameObject** button. It creates the objects that we need. Create two empty **GameObjects** and position one at the base of the turret and the other at the base of the cannon. In addition, rename them as **TurretPivot** and **CannonPivot** respectively. This can be done with the textbox at the very top of the **Inspector** window if the object is selected.

4. In the **Hierarchy** window, drag `TurretPivot` onto `Tank`. This changes the parent of `TurretPivot` to `Tank`. Then, drag the object, that is, the turret mesh, onto `TurretPivot`. In the code, we will be rotating the pivot point and not the mesh directly. When a parent object moves or rotates, all of the children objects move with it. When you make this change, Unity will complain about the change to the original hierarchy of the object; it does this just to make sure that it is a change that you want to make and not an accident:



5. As losing the connection to the prefab can potentially break a game, Unity just wants to be sure that we actually want it to happen. So, click on **Continue** and we can finish working with the tank without other complaints from Unity. We also need to make `CannonPivot` a child of `TurretPivot` and the cannon a child of `CannonPivot`.
6. To finish our hierarchy changes, we need to place the camera. Since we want the player to appear as if they are in the tank, the camera should be placed behind and above the tank with a tilt slightly downward to focus on a spot a few tank lengths ahead. Once it is positioned, make it a child of `TurretPivot` as well.

We have set up the basic structure that our tank will use. By making use of multiple objects in this way, we can control their movements and actions independently from each other. At this point, instead of having a rigid tank that only points forward, we can tilt, rotate, and aim each piece independently.



Also, the tank should be centered above the point at which you want the whole thing to pivot around. If yours is not, you can select everything that is under the base tank object in the **Hierarchy** window and move it around.

Keeping score

A short script for keeping track of a player's score and the addition of a text element will constitute the focus of this short section. The following are the steps for the creation of our script:

1. The first script that is needed to make our tank work is fairly simple. Create a new script and name it `ScoreCounter`. It will, as the name implies, track the score. Create it in the `Scripts` folder and clear out the default functions, just like every other script that we have made so far.
2. Just like we did in the previous chapter, since any script that needs access to any of our GUI elements needs an extra line at the very top of the script, add the following line of code right after the line that says `using UnityEngine;`. This allows us to use and change the text element we need to display the score:

```
using UnityEngine.UI;
```

3. The following line of code should look familiar from the previous chapter. First, we define an integer counter. As it is static, other scripts (such as the ones we will create for the targets) will be able to modify this number and give us the score:

```
public static int score = 0;
```

4. We will then add a variable to store the text element for our interface. It will work like the turn indicator from the previous chapter, giving us a location to update and display the player's score:

```
public Text display;
```

5. The last bit of code for this script is an `Update` function. This function is called automatically by Unity for every single frame. This is the perfect spot for us to put any code and logic that needs to change regularly without the player's direct input. For our purpose, we will update the text element and make certain that it always has the most up-to-date score to display. By adding the score to double quotes, we are changing the number into a word so that it can be used properly by the text element:

```
public void Update() {  
    display.text = "" + score;  
}
```

That's it for this very simple script. It will track our score throughout the game. In addition, instead of doing any of the score increments itself, other scripts will update the counter to give points to the player.

Repeat buttons

The buttons that we have used so far only perform an action when they are pressed and released. Our players will need to hold down the buttons to control their tank. So, we need to create a repeat button; a button that performs an action as long as it is held down. Follow these steps to create a repeat button:

1. Create a new script that should be named as `RepeatButton`.
2. To give this script access to the parts of Unity that it needs in order to work, just like the previous script, we need to add the following two lines right after the one that says `using UnityEngine;`. The first will give us access to the `Selectable` class: the one from which all interactive interface elements are derived. The second will let us handle the events that occur when our players interact with our new button:

```
using UnityEngine.UI;  
using UnityEngine.EventSystems;
```

3. Next, we need to update the `public class` line of our code. Any normal script that will provide functionality to the objects in our game expands upon the `MonoBehaviour` class. We need to change the line to the following, so that our script can instead exist in and expand the functionality of the interface:

```
public class RepeatButton : Selectable {
```

4. Our script will have a total of four variables. The first allows it to keep track of whether or not it is being pressed:

```
private bool isPressed = false;
```

5. The next three variables will provide the same functionality as the button did in the previous chapter. For the button, we had to select an object, followed by a function from a specific script, and finally some value to send. Here, we are going to do the same thing. The first variable here keeps track of which object in the scene we are going to interact with. The second will be the name of a function that is on some of the script attached to the object. The last will be a number to send along for the function, and it will provide more specific input:

```
public GameObject target;  
public string function = "";  
public float value = 0f;
```

6. The first function for this script will override a function provided by the `Selectable` class. It is called the moment the player clicks on the button. It is given some information about how and where it was clicked, which is stored in `eventData`. The second line just calls the function of the same name on the parent class. The last thing the function does is set our Boolean flag to mark that the button is currently being pressed by the player:

```
public override void OnPointerDown(PointerEventData eventData) {  
    base.OnPointerDown(eventData);  
    isPressed = true;  
}
```

7. The next function does the exact same thing as the previous function. The main difference is that it is called when the mouse or touch from the player is no longer over the button in the interface. The second difference is that it sets the Boolean to `false` because when our player drags their finger off the button, they are no longer pressing it, and we want to stop performing our action in that case:

```
public override void OnPointerExit(PointerEventData eventData) {  
    base.OnPointerExit(eventData);  
    isPressed = false;  
}
```

8. The following function is like the first two. However, it is called when the button is released:

```
public override void OnPointerUp(PointerEventData eventData) {  
    base.OnPointerUp(eventData);  
    isPressed = false;  
}
```

9. The final function of this script is again our `Update` function. It first checks whether the player is currently pressing this button. It then calls the `SendMessage` function on our target object, telling it what function to perform and what number to use. The `SendMessage` function is only available for **GameObject** and **MonoBehaviour** components. It takes the name of a function and tries to find it on the `GameObject` to which the message was sent:

```
public void Update() {  
    if(isPressed) {  
        target.SendMessage(function, value);  
    }  
}
```

Another script done! This one allows us to hold buttons rather than be forced to press them repeatedly to move through our game.

Controlling the chassis

A normal tank rotates in place, and it can easily move forward and back. We will make our tank do this with the creation of a single script. Perform these steps to create our second script for the tank:

1. The second script is called `ChassisControls`. It will make our tank move around. We will create it in the `Scripts` folder as well.
2. The first three lines of the script define the variables that the tank will need to move around. We will also be able to change them in the **Inspector** window, in case our tank is too fast or too slow. The first line defines a variable that holds a connection to a `CharacterController` component. This component will not only move the tank around easily but it will also allow it to stop by walls and other colliders. The next two lines of code define how fast we move and rotate:

```
public CharacterController characterControl;  
public float moveSpeed = 10f;  
public float rotateSpeed = 45f;
```

3. We start the following line of code by defining our `MoveTank` function; it needs to be passed a `speed` value to dictate how far and in which direction the tank should go. A positive value will make the tank go forward and a negative value will make it go backwards:

```
public void MoveTank(float speed) {
```

4. In order to move in a three-dimensional space, we need a vector—a value with both direction and magnitude. Therefore, we define a movement vector and set it to the tank's forward direction, multiplied by the tank's speed, and again multiplied by the amount of time that has elapsed since the last frame.
 - If you remember from geometry class, 3D space has three axes: `x`, `y`, and `z`. In Unity, the following convention applies: `x` is to the right, `y` is up, and `z` is forward. The `transform` component holds these values for an object's position, rotation, and scale. We can access the `transform` component of any object in Unity by calling upon the `transform` variable that Unity provides. The `transform` component also provides a `forward` variable that will give us a vector that points in the direction in which the object is facing.

- In addition, we want to move at a regular pace, for example, a certain number of meters per second; therefore, we make use of `Time.deltaTime`. This is a value provided by Unity that holds how many seconds it has been since the last frame of the game was drawn on screen. Think of it like a flip book. In order to make it look like a guy is walking across the page, he needs to move slightly on each page. In the case of a game, the pages are not flipped regularly. So, we have to modify our movement by how long it has taken to flip to the new page. This helps us to maintain an even pace.

```
Vector3 move = characterControl.transform.forward * speed * Time.deltaTime;
```

5. Next, we want to stay on the ground. In general, any character you want to control in a game does not automatically receive all of the physics, such as gravity, that a boulder would. For example, when jumping, you temporarily remove gravity so that the character can go up. That is why the next line of code does a simple implementation of gravity by subtracting the normal speed of gravity and then keeping it in pace with our frame rate:

```
move.y -= 9.8f * Time.deltaTime;
```

6. Finally, for the `MoveTank` function, we actually do the moving. The `CharacterController` component has a special `Move` function that will move the character but constrain it by collisions. We just need to tell it how far and in which direction we want to move this frame by passing the `move` vector to it. This final curly brace, of course, closes off the function:

```
characterControl.Move(move);  
}
```

7. The `RotateTank` function also needs a speed value to dictate how fast and in which direction to rotate. We start by defining another vector; however, instead of defining in which direction to move, this one will dictate in which direction to rotate around. In this case, we will be rotating around our up direction. We will then multiply that by our speed and `Time.deltaTime` parameters to move fast enough and keep pace with our frame rate.

```
public void RotateTank(float speed) {  
    Vector3 rotate = Vector3.up * speed * Time.deltaTime;
```

8. The last bit of the function actually does the rotation. The **Transform** component provides a `Rotate` function. Rotation, especially in 3D space, can become weird and difficult very quickly. The `Rotate` function handles all of that for us; we just need to supply it with the values to apply for rotation. In addition, don't forget the curly brace to close off the function:

```
characterControl.transform.Rotate(rotate);  
}
```

We created a script to control the movement of our tank. It will use a special `Move` function from the `CharacterController` component so that our tank can move forwards and backwards. We also used a special `Rotate` function provided by the `Transform` component to rotate our tank.

Controlling the turret

This next script will allow the player to rotate their turret and aim the cannon:

1. The last script that we need to create for our tank is `TurretControls`. This script will allow players to rotate the turret left and right and tilt the cannon up and down. As with all of the others, create it in the `Scripts` folder.
2. The first two variables that we define will hold pointers to the turret and cannon pivots- the empty `GameObjects` that we created for our tank. The second set is the speed at which our turret and cannon will rotate. Finally, we have some limit values. If we didn't limit how much our cannon could rotate, it would just spin around and around, passing through our tank. This isn't the most realistic behavior for a tank, so we must put some limits on it. The limits are in the range of 300 because straight ahead is zero degrees and down is 90 degrees. We want it to be in the upward angle, so it is in the range of 300. We can also use 359.9 because Unity will change 360 to zero so that it can continue to rotate:

```
public Transform turretPivot;  
public Transform cannonPivot;  
  
public float turretSpeed = 45f;  
public float cannonSpeed = 20f;  
  
public float lowCannonLimit = 315f;  
public float highCannonLimit = 359.9f;
```

3. Next is the `RotateTurret` function. It works in exactly the same way as the `RotateTank` function. However, instead of looking at a `CharacterController` component's `transform` variable, we act upon the `turretPivot` variable:

```
public void RotateTurret(float speed) {
    Vector3 rotate = Vector3.up * speed * Time.deltaTime;
    turretPivot.Rotate(rotate);
}
```

4. The second and last function, `RotateCannon`, gets a little more down-and-dirty with rotations. The fault completely lies with the need to put limits on the rotation of the cannon. After opening the function, the first step is to figure out how much we are going to be rotating this frame. We use a float value instead of a vector because we have to set the rotation ourselves:

```
public void RotateCannon(float speed) {
    float rotate = speed * Time.deltaTime;
```

5. Next, we define a variable that holds our current rotation. We do this because Unity will not let us act on the rotation directly. Unity actually keeps track of rotation as a quaternion. This is a complex method of defining rotations that is beyond the scope of this book. Luckily, Unity gives us access to an x, y, and z method of defining rotations called `EulerAngles`. It is a rotation around each of the three axes in 3D space. The `localEulerAngles` value of a **Transform** component is the rotation relative to the parent **GameObject**.

```
Vector3 euler = cannonPivot.localEulerAngles;
```



It is called `EulerAngles` because of Leonhard Euler, a Swiss mathematician, who came up with this method of defining rotations.

6. Next, we adjust the rotation and apply the limits in one go through the use of the `Mathf.Clamp` function. `Mathf` is a group of useful mathematical functions. The `clamp` function takes a value and makes it no lower and no higher than the other two values passed to the function. So, we first send it our x axis rotation, which is the result of subtracting `rotate` from the current x rotation of `euler`. As the positive rotation is clockwise around an axis, we have to subtract our rotation to go up instead of down with a positive value. Next, we pass our lower limit to the `Clamp` function, followed by our higher limit: these are the `lowCannonLimit` and `highCannonLimit` variables that we defined at the top of the script:

```
euler.x = Mathf.Clamp(euler.x - rotate, lowCannonLimit,
highCannonLimit);
```

- Finally, we have to actually apply the new rotation to our cannon's pivot point. This involves simply setting the `localEulerAngles` value of the **Transform** component to the new value. Again, be sure to use the curly brace to close off the function:

```
cannonPivot.localEulerAngles = euler;  
}
```

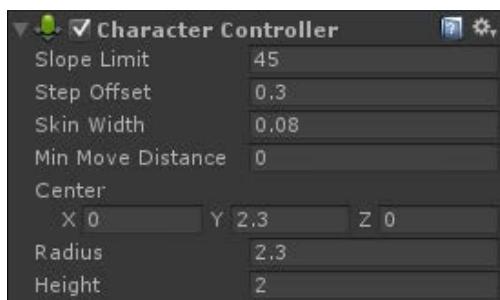
We have now created a script that will control the turret of the tank. The player will be able to control the tilt of the cannon and rotation of the turret. This script functioned in a very similar manner to the `ChassisControls` script that we created earlier – the difference was in limiting the amount the cannon can tilt.

Putting the pieces together

That was the last of the scripts for the moment. We have our tank and our scripts; the next step is to put them together:

- Now, we need to add the scripts to our tank. Remember how we added our Tic-tac-toe script to the camera in the last chapter? Start by selecting your tank in the **Hierarchy** window. Before these scripts work, we will first need to add the `CharacterController` component to our tank. So, go to the top of the Unity Editor and select **Component**, then select **Physics**, and finally click on the **Character Controller** option.

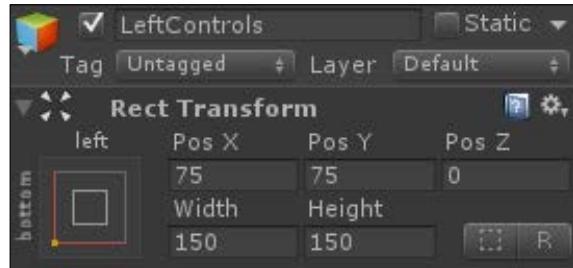
You will notice that a green capsule appears on the tank in the **Scene** view as soon as you add the new component. This capsule represents the space that will collide and interact with other colliders. The values on the **Character Controller** component let us control how it interacts with other colliders. For most cases, the defaults for the first four parameters are just fine.



The parameters in **Character Controller** are as follows:

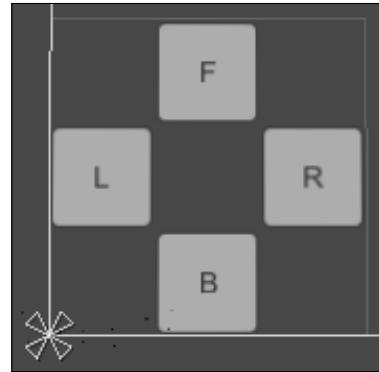
- **Slope Limit:** This attribute shows us how steep an incline the controller can move up.
 - **Step Offset:** This attribute shows us how high a step can be before it starts to block movement.
 - **Skin Width:** This defines how far another collider can penetrate this controller's collider before it is completely stopped. This is mostly used for squeezing between objects.
 - **Min Move Distance:** This attribute is for limiting jitter. It is the minimum amount of movement that has to be applied in a frame before it will actually move.
 - **Center/Radius/Height:** These attributes define the size of the capsule that you see in the **Scene** view. They are used for the collision.
2. The last three values are the most important right now. We need to adjust these values as closely as possible to match our tank's size. Admittedly, the capsule is round and our tank is square, but a **CharacterController** component is the easiest way to move a character with collision, and it will be used the most often. Use values of `2.3` for the **Radius** attribute and the **Y** portion of the **Center** attribute; everything else can be left as the default values.
 3. It is now time to add the scripts to our tank. Do this by selecting the tank in the **Hierarchy** window and dragging the **ChassisControls**, **TurretControls**, and **ScoreCounter** scripts onto the **Inspector** window. This is just as we did in the previous chapters.
 4. Next, we need to finish creating the connections that we started in our scripts. Start by clicking the **CharacterController** component's name and dragging it to the **Character Control** slot that is on our new **ChassisControls** script component. Unity lets us connect object variables in the Unity Editor so that they do not have to be hardcoded.
 5. We also need to connect our turret and cannon pivot points. So, click and drag the points from the **Hierarchy** window to the corresponding variable on the **TurretControls** script component.
 6. Before we can test our game, we need to create a bunch of GUI buttons to actually control our tank. Start by creating a canvas, just like we did in the previous chapter, and one empty GameObject.
 7. The empty GameObject needs a **Rect Transform** component, and it needs to be made a child of **Canvas**.

8. Rename it to `LeftControls` and set its anchor to **bottom left**. In addition, set **Pos X** to 75, **Pos Y** to 75, **Pos Z** to 0, **Width** to 150, and **Height** to 150 as shown in the following screenshot:

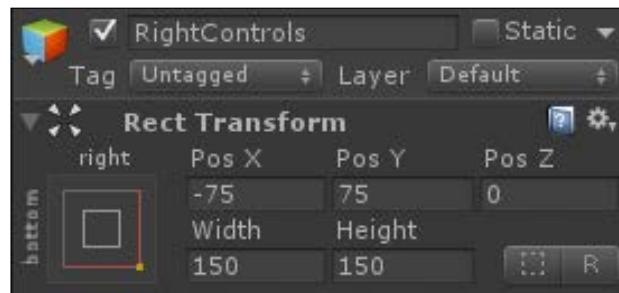


9. Next, we need four buttons to be the children of `LeftControls`. As in the last chapter, they can be found at the top of the editor by navigating to **GameObject | UI | Button**.
10. Rename the four buttons to `Forward`, `Back`, `Left`, and `Right`. While you're at it, you can also change their text child to have the relevant text, such as `F`, `B`, `L`, and `R`.
11. The button only activates when the player clicks and releases it. Clicking repeatedly just to make the tank move will not work very well. So, click on the gear to the right of each of their **Button** components and select **Remove Component**.
12. Now, add our `RepeatButton` script to each. As we extended that `Selectable` class, you can see that we have all the same controls over our button that we did on the other buttons.
13. Set the values of **Width** and **Height** of all the four buttons to 50. Their positions become as follows:

Button	Pos X	Pos Y
Forward	0	50
Left	-50	0
Back	0	-50
Right	50	0



14. Now that we have our four movement buttons, we need to connect them to our tank. For each of the buttons, drag Tank from the **Hierarchy** panel and drop it in the **Target** slot in the **Inspector** panel.
15. When we next set the **Function** and **Value** slots, spelling is very important. If something is a little bit off, your function will not be found, lots of errors will appear, and the tank will not work. For the Forward button, set the **Function** slot to MoveTank and the **Value** slot to 1. The Back button also needs the value of MoveTank in the **Function** slot, but it needs a value of -1 in the **Value** slot. The Left button needs a value of RotateTank in the **Function** slot and a value of -1 in the **Value** slot. The Right button needs a value of RotateTank in the **Function** slot and 1 in the **Value** slot.
16. Next, we need to set up our turret controls. Right-click on **LeftControls** in the **Hierarchy** window and select **Duplicate** from the new menu. Rename the new copy to **RightControls**.
17. This new control set needs an anchor set of **bottom right**, a **Pos X** of -75, and **Pos Y** of 75 (as shown in the following screenshot):



18. The buttons under this set will need to be renamed as Up, Down, Left, and Right. Their text can be changed to U, D, L, and R respectively.

19. The **Function** slot of the Up button should be set to RotateCannon with the value of the **Value** slot as 1. The Down button has a **Function** slot value of RotateCannon and a **Value** slot value of -1. The Left button needs RotateTurret as the value of the **Function** slot with a value of -1 for the **Value** slot. Finally, the Right button needs a **Function** slot value of RotateTurret with a **Value** slot value of 1.
20. The last thing to do is to create a new Text element that can be found by navigating to **GameObject | UI | Text** and rename it as Score.
21. Finally, select your Tank and drag Score from the **Hierarchy** window to the **Display** slot of the **Score Counter (Script)** component.
22. Save the scene as TankBattle and try it out.

We just finished putting our tank together. Unless you look at the **Scene** view while using the movement controls, it is hard to tell that the tank is moving. The turret controls can be seen in the **Game** view though. Other than not having a point of reference for whether or not our tank is moving, it runs pretty well. The next step and the next section will give us that reference point as we add our city.

You might notice a quick jump when you first try to tilt the cannon. Such behavior is annoying and makes the game look broken. Try adjusting the cannon to fix it. If you are having trouble with it, take a look at the cannon's starting rotation. It has to do with the way the rotation is clamped every time we try to move it.

Creating materials

In Unity, the materials are the defining factor for how models are drawn on the screen. They can be as simple as coloring it all blue or as complex as reflecting water with waves. In this section, we will cover the details of the controls for a material. We will also create our city and some simple materials to texture it with.

The city

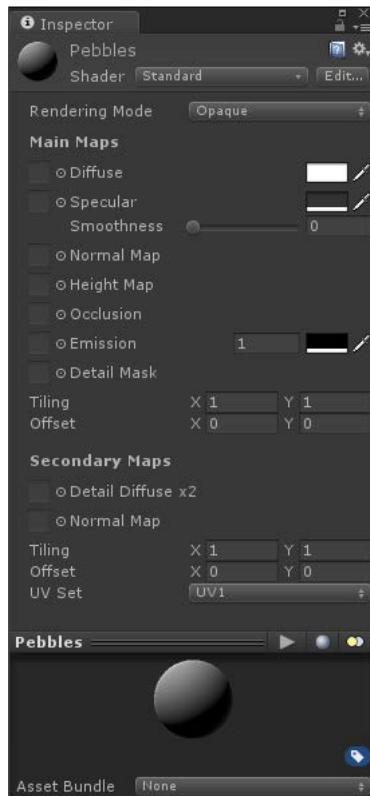
Creating a city gives our tanks and our players a good place to play. Follow these steps to create our city:

1. For the purpose of this section, no part of the city provided with the code bundle of this book was given a specific texture. It was just unwrapped and some tileable textures were created. So, we need to start by importing the city and the textures to the **Environment** folder. Do this in the same way in which we imported the tank.



The files are TankBattleCity.blend, brick_001.png, brick_002.png, brick_003.png, dirt_001.png, dirt_003.png, pebbles_001.png, rocks_001.png, rubble_001.png, and water_002.png.

2. As the city is unwrapped, Unity will still create a single material for it. However, textures were never applied in any modeling program. So, the material is plain white. We have several extra textures, so we are going to need more than just that one material for the whole city. Creating a new material is simple; it is done just like creating a new script. Right-click on the Materials folder inside the Environment folder, select **Create**, and then click on **Material**, which is about halfway down the menu.
3. This will create a new material in the folder and immediately allow us to name it. Name the material as **Pebbles**.
4. With your new material selected, take a look at the **Inspector** window. When we have selected a material, we get the options that are needed to change its look:



5. You can see the following things from the preceding screenshot:
 - At the very top of the **Inspector** window, we have the material's name followed by a **Shader** drop-down list. A shader is essentially a short script that tells the graphics card how to draw something on the screen. You will use the **Standard** shader most often; it is essentially an all-inclusive shader, so it is always selected by default. This is where you would select any special effect or custom shaders.
 - The **Rendering Mode** drop-down menu lets you pick whether or not this material will use any amount of transparency. **Opaque** means it will be solid. The **Cutout** option will render with a sharp edge around transparent areas of your texture, based on the value of **Alpha Cutoff**. The **Transparent** option will give you a smooth edge that is based on the alpha channel of your texture.

Main Maps

The **Main Maps** section has the following options:

- The **Main Maps** section starts with **Diffuse**, where you put your main color texture. It can be tinted with the color picker to the right of the slot.
- The **Specular** option defines the shininess of your material; think of it like the glare from the light on your device's screen. You can either use an image to control it, or you can use the color picker to determine what color is reflected and the smoothness to control how sharp the glare is.
- The **Normal Map** option lets you add a texture that controls shading across the surface of your material. These textures need to be specially imported. If the texture you pick hasn't been set properly, a warning box will appear where you can select **Fix Now** to change it. A slider will also appear, giving you control over how much effect the texture has.
- The **Height Map** option works in a manner similar to **Normal Map**. It adjusts the bumpiness of your material and gives a slider to adjust it.
- The **Occlusion** option lets you add an ambient occlusion texture to the material, controlling the darkness or lightness of the material based on the proximity of objects to each other in the model.
- The **Emission** option gives you control over the projected light and color that a material gives off. This only affects lightmaps and the appearance of this material. To actually give off light dynamically, it has to be faked with the addition of a real-time light.
- The **Detail Mask** option lets you control where the textures in **Secondary Maps** appear on your material.

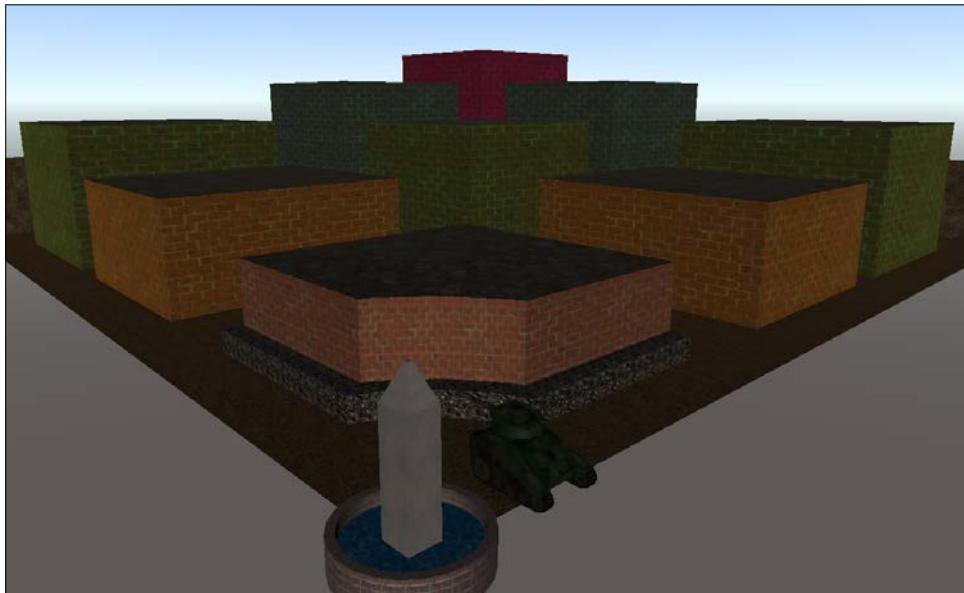
- The values of **Tiling** and **Offset** control the size and position of the textures. The values of **Tiling** dictate how many times the texture will repeat across the normalized UV space in the x and y directions. The **Offset** parameter is how far from zero the texture starts in the normalized UV space. You can select the number fields and input values to modify them. By doing so, and paying attention to the **Preview** window at the bottom, you will see how they change the texture. Tiling textures are most commonly used for large surfaces where the texture is similar across the surface and a particular texture just repeats.

Secondary Maps

The **Secondary Maps** section has the following options:

1. **Secondary Maps** starts with **Detail Diffuse x2**, which is an extra diffuse texture to be blended on top of your main diffuse texture. It could be used to add a bumpy variation across the surface of your boulder.
2. **Normal Map** works like the main **Normal Map** slot and controls the shading of the detail textures.
3. The second set of **Tiling** and **Offset** values work like the first and just control the detail textures. Usually these are set higher than the first to add extra interest across the surface of the material.
4. **UV Set** just lets you select the model unwrap set that the detail textures are going to use on the model to which the material is applied.
5. Add the `pebbles_001` texture to this material by dragging it from the **Project** window and dropping it on the square to the right of the **Diffuse** slot.
6. To make the color of the texture better, use the color picker to the right of the **Diffuse** slot and pick a light tan color.
7. A value of 30 for both the **X** and **Y** values of the main **Tiling** will make it easier to see when **Tiling** is applied to our city streets.
8. To see our new material in action, first drag your city to the **Hierarchy** window so that it is added to the **Scene** view. By right-clicking and dragging, you can look around in your **Scene** view, and by using *W*, *A*, *S*, and *D*, you can move around. Look over at the streets of the city.
9. Now, drag your new material from the **Project** window into your **Scene** view. While dragging the material around, you should see that the meshes change to appear as if they are using the material. Once you are over the streets, let go of your left mouse button. The material is now applied to the mesh.

10. However, we currently have a whole quarter of a city to texture. So, create more materials and use the remaining textures on the rest of the city. Create a new material for each extra texture, and four extra textures of `brick_002`, so that we can have different colors for each building's height.
11. Apply your new materials to the city, either by looking at the following screenshot or through your own artistic sensibility:



 When you are trying to get to the center fountain, if your tank is in the way, select your tank in the **Hierarchy** window and use the **Gizmo** in the **Scene** view to drag it out of the way.

If you were to try to play the game now, you might notice that we have a couple of problems. For starters, we only have a quarter of a city; perhaps you have more if you made your own city. In addition, there is still no collision on the city, so we will fall right through it when we move.

12. Changing the size of our tank is pretty simple. Select it in the **Hierarchy** window and look for the **Scale** label in our **Transform** component. Changing the **X**, **Y**, and **Z** values under **Scale** will change the size of our tank. Be sure to change them evenly or some weirdness will occur when we start rotating the tank. Values of `0.5` make the tank small enough to fit through the small streets.

13. Next up is collision for the city. For the most part, we will be able to get away with simple collision shapes that are faster to process. However, the circular center of the city will require something special. Start by double-clicking on the walls of one of the square buildings in the **Scene** view.

 When dealing with prefabs, which the city still is, clicking on any object that makes up the prefab will select the root prefab object. Once a prefab is selected, clicking on any part of it will select that individual piece. Because this behavior is different from non-prefab objects; you need to be mindful of this when you select objects in the **Scene** view.

14. With a set of walls selected, go to the top of the Unity Editor and select **Component**, followed by **Physics**, and finally select **Box Collider**.
15. As we are adding the collider to a specific mesh, Unity does its best to automatically fit the collider to the shape. For us, this means that the new **BoxCollider** component is already sized to fit the building. Continue by adding **BoxCollider** components to the rest of the square buildings and the outer wall. Our streets are essentially just a plane, so a **BoxCollider** component will work just fine for them as well. Though it is pointed at the top, the obelisk at the center of the fountain is essentially just a box; so another **BoxCollider** will suit it fine.
16. We have one last building and the fountain ring to deal with. These are not boxes, spheres, or capsules. So, our simple colliders will not work. Select the walls of the last building, the one next to the center fountain. A few options down from where you selected **Box Collider**, there is a **Mesh Collider** option. This will add a **MeshCollider** component to our object. This component does what its name suggests; it takes a mesh and turns it into a collider. By adding it to a specific mesh, the **MeshCollider** component automatically selects that mesh to be "collideable". You should also add **MeshCollider** components to the short ledge around the center building and the ring wall around the fountain.
17. The last problem to solve is the duplication of our city quarter. Start by selecting the root city object in your **Hierarchy** window, select **TankBattleCity**, and remove the **Animator** component from it. The city is not going to animate, so it does not need this component.
18. Now, right-click on the city in the **Hierarchy** window and click on **Duplicate**. This creates a copy of the object that was selected.

19. Duplicate the city quarter two more times and we will have the four parts of our city. The only problem is that they will all be in the exact same position.
20. We need to rotate three of the pieces to make a full city. Select one and set the value of **Y Rotation** in the **Transform** component to 90. This will rotate it 90 degrees around the vertical axis and give us half of a city.
21. We will complete the city by setting one of the remaining pieces to 180 and the other to 270.
22. That leaves one last thing to do. We have four center fountains. In three of the four city pieces, select the three meshes that make up the center fountain (the Obelisk, Wall, and Water) and hit the *Delete* key on your keyboard. Confirm that you want to break the prefab connection each time, and our city will be complete, as shown in the following figure:



Try out the game now. We can drive the tank around the city and rotate its turret. This is so much fun. We created materials and textured the city, and after making it possible for the player to collide with the buildings and road, we duplicated the section so that we could have a whole city.

Now that you have all the skills needed to import meshes and create materials, the challenge is to decorate the city. Create some rubble and tank traps and practice importing them to Unity and setting them up in the scene. If you really want to go above and beyond, try your hand at creating your own city; choose something from the world, or do something using your imagination. Once it is created, we can release the tanks in it.

Moving treads

There is just one thing left to do, and then we will be done with materials and can go on to make the game even more fun. Remember the **Offset** value of the materials? It turns out that we can actually control it with a script. Perform these steps to make the treads move with our tank:

1. Start by opening the `ChassisControls` script.
2. First, we need to add a few variables at the beginning of the script. The first two will hold references to our tank tread renderers, the part of the mesh object that keeps track of the material that is applied to the mesh and actually does the drawing. This is similar to how the `characterControl` variable holds a reference to our `CharacterController` component:

```
public Renderer rightTread;  
public Renderer leftTread;
```

3. The next two variables will keep track of the amount of offset applied to each tread. We store it here because this is a faster reference than trying to look it up from the tread's material for each frame.

```
private float rightOffset = 0;  
private float leftOffset = 0;
```

4. To make use of the new values, the following lines of code need to be added at the end of the `MoveTank` function. The first line here adjusts the offset for the right tread as per our speed, and keeps in time with our frame rate. The second line utilizes the material value of a `Renderer` component to find our tank's tread material. The `mainTextureOffset` value of the material is the offset of the primary texture in the material. In the case of our diffuse materials, this is the only texture. Then, we have to set the offset to a new `Vector2` value that will contain our new offset value. `Vector2` is just like `Vector3`, which we used for moving around, but it works in 2D space instead of 3D space. A texture is flat; therefore, it is a 2D space. The last two lines of the code do the same thing as the other two, but for the tank's left tread instead:

```
rightOffset += speed * Time.deltaTime;  
rightTread.material.mainTextureOffset = new Vector2(rightOffset,  
0);
```

```
leftOffset += speed * Time.deltaTime;  
leftTread.material.mainTextureOffset = new Vector2(leftOffset, 0);
```

5. To make the connections to the Renderer components of our treads, do the same thing that we did for the pivot points: drag the tread meshes from the **Hierarchy** window to the corresponding value in the **Inspector** window. Once this is done, be sure to save it and try it out.

We updated our `ChassisControls` script to make the tank's treads move. As the tank is driven around, the textures pan in the appropriate direction. This is the same type of functionality that is used to make waves in water and other textures that move.

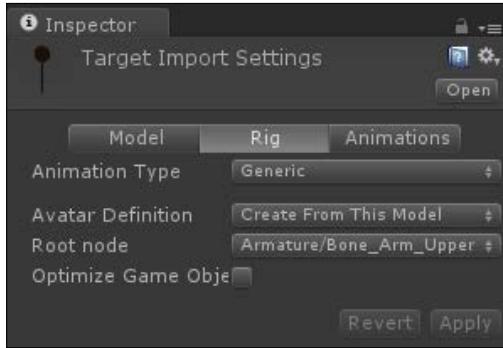
The movement of the material doesn't quite match the speed of the tank. Figure out how to add an extra speed value for the tank's treads. In addition, it would be cool if they moved in opposite directions when the tank is rotating. Real tanks turn by making one tread go forward and the other back.

Animations in Unity

The next topic that we will be covering is animation. As we explore animations in Unity, we will create some targets for our tank to shoot at. Much of the power of Unity's animation system, **Mecanim**, lies in working with humanoid characters. But, setting up and animating human type characters could fill an entire book in itself, so it will not be covered here. However, there is still much that we can learn and do with Mecanim.

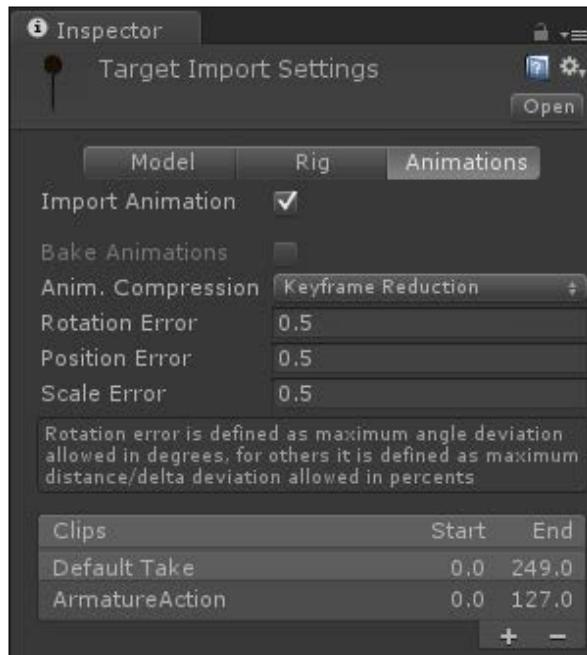
The following bullet points will explain all of the settings that are available for importing animations:

- Before we continue with the explanation of the animation import settings, we need an animated model to work with. We have one last set of assets to import to our project. Import the `Target.blend` and `Target.png` files into the `Targets` folder of our project. Once they are imported, adjust the **Import Settings** window on the **Model** page for the target, just as we did for the tank. Now, switch to the **Rig** tab (as shown in the following screenshot):



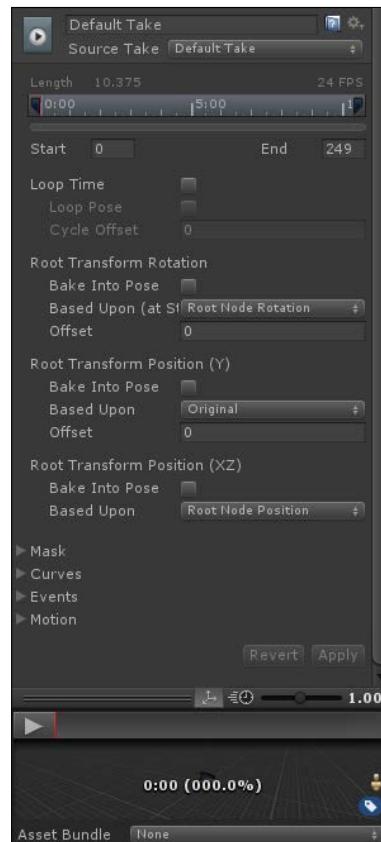
- The **Animation Type** attribute tells Unity what type of skeleton the current model is going to use when animation is going to be done. Models with different types are unable to share animations. The different options under **Animation Type** are as follows:
 - The **Humanoid** option adds many more buttons and switches to the page for working with human type characters. But again, this is too complex to cover here.
 - A **Generic** rig still uses Mecanim and many of its features. In reality, this is just any animation skeleton that does not resemble a human in structure.
 - The third option, **Legacy**, utilizes Unity's old animation system. However, this system will be phased out over the next few versions of Unity, so this will not be covered here either.
 - The last option, **None**, indicates that the object will not animate. You could select this option for both the tank and the city because it also keeps Unity from adding the Animator component, and saves space in the final project's size.
- The **Root Node** value is a list of every object that is in the model file. Its purpose is to select the base object of your animation rig. For this target, select **Bone_Arm_Upper**, which is underneath the second **Armature** option.

- The **Optimize Game Object** option will hide the whole skeleton of your model when it is checked. Hitting the plus sign on the new box that appears will allow you to select specific bones, which you still want access to when you view the model in the **Hierarchy** window. This is an especially useful option when dealing with any complex rig that has a great many bones.



- The last tab of the import settings, **Animations**, contains everything that we need to get the animations from our files into Unity. At the top of the **Target Import Settings** window, we have the **Import Animation** checkbox. If an object is not going to animate, it is a good idea to turn this option off. Doing so will also save space in your project.
- The option below that, **Bake Animations**, is only used when your animations contain kinematics and are from 3ds Max or Maya. This target is from Blender, so the option is grayed out.
- The next four options, **Anim. Compression**, **Rotation Error**, **Position Error**, and **Scale Error**, are primarily used for smoothing jittery animations. Nearly all the time, the defaults will be just fine for use.

- The **Clips** section is what we are really concerned about here. This will be a list of every animation clip that is currently being imported from the model. On the left-hand side of the list, we have the name of the clip. On the right-hand side, we can see the start and end frames of the clip. The various parameters under the **Clips** section are as follows:
 - Unity will add a default animation to every new model. This is a clip generated from the default preview range of your modeling program when the file was saved. In the case of our target, this is **Default Take**.
 - In Blender, it is also possible to create a series of actions for each rig. By default, they are imported by Unity as animation clips. In this case, the **ArmatureAction** clip is created.
 - Below and to the right-hand side of the clips, there is a little tab with the + and - buttons. These two buttons add a clip to the end and remove the selected clip respectively.



- When a clip is selected, the next section appears. It starts with a text field for changing the name of the clip.
- Below the text field, when working with Blender, there is a **Source Take** drop-down list. This list is the same as the default animations. Most of the time, you will just use **Default Take**; but, if your animation is forever appearing wrong or is missing, try changing the **Source Take** drop-down list first.
- Then, we have a small timeline, followed by input fields for the **Start** and **End** frames of the animation clip. Clicking on the two blue flags and dragging them in the timeline will change the numbers in the input fields.
- Next, we have **Loop Time**, **Loop Pose**, and **Cycle Offset**. If we want our animation to repeat, check the box next to **Loop Time**. **Loop Pose** will cause the positions of the bones in the first and last frames of the animation to match. When an animation is looping, **Cycle Offset** will become available. This value lets us adjust the frame on which the looping animation starts.
- The next three small sections, **Root Transform Rotation**, **Root Transform Position (Y)**, and **Root Transform Position (XZ)**, allow us to control the movement of a character through the animation. The controls under these sections are as follows:
 - All three of these sections have a **Bake into Pose** option. If these are left unchecked, the movement of the root node (we selected it under the **Rig** page) within the animation is translated into movement of the whole object. Think of it like this: say you were to animate a character running to the right inside the animation program, you will actually move them rather than animating in place as normal.
 - With Unity's old animation system, for the physical part of a character to move the collider, the GameObject had to be moved with the code. So, if you were to use the animation, the character would appear as if it had moved, but it would have no collision. With this new system, the whole character will move when the animation is played. However, this requires a different and more complex setup to work completely. So, we did not use this on the tank, though we could have used it.
 - Each of the three sections also has a **Based Upon** drop-down option. The choice of this option dictates the object's center for each of the sections. There are more choices if you are working with humanoid characters, but for now we only have two. A choice of **Root Node** means the pivot point of the root node object is the center. A choice of **Original** means that the origin, as defined by the animation program, is the center of the object.

- There is also an **Offset** option for the first two of these sections that works to correct errors in the motion. When animating a walk cycle for a character, if the character is pulling to the side slightly, adjusting the **Offset** option under **Root Transform Rotation** will correct it.
- The next option for our animation clip is a **Mask**. By clicking on the arrow to the left, you can expand a list of all objects in the model. Each object has a checkbox next to it. The objects that are not checked will not be animated when this clip is played. This is useful in the case of a hand-waving animation. Such an animation would only need to move the arm and hand, so we would uncheck all of the objects that might make up the body of the character. We could then layer animations, making our character capable of waving while standing, walking, or running without the need to create three extra animations.
- The **Curves** option will give you the ability to add a float value to the animation, which will change over the course of the animation. This value can then be checked by your code while the animation plays. This could be used to adjust the gravity affecting your character while they jump, change the size of their collider as they crouch into a ball, or do a great many other things.
- Events work similar to how we used the `SendMessage` function in our `RepeatButton` script. At a specific point in your animation, a function can be called to perform some action.
- The **Motion** option lets you define which bone in your animation controls the model motion. This can override the one chosen on the **Rig** tab. Our target does not move, so it is not particularly relevant to our situation.
- Finally, we have our **Revert** button, **Apply** button, and the **Preview** window at the bottom. Just as with all of our other import settings, we have to hit one of these buttons when changes are made. This **Preview** window is made special by the speed slider in the top-right corner and the big play button in the top-left corner. By clicking on this button, we can preview the selected animation. This lets us detect the errors in motion that we discussed earlier, and it generally makes sure that the animation is what we want it to be.

There are a lot of settings that are available to us when we are working with animations in Unity. They let us control the frames from the original animation program that we want to import. In addition, they can be used to control how the animation interacts with your scripts. No matter what settings you choose, the most important thing is the animation clip's name. If this is not set, it can be extremely difficult to work with several animations that have the same name.

The target's animations

So, now that the description is all out of the way, let's actually make something with it. We will start by setting up the animations for the target. Using the knowledge that we just gained, we can now set up our target's animations as follows:

1. For starters, if you missed or skipped it earlier, be sure to import the `Target.blend` and `Target.png` files to the `Targets` folder. In addition, on the **Rig** page of the import settings, ensure that the **Animation Type** attribute is set to **Generic** and the **Root Node** attribute is set to **Bone_Arm_Upper**.
2. We need a total of six animations. By clicking on the **+** button in the **Clips** section, you can add four more animations. If you have added too many, click on the **-** button to remove the extra clips.
3. All of these clips should have a **Source Take** drop-down list of **Default Take** and all of the **Bake into Pose** options should be checked because the target will not move from its starting location.
4. First, let's create our idle animations. Select the first clip and rename it as `Idle_Retract`. As this is a mechanical object, we can get away with a really short animation; it is so short that we are just going to use the first frame. Set the starting frame to `0 . 9` and the ending frame to `1`.
5. We also need to turn on **Loop Pose** because idle animations are, of course, looping.
6. The extended idle animation is created in almost exactly the same manner. Select the second clip and rename it as `Idle_Extend`. The starting frame here is `14` and the ending frame is `14 . 1`. In addition, this animation needs to loop.
7. Our next two animations are for a situation when the target extends and retracts. They will be called `Extend` and `Retract`, so rename the next two clips. The `Extend` animation will start at frame `1` and end at frame `13`. The `Retract` animation will start at frame `28` and ends at frame `40`. Neither of these will loop.
8. The last two animations also will not loop. They are for when we shoot the targets. There is one for being shot in the front and one for being shot from behind. The `Hit_Front` animation will be from frame `57` to frame `87`. The `Hit_Back` animation will be from frame `98` to frame `128`.
9. Once all of the changes are made, make sure to click on **Apply** or they will not be saved.

We have now set up the animations that will be used by our targets. There are six in total. They may not seem like much now, but the next section would not be possible without them.

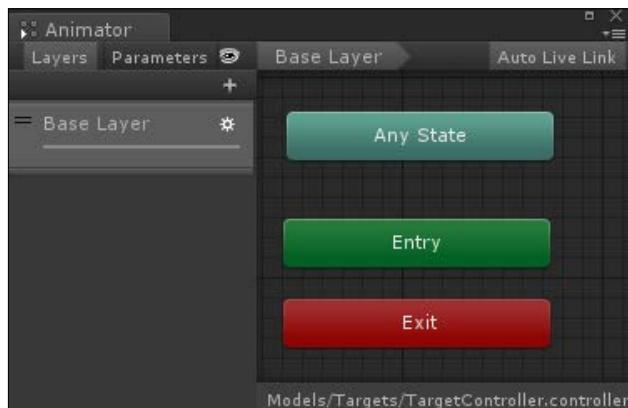
State machines to control animations in Unity

In order for us to control these new animations in Unity, we need to set up a state machine. A state machine is just a fancy object that keeps track of what an object can do, and how to transition between things. You can think of it in terms of a builder from a real-time strategy game. The builder has a walk state that is used when moving to the next construction site. When the builder gets there, it switches to a build state. If an enemy shows up, the builder will enter a runaway state until the enemy is gone. Finally, there is an idle state for when the builder does nothing. In Unity, these are called Animator controllers when you work with animations and Mecanim.

Target state machine

The use of a state machine allows us to focus more on what the target is doing, while letting Unity handle the *how it is going to do it* part. Perform these steps to create the state machine and control the target:

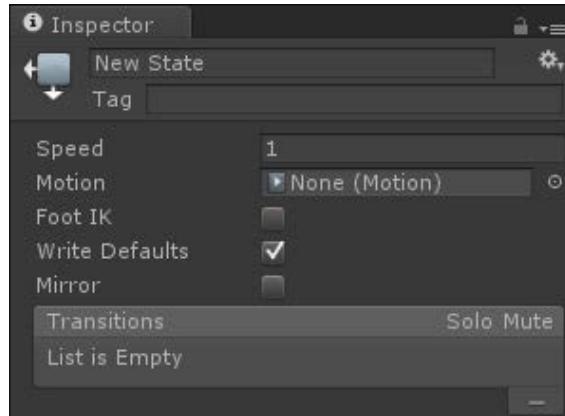
1. Creating an **Animator** controller is simple and this is done just as we have been doing for our scripts and materials. The option is approximately in the middle of the **Create** menu. Create an Animator controller in the Targets folder and name it `TargetController`.
2. Double-click on `TargetController` to open a new window (as shown in the following screenshot):



The **Animator** window is where we edit our state machines. The various parts of the **Animator** window are as follows:

- In the top-left side is a **Layers** button. Clicking on it will display a list of all of the blendable layers that make up your animation system. Every state machine will have at least the **Base Layer**. Adding more layers would allow us to blend state machines. Let's say we have a character that walks around normally when he is at full health. When his health drops below half, he starts to limp. If the character has only ten percent of his health left, he starts to crawl. This would be achieved through the use of layers to prevent the need of creating extra animations for each type of movement.
 - To the right of that is a **Parameters** button that will display the list of parameters. Clicking on the + button will add a new parameter to the list. These parameters can be **Float**, **Int**, **Bool**, and **Trigger**. The transitions between the states are most often triggered by changes in these parameters. Any scripts working with the state machine can modify these values.
 - The next bit is a breadcrumb trail, like one that you might find on a website. It lets us see where we are in the state machine at a glance.
 - The **Auto Live Link** button at the top-right corner controls our ability to see the state machine's update in real time within the game. This is useful for debugging transitions and controls for the character.
 - In the center of the **Animator** window, there are three boxes: **Any State**, **Entry**, and **Exit**. (If you can't see them, click on the middle mouse button and drag on the grid to pan the view around.) These boxes are the base controls for your animation state machine. The **Any State** box will allow your object to transition into specific animations, no matter where in the state machine they may be, such as moving to a death animation irrespective of the action the player was performing. The **Entry** box is used when you first start your state machine. All of the transitions are analyzed and the first suitable and subsequent animation becomes the starting location. The **Exit** box is used primarily for substate machines and allows you to transition out of the group without a lot of extra convoluted connections.
3. To create a new state, right-click on the grid that is inside our **Animator** window. Hover your mouse over **Create State** and select **Empty**. This creates a new empty state for our state machine. Normally, new states are gray, but since this is the first state in our machine, it is orange, which is the color of the default state.

4. Every state machine will start in its default state. Click on the state to select it, and we can take a look at it in the **Inspector** window (as shown in the following screenshot).

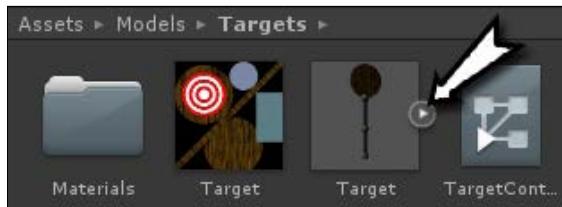


You can see the following fields in the preceding screenshot:

- At the top, there is a text field for changing the name of the state.
- Below that, you can add a **Tag** for organizational purposes.
- Next, there is a **Speed** field. This field controls the playback speed of the animation.
- The **Motion** field is where we will add connections to the animation clips that we created earlier.
- The **Foot IK** option lets us decide whether we want to let a part of the animation to be calculated with **Inverse Kinematics (IK)**, which is the process of calculating how a chain of bones will be laid out based on the position of the target bone at the end. We did not set up any IK for these animations, so we do not need to worry about this option.
- With the **Write Defaults** option, we can control whether animated properties remain changed after the animation ends.
- The last option, **Mirror**, is used to flip the left and right axis (or *x* axis) of the animation. If you created a right-hand-waving animation, this option would let you change it to a left-hand-waving animation.
- Below that, there is the list of transitions that go from the current state to another state. These are transitions that are out of the state and not into it. As you will soon see, a transition in this list appears as the name of the current state with an arrow to the right, followed by the name of the state it is connected to.

- Checkboxes also appear under the **Solo** and **Mute** labels on the right. These are for debugging transitions between the states. Any number of transitions can be muted at one time, but only one can be soloed at a time. When a transition has been muted, it means that the state machine will ignore it when deciding which transition to make. Checking the **Solo** box is the same as muting all but one of the transitions; this is just a quick way to make it the only active transition.

5. We are going to need one state for each of our target's animations. So, create five more states and rename all six to match the names of the animation clips that we created earlier. The default state, the first one you created that will appear orange on your screen, should be named `Idle_Retract`.
6. In the **Project** window, click on the little triangle on the right side of the **Target** model (as highlighted in the following screenshot):

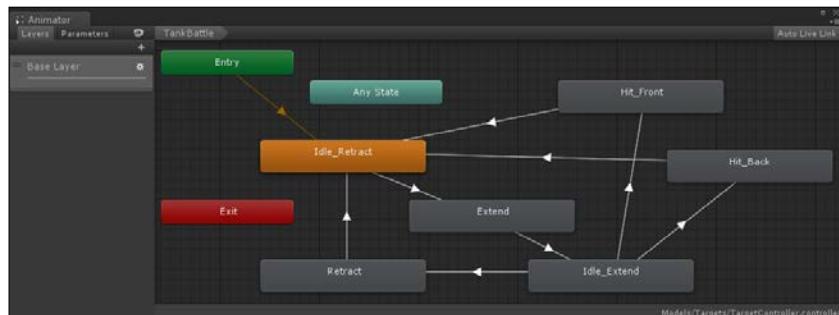


This expands the model so that we can see all of the objects that make up that model in Unity. The first group consists of the actual objects that make up the model. Next are the raw meshes that are used in the model. These are followed by the animation clips (they will appear on your screen as a blue box with a big play button at the center); these are what we are interested in right now. Last is an avatar object; this is what keeps track of the **Rig** setup.

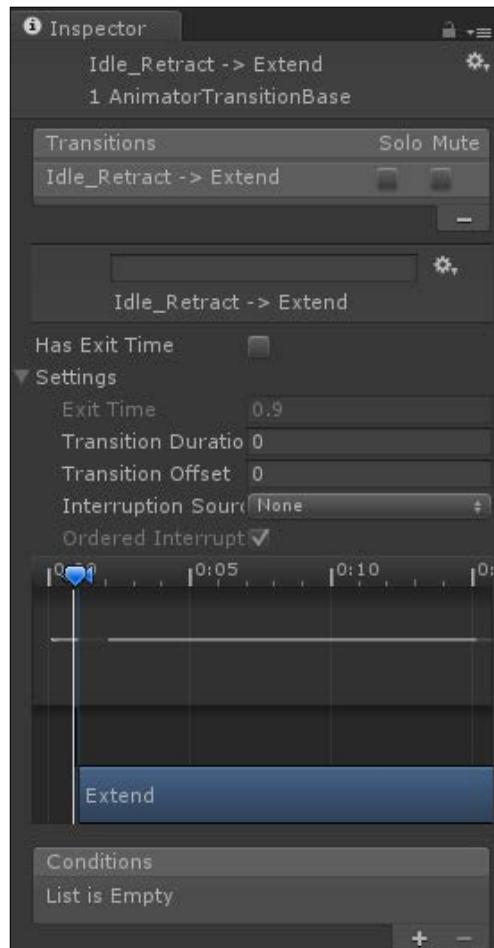
7. Select each state in your **Animator** window and pair it with the correct clip by dragging an animation clip from the **Project** window, and dropping it onto the **Motion** field in the **Inspector** window.
8. Before we can create our transitions, we need a few parameters. Open the parameters list by clicking on the **Parameters** button in the top-left corner. Then, click on the **+** button and select **Float** from the menu that appears. A new parameter should now appear in the list.
9. The new field on the left-hand side is the name of the parameter; it can be renamed at any time by double-clicking on it. Rename this one to `time`. The field on the right is the current value of this parameter. When debugging our state machine, we can modify these values here to trigger changes in the state machine. Any changes made by the scripts while the game is running will also appear here.

10. We need two more parameters. Create two **Bool** parameters and rename them as `wasHit` and `inTheFront`. These will trigger the machine to change to the getting hit states, while the time parameter will trigger the machine to utilize the `extend` and `retract` states.
11. To create a new transition, right-click on a state and select **Make Transition** from the menu that pops up. A transition line is now connected from the state to your mouse. To complete the transition creation, click on the state that you wish to connect to. There will be an arrow on the line, indicating the direction of the transition. We need the following transitions:
 - We need a transition from `Idle_Retraction` to `Extend`.
 - We also need a transition from `Extend` to `Idle_Extend`.
 - `Idle_Extend` needs three transitions, one going to `Retraction`, the other to `Hit_Front`, and the last to `Hit_Back`.
 - The `Retraction`, `Hit_Front`, and `Hit_Back` animations need a transition that goes to `Idle_Retraction`.

Use the following screenshot as a reference. If you create a transition or state that you do not want, select it and hit the *Delete* key on your keyboard to remove it.



12. If you click on one of the transition lines, then we can take a look at its settings (as shown in the following screenshot):



You can see the following things in the screenshot:

- At the top of the **Inspector** window, we have the same indicators of which states we are transitioning between that we had in the state – the name of the state the transition starts in, followed by an arrow, and finally the name of the state the transition ends in.

- Underneath the familiar **Transitions** list, there is a text field where we can give our transitions specific names. This is useful if we have several different types of transitions between the same two states.
- The **Has Exit Time** checkbox dictates whether the transition will wait until the animation is close to its end before changing to the next animation. This is good for things like smoothly transitioning between walk and idle animations.
- The first value in **Exit Time** under **Settings** sets when the transition would start. This is only relevant when the checkbox above it is checked. It should have a value from zero to start the animation, and to one to end the animation.
- The **Transition Duration** setting defines how long the transition will take. It again takes a value between zero and one.
- The **Transition Offset** setting defines where in the target animation the transition will begin.
- The **Interruption Source** and **Ordered Interruption** options determine whether another transition can occur while it is in the process of going through this one. They also set which set of transitions will have precedence and in which order they will be processed.
- Next is a timeline block that lets us preview the transition between animations. By dragging the little flag left and right, we can watch the transition in the **Preview** window. The top half of this block holds waveforms that indicate the movement contained in an animation. The bottom half shows the states as boxes that overlap where the transition actually occurs. Either one of these boxes can be dragged to change the length of the transition.



Since our two idle animations are of negligible length, this normally can't easily be seen in our setup. If you create a temporary transition between the `extend` and `retract` states, it would be visible.

- Lastly, we have a **Conditions** list. Using the parameters that we set up, we can create any number of conditions here that must be met before this transition can take place.



There is another **Preview** window at the bottom of the **Inspector** panel. It functions just like the one for the **Animation Import Settings** page, but this one plays the transition between the two relevant animations.

13. Select the transition between the `Idle_Retract` state and the `Extend` state. We want the targets to randomly pop up. This will be controlled by a script that will change the time parameter.
14. Click on the **+** under the **Conditions** list to add a new condition. Then, click on the arrow in the middle of the condition to select **time** from the list of parameters.
15. In order to turn a **Float** value into a conditional statement, we need to compare it with another value. That is why we got a new drop-down button with comparison options when we selected the parameter. A **Float** value will be either greater than or less than the value on the right. Our time will be counting down, so select **Less** from the list and leave the value as zero.
16. Add a condition so that the transition between the `Idle_Extend` and `Retract` states will be the same.
17. For the transition between the `Idle_Extend` state and the `Hit_Front` state, we will use both the **Bool** parameters that were created. Select the transition and click on the **+** button under **Conditions** until you have two conditions.
18. For the first condition, select **wasHit**, and select **inTheFront** for the second condition. A **Bool** parameter is either `true` or `false`. In the case of transitions, it needs to know which of the values it is waiting for. For this transition, both should be left as `true`.
19. Next, set up the conditions for the transition between `Idle_Extend` and `Hit_Back`, just as you did for the previous transition. The only difference is that `false` needs to be selected from the drop-down list next to the **inTheFront** conditional.

Here, we created a state machine that will be used by our targets. By linking each state to an animation and connecting all of them with transitions, the target will be able to switch between animations. This transitioning is controlled by adding conditionals and parameters.

Scripting the target

We only need one more piece before we can finish putting the target together – a script:

1. Create a new script in our `Scripts` folder and name it `Target`.
2. First, in order to interact with our state machine, we need a reference to the `Animator` component. It is the component that you removed from the tank and the city. The `Animator` component is what ties all of the pieces of the animation together:

```
public Animator animator;
```

3. This is followed by two float values that will dictate the range of time, in seconds, during which our targets will sit in their idle states:

```
public float maxIdleTime = 10f;  
public float minIdleTime = 3f;
```

4. Next, we have three values that will hold the ID numbers of the parameters that we need to change. It is technically possible to just use the names of the parameters to set them, but using the ID number is much faster:

```
private int timeId = -1;  
private int wasHitId = -1;  
private int inTheFrontId = -1;
```

5. The last two variables will hold the ID numbers of the two idle states. We need these for checking which state we are in. All of the IDs are initially set to `-1` as a dummy value; we set them to their actual values with the function in the next step:

```
private int idleRetractId = -1;  
private int idleExtendId = -1;
```

6. The `Awake` function is a special function in Unity that is called on every script at the beginning of the game. Its purpose is initialization before the game gets underway, and it is perfect for initially setting our ID values.

```
public void Awake() {
```

7. For each ID, we make a call to the `Animator.StringToHash` function. This function calculates the ID number of the parameter or state whose name we give it. The state names also needs to be prefixed with `Base Layer`. This is because Unity wants us to be specific when it is possible to have several different layers with states that are named the same. It is also very important that the name here exactly matches the name in the **Animator** window. If it does not, IDs will not match, errors will occur, and the script will not function correctly.

```
timeId = Animator.StringToHash("time");
wasHitId = Animator.StringToHash("wasHit");
inTheFrontId = Animator.StringToHash("inTheFront");
idleRetractId = Animator.StringToHash("Base Layer.Idle_Retract");
idleExtendId = Animator.StringToHash("Base Layer.Idle_Extend");
}
```

8. To make use of all of these IDs, we turn to our very good friend—the `Update` function. At the beginning of the function, we use the `GetCurrentAnimatorStateInfo` function to figure out which state is the current one. We send a zero to this function because it wants to know the index of the layer we are inquiring about, of which we only have one. The function returns an object with the information about the current state, and we grab the `nameHash` value (also known as the ID value) of this state right away and set our variable to it.

```
public void Update() {
    int currentStateId = animator.GetCurrentAnimatorStateInfo(0).nameHash;
```

9. The next line of code is a comparison with our idle state IDs to figure out whether we are in one of those states. If we are, we call upon the `SubtractTime` function (which we will write in a moment) to reduce the time parameter.

```
if(currentStateId == idleRetractId || currentStateId ==
idleExtendId) {
    SubtractTime();
}
```

10. If the target is not currently in one of its idle states, we start by checking to see whether we were hit. If so, the hit is cleared using the `ClearHit` function and the time parameter is reset using the `ResetTime` function. We will write both these functions in a moment. Finally, we check to see whether our timer has dropped below zero. If it has, we again reset the timer.

```
else {
    if(animator.GetBool(wasHitId)) {
        ClearHit();
        ResetTime();
    }

    if(animator.GetFloat(timeId) < 0) {
        ResetTime();
    }
}
```

11. In the `SubtractTime` function, we use the `GetFloat` function of our `Animator` component to retrieve the value of a float parameter. By sending our `timeId` variable to it, we can receive the current value of the time parameter. Like we did with the tank, we then use `Time.deltaTime` to keep pace with our frame rate and subtract time from the timer. Once this is done, we need to give the state machine the new value, which is done with the `SetFloat` function. We tell it which parameter to change by giving it an ID value, and we tell it what to change by giving it our new time value.

```
public void SubtractTime() {
    float curTime = animator.GetFloat(timeId);
    curTime -= Time.deltaTime;
    animator.SetFloat(timeId, curTime);
}
```

12. The next function to create is `ClearHit`. This function uses `SetBool` from the `Animator` component to set Boolean parameters. It functions just like the `SetFloat` function. We just give it an ID and a value. In this case, we set both of our Boolean parameters to `false` so that the state machine no longer thinks that it has been hit.

```
public void ClearHit() {
    animator.SetBool(wasHitId, false);
    animator.SetBool(inTheFrontId, false);
}
```

13. The last function for the script is `ResetTime`. This is another quick function. First, we use the `Random.Range` function to get a random value. By passing it a minimum and maximum value, our new random number will be between them. Finally, we use the `SetFloat` function to give the state machine the new value.

```
public void ResetTime() {
    float newTime = Random.Range(minIdleTime, maxIdleTime);
    animator.SetFloat(timeId, newTime);
}
```

We have created a script to control the state machine of our target. For comparing states and setting parameters, we gathered and used IDs. For now, do not worry about when the hit states are activated. It will be made clear in the following section when we finally make the tank shoot.

Creating the prefab

Now that we have the model, animations, state machine, and script, it is time to create the target and turn it into a prefab. We have all the pieces, so let's put them all together:

1. Start by dragging the **Target** model from the **Project** window to the **Hierarchy** window. This creates a new instance of the target object.
2. By selecting the new target object, we can see that it already has an **Animator** component attached to it; we just need to add a reference to the **AnimatorController** that we created. Do this by dragging **TargetController** from the **Project** window and dropping it onto the **Animator** component's **Controller** field, just like all the other object references that we have set up so far.
3. Also, we need to add the **Target** script to the object and connect a reference to the **Animator** component in its relevant field.
4. The last thing to do to the target object is to add a collider to actually receive our cannon shots. Unfortunately, since the **Target** object uses bones and a rig to animate, it is not as simple as adding a collider directly to the mesh at which we will be shooting. Instead, we need to create a new empty **GameObject**.
5. Rename this to **TargetCollider** and make it a child of the target's **Bone_Target** bone.
6. Add a **MeshCollider** component to the new **GameObject**.

7. Now, we need to provide this component with some mesh data. Find the **Target** mesh data in the **Project** window, underneath the **Target** model. Drag it to the **Mesh** value of the **MeshCollider** component. This causes a green cylinder to appear in the **Scene** view. This is our collision, but it is not yet aligned to the target.
8. Use the **Transform** component to set the **GameObject** position to 4 for the **X** value and 0 for both **Y** and **Z**. The rotation needs to be changed to 0 for **X**, -90 for **Y**, and 90 for **Z**.
9. As we made the changes, you probably noticed that the font of everything that was new or changed became bold. This is to indicate that something is different with this prefab instance as compared to the original. Remember, models are essentially prefabs; the problem with them is that we cannot directly make changes, such as adding scripts. To make this target into a new prefab, simply drag it from the **Hierarchy** window and drop it onto the **Prefabs** folder in the **Project** window.
10. After this spiffy new prefab is created, populate the city with it.
11. When you placed all of these targets, you probably noticed that they are a little large. Instead of editing each target individually or even all of them as a group, we only have to make a change to the original prefab. Select the **Target** prefab in the **Project** window. The **Inspector** window displays the same information for a root prefab object as it does for any other object in the scene. With our prefab selected, half the scale and all of the instances already in the scene will automatically be updated to match. We can also make changes to the min and max idle times and make it affect the whole scene.

We just finished creating the targets for our tank. By making use of Unity's prefab system, we can also duplicate the target throughout our game and easily make changes that affect them all.

If you wanted one of the targets to be larger than all of the others, you could change it in the scene. Any changes made to a prefab instance are saved, and they take precedence over changes made to the root prefab object. In addition, when you look at an instance in the **Inspector** window, there will be three new buttons at the top of the window. The **Select** button selects the root prefab object in the **Project** window. The **Revert** button will remove any unique changes made to this instance, whereas the **Apply** button updates the root object with all the changes that were made in this instance.

Using all that you have learned about animations and state machines, your challenge here is to create a second type of target. Play around with different movements and behaviors. You can perhaps create one that transitions from waving around to standing still.

Ray tracing to shooting

Play the game now; it is pretty cool. We have our drivable tank and textured city. We even have fancy animated targets. We are just missing one thing: how do we shoot? We need to make one more script and we can shoot targets to our heart's content. Follow these steps to create the script and set it up:

1. First, we need to add an empty **GameObject** to our tank. Rename it to **MuzzlePoint** and make it a child of the cannon's pivot point object. Once this is done, position it at the end of the cannon so that the blue arrow points away from the tank, along the same direction as the cannon. This will be the point where our bullets will come from.
2. We also need something to indicate where we are shooting. The explosions are covered in future chapters, so choose **Sphere** from the **3D Object** menu underneath **GameObject** and rename it to **TargetPoint**.
3. Set the sphere's scale to **0.2** for each axis and give it a red material. This way, it can be more easily seen without being completely obtrusive. It does not matter where it starts in our scene, our next script will move it around when we shoot.
4. Remove the **SphereCollider** component from **TargetPoint**. The **SphereCollider** has to be removed because we don't want to shoot our own target indicator.
5. Now, create a new script and call it **FireControls**.
6. This should start to look familiar to you. We start with variables to hold references to our muzzle and targeting objects that we just created.

```
public Transform muzzlePoint;  
public Transform targetPoint;
```

7. The **Fire** function starts by defining a variable that will hold the detailed information about what was shot:

```
public void Fire() {  
    RaycastHit hit;
```

8. It is followed by an **if** statement that checks the **Physics.Raycast** function. The **Raycast** function works just like shooting a gun. We start with a position (the muzzle point's position) pointing to a specific direction (forward relative to the muzzle point along that blue axis) and get out what was hit. If we hit something, the **if** statement evaluates to **true**; otherwise, it is **false** and we would skip ahead.

```
if(Physics.Raycast(muzzlePoint.position, muzzlePoint.forward, out  
hit)) {
```

9. When we do hit something, we first move our target point to the point that was hit. We then use the `SendMessage` function to tell what we hit that it has been hit, the same way we used it in our `RepeatButton` script earlier. We use `hit.transform.root.gameObject` to get at the `GameObject` that was hit. We also provide it with a value, `hit.point`, to tell the object where it was hit. The `SendMessageOptions.DontRequireReceiver` part of the line keeps the function from throwing an error if it is unable to find the desired function. Our targets have the function, but the city walls do not and they would throw an error.

```
targetPoint.position = hit.point;
hit.transform.root.gameObject.SendMessage("Hit", hit.point,
SendMessageOptions.DontRequireReceiver);
}
```

10. The last part of our `Fire` function occurs if we didn't hit anything. We send our target point back to the world origin so that the player knows that they missed everything:

```
else {
    targetPoint.position = Vector3.zero;
}
}
```

11. The last thing to add is the `Hit` function at the end of our `Target` script. We start the function by getting the current state ID, just as we did earlier in the script. However, this time we only check against our extended idle ID. If they do not match, we use `return` to exit the function early. We do this because we don't want to let the player shoot any targets that are down or in mid-transition. If our state is correct, we continue by telling the animation that we were hit by using the `SetBool` function:

```
public void Hit(Vector3 point) {
    int currentStateId = animator.GetCurrentAnimatorStateInfo(0).nameHash;
    if(currentStateId != idleExtendId) return;
    animator.SetBool(wasHitId, true);
```

12. The rest of the `Hit` function figures out on which side the target was hit. To do this, we first have to convert the point that we received from world space into local space. The `InverseTransformPoint` function from our **Transform** component does this nicely. We then do a check to see where the shot came from. Due to the way that the target is constructed, if the shot was positive on the *x* axis, it came from behind. Otherwise, it came from the front. Either way, we set the `inTheFront` parameter from our state machine to the proper value. Then, we give the player some points by incrementing the static variable that we created in our `ScoreCounter` script, way back at the beginning of the chapter:

```
Vector3 localPoint = transform.InverseTransformPoint(point);
if(localPoint.x > 0) {
    animator.SetBool(inTheFrontId, false);
    ScoreCounter.score += 5;
}
else {
    animator.SetBool(inTheFrontId, true);
    ScoreCounter.score += 10;
}
```

13. Next, we need to add the new `FireControls` script to the tank. You also need to connect the references to the `MuzzlePoint` and `TargetPoint` objects.
14. Finally, we need to create a new button to control and trigger this script. So, navigate to **GameObject | UI | Button** and rename the button to `Fire`.
15. Next, we need to hit the little plus sign in the bottom right of the button's **Inspector** window and select `Tank` for the **Object** slot, exactly like we did for our Tic-tac-toe game. Then, navigate to **FireControls | Fire ()** from the function drop down.

We have created a script that allows us to fire the cannon of our tank. The method of using ray tracing is the simplest and most widely used. In general, bullets fly too fast for us to see them. Ray tracing is like this, that is, it is instantaneous. However, this method does not take gravity, or anything else that might change the direction of a bullet, into account.

Now that all of the buttons and components are in place, make them look better. Use the skills you gained from the previous chapter to style the GUI and make it look great. Perhaps you could even manage to create a directional pad for the movement.

Summary

And, that is it! The chapter was long and we learned a lot. We imported meshes and set up a tank. We created materials so that color could be added to a city. We also animated some targets and learned how to shoot them down. It was a lot and it is time for a break. Play the game, shoot some targets, and gather those points. The project is all done and ready to be built in your device of choice. The build process is the same as both the previous projects, so have fun!

The next chapter is about special camera effects and lighting. We will learn about lights and their types. Our Tank Battle game will expand through the addition of a skybox and several lights. We will also take a look at distance fog. With the addition of shadows and lightmaps, the city in which we battle really starts to become interesting and dynamic.

4

Setting the Stage – Camera Effects and Lighting

In the previous chapter, you learned about the basic building blocks of any game: meshes, materials, and animations. We created a *Tank Battle* game that utilized all of these blocks.

In this chapter, we will expand upon the Tank Battle game. We will start with the addition of a skybox and distance fog. The exploration of camera effects continues with a target indicator overlay that uses a second camera. The creation of a turbo boost effect for the tank will round out our look at camera effects. Continuing with a look at lighting, we will finish off our tank environment with the addition of lightmaps and shadows.

In this chapter, we will cover the following topics:

- Skyboxes
- Distance fog
- Using multiple cameras
- Adjusting the field of view
- Adding lights
- Creating lightmaps
- Adding cookies

We will be directly piggybacking off the project from *Chapter 3, The Backbone of Any Game – Meshes, Material, and Animations*. So, open the project in Unity and we will get started.

Camera effects

There are many great camera effects that you should add in order to give your game the last great finishing touch. In this chapter, we will be covering a few options that are easy to add. These will also give our tank game a finished look.

Skyboxes and distance fog

When a camera renders the frame of a game, it starts by clearing the screen. The default camera in Unity does this by coloring everything with a gradient, simulating the look of a skybox. All of the game's meshes are then drawn on top of this blank screen. While the gradient looks better than a solid color, it is still rather boring for an exciting battle of tanks. Luckily for us, Unity allows us to change the skybox. A skybox is just a fancy word for the series of images that form the background sky of any game. Distance fog works in conjunction with the skybox by easing the visual transition between models and the background.

The very first thing we need is a new skybox. We can create our own, however, Unity provides us with several excellent ones that will fit our needs just fine. Let's use the following steps to get a skybox now:

1. At the top of the Unity Editor, select **Assets** and then click on **Import Package**. About halfway down this list, select **Skyboxes**.
2. After a little bit of processing, a new window will pop up. A package in Unity is just a compressed group of assets that have already been set up in Unity. This window displays the contents and allows you to selectively import them. We want them all, so we just click on **Import** in the bottom-right corner of this window.
3. A new folder, **Standard Assets**, will be added to the **Project** window. This contains a folder, **Skyboxes**, which contains various skybox materials. Select any one of these. You can see in the **Inspector** window that they are normal materials that make use of the skybox shader. They each have six images, one for each direction of a box.
4. You will also notice that there are warning messages with a **Fix Now** button under each image. This is because all the images were compressed to save import time and space, but the skybox shader needs them in a different format. Just click on the **Fix Now** button each time and Unity will automatically fix it for you. It will also get rid of all of the odd blackness in the material preview.

5. To add a skybox of your choice to the game, first make sure that you have the correct scene loaded. If you do not, simply double-click on the scene in the **Project** window. This is necessary because the settings we are about to change are specific to each scene.
6. Go to the top of the Unity Editor and select **Edit** and then click on **Scene Render Settings**. The new group of settings will appear in the **Inspector** window.
7. At the moment, we are concerned with the value at the top, **Skybox Material**. Just drag and drop the new skybox material into the **Skybox Material** slot and it will be automatically updated. The change can be viewed right away in the **Game** and **Scene** windows.
8. To add distance fog, we also adjust this setting in **Scene Render Settings**. To turn it on, simply tick the **Use Fog** checkbox.
9. The next setting, **Fog Color**, allows you to pick a color for the fog. It is good to pick a color that is close to the general color of the skybox.
10. The **Fog Mode** setting is a drop-down list of options that dictate the method that Unity will use to calculate the distance fog. For nearly all cases, the default setting of **Exponential Squared** is suitable.
11. The next three settings, **Density**, **Start**, and **End**, determine how much fog there is and how close it starts. They will only appear for the fog modes that use them. **Density** is used for the **Exponential** and **Exponential Squared** fog modes, while the others are used for the **Linear** fog mode. Settings that put the fog at the edge of sight will, in general, give the best-looking effect. Leave these settings on **Exponential Squared** and choose **0.03** for the **Density** in order to get a good look.



We have imported several skyboxes and added them to the scene. The distance fog settings are also turned on and adjusted. Now, our scene has started to look like a real game.

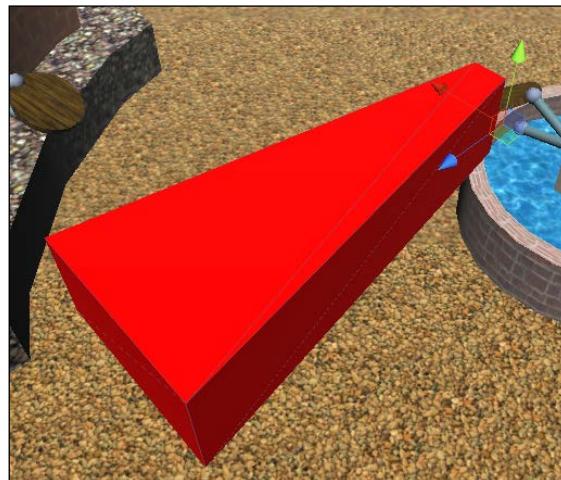
Target indicator

Another camera effect that is rather interesting is the use of multiple cameras. A second camera can be used to make a 3D GUI, a minimap, or perhaps a security camera popup. In the next section, we will be creating a system that will point at targets that are nearby. Using a second camera, we will make the indicators appear above the player's tank.

Creating the pointer

We are going to start by creating an object that will point at targets. We will be making a prefab that can be used repeatedly. However, you will need to import the `IndicatorSliceMesh.blend` starting asset for this chapter, so we have something for the player to see. It is a pie-slice-shaped mesh. Let's perform the following steps to create the pointer:

1. Once you have the mesh imported, add it to the scene.
2. Create an empty **GameObject** component and rename it to `IndicatorSlice`.
3. Make the mesh a child of `IndicatorSlice` and position it so that it points along the z axis of `GameObject`, with the small end of the pie slice being at the position of `IndicatorSlice`. The `IndicatorSlice` `GameObject` will be centered in our indicator. Each slice that is created will have its z axis pointing in the direction of a target, as shown in the following figure:



4. Now, we need to create a new script that will control our indicator. Create a new script called **TargetIndicator** in the **Project** window.
5. We start off this script with a pair of variables. The first variable will hold a reference to the target that this indicator piece will point at. The indicator is also going to grow and shrink, based on how far away the target is. The second variable will control the distance at which the indicator will start to grow:

```
public Transform target;  
public float range = 25;
```

6. The next function will be used to set the `target` variable when the indicator piece is created:

```
public void SetTarget(Transform newTarget) {  
    target = newTarget;  
}
```

7. The last set of code goes in the `LateUpdate` function. The `LateUpdate` function is used so that the indicator pieces can point at a target after our tank moves in the `Update` function:

```
public void LateUpdate() {
```

8. We start the function by checking whether the `target` variable has a value. If it is null, the indicator slice is destroyed. The `Destroy` function can be used to remove any object that exists from the game. The `gameObject` variable is automatically provided by the `MonoBehaviour` class and holds a reference to the **GameObject** component that the script component is attached to. Destroying this component will also destroy everything that is a child of (or attached to) it:

```
if(target == null) {  
    Destroy(gameObject);  
    return;  
}
```

9. Next, we determine how far this indicator slice is from its target. By using `Vector3.Distance`, we can easily calculate the distance without doing the math ourselves:

```
float distance = Vector3.Distance(transform.position, target.  
position);
```

10. This line of code determines the vertical scale, *y* axis, of the slice. It does so by using a bit of carefully applied math and the `Mathf.Clamp01` function. This function limits the supplied value to be between zero and one:

```
float yScale = Mathf.Clamp01((range - distance) / range);
```

11. We use the calculated scale to set the indicator slice's local scale. By adjusting the local scale, we can easily control how big the whole indicator is just by changing the scale of the parent object:

```
transform.localScale = new Vector3(1, yScale, 1);
```

12. The transform.LookAt function is just a fancy, automatic way of rotating a GameObject so that its z axis points to a specific spot in the world. However, we want all the indicator slices to lie flat on the ground and not point into the air at any targets that might be above us. So, we first collect the target's position. By setting the variable's y value to the position of the slice, we ensure that the slice remains flat. That last line, of course, closes off the LateUpdate function:

```
Vector3 lookAt = target.position;  
lookAt.y = transform.position.y;  
transform.LookAt(lookAt);  
}
```

13. The preceding code is the last code for this script. Return to Unity and add the TargetIndicator script to the IndicatorSlice object in the scene.
14. To finish off the indicator, create a prefab of it. Do it just like we did for our target objects.
15. Lastly, delete the IndicatorSlice object from the scene. We will be creating slices dynamically when the game starts. This requires the prefab, but not the one in the scene.

We created a prefab of the object we will be using to indicate the direction of targets. The script that was created and attached will rotate each instance of the prefab to point at the targets in the scene. It will also adjust the scale to indicate how far away the targets are from the player.

Controlling the indicator

We now need to create a script that will control the indicator slices. This will include creating new slices as they are needed. Also, the **GameObject** component it is attached to will act as a center point for the indicator slices, which we just created, to rotate around. Let's perform these steps to do this:

1. Create a new script and name it **IndicatorControl**.

2. We start off this script with a pair of variables. The first variable will hold a reference to the prefab that was just created. This will allow us to spawn instances of the prefab whenever we desire. The second is a static variable, which means that it can be easily accessed without a reference to the component that exists in the scene. It will be filled when the game starts with a reference to the instance of this script that is in the scene:

```
public GameObject indicatorPrefab;  
private static IndicatorControl control;
```

3. The next function will be used by the targets. Soon, we will be updating the target's script to call this function at the beginning of the game. The function is static, just like the preceding variable:

```
public static void CreateSlice(Transform target) {
```

4. This function starts by checking whether there is a reference to any object in the static variable. If it is empty, equal to null, `Object.FindObjectOfType` is used to fill the variable. By telling it what type of object we want to find, it will search in the game and try to find one. This is a relatively slow process and should not be used often, but we use this process and the variable so that we can always be sure that the system can find the script:

```
if(control == null) {  
    control = Object.FindObjectOfType(typeof(IndicatorControl)) as  
    IndicatorControl;  
}
```

5. The second part of the `CreateSlice` function checks to make sure that our static variable is not empty. If so, it tells the instance to create a new indicator slice and passes the target to the slice:

```
if(control != null) {  
    control.NewSlice(target);  
}
```

6. There is one more function for this script: `NewSlice`. The `NewSlice` function does as its name implies; it will create new indicator slices when called:

```
public void NewSlice(Transform target) {
```

7. The function first uses the `Instantiate` function to create a copy of `indicatorPrefab`:

```
GameObject slice = Instantiate(indicatorPrefab) as GameObject;
```

8. Next, the function makes the new slice a child of the control's transform, so it will stay with us as we move around. By zeroing out the local position of the new slice, we also insure that it will be at the same location as our control:

```
slice.transform.parent = transform;  
slice.transform.localPosition = Vector3.zero;
```

9. The last line of the function uses the slice's SendMessage function to call the SetTarget function that we created previously and passes it the desired target object:

```
slice.SendMessage("SetTarget", target);  
}
```

10. Now that the script is created, we need to use it. Create an empty **GameObject** component and name it **IndicatorControl**.

11. The new **GameObject** component needs to be made a child of the tank, followed by having its position set to zero on each axis.

12. Add the script we just created to the **IndicatorControl** object.

13. Finally, with the GameObject selected, add the reference to the **IndicatorSlice** prefab. Do this by dragging the prefab from the **Project** window to the proper slot in the **Inspector** window.

We created a script that will control the spawning of our target indicator slices. The **GameObject** component we created at the end will also allow us to control the size of the whole indicator with ease. We are almost done with the target indicator.

Working with a second camera

If you were to play the game now, it will still look no different. This is because the targets do not make the call yet to create the indicator slices. We will also be adding the second camera in this section as we finish off with the target indicator. These steps will help us do it well:

1. Start by opening the **Target** script and adding the following line of code at the end of the **Awake** function. This line tells the **IndicatorControl** script to create a new indicator slice for this target:

```
IndicatorControl.CreateSlice(transform);
```



2. If you play the game now, you can see the indicator in action. However, it is probably too large and certainly appears inside the tank. A bad solution will be to move the `IndicatorControl` object until the whole thing appears above the tank. However, when explosions occur and things start flying through the air, they will obscure the target indicator all over again. A better solution is to add a second camera. You can do so now by selecting **GameObject** from the top of the Unity Editor and then clicking on **Camera**.
3. Additionally, make the camera a child of `Main Camera`. Be sure to set the new camera's position and rotation values to 0.
4. By default, every camera in Unity is given a variety of components: **Camera**, **Flare Layer**, **GUI Layer**, and **Audio Listener**. Besides the **Camera** component, the others are generally unimportant to every other camera, and there should only be one **Audio Listener** component in the whole of the scene. Remove the excess components from the camera, leaving just the **Camera** component.
5. Before we do anything else with the camera, we need to change the layer that the `IndicatorSlice` prefab is on. Layers are used for selective interaction between objects. They are used primarily for physics and rendering. First select the prefab in the **Project** window.
6. At the top of the **Inspector** window is the **Layer** label with a drop-down list that reads **Default**. Click on the drop-down list and select **Add Layer...** from the list.

7. A list of layers will now appear in the **Inspector** window. These are all the layers that are used in the game. The first few are reserved for use by Unity; hence, they have been grayed out. The rest are for our use. Click on the input box at the right-hand side of **User Layer 8** and name it **Indicator**.
8. Select the **IndicatorSlice** prefab again. This time, select the new **Indicator** layer from the **Layer** drop-down list.
9. Unity will ask whether you want to change the layer of all the child objects as well. We want the whole object rendered on this layer, so we need to select **Yes, change children** and we will be able to do so.
10. Now, let's get back to our second camera. Select the camera and take a look at the **Inspector** window.
11. The first attribute of the **Camera** component is **Clear Flags**. This list of options dictate what the camera will fill the background with before drawing all the models in the game. The second camera should not block out everything drawn by the first camera. We select **Depth only** from the **Clear Flags** drop-down list. This means that instead of putting the skybox in the background, it will leave what was already rendered and just draw new things on top.
12. The next attribute, **Culling Mask**, controls which layers are rendered by the camera. The first two options, **Nothing** and **Everything**, are for the quick deselection and selection of all the layers. For this camera, deselect all other layers so that only the **Indicator** layer has a check next to it.
13. The last thing to do is to adjust the scale of **IndicatorControl** so that the target indicator is not too large or small.



We created a system to indicate the direction of potential targets. To do this, we used a second camera. By adjusting the layers in the **Culling Mask** attribute, we can make a camera render only a part of the scene. Also, by changing the **Clear Flags** attribute to **Depth only**, the second camera can draw on top of what was drawn by the first camera.

It is possible to change where the indicator is drawn by moving the camera. If you were to move the `IndicatorControl` object instead, it will change how the distance from the targets and the directions to target are calculated. Move and angle the second camera so that there is a more pleasing view of the target indicator.

When you move the second camera or when you use the boost (from the next section), you will probably notice that the target indicator can still be seen in the tank. Adjust the main camera so that it does not render the target indicator. This is done similarly to how we made the second camera only render the target indicator objects.

Turbo boost

The last camera effect that we will be looking at in this chapter is a turbo boost. It is going to be a button on the screen that will propel the player forward rapidly for a short amount of time. The camera effect comes in because a simple adjustment to the **Field of View** attribute can make it look as if we are going much faster. A similar method is used in movies to make car chases look even faster than they are.

We will only be making a single script in this section. The script will move the tank in a similar manner to the `ChassisControls` script we created in the last chapter. The difference is that we won't have to hold down a button for the boost to work. Let's get to it with these steps:

1. Start by creating a new script and calling it `TurboBoost`.
2. To start off the script, we need four variables. The first variable is a reference to the `CharacterController` component on the tank. We need this for movement. The second variable is how fast we will be moving while boosting. The third is for how long, in seconds, we will be boosting. The last is used internally for whether or not we can boost and when we should stop:

```
public CharacterController controller;  
public float boostSpeed = 50;  
public float boostLength = 5;  
public float startTime = -1;
```

3. The `startBoost` function is pretty simple. It checks whether the `startTime` variable is less than zero. If it is, the variable is set to the current time as provided by `Time.time`. The value of the variable being less than zero means that we are not boosting currently:

```
public void StartBoost() {  
    if(startTime < 0)  
        startTime = Time.time;  
}
```

4. The last function we are going to use is the `Update` function. It begins with a check of `startTime` to see whether we are currently boosting. If we are not boosting, the function exits early. The next line of code checks to make sure that we have our `CharacterController` reference. If the variable is empty, then we can't make the tank move:

```
public void Update() {  
    if(startTime < 0) return;  
    if(controller == null) return;
```

5. The next line of code should look familiar. This is the line that makes the tank move:

```
controller.Move(controller.transform.forward * boostSpeed * Time.  
deltaTime);
```

6. Next, check whether we are in the first half-second of the boost. By comparing the current time with the time that was recorded when we started, we can easily figure out for how long we have been boosting:

```
if(Time.time - startTime < 0.5f)
```

7. If the time is right, we transition the camera by adjusting the `fieldOfView` value. The `Camera.main` value is just a reference provided by Unity to the main camera used in the scene. The `Mathf.Lerp` function takes a starting value and moves this value toward the goal value based on a third value between zero and one. Using this, the camera's `fieldOfView` value is moved toward our goal over the half-second:

```
Camera.main.fieldOfView = Mathf.Lerp(Camera.main.fieldOfView, 130,  
(Time.time - startTime) * 2);
```

8. The next piece of code does the same thing as the previous two, except for the last half-second of our boost, and uses the same method to transition the `fieldOfView` value back to the default:

```
else if(Time.time - startTime > boostLength - 0.5f)  
    Camera.main.fieldOfView = Mathf.Lerp(Camera.main.fieldOfView, 60,  
(Time.time - startTime - boostLength + 0.5f) * 2);
```

9. The last bit of code checks whether we are done with boosting. If so, `startTime` is set to `-1` in order to indicate that we can start another boost. That last curly brace, of course, closes off the `Update` function:

```
if(Time.time > startTime + boostLength)
    startTime = -1;
}
```

10. Next, add the script to your tank and connect the `CharacterController` reference.
11. We are almost done. We need to create a new button. We can do this just like we have done before. Anchor the button to the bottom-right corner of `Canvas` and position it just above the chassis' movement controls.
12. Last, be sure to select `Tank` for the **OnClick** object and navigate to **Turbo Boost | StartBoost ()** for the function.
13. Try this out.



We created a turbo boost here. The same method of movement that we used in the previous chapter moves the tank here. By adjusting the **Field of View** attribute of the camera, we make it look like the tank is moving even faster.

You might notice while playing the game that you can turn even when boosting. Try adding a check to the `ChassisControls` script in order to lock the controls, at the time of boosting. You need to add a reference to the `TurboBoost` script to do this.

For an additional, extra challenge, try adding a cooldown to the boost. Make it such that the player can't constantly use the boost. Also, try canceling the boost if the tank runs into something. This is a hard one, so here's a hint to start with: take a look at `OnControllerColliderHit` in the Unity documentation.

Lights

Unity provides a variety of light types for brightening the game world. They are **Directional Light**, **Spotlight**, **Point Light**, and **Area Light**. Each of these projects light in a different way; they are explained in detail as follows:

- **Directional Light:** This functions much like the sun. It projects all of its light in a single direction. The position of the light does not matter, only the rotation does. Light is projected over the entire scene in one direction. This makes it perfect for initially adding light to a scene.
- **Spotlight:** This functions just like the lights on a stage. Light is projected in a cone-like shape in a specific direction. Because of this, it is also the most complex light type for the system to calculate. Unity has made significant improvements on how it calculates lights, but an overuse of these lights should be avoided.
- **Point Light:** This is the primary light type that will be used in your games. It emits light in every direction. This functions just like a light bulb.
- **Area Light:** This is a special-use light. It emits light in a single direction from a plane. Think of it as a big neon sign used to advertise a hotel or restaurant. Because of their complexity, these lights can only be used when baking shadows. There are too many calculations for them to be used when the game is running.

The next obvious question when talking about lights concerns shadows, especially real-time shadows. While real-time shadows add a lot to a scene and are technically possible on any platform, they are very expensive. On top of that, they are a Unity Pro feature for all light types, except **Directional Lights**. All in all, this makes them a bit too much for your average mobile game.

On the other hand, there are perfectly viable alternatives that do not cost nearly as much and often look more realistic than real-time shadows. The first alternative is for your environment. In general, the environment in a game never moves and never changes within a specific scene. For this, we have lightmaps. They are extra textures that contain shadow data. Using Unity, you can create these textures while making your game. Then, when the game is running, they are automatically applied and your shadows appear. This, however, does not work for dynamic objects (anything that moves).

For dynamic objects, we have cookies. These are not your grandmother's cookies. In lighting, a cookie is a black and white image that is projected onto meshes in the game. This is similar to shadow puppets. Shadow puppets use a cutout to block a part of the light, whereas cookies use black and white images to tell the light where it can cast its light.

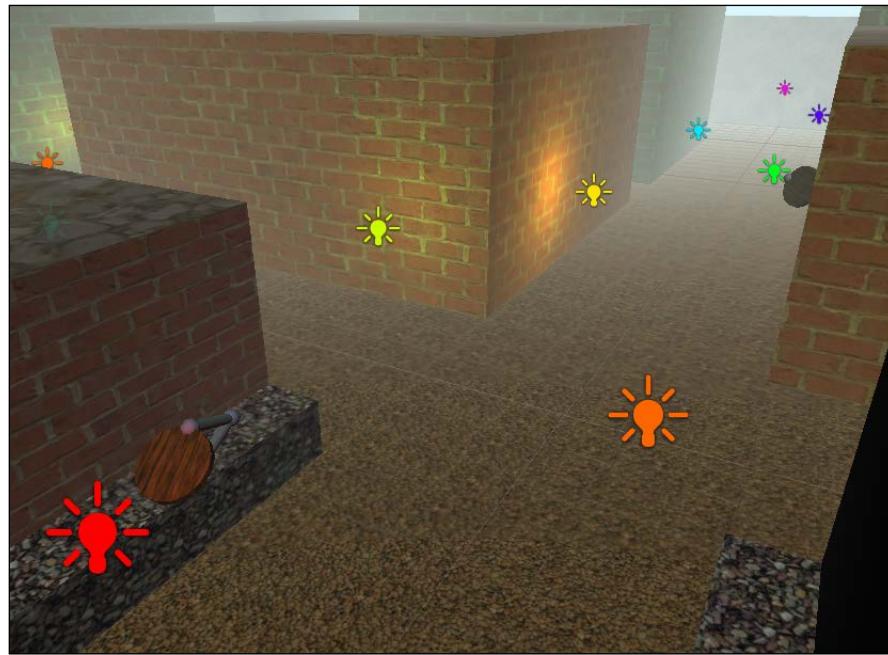
Cookies can also be used to create other good effects, both static and dynamic, such as a cloud cover that pans across the scene or, perhaps, light projecting out from a cage. Or, you can use them to make the uneven focus point of a flashlight.

Adding more lights

It is rather simple to add additional lights to the scene. Also, as long as one sticks to point lights, the cost to render them stays low. Let's use these steps to light up our game:

1. At the top of the Unity Editor, navigate to **GameObject | Light | Point Light**.
2. With the new light selected, these are a few attributes that we should be concerned about in the **Inspector** window:
 - **Range:** This is how far light will be emitted from the object. The light emitted from this point is brightest at the center and fades to nothing as it reaches the extent of the range. The range is additionally represented as a yellow-colored wire sphere in the **Scene** view.
 - **Color:** This is simply the color of the light. By default, it is white; however, any color can be used here. This setting is shared between all the light types.
 - **Intensity:** This denotes the brightness of the light. The greater the intensity of the light, the brighter will be the light at its center. This setting is also shared between all the light types.
3. Create and position several more lights, arranging them along the streets to add some more interesting elements to the environment.

4. Press **Ctrl + D** to duplicate the selected object. This can greatly speed up the creation process (like the one shown in the following screenshot):



5. While adding these lights, you probably noticed one of their major drawbacks. There is a limit to the number of lights that will affect a surface in real time. It is possible to somewhat work around this by using more complex meshes. A better option is to use lightmaps, which we'll be seeing in the next section.
6. At the top of the Unity Editor again, navigate to **GameObject | Light | Spotlight**.
7. Select a new light and take a look at it in the **Inspector** window.

Spot Angle: This is unique to this type of light. It dictates how wide the cone of the emitted light will be. Together with **Range**, it is represented by a yellow-colored wire cone in the **Scene** view.

8. Add a few spotlights around the fountain in the center of our Tank Battle city, as shown in the following screenshot:



9. Having so many objects in a scene clutters the **Hierarchy** window, making it hard to find anything. To organize it, you can use empty **GameObjects**. Create a **GameObject** and name it **PointLights**.
10. By making all of your point lights children of this empty **GameObject**, the **Hierarchy** window becomes significantly less cluttered.

We added several lights to the game. By changing the colors of the lights, we make the scene much more interesting to look at and play in. However, a drawback of the lighting system is revealed. The city we are using is very simple and there is a limit to the number of lights that can affect a plane at one time. While the look of our scene is nevertheless improved, much of the impressiveness is stolen by this drawback.

Lightmaps

Lightmaps are great for complex lighting setups that will be too expensive or simply won't work at runtime. They also allow you to add detailed shadows to your game world without the expense of real-time shadows. However, it will only work for objects that do not move over the course of a game.

Lightmaps are a great effect for any game environment, but we need to explicitly tell Unity which objects will not move and then create the lightmaps. The following steps will help us do this:

1. The first thing to do is to make your environment meshes static. To do this, start by selecting a piece of your city.
2. In the top-right corner of the **Inspector** window, to the right-hand side of the object name field, are a checkbox and a **Static** label. Checking this box will make the object static.
3. Make all of the city's meshes static, as follows:
 - Instead of selecting each checkbox one by one, if you have any sort of grouping (as we just did for the lights), this step can be completed much faster. Select the root object of your city, the one that is the parent to all the pieces of your city, buildings, and streets.
 - Now, go and select the **Static** checkbox.
 - In the new popup, select **Yes, change children** to cause all the subobjects to become static as well.
4. Any mesh that is either not unwrapped or has UV positions outside the normalized UV space will be skipped when Unity generates a lightmap. In the **Model Import Settings** window, there is an option to have Unity automatically generate lightmap coordinates, **Generate Lightmap UVs**. If you are using TankBattleCity for your environment, this option should be turned on now.
5. Go to the top of the Unity Editor and select **Window** and then click on **Lighting**, near the bottom.
6. Most of your time will be spent on the **Scene** page when looking at this window. Select **Scene** at the top of the window to switch to that window.
7. The first thing you will notice about this page is that it has the same **Sky Light** section that we saw in **Scene Render Settings**, where we changed the skybox. We also have all the **Fog** settings toward the bottom of the window.

8. The section we are interested in is **General GI Settings**, as shown in the following screenshot:



The preceding screenshot has the following settings:

- **Workflow:** This setting determines which method you are going to use in order to work with your lightmaps. By default, **Legacy** is selected, which is the old method. We want to change it to **On Demand**. (**Iterative** is the same as **On Demand**, but it attempts to update the lightmaps while you are adjusting settings. This is only recommended if you have a computer that is powerful enough to handle it.)
- **Global Parameters:** This setting lets you create settings that you might want to be able to quickly select. This will be especially useful if you have many scenes that need to be changed. However, we only have one scene, so we can ignore it for now.

- **Sky Light:** This setting affects how much ambient light is in the scene. A lower value will make the overall scene darker, perhaps giving you a night scene. A higher value will make everything brighter, perhaps a daytime scene. The **Realtime Sky** checkbox below this setting dictates whether this calculation is made while the game is running or only when you are baking the lightmaps. Unchecking the box will save on processing, but leaving it checked will allow you to change the brightness of your scene while the game is running. So, you can see your lights in the game, set **Sky Light** to `0.2`, and uncheck **Realtime Sky**.
- **Albedo Scale:** This setting affects how much light bounces off of surfaces. The **Indirect Scale** option affects how much light is in the overall scene from the light sources that do not point directly at an object. For our purposes, both of these can be left at their default values.
- **Realtime GI Settings:** This section is only available with Unity's new lightmapping system. It holds the controls for lightmaps that are calculated while the game is running. The **Realtime Resolution** and **Realtime Atlas Size** options adjust how much detail is present in these lightmaps. The **CPU Usage** option controls the amount of effort that the system will put into calculating the values you see while the game is running. Since we are working on a mobile platform, we need to keep our processing costs down, so leaving all of these at their low defaults works fine for us.
- **Baked GI Settings:** These settings hold the controls for adjusting the precalculated lightmaps. This is where most of your adjustments will take place. Right off the bat, we have a **Directional Mode** checkbox that dictates whether we are going to use a single set of lightmaps when unchecked. Or, if we are going to use two sets, where one set is for color and direct light and the second set is for indirect light. Using two sets of lightmaps can give you greater detail, especially in dark areas, but is more costly to calculate and use. So, we are going to leave it unchecked.
- **Baked Resolution:** This setting controls how much detail is put into an object based on its size. After the number field, you can see the **texels per unit** setting. A texel is just a fancy lightmap pixel. So, it is really just the amount of pixel details in the lightmap for each unit in the scene. For our purposes, a value of `30` will give us a good amount of detail without overloading our computers.



The **Baked Resolution** setting will most quickly affect how long it takes to actually bake your lightmaps. It is always better to start working with low values and only increase the values once your lighting setup comes close to what you want the final product to look like.

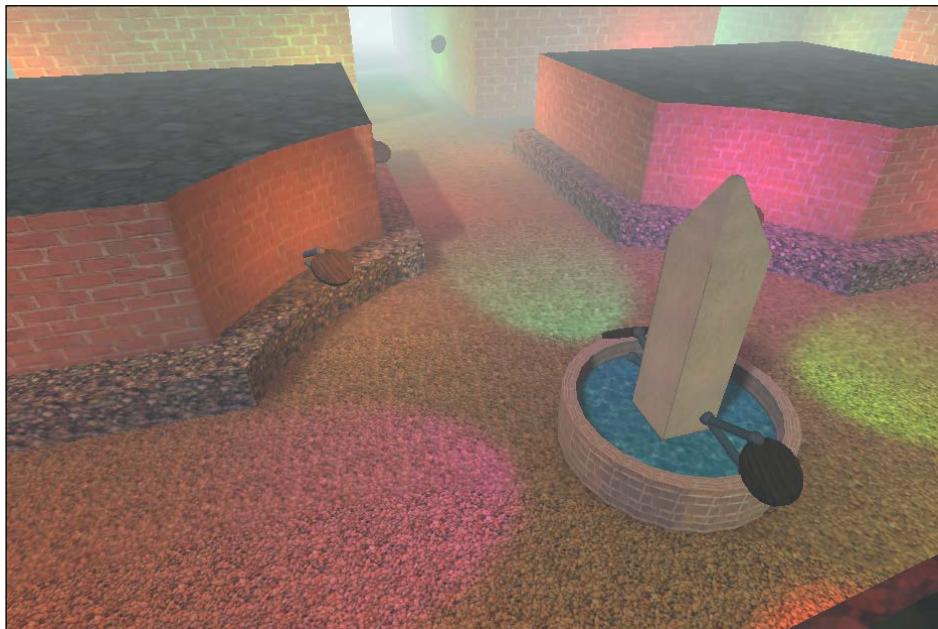
- **Baked Atlas Size:** This setting controls the ultimate resolution of the final lightmap images. Smaller resolution sizes will be easier to process, but you need to limit the overall details of the largest objects in your scene. No matter what resolution you have chosen, a single plane of your models cannot have more detail than a single lightmap atlas. The default of **1024** is an excellent compromise between detail and processing cost.
 - **Padding:** The value of this setting adjusts the space in the lightmap between objects. A value that is too low will cause the shading to bleed onto the edges of other objects that share the lightmap. A value that is too high will lead to a great amount of wasted space in your lightmaps. Again, the default here will work just fine for us.
 - **Direct Scale:** This setting will scale the intensity of lights when baked into your lightmap. It will let you change the overall brightness of your scene. The default will work just fine here as well.
 - **AO Exponent:** This setting adjusts the contrast of the ambient lighting. This will make the dark areas in your scene look darker and the light areas look brighter. Leaving it at the default of **1** will be fine for us.
9. At the bottom of the page is a **Bake** button. Clicking on this button will start the render process. A loading bar will appear in the bottom-right corner of Unity, so you can monitor the progress.



Be warned as this process is likely to take a while. Especially as the complexity of the environment and the number of lights increases and the detail settings are ramped up, this will take longer and longer to run. Also, unless you have a superior computer, there isn't much you can do in Unity while it is running.

10. If you clicked on the button and realized that you have made a mistake, don't fret. After **Bake** is selected, the button changes to **Cancel**. At this time, it is possible to select it and stop the process from continuing. However, once the textures have been created and Unity starts to import them, there is no stopping it.

11. At the left-hand side of the **Bake** button is **Clear**. This button is the quickest and easiest way to delete and remove all of the lightmaps that are currently being used in the scene. This cannot be undone.
12. In order to add shadows to your buildings, select **Directional Light** in your scene, from **Hierarchy**, and take a look at the **Inspector** window.
13. From the **Shadow Type** drop-down list, select **Soft Shadows**. This simply turns on the shadows for this light. It turns them on for both lightmaps and real-time lighting. The greater the number of lights with shadows turned on, the more expensive they become to render. It is a good idea to turn on the shadows for your lightmap, but be sure to turn them off afterwards. This will conserve processing in your final game, while still giving your static scene a good look.
14. When all your lights and settings match your expectations, select **Bake** and once it has finished processing, gaze in wonder at the now beautiful scene before you, as shown here:



We added lightmaps to our game world. The length of time it takes to just process this step makes it difficult to make minor tweaks. However, our lighting has vastly improved with a few clicks. While earlier the lights were broken by the meshes, we now have smooth patches of color and light.

When playing a game, there is only one type of light that people will not question the source of: sunlight. Every other light looks weird if a source is not seen. Create a mesh and add it to the game in order to give a reason for the lights you are using. This can be something along the lines of torches, lamp posts, or even glowing alien goo balls. Whatever they end up being, having them adds that touch of completeness that makes the difference between an OK-looking game and a great-looking game.

As a second challenge, take a look at your lightmap's quality. Play with the various quality settings we discussed to see what the differences are. Also, find out how low the resolution can be before you notice any pixelation. Can the settings go even lower when running on smaller mobile device screens? Go find out.

Cookies

Cookies are a great way to add interest to the lights in your game. They use a texture to adjust how the light is emitted. This effect can cover a wide range of uses, from sparkling crystals to caged industrial lights and, in our case, headlights.

By giving our tank headlights, we give the player the ability to control the light in their world. Using cookies, we can make them look more interesting than just circles of light. Let's add those lights with these steps:

1. Start by creating a spotlight.
2. Position the light in front of the tank and pointing away.
3. In the **Inspector** window, increase the value of the **Intensity** attribute to 3. This will make our headlights bright, like real headlights.
4. Now, we need some cookie textures. At the top of the Unity Editor, navigate to **Assets | Import Package | Light Cookies**.
5. In the new window, click on **Import** and wait for the loading bar to finish.
6. We now have a few options to choose from. Inside the **Standard Assets** folder, a new folder was created, **Light Cookies**, that contains the new textures. Drag **Flashlight** from the **Project** window and drop it onto the **Cookie** field on **Spotlight** in the **Inspector** window. It is as simple as that to add a cookie to a light.
7. You may still not be able to see your cookie in action. It is the result of the same issue we were having before; too many lights can't shade the same object. Unfortunately, a light that is meant to move around cannot be baked into the lightmaps. To fix this, change the light's **Render Mode** attribute to **Important** in the **Inspector** panel. This will give the light priority and make it light an object before the other objects in the scene.

8. If you were to bake your lights again now, you would end up with the cookie shape stuck on the wall of a building. We need to change **GI Mode** to **Realtime** so that the light is ignored by the lightmapping process but is still able to affect the scene.
9. To finish off, duplicate the light for the second headlight and make them both children of the tank. What good is it to have headlights if they don't come with us?



We performed a few short steps and created a pair of headlights for our tank using cookies. This is exactly how many other games, especially horror games, create flashlight effects.

Try making a script that will allow the player to turn the headlights on and off. It should be a simple button that toggles the lights. Take a look at the enabled variable that is supplied as part of the light.

As a simple challenge, create a lamp that sits on the turret of the tank. Give it a light as well. With this, the player can point a light to where they are shooting and not just in the direction in which their tank is pointing.

Blob shadows

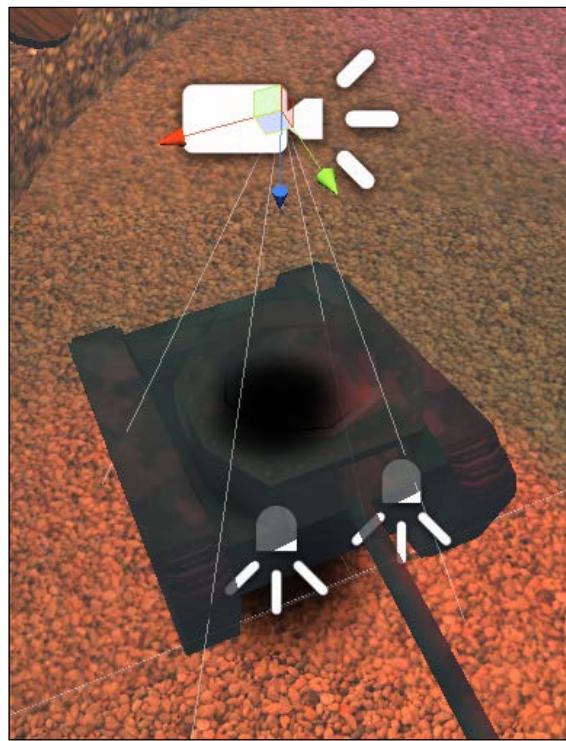
Blob shadows are a simple and cheap method by which you can add a shadow to a character. They have been around since the dawn of video games. A normal shadow is a solid, dark projection of an object onto another surface. The contours of the shadow exactly match the shape of the object. This becomes expensive to calculate when characters start to move around randomly.

A blob shadow is a blot of black texture underneath a character or an object. It usually does not have a clearly definable shape and never matches the exact shape of the object it is meant to be the shadow of. The blob shadow also, generally, does not change sizes. This makes it significantly easier to calculate, making it the shadow of choice for many generations of video games. This also means that it is a better option for our mobile devices where processing speed can quickly become an issue.

We are going to add a blob shadow to our tank. Unity has already done the bulk of the work for us; we just need to add it to the tank. With these steps, we can add a blob shadow:

1. We start this one off by importing Unity's blob shadow. Go to the top of the Unity Editor and navigate to **Assets | Import Package | Projectors**.
2. Click on **Import** in the new window and look in the **Project** window for a new folder called **Projectors** created under **Standard Assets**.

3. Drag the **Blob Shadow Projector** prefab from the **Project** window to the scene and position it above the tank, as shown in the following screenshot:

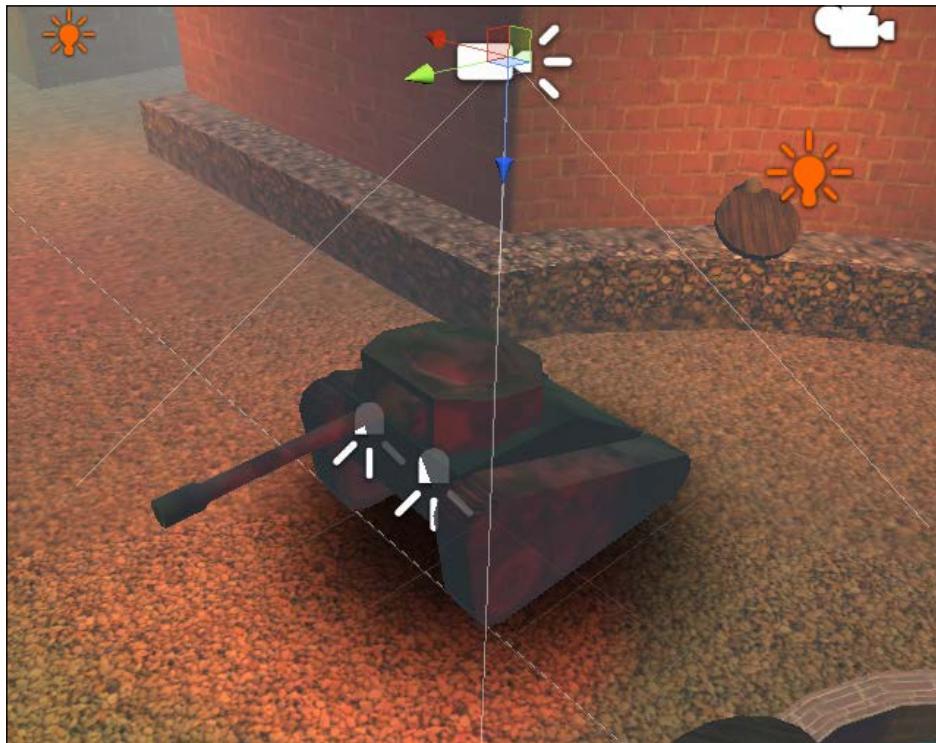


4. Unfortunately, the shadow appears on top of our tank. To fix this, we again need to make use of layers. So, select the tank.
5. From the **Layer** drop-down list, select **Add Layer....**
6. Click on the text field at the right-hand side of **User Layer 9** and give it the name **PlayerTank**.
7. Select your tank once more, but select **PlayerTank** from the **Layer** drop-down list this time.
8. When the new window pops up, be sure to select **Yes, change children** to change the layer of the whole tank. If you don't select this, the blob shadow may appear on some parts of the tank, while it may not appear on other parts.
9. Now, select **Blob Shadow Projector** from the **Hierarchy** window.



The blob shadow is created by the **Projector** component. This component functions in a similar manner to the **Camera** component. However, it puts an image on the world rather than turning the world into an image and putting it on your screen.

10. Take a look at the **Inspector** window. The value we are concerned with right now is that of **Ignore Layers**. Right now, it is set to **Nothing**.
11. Click on **Nothing** and select **PlayerTank** from the **Layers** drop-down list. This will make the projector ignore the tank and only make the blob shadow appear underneath it.
12. The next step is to change the size of the shadow to roughly match the size of the tank. Adjust the value of the **Field of View** attribute until the size is just about right. A value of 70 seems to be a good size to start with.



13. The final step is to make **Blob Shadow Projector** a child of the tank. We need to be able to bring our shadow with us; we don't want to lose it.

We gave our tank a shadow. Shadows are great for making objects, and especially characters, look like they are actually touching the ground. The blob shadow that we used is better than a real-time shadow because it is processed faster.

The texture that the blob shadow comes with is round, but our tank is mostly square. Try creating your own texture for the blob shadow and use it. Some sort of a rectangle should work well. If you end up with long black streaks on your scene, make sure that your texture has a completely white border around the edge of the image.

If you managed to add your own texture to the blob shadow, then how about taking a look at that cannon? The cannon sticks out of our tank and ruins its otherwise square profile. Use a second blob shadow, attached to the turret, to project a shadow for the cannon. The texture for this will also have to be rectangle shaped.

Summary

At this point, you should be well and truly familiar with camera effects and lights.

In this chapter, we started by taking a look at using multiple cameras. We then played around with a turbo boost camera effect. The chapter continued with the lighting of our city. The lights improved greatly when we made use of lightmaps. We finished it off with a look at cookies and blob shadows for use with some special lighting effects.

In the next chapter, we will see the creation of enemies for our game. We will use Unity's pathfinding system to make them move around and chase the player. After this, the player will need to be much more active if they hope to keep their points.

5

Getting Around – Pathfinding and AI

In the previous chapter, we learned about camera and lighting effects. We added skybox, lights, and shadows to our Tank Battle game. We created lightmaps to make our scene dynamic. We took a look at cookies by giving our tank headlights. We also took a look at projectors by creating a blob shadow for the tank. A turbo boost was also created for the tank. By adjusting the viewing angle of the camera, we were able to make the tank look as if it was going much faster than it really was. When we finished the chapter, we had a dynamic and exciting-looking scene.

This chapter is all about the enemy. No longer will the player be able to just sit in one place to gather points. We will be adding an enemy tank to the game. By using Unity's NavMesh system, the tanks will be able to do pathfinding and chase the player. Once the player is found, the tanks will shoot and reduce the player's score.

In this chapter, we will cover the following topics:

- NavMesh
- NavMeshAgent
- Pathfinding
- Chase and attack AI
- Spawn points

We will be adding modifications to the Tank Battle game from *Chapter 4, Setting the Stage – Camera Effects and Lighting*, so load it up and we can begin.

Understanding AI and pathfinding

AI is, as you might have guessed, **Artificial Intelligence**. In the broadest sense, this is anything an inanimate object might do to appear to be making decisions. You are probably most familiar with this concept from video games. When a character, not controlled by the player, selects a weapon to use and a target to use it on, this is AI.

In its most complex form, AI attempts to mimic full human intelligence and learning. However, there is still far too much happening incredibly fast for this to truly succeed. Video games do not need to reach this far. We are primarily concerned with making our characters appear intelligent but still conquerable by our players. Usually, this means not allowing characters to act on more information than what a real player might have. Adjusting how much information characters have and can act on is a good way to adjust the level of difficulty in a game.

Pathfinding is a subset of AI. We use it all the time, though you have probably never realized it. Pathfinding is, as the word suggests, the act of finding a path. Every time you need to find your way between any two points, you are doing pathfinding. As far as our characters are concerned, the simplest form of pathfinding is to follow a straight line to the goal point. Obviously, this method works best on an open plain, but tends to fail when there are any obstacles in the way. Another method is to overlay the game with a grid. Using the grid, we can find a path that goes around any obstacles and reaches our target.

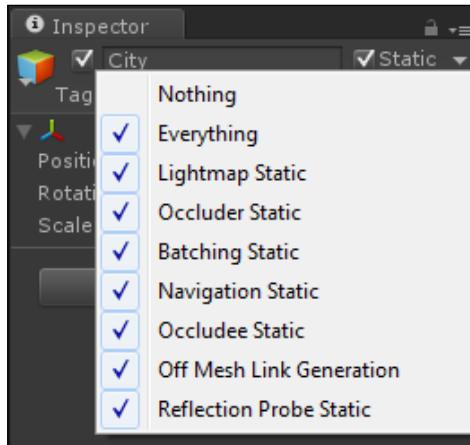
An alternative method to pathfinding, and perhaps the one most often chosen, makes use of a special navigation mesh, or NavMesh. This is just a special model that is never seen by the player but covers all of the area that a computer character can move around in. The player is then navigated in a way that is similar to the grid; the difference is that the triangles of the mesh are used rather than the squares of the grid. This is the method that we will be using in Unity. Unity provides a nice set of tools for creating the NavMesh and utilizing it.

The NavMesh

Creating the navigation mesh in Unity is very simple. The process is similar to the one that we used for making lightmaps. We just mark some meshes to be used, adjust some settings in a special window, and hit a button. So, load up the Tank Battle game in Unity if you haven't already done so, and we can get started.

Unity can automatically generate a NavMesh from any meshes that exist in a scene. To do so, the mesh must first be marked as static, just as we did for lightmaps. However, we do not want or need to be able to navigate the roofs of our city, so we make use of a special list of settings to dictate what type of static each object will be. Let's start with the following steps:

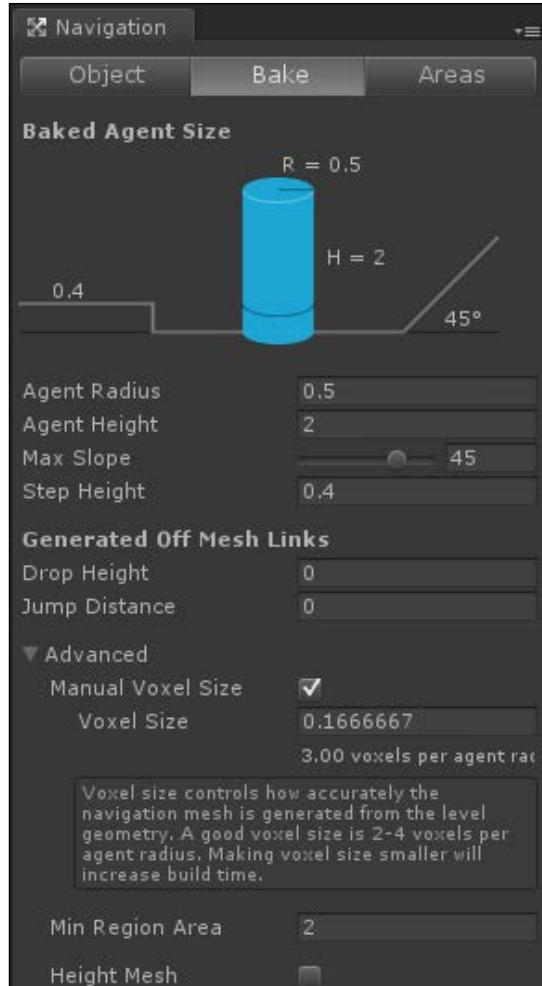
1. Select the city from the **Hierarchy** window and click on the down arrow to the right of **Static** in the **Inspector** window:



We can take a look at the options available for static objects as follows:

- **Nothing:** This option is used to quickly deselect all the other options. If all the other options are unchecked, this one will be checked.
- **Everything:** Using this option, you can quickly select all the other options. When all of them are checked, this one will also be checked. The checkbox next to the **Static** label in the **Inspector** window performs the same function as checking and unchecking the **Everything** checkbox.
- **Lightmap Static:** This option needs to be checked when working with lightmaps in order for them to work. Any mesh that does not have this checked will not be lightmapped.
- **Occluder Static:** This is an option for working with occlusion. **Occlusion** is a method of runtime optimization that involves only rendering objects that can actually be seen, whether or not they are within the camera's view space. An **occluder** is an object that will block other objects from being seen. It works in conjunction with the **Occludee Static** option. The best object choices for this option are large and solid.
- **Batching Static:** This is another option for runtime optimization. Batching is the act of grouping objects together before rendering them. It greatly increases the overall render speed of a game.

- **Navigation Static:** This is the option that we are primarily concerned with at this point. Any mesh that has this option checked will be used when calculating the NavMesh.
 - **Occludee Static:** As mentioned a moment ago, this option works in conjunction with **Occluder Static** for the good of occlusion. An **occludee** is an object that will be obscured by other objects. When covered by an occluder, this object will not be drawn.
 - **Off Mesh Link Generation:** This option also works with the NavMesh calculation. An off-mesh link is a connection between two parts of the NavMesh that aren't physically connected, such as the roof and the street. Using a few settings in the **Navigation** window and this option, the links are automatically generated.
 - **Reflection Probe Static:** The last option allows the object to be recorded by reflection probes. These record everything around them and generate a cubemap that can be used by reflective shaders.
2. In order to make the NavMesh work properly, we need to change the settings so that only the streets of the city can be navigated. When was the last time you saw a tank jump or fall from the roof of a building? So, we need to change the static options so that only the streets have **Navigation Static** checked. This can be done in one of the following two ways:
- The first way is to go through and uncheck the option for each object that we want changed.
 - The second is to uncheck **Navigation Static** for the top-level object in the **Hierarchy** window, and when Unity asks whether we want to make the change for all children objects, reply with a yes. Then, go to just the objects that we want to navigate and recheck the option.
3. Now, open the **Navigation** window by going to Unity's toolbar and click on **Window** and then click on **Navigation** at the bottom of the menu. The following screenshot displays the window where all the work of making the NavMesh happens:



4. This window consists of three pages and a variety of settings:

When an object is selected, the settings will appear on the **Object** page. The two checkboxes correspond directly with the **Static** options of the same name that we set a moment ago. The drop-down list in **Navigation Area** lets us group different parts of our NavMesh. These groups can be used to affect the pathfinding calculation. For example, a car can be set to only travel on the road area and the human can follow the sidewalk area.

The **Bake** page is the one that we are interested in; it is full of options to change how the NavMesh will be generated. It even includes a nice visual representation of the various settings at the top:

- **Agent Radius:** This should be set to the size of the thinnest character. It is used to keep characters from walking too close to walls.
- **Agent Height:** This is the height of your characters. Using this, Unity can calculate and remove areas that are too low for them to pass. Anything lower than this value is deemed too small, so it should be set to the height of your shortest character.
- **Max Slope:** Anything steeper than this value is ignored when calculating the NavMesh.
- **Step Height:** When making use of stairs, one must use this value. This is the maximum height of a stair that a character can step on.
- **Drop Height:** This is the height from which characters can fall. With it, paths will include jumping off ledges, if it is faster to do so.
- **Jump Distance:** Using this value, characters can jump across gaps in the NavMesh. This value represents the longest distance that can be jumped.
- **Manual Voxel Size / Voxel Size:** By checking the **Manual Voxel Size** box, you can adjust the value of **Voxel Size**. This is a level of detail for the NavMesh. Lower values will make it more accurate to the visible mesh, but it will take longer to calculate and require more memory to store.
- **Min Region Area:** If parts of the NavMesh are smaller than this value, they will not be used in the final NavMesh.
- **Height Mesh:** With this option checked, the original height information is maintained in NavMesh. Unless you have a special need for it, this option should remain off. It takes the system longer to calculate and requires more memory to store.

The third page, **Areas**, allows us to adjust the cost of movement for each of our defined areas. Essentially, how difficult is it to move though different parts of our game world? With cars, we could adjust the layers so that it is twice as costly for them to move through the field than to move along the road.

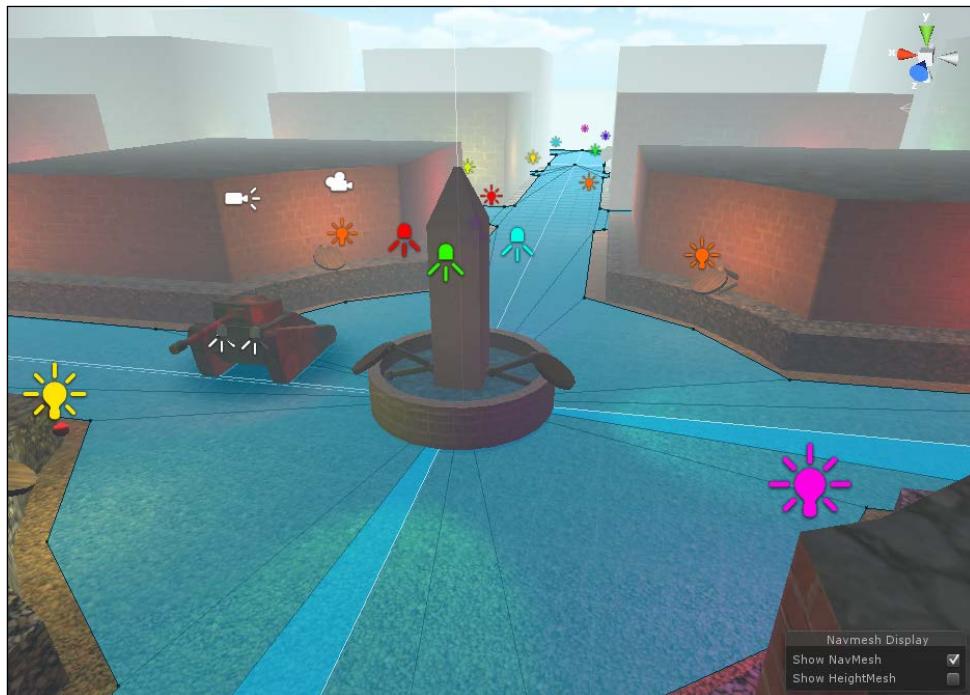
At the bottom of the window, we have the following two buttons:

- **Clear:** This button removes the previously created NavMesh. After using this button, you will need to rebake the NavMesh before you can make use of pathfinding again.
- **Bake:** This button starts the work and creates the NavMesh.

5. Our city is very simple, so the default values will suit us well enough. Hit **Bake** and watch the progress bar in the bottom-right corner. Once it is done, a blue mesh will appear. This is the NavMesh and it represents all of the area that a character can move through.

 It may happen that your tanks will poke through the walls of the buildings a little as they move around. If they do, increase the **Agent Radius** in the **Navigation** window until they no longer do this.

6. There is one last thing we need to do. Our NavMesh is just right, but if you look closely, it goes through the fountain at the center of the city. It would be just wrong if enemy tanks start driving through the fountain. To fix this, start by selecting the mesh that forms the wall around the fountain.
7. In Unity's toolbar, click on **Component**, followed by **Navigation**, and finally **Nav Mesh Obstacle**. This simply adds a component that tells the navigation system to go around when searching for a path. Since we had already selected the wall, the new component will be sized to fit; we just need to select **Capsule** from the **Shape** drop-down list. You can see it represented as a wire cylinder in the **Scene** view.



We created the NavMesh. We made use of the **Navigation** window and the **Static** options to tell Unity which meshes to use when calculating the NavMesh. The Unity team put a lot of work into making this process quick and easy.

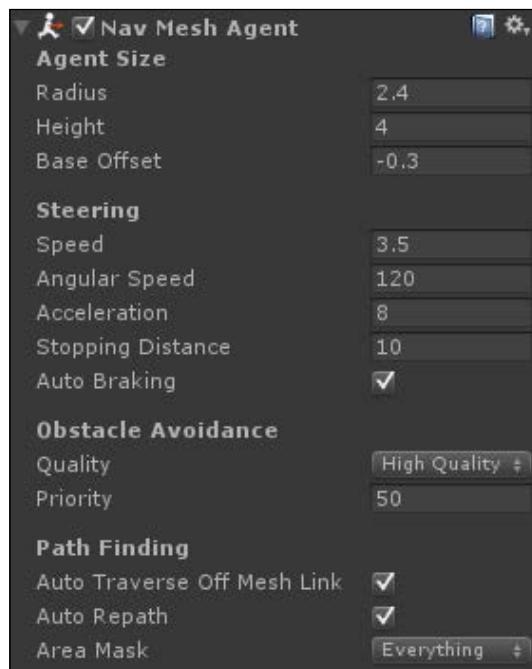
Remember, in *Chapter 3, The Backbone of Any Game – Meshes, Materials, and Animations*, when the challenge was to create obstacles for the player, you were encouraged to create additional meshes, such as tank traps and rubble. It would be a bad idea to let the enemy tanks drive through these as well. So, have a go at turning these into obstacles for the navigation system. This will be done just as with the fountain.

The NavMeshAgent component

You might be thinking that it is all well and good that we have a NavMesh, but there are no characters to navigate it. In this section, we will start the creation of our enemy tank. We will need to import and do a little setup for this second tank before we can do any AI type of programming. Using these steps, we can create it:

1. Select `Tanks_Type03.png` and `Tanks_Type03.blend` from the starting assets for the chapter and import them to the `Tanks` folder under the `Models` folder.
2. Once Unity has finished importing, select the new tank in the **Project** window and take a look at it in the **Inspector** window.
3. This tank has no animations, so the **Animation Type** can be set to **None** and **Import Animation** can be unchecked from the **Rig** and **Animations** pages respectively.
4. Drag the tank from the **Project** window to the **Scene** window; any clear patch of street will work just fine.
5. For starters, rename the model in the **Scene** view to `EnemyTank`.
6. Now, we need to change the parenting of the tank so that the turret can turn and the cannon will follow, just as we did for the player's tank. To do this, create an empty **GameObject** and rename it as `TurretPivot`.
7. Position `TurretPivot` to be at the base of the turret.
8. In the **Hierarchy** window, drag and drop `TurretPivot` onto `EnemyTank` to make `EnemyTank` its parent.
9. Next, make another empty **GameObject** and rename it as `CannonPivot`.
10. The `CannonPivot` **GameObject** must be made a child of `TurretPivot`.
11. In the **Hierarchy** window, make the turret mesh a child of `TurretPivot` and the cannon mesh a child of `CannonPivot`. When Unity asks whether you are sure that you want to break the prefab connections, be sure to click on **Yes**.

12. The tank is a little large, so adjust the **Scale Factor** of the tank's **Import Settings** in the **Inspector** window to **0.6** to give us a tank that is similar to the size of the player's tank.
13. In order for the tank to navigate our new NavMesh, we need to add a **NavMeshAgent** component. First, select **EnemyTank** in the **Hierarchy** window, go to Unity's toolbar and navigate to **Component | Navigation | Nav Mesh Agent**. In the **Inspector** window, we can see the new component and the settings associated with it, as shown in the following screenshot:



All of these settings let us control how the NavMeshAgent interacts with our game world. Let's take a look at what each of them does:

- **Radius**: This is simply how big the agent is. By working in conjunction with the value of **Radius** that we set in the **Navigation** window, this keeps the object from walking partly in the walls and into other agents.
- **Height**: This setting affects the cylinder that appears in the editor, around the agent. It simply sets the height of the character and affects what overhangs they might be able to walk under.

- **Base Offset:** This is the vertical offset of the colliders that is attached to the agent. It allows you to adjust what the **NavMeshAgent** component considers to be the bottom of your character.
- **Speed:** The **NavMeshAgent** component automatically moves the connected object when it has a path. This value dictates how fast to follow the path in units per second.
- **Angular Speed:** This is the degrees per second that the agent can turn. A person would have a very high angular speed, while a car's angular speed would be low.
- **Acceleration:** This is how many units per second in speed that the agent gains until it reaches its maximum capacity.
- **Stopping Distance:** This is the distance from the target destination at which the agent will start to slow down and stop.
- **Auto Braking:** With this box checked, the agent will stop as soon as it reaches the destination, rather than overshooting because of the irregular frame rate that tends to average to around 60 to 90 FPS for most games.
- **Obstacle Avoidance Quality / Priority:** The quality is how much effort the agent will put in to find a smooth path around obstacles. A higher quality means more effort is made to find the path. The **Priority** option dictates who has the right of way. An agent with a high value will go around an agent with a low value.
- **Auto Traverse Off Mesh Link:** With this box checked, the agent will use the off-mesh links when pathfinding, such as jumping gaps and falling off ledges.
- **Auto Repath:** If the path that was found is incomplete for any reason, this checkbox allows Unity to automatically try to find a new one.
- **Area Mask:** Remember the areas that were mentioned earlier when discussing the **Navigation** window? This is where we can set which areas the agent is able to traverse. Only the areas in this list that are checked will be used for pathfinding by the agent.

14. Now that we understand the settings, let's use them. For the enemy tank, a value of `2.4` for the **Radius** and `4` for the **Height** will work well. You should be able to see another wire cylinder in the **Scene** window, which is our enemy tank.
15. The last thing to do is to turn `EnemyTank` into a prefab. Do this just as we did with the targets, by dragging it from the **Hierarchy** window and dropping it on the **Prefabs** folder in the **Project** window.

Here, we created an enemy tank. We also learned about the settings for the **NavMeshAgent** component. However, if you try to play the game now, nothing will appear to happen. This is because the **NavMeshAgent** component is not being given a destination. We will resolve this in the next section.

Making the enemy chase the player

Our next task is to make our enemy tank chase the player. We will need two scripts for this. The first will simply advertise the player's current position. The second will use that position and the **NavMeshAgent** component that we set up earlier to find a path to the player.

Revealing the player's location

With a very short script, we can easily allow all our enemies to know the location of the player. A few short steps to create it are as follows:

1. Start by creating a new script in the **Scripts** folder of the **Project** window. Name it **PlayerPosition**.
2. This script will start with a single static variable. This variable will simply hold the current position of the player. As it is static, we will be able to easily access it from the rest of our scripts.

```
public static Vector3 position = Vector3.zero;
```

We chose to use a static variable here for its simplicity and speed. Alternatively, we could have added a few extra steps to our enemy tank; it could have used the **FindWithTag** function when the game started to actually find the player tank and store it in a variable. Then, it could query that variable when it looks for the player's position. This is just one more way, among the multitude of ways, in which we could have gone about it.

3. For the next few lines of code, we make use of the **Start** function. This function is automatically called when a scene is first loaded. We use it so that the **position** variable can be filled and used as soon as the game starts.

```
public void Start() {
    position = transform.position;
}
```

4. The last segment of the code simply updates the position variable in every frame to the player's current position. We also do this in the `LateUpdate` function so that the update is done after the player has moved. The `LateUpdate` function is called at the end of every frame. With this, the player is able to move during the `Update` function and their position is updated later.

```
public void LateUpdate() {  
    position = transform.position;  
}
```

5. The last thing to do with this script is to add it to the player's tank. So, return to Unity and drag and drop the script from the **Project** window to the tank to add it as a component, just as we did it for all our other scripts.

Here, we created the first script that is needed for our chase AI. This script simply updates a variable with the player's current position. We will make use of it in our next script where we will make the enemy tank move around.

Chasing the player

Our next script will control our simple chase AI. Since we are making use of the **NavMesh** and **NavMeshAgent** components, we can leave nearly all the difficult portions of pathfinding to Unity. Let's create the script by performing these steps:

1. Again, create a new script. This time, name it `ChasePlayer`.
2. The first line of this script holds a reference to the **NavMeshAgent** component that we set up earlier. We need access to this component in order to move the enemy tank.

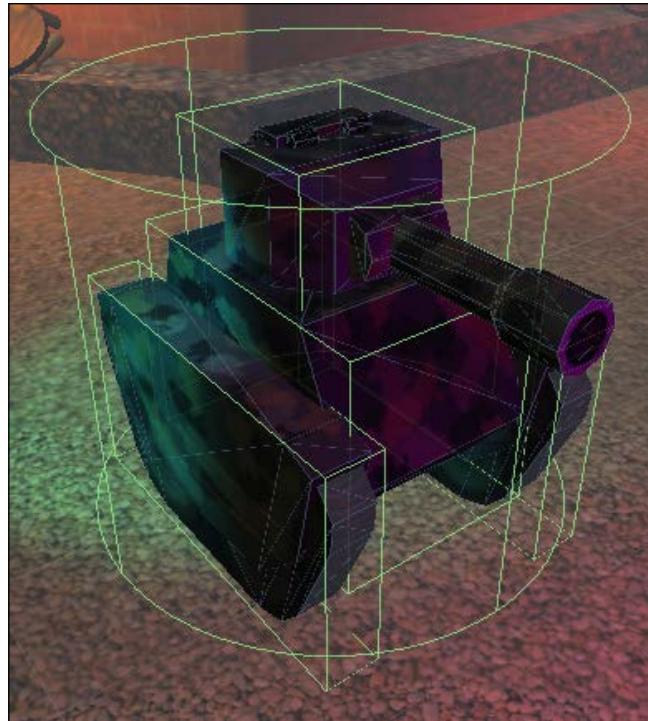
```
public NavMeshAgent agent;
```

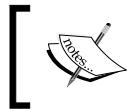
3. The last segment of the code first makes sure that we have our **NavMeshAgent** reference and then updates our goal destination. It uses the `PlayerPosition` script's variable that was set up earlier and the `SetDestination` function from the **NavMeshAgent**. Once we tell the function where to go, the **NavMeshAgent** component does all the hard work of getting us there. We update our goal destination in the `FixedUpdate` function because we do not need to update the destination in every frame. Updating this too often could cause a serious lag issue if there are a whole lot of enemies. The `FixedUpdate` function is called at regular intervals and is slower than the frame rate, so it is perfect.

```
public void FixedUpdate() {  
    if(agent == null) return;
```

```
    agent.SetDestination(PlayerPosition.position);  
}
```

4. We now need to add the script to our enemy tank. Select the prefab in the **Project** window and drag and drop the script in the **Inspector** panel, underneath the **NavMeshAgent** component.
5. Be sure to connect the reference, as we did previously. Drag the **NavMeshAgent** component to the **Agent** value in the **Inspector** window.
6. Play the game now to try it out. Irrespective of the location where the enemy starts, it finds its way around all the buildings and makes it to the player's position. As you drive around, you can watch the enemy follow. However, the enemy tank could end up going through our tank and we could drive through it as well.
7. The first step to fix this is to add some colliders. Add a **Box Collider** component by using the **Physics** option in the **Component** menu for the turret, chassis, and each of the **TreadCase** objects. Neither the cannon nor the treads need colliders. The tread casings already cover the area of the treads, and the cannon is too small a target to be shot at properly.





If you are making any of these changes in the **Scene** view, be sure to click on the **Apply** button in the **Inspector** window to update the root prefab object.

8. The last thing to change is the **Stopping Distance** property on the **NavMeshAgent** component. When the tanks engage, they move into range and start firing. They do not try to occupy the same space as the enemy, unless that enemy is small and squishy. By setting **Stopping Distance** to 10, we will be able to replicate this behavior.



In this section, we created a script that causes a **NavMeshAgent** component, in this case our enemy tank, to chase the player. We added colliders to stop us from driving through the enemy. In addition, we adjusted the value of **Stopping Distance** to give us a better tank behavior.

Try adding a blob shadow to the enemy tank. This will give it a better visual sense of being grounded. You can just copy the one that was made for the player's tank.

Being attacked by the enemy

What fun is a game without a little conflict; the nagging choice is whether to fight to the death or the doom of the cosmos? Every game needs some form of conflict to drive the player towards seeking a resolution. Our game will become a battle for points. Before, this just involved shooting some targets and getting some points.

Now, we will make the enemy tank shoot at the player. Every time the enemy scores a hit, we will reduce the player's score by a few points. The enemy will shoot in a similar manner to how the player fires, but we will use some basic AI to control the direction and firing speed and replace the player's input controls. These steps will help us do it:

1. We will start this off with a new script called `ShootAtPlayer`. Create it in the `Scripts` folder.

2. As with all our other scripts, we start this one out with two variables. The first variable will hold the last position of the enemy tank. The tank will not be shooting if it is in motion, so we need to store its last position to see whether it has moved. The second variable will be the maximum speed at which we can move and shoot. If the tank moves faster than this, it will not fire.

```
private Vector3 lastPosition = Vector3.zero;
public float maxSpeed = 1f;
```

3. The next two variables dictate how long it takes for the tank to ready a shot. It is unrealistic to be shooting at the player in every single frame. So, we use the first variable to adjust the length of time it takes to ready a shot and the second to store when the shot will be ready:

```
public float readyLength = 2f;
private float readyTime = -1f;
```

4. The next variable contains the value of how fast the turret can rotate. While the tank is readying its shot, the turret will not be rotating to point at the player. That gives the player an opportunity to move out of the way. However, we need a speed variable to keep the turret from snapping to face the player after it has finished shooting.

```
public float turretSpeed = 45f;
```

5. The last three variables here hold references to other parts of the tank. The `turretPivot` variable is, of course, the pivot of the turret that we will rotate. The `muzzlePoint` variable will be used as the point from where our cannon will be fired. These will be used in the same manner as the ones for the player's tank.

```
public Transform turretPivot;  
public Transform muzzlePoint
```

6. For the first function of the script, we will make use of the `Update` function. It starts by calling a function that will check to see whether it is possible to fire the cannon. If we can fire, we will perform some checks on our `readyTime` variable. If it is less than zero, we have not yet begun to ready our shot and call a function to do so. However, if it is less than the current time, we have finished the preparation and call the function to fire the cannon. If we are unable to fire, we first call a function to clear any preparations and then rotate the turret to face the player.

```
public void Update() {  
    if(CheckCanFire()) {  
        if(readyTime < 0) {  
            PrepareFire();  
        }  
        else if(readyTime <= Time.time) {  
            Fire();  
        }  
    }  
    else {  
        ClearFire();  
        RotateTurret();  
    }  
}
```

7. Next, we will create our `CheckCanFire` function. The first part of the code checks to see whether we have moved too fast. First, we use `Vector3.Distance` to see how far we have moved since the last frame. By dividing the distance by the length of the frame, we are able to determine the speed with which we moved. Next, we update our `lastPosition` variable with our current position so that it is ready for the next frame. Finally, we compare the current speed with `maxSpeed`. If we moved too fast in this frame, we will be unable to fire and return a result as `false`:

```
public bool CheckCanFire() {  
    float move = Vector3.Distance(lastPosition, transform.position);  
    float speed = move / Time.deltaTime;
```

```
lastPosition = transform.position;  
  
if (speed > maxSpeed) return false;
```

8. For the second half of the `CheckCanFire` function, we will check to see whether the turret is pointed at the player. First, we will find the direction to the player. By subtracting the second point's location from that of any given point in space, we will get the vector value of the first point with respect to the second point. We will then flatten the direction by setting the `y` value to 0. This is done because we do not want to be looking up or down at the player. Then, we will use `Vector3.Angle` to find the angle between the direction to the player and our turret's forward direction. Finally, we will compare the angle to a low value to determine whether we are looking at the player and return the result:

```
Vector3 targetDir = PlayerPosition.position - turretPivot.  
position;  
targetDir.y = 0;  
  
float angle = Vector3.Angle(targetDir, turretPivot.forward);  
  
return angle < 0.1f;  
}
```

9. The `PrepareFire` function is quick and easy. It simply sets our `readyTime` variable to the time in the future when the tank would have prepared its shot:

```
public void PrepareFire() {  
    readyTime = Time.time + readyLength;  
}
```

10. The `Fire` function starts by making sure that we have a `muzzlePoint` reference to shoot from:

```
public void Fire() {  
    if (muzzlePoint == null) return;
```

11. The function continues with the creation of a `RaycastHit` variable to store the result of our shot. We use `Physics.Raycast` and `SendMessage`, just as we did in the `FireControls` script, to shoot at anything and tell it that we hit it:

```
RaycastHit hit;  
if (Physics.Raycast(muzzlePoint.position, muzzlePoint.forward, out  
hit)) {  
    hit.transform.gameObject.SendMessage("RemovePoints", 3,  
    SendMessageOptions.DontRequireReceiver);  
}
```

12. The `Fire` function finishes by clearing the fire preparations:

```
ClearFire();  
}
```

13. The `ClearFire` function is another quick function. It sets our `readyTime` variable to be less than zero, indicating that the tank is not preparing to fire:

```
public void ClearFire() {  
    readyTime = -1;  
}
```

14. The last function is `RotateTurret`. It begins by checking the `turretPivot` variable and cancels the function if the reference is missing. This is followed by the finding of a direction that points at the player, just as we did earlier. This direction is flattened by setting the `y` axis to 0. Next, we will create the `step` variable to specify how much we can move this frame. We use `Vector3.RotateTowards` to find a vector that is closer to pointing at our target than the current forward direction. Finally, we use `Quaternion.LookRotation` to create a special rotation that points our turret in the new direction.

```
public void RotateTurret() {  
    if(turretPivot == null) return;  
  
    Vector3 targetDir = PlayerPosition.position - turretPivot.  
position;  
    targetDir.y = 0;  
  
    float step = turretSpeed * Time.deltaTime;  
  
    Vector3 rotateDir = Vector3.RotateTowards(  
        turretPivot.forward, targetDir, step, 0);  
    turretPivot.rotation = Quaternion.LookRotation(rotateDir);  
}
```

15. Now, by returning to Unity, create an empty **GameObject** and rename it as **MuzzlePoint**. Position **MuzzlePoint** like we did for the player, at the end of the cannon.
16. Make **MuzzlePoint** a child of the cannon and zero out any **Y** rotation that might be on it in the **Inspector** window.
17. Next, add our new **ShootAtPlayer** script to the enemy tank. Additionally, connect the references to the **TurretPivot** and **MuzzlePoint** variables.
18. Finally, for the enemy tank, hit the **Apply** button in the **Inspector** window to update the prefab.
19. If you play the game now, you will see the enemy rotating to point at you, but our score will not decrease. This is because of two reasons. First, the tank is slightly floating. It doesn't matter where in the world you place it; when you play the game, the tank will slightly float. This is because of the way the **NavMeshAgent** component functions. The fix is simple; just set **BaseOffset** to **-0.3** in the **Inspector** window. This adjusts the system and puts the tank on the ground.
20. The second reason the score isn't changing is because the player is missing a function. To fix this, open the **ScoreCounter** script.
21. We will add the **RemovePoints** function. Given an amount, this function simply removes that many points from the player's score:

```
public void RemovePoints(int amount) {  
    score -= amount;  
}
```



If your enemy tank is still unable to hit the player, it may be too big and is shooting over the player. Just tilt the tank's cannon down so that when it is shooting at the player, it also points towards the center of the player's tank.

If you take a look at the score counter in the top-right corner, the score will go down when the enemy gets close. Remember, it will not start dropping immediately because the enemy needs to stop moving, to ready the cannon, before they can shoot.



We gave the enemy the ability to attack the player. The new `ShootAtPlayer` script first checks to see whether the tank has slowed down and the cannon is trained on the player. If so, it will take regular shots at the player to reduce their score. The player is going to need to keep moving and aim at targets fast if they hope to be left with any points at the end of the game.

Unless you are paying close attention to your score, it is difficult to tell when you are being shot at. We will be working with explosions in a future chapter, but even so, the player needs some feedback to tell what is going on. Most games will flash a red texture on the screen when the player is hit, whether or not there are any explosions. Try creating a simple texture and drawing it on the screen for half a second when the player is hit.

Attacking the enemy

Players tend to become frustrated quickly when faced with an enemy that they are unable to fight against. So, we are going to give our player the ability to damage and destroy the enemy tank. This will function in a similar manner to how the targets are shot.

The easiest way to weaken our enemies is to give them some health that will reduce when they are shot. We can then destroy them when they run out of health. Let's create a script with these steps to do this:

1. We will start by creating a new script and naming it `Health`.
2. This script is rather short and starts with a single variable. This variable will keep track of the remaining health of the tank. By setting the default value to 3, the tank will be able to survive three hits before being destroyed.

```
public int health = 3;
```

3. This script also contains only one function, `Hit`. As in the case of the targets, this function is called by the `BroadcastMessage` function when the player shoots at it. The first line of the function reduces `health` by one point. The next line checks to see whether `health` is below zero. If it is, the tank is destroyed by calling the `Destroy` function and passing the `gameObject` variable to it. We also give the player a handful of points.

```
public void Hit() {  
    health--;  
    if(health <= 0) {  
        Destroy(gameObject);  
        ScoreCounter.score += 5;  
    }  
}
```

4. It really is just that simple. Now, add the new script to the `EnemyTank` prefab in the **Project** window, and it will update all the enemy tanks that you currently have in the scene.
5. Try this out: add a few extra enemy tanks to the scene and watch them follow you around and disappear when you shoot them.

Here, we gave the enemy tank a weakness, `health`. By creating a short script, the tank is able to track its health and detect when it has been shot. Once the tank runs out of health, it is removed from the game.

We now have two targets to shoot at: the animated ones and the tank. However, they are both indicated with red slices. Try to make the ones that point at tanks to be of a different color. You will have to make a duplicate of the `IndicatorSlice` prefab and change the `IndicatorControl` script so that it can be told which type of slice to use when the `CreateSlice` and `NewSlice` functions are called.

As a further challenge, the moment we give a creature some health, players want to be able to see how much damage they have done to it. There are two ways you could do this. First, you could put a cluster of cubes above the tank. Then, each time the tank loses health, you will have to delete one of the cubes. The second option is a little more difficult—drawing the bar in the GUI and changing its size based on the remaining health. To make the bar stay above the tank as the camera moves around, take a look at `Camera.WorldToScreenPoint` in the documentation.

Spawning enemy tanks

Having a limited number of enemies in the game at the beginning is not suitable for our game to have lasting fun. Therefore, we need to make some spawn points. As tanks are destroyed, these will make new tanks appear to keep the player on their toes.

The script that we will create in this section will keep our game world populated with all the enemies that our player might want to destroy. These steps will let us spawn the enemy tanks:

1. We need another new script for this section. Once this is created, name it `SpawnPoint`.

2. This script begins simply with a few variables. The first variable will hold a reference to our `EnemyTank` prefab. We need it so that we can spawn duplicates.

```
public GameObject tankPrefab;
```

3. The second variable tracks the spawned tank. When it is destroyed, we will create a new one. Using this variable, we prevent the game from becoming overwhelmed with the enemy. There will only be as many tanks as spawn points.

```
private GameObject currentTank;
```

4. The third variable is for setting a distance between the spawning tanks and the player to prevent the spawning tanks from appearing on top of the player. If the player is outside this distance, a new tank can be spawned. If they are within, a new tank will not be spawned.

```
public float minPlayerDistance = 10;
```

5. The first function that we will use is `FixedUpdate`. This will start by checking a function to see whether it needs to spawn a new tank. If it does, it will call the `SpawnTank` function to do so:

```
public void FixedUpdate() {
    if (CanSpawn())
        SpawnTank();
}
```

6. Next, we create the `CanSpawn` function. The first line of the function checks to see whether we already have a tank and returns `false` if we do. The second line uses `Vector3.Distance` to determine how far away the player currently is. The last line compares that distance with the minimum distance that the player needs to be before we can spawn anything, and it then returns the result:

```
public bool CanSpawn() {
    if(current != null) return false;

    float currentDistance = Vector3.Distance(PlayerPosition.
position, transform.position);
    return currentDistance > minPlayerDistance;
}
```

7. The last function, `SpawnTank`, starts by checking to make sure that the `tankPrefab` reference has been connected. It can't continue if there is nothing to spawn. The second line uses the `Instantiate` function to create a duplicate of the prefab. In order to store it in our variable, we use `as GameObject` to make it the proper type. The last line moves the new tank to the spawn point's position as we don't want the tanks to appear at random locations.

```
public void SpawnTank() {
    if(tankPrefab == null) return;

    currentTank = Instantiate(tankPrefab) as GameObject;
    currentTank.transform.position = transform.position;
}
```

We again chose to use the `Instantiate` and `Destroy` functions to handle the creation and deletion of our enemy tanks due to their simplicity and speed. Alternatively, we could have created a list of available enemies. Then, every time our player kills one, we could turn it off (instead of completely destroying it), just move an old one to where we need it (instead of creating a new one), reset the old one's stats, and turn it on. There will always be multiple ways to program everything, and this is just one alternative.

8. Return to Unity, create an empty **GameObject**, and rename it as `SpawnPoint`.
9. Add the `SpawnPoint` script, which we just created, to it.
10. Next, with the spawn point selected, connect the prefab reference by dragging the `EnemyTank` prefab from the `Prefabs` folder and drop it on the appropriate value.

11. Now, turn the **SpawnPoint** object into a prefab by dragging and dropping it from the **Hierarchy** window into the **Prefabs** folder.
12. Finally, populate the city with the new points. Positioning one in each corner of the city will work well.



Here, we created spawn points for the game. Each point will spawn a new tank. When a tank is destroyed, a new one will be created at the spawn point. Feel free to build the game and try it out on your device. This section and chapter are now complete and ready to be wrapped up.

Having one spawn point per tank is great, until we want many tanks or we wish them all to spawn from the same location. Your challenge here is to make a single spawn point to track multiple tanks. If any one of the tanks is destroyed, a new one should be created. You will definitely need an array to keep track of all the tanks. In addition, you could implement a delay for the spawn process as you won't want multiple tanks spawning on top of each other. This could cause them to suddenly jump as the **NavMeshAgent** component does its best to keep them from occupying the same space. In addition, the player might also think that they are only fighting one tank, when in fact there are several tanks in the same spot.

Now that you have all the knowledge and tools that you need, as a further challenge, try to create other types of enemy tanks. You can experiment with size and speed. They can also have different strengths, or you could give more points when enemy tanks are destroyed. Perhaps, there is a tank that actually gives the player points when shooting at them. Play around with the game and have some fun with it.

Summary

In this chapter, we learned about NavMeshes and pathfinding. We also did a little work with AI. This was perhaps one of the simplest types of AI, but chase behaviors are highly important to all types of games. To utilize all of this, we created an enemy tank. It chased the player and shot at them to reduce their score. To give the edge back to the player, we gave health to the enemy tanks. The player could then shoot the enemy tanks as well as the targets for points. We also created some spawn points so that every time a tank was destroyed, a new one would be created. In terms of general game play, our Tank Battle game is pretty much complete.

In the next chapter, we will create a new game. In order to explore some of the special features of the mobile platform, we will create a Monkey Ball game. We will remove nearly all of the buttons from the screen in favor of new control methods. We will be turning the device's tilt sensors into our steering method. In addition, we will use the touchscreen to destroy enemies or collect bananas.

6

Specialities of the Mobile Device – Touch and Tilt

In the previous chapter, we learned about pathfinding and AI. We expanded our Tank Battle game to include enemy tanks. We created points for them to spawn at and made them shoot at the player. In addition, the player was given the ability to destroy the tanks. Once they were destroyed, the player received some points and a new enemy tank was spawned.

In this chapter, we will work on a new game as we explore some of the specialties of mobile devices. We will create a **Monkey Ball** game. The player will take control of a monkey in an oversized hamster ball and try to reach the end of the maze before time runs out, while collecting bananas. To move around, they will have to tilt the mobile device. To collect bananas, the player will have to touch the screen where the banana is.

In this chapter, we will cover the following topics:

- Touch controls
- Tilt controls
- The Monkey Ball game

We will be creating a new project for this chapter, so start Unity and we will begin.

Setting up the development environment

As with every project, we need a little bit of preparation work in order to prepare our development environment. Don't worry; the setup for this chapter is simple and straightforward. Let's follow these steps to do it:

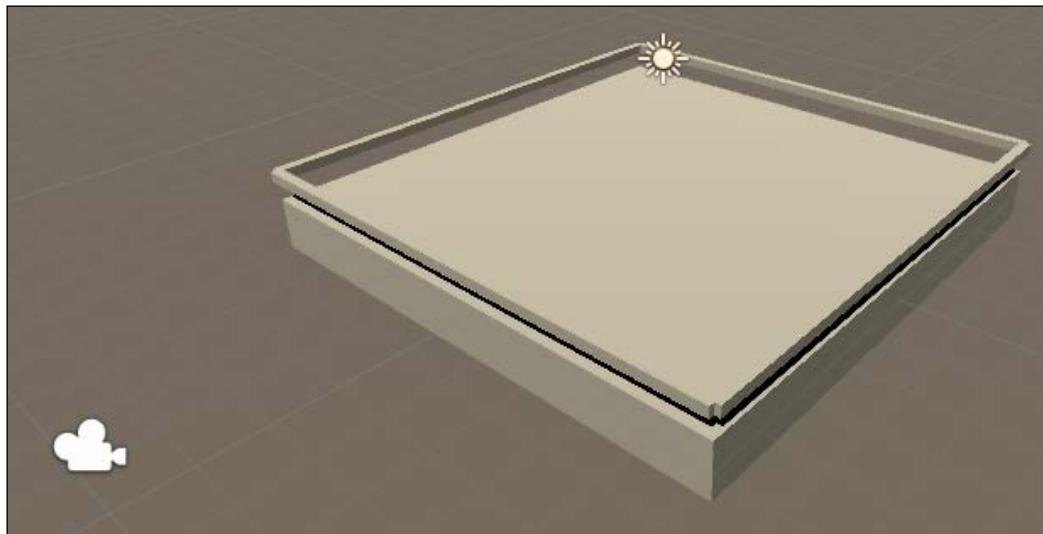
1. The first step is, of course, to start Unity and create a new project. It will need to be a 3D project and naming it Ch6_MonkeyBall will work well.
2. Once Unity has finished initializing, this is the perfect opportunity to set our build settings. Open the **Build Settings** window, select **Android** from the list of platforms and hit **Switch Platform** to change the target platform.
3. While you are in the **Build Settings** window, select **Player Settings** to open the player settings in the **Inspector**. Adjust the **Company Name**, **Product Name**, and, most importantly, the **Bundle Identifier**.
4. When a user tilts their device, the whole screen will adjust its orientation when a new side becomes the bottom. Since the whole game is based around tilting the device, the screen orientation might change at any moment when a player is playing and thus spoil their game. Therefore, in **Player Settings**, find the **Resolution and Presentation** section and ensure that the **Default Orientation** is not set to **Auto Rotation**, which would cause Unity to change a game's orientation when we are playing. Any of the other options will work for us here.
5. We need to create a few folders to keep the project organized. The **Scripts**, **Models**, and **Prefabs** folders should be created in the **Project** window. Since we may end up with dozens of levels and maps in the future, it would be a good idea to make a **Scenes** folder as well.
6. Lastly, we must import the assets for this project. We are going to need a monkey for the player, a banana to collect, a sample map, and some fences. Luckily, all of these have already been prepared and are available with the starting assets for this chapter. Import **Monkey.blend**, **Monkey.psd**, **Ball.psd**, **Banana.blend**, **Banana.psd**, **MonkeyBallMap.blend**, **Grass.psd**, **Fence.blend**, and **Wood.psd** to the **Models** folder that you just created.

We have just finished the setup for this chapter's project. Once again, a little bit of effort at the beginning of the project will save time and avoid frustration later; as the project grows in size, the organization done at the beginning becomes very important.

A basic environment

Before we dive into all the fun of tilt and touch controls, we need a basic testing environment. When working with new control schemes, it is always best to work in a simple and well-controlled environment before introducing the complexities of a real level. Let's make ours with these steps:

1. Go to the top of Unity and select **Cube** by navigating to **GameObject | 3D Object** to create a new cube, which will be the base of our basic environment. Rename it as **Ground** so that we can keep track of it.
2. Set the cube's **Position** in the **Inspector** panel to 0 on each axis, allowing us to work around the world origin. Also, set its **X** and **Z Scale** in the **Inspector** to 10, giving us enough space to move around and test our monkey.
3. Next, we need a second cube, named **Fence**. This cube should have a **Position** value of -5 for **X**, 1 for **Y**, and 0 for **Z**, along with a scale of 0.2 for **X** and **Y** and 10 for **Z**.
4. With **Fence** selected in the **Hierarchy** window, you can hit *Ctrl + D* on your keyboard to make a duplicate. We are going to need a total of four, positioned along each side of our **Ground** cube:



We now have a basic testing environment that will allow us to work with our controls and not worry about all the complexities of a whole level. Once our controls work in this environment the way we want them to, we will introduce our monkey to a new environment.

Controlling with tilt

Modern mobile devices provide a broad variety of internal sensors to detect and provide information about the surrounding world. Though you may not have thought of them in such a way, you must be certainly very familiar with the microphone and speaker that are required for making calls. There is also a Wi-Fi receiver for connecting to the Internet and a camera for taking pictures. In addition, your device almost certainly has a magnetometer, to work with your GPS and provide directions.

The sensor that we are interested in right now is the **gyroscope**. This sensor detects local rotation of the device. In general, it is one of the many sensors in your phone that is used to determine the orientation and movement of the device in the world. We are going to use it to steer our monkey. When the user tilts their device left and right, the monkey will move left and right. When the device is tilted up and down, the monkey will go forward and backward. With these steps, we can create the script that will let us control our monkey in this manner:

1. To start this off, create a new script and name it `MonkeyBall`.
2. Our first variable will hold a reference to the **Rigidbody** component that will be attached to the ball. This is what will allow us to actually make it roll around and collide with the things in the world:

```
public Rigidbody body;
```

3. The next two variables will let us control how the tilting of the device affects the movement in the game. The first will allow us to get rid of any movements that are too small. This lets us avoid random movements from the environment or a sensor that perhaps isn't entirely accurate. The second will let us scale the tilt input up or down in case the control feels either sluggish and slow or uncontrollably fast:

```
public float minTilt = 5f;  
public float sensitivity = 1f;
```

4. The last variable for now will keep track of how much the device has been tilted. It forces the user to tilt their device back and forth, countering movement if they want to go in the opposite direction:

```
private Vector3 totalRotate = Vector3.zero;
```

5. Our very first function for this script is nice and short. In order to get input from the gyroscope, we must first turn it on. We will do this in the `Awake` function so that we can start tracking it at the very beginning of the game:

```
public void Awake() {  
    Input.gyro.enabled = true;  
}
```

- The next function for our script will be `Update`. It starts by grabbing the value of `rotationRate` from the gyroscope. This is a value in radians per second, indicating how fast the user has tilted their device along each axis. To make it a little more understandable, we multiply the value of `rotationRate` by `Mathf.Rad2Deg` to convert it into degrees per second before we store it in a variable:

```
public void Update() {
    Vector3 rotation = Input.gyro.rotationRate * Mathf.Rad2Deg;
```



When holding your device with the screen facing you, the *x* axis of your device points to the right. The *y* axis is straight up, at the top of the device and the *z* axis points directly towards you from the center of the screen.

- Next, we make sure that there is enough movement along each axis to actually make our monkey move. By using `Mathf.Abs` on each value, we find the absolute value of the axis movement. We then compare it to the minimum amount of tilt that we are looking for. If the movement is too little, we zero it out in our `rotation` variable:

```
if(Mathf.Abs(rotation.x) < minTilt) rotation.x = 0;
if(Mathf.Abs(rotation.y) < minTilt) rotation.y = 0;
if(Mathf.Abs(rotation.z) < minTilt) rotation.z = 0;
```

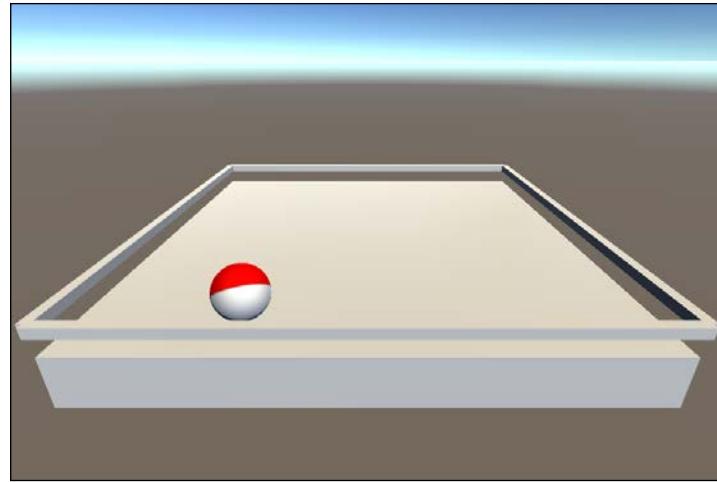
- Finally, for our `Update` function, we track the new movement by adding it to our `totalRotate` variable. To do this properly, we need to rearrange the values. The player expects to be able to tilt the top of their device towards them to go backwards and away to go forwards. This is the *x* axis movement, but it comes in backwards from our device compared to what we need to move the monkey, hence the negative sign before the value. Next, we swap the *y* and *z* axes' rotation because the player is going to expect to tilt their device left and right to go left and right, which is a *y* axis movement. If we applied that to the *y* axis of our monkey, he would just spin in place. So, the movement is treated to be speed per second rather than speed per frame; we have to multiply by `Time.deltaTime`:

```
TotalRotate += new Vector3(-rotation.x, rotation.z, -rotation.y) *
Time.deltaTime;
}
```

9. The last function for now is the `FixedUpdate` function. When making changes to and dealing with rigidbodies, it is best to do it in `FixedUpdate`. The `Rigidbody` is what actually connects us into Unity's physics engine, and it only updates during this function. All we are doing here is adding some torque, or rotational force, to the `Rigidbody`. We use the total that we have been collecting and multiply it by our `sensitivity` to give our players the speed of control that they will expect:

```
public void FixedUpdate() {  
    body.AddTorque(totalRotate * sensitivity);  
}
```

10. In order to make use of our new script, we need to make some changes to the ball. Start by creating a sphere for us to work with; this can be found by navigating to **GameObject | 3D Object | Sphere**. Rename it as `MonkeyBall` and position it a little above our **Ground** cube.
11. Next, give it the `Ball.psd` texture in a material so that we can see it rotate and not just move. The two-tone nature of the texture will let us easily see it roll around the scene.
12. The **Rigidbody** component can be found by navigating to **Component | Physics | Rigidbody** at the top of Unity. Add a new **Rigidbody** component.
13. In addition, add our `MonkeyBall` script to the sphere and drag the new **Rigidbody** component to the **Body** slot in the **Inspector** panel.
14. This is the point where it is especially important to have **Unity Remote**. With your device attached and *Unity Remote* running, you can hold it up and steer the ball. Feel free to adjust the sensitivity and minimum tilt until you find settings that feel natural to control. Due to the great variety of devices, their hardware, and the architecture used, the rate of tilt can easily differ from one device to the next. However, especially at this stage, you must find settings that work for your device now and worry about what will work for other devices once the game is more complete.
15. If you are having trouble seeing the ball roll around, move the camera so that you have a better view. However, make sure that it continues to point forward along the world's z axis.
16. Once all your settings are in place, ensure that you save the scene. Name it `MonkeyBall`.



We made use of the gyroscope to provide you with the steering control of a ball. By measuring how the player is tilting his or her device, we are able to add motion to the ball accordingly. By rolling around a simple map, we can fine-tune our controls and make sure everything is working correctly.

Following with the camera

To really make the player feel like they are controlling the ball, the camera needs to follow it around. This is particularly necessary when the maps and levels become larger and more complex than what can be shown in a single camera shot. The simplest solution would be to just make the camera a child of the ball, but that will make it spin with the ball and our controls will become confusing as well. So, let's use these steps to set up our camera to follow the ball around:

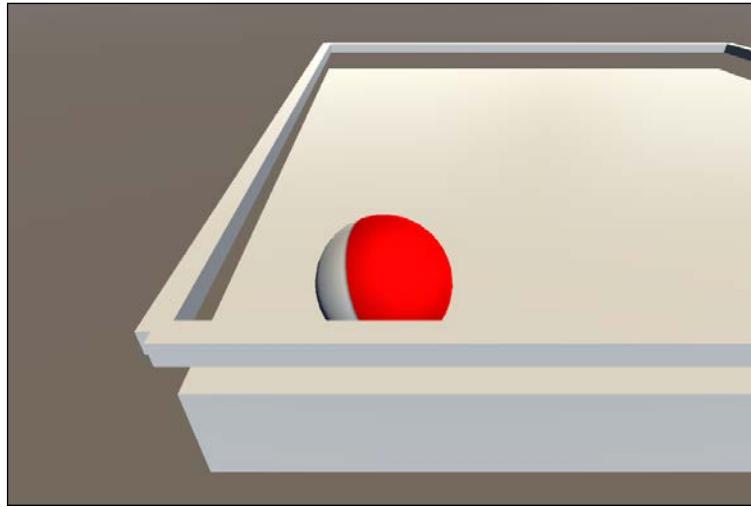
1. We need to first create a new script and name it `CameraFollow`.
2. This script is really simple. It has a single variable to keep track of what is being followed:

```
public Transform ball;
```

3. The only function in the script is the `LateUpdate` function. We use this function because it executes after everything else has had a chance to do their normal update. The only thing the script is going to do is move to the new position of the ball:

```
public void LateUpdate() {  
    transform.position = ball.position;  
}
```

4. To make use of this script, we need a new empty **GameObject** component. Name it **CameraPivot**.
5. Position it at (approximately) the center of the ball. This is the point that will actually move to follow the ball around. At this point, the created **GameObject** doesn't have to be perfectly positioned; it just needs to be close enough so that it's easier to line up the camera.
6. Next, find the **Main Camera** in the **Hierarchy** window and make it a child of **CameraPivot**.
7. Set the **Main Camera** component's **X** position to 0. As long as **X** stays at zero and the camera continues to point relatively forward along the **z** axis, you can freely move it to find a good position from which to observe the ball. Values of 2 for the **Y** position, -2.5 for the **Z** position, and 35 for the **X** rotation also work well.
8. Next, add the **CameraFollow** script to the **CameraPivot** object.
9. Finally, drag **MonkeyBall** from the scene and drop it on the **Ball** slot of the new **CameraFollow** script component. Then, go try it out!



We now have a ball that rolls around and a camera that follows it. The camera is simply updating its position to keep pace with the ball, but it works well as an effect. As a player, we will definitely feel that we are taking control of the ball and its motion.

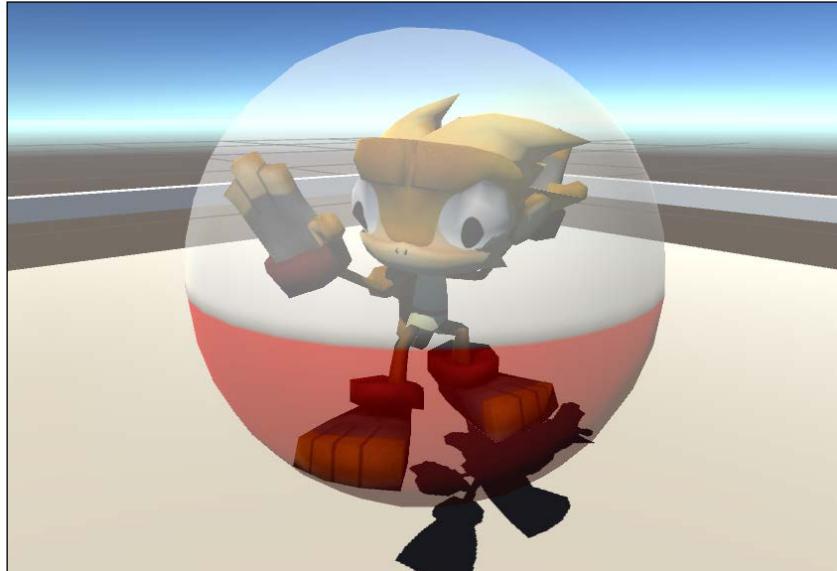
Adding the monkey

Now that we are close to the ball and following it around, we need something a little more interesting to look at. In this section, we are going to add the monkey to the ball. In addition, to ensure that he isn't being spun around wildly, we will make a new script to keep him upright. Let's do all of that by following these steps:

1. Create a new empty **GameObject** and rename it `MonkeyPivot`.
2. Make it a child of the `MonkeyBall` script and zero out its position.
3. Next, add the monkey to the scene and make it a child of the `MonkeyPivot` **GameObject**.
4. To make it easier to see the monkey inside the ball, we need to make it slightly transparent. Select `MonkeyBall` and find the **Rendering Mode** setting on the material at the bottom. By changing it to **Transparent**, we will be able to adjust it.
5. Now, click on the **Color Picker** box to the right of **Albedo** and change the **A** slider, alpha, to 128; this will allow us to now see through the ball.
6. Scale and move the monkey until he fills the center of the ball.



You can also take this opportunity to pose the monkey. If you expand the monkey in the **Hierarchy** window, you will be able to see all of the bones that make up his skeleton rig. Giving him a cool pose now will make the game much better for our players later.



7. Our monkey and the ball are looking really cool right now, until we actually hit play and the monkey spins around dizzily in the ball. We need to open our `MonkeyBall` script and fix his spinning antics:
8. First, we need two new variables at the top of the script. The first will keep track of the empty **GameObject** that we created a moment ago. The second will give us the speed for updating the rotation of the monkey. We want it to look like the monkey is moving the ball, so he needs to face the direction in which the ball is moving. The speed here is how fast he will turn to face the right direction:

```
public Transform monkeyPivot;  
public float monkeyLookSpeed = 10f;
```

9. Next, we need a new `LateUpdate` function. This double-checks whether the `monkeyPivot` variable has actually been filled for the script. If it isn't there, we can't do anything else:

```
public void LateUpdate() {  
    if(monkeyPivot != null) {
```

10. We first need to figure out which direction the ball is moving in. The easiest way to do this is to grab `velocity` of the **Rigidbody** component, our `body` variable. It is a `Vector3` that indicates how fast and in which direction we are currently moving. Since we do not want our monkey to point up or down, we zero out the `y` axis movement:

```
Vector3 velocity = body.velocity;  
velocity.y = 0;
```

11. Next, we need to figure out which direction the monkey is currently facing. We have used the `forward` value before, with our tanks. It is simply the direction in 3D space in which we are facing. Again, to avoid looking up or down, we zero out the `y` axis:

```
Vector3 forward = monkeyPivot.forward;  
forward.y = 0;
```

12. To prevent suddenly changing direction as we move and to keep pace with the frame rate, we must calculate a `step` variable. This is how much we can rotate this frame, based on our speed and the time that has elapsed since the last frame:

```
float step = monkeyLookSpeed * Time.deltaTime;
```

13. We then need to find a new direction to face by using `Vector3`.
`RotateTowards`. It takes the direction we were facing, followed by the direction we want to face and two speeds. The first speed specifies how much the angle can change in this frame and the second specifies how much the magnitude, or length, of the vector can change. We are not concerned with a change in magnitude, so it is given a zero value:

```
Vector3 newFacing = Vector3.RotateTowards(forward, velocity, step,  
0);
```

14. Finally, the new rotation is calculated using `Quaternion.LookRotation` by passing the `newFacing` vector to it and applying the result to the monkey's rotation. This will turn the monkey to face in the direction of the movement and keep him from spinning with the ball:

```
monkeyPivot.rotation = Quaternion.LookRotation(newFacing);  
}  
}
```

15. To make it work, drop the `MonkeyPivot` object on the **Monkey Pivot** slot on the **MonkeyBall** script component. The monkey will rotate to face the direction of the ball's movement while staying upright:



We've just finished adding the monkey to the ball. By giving him a cool pose, the player will be more engaged with it as a character. However, it looks a little weird when the monkey spins wildly within the ball, so we updated our script to keep him upright and facing the direction in which the ball is moving. Now, it almost looks as though the monkey is in control of the ball.

Keeping the monkey on the board

What fun is a game if there is no risk of failure? In order to test our monkey and tilt controls, we put a safety fence around our basic environment to keep them from falling over. However, every game needs a little risk to make it exciting. By removing the safety fence, we introduce a risk of falling over and losing the game. However, usually there is an option to retry the game if you fall. To this end, we will now create what is traditionally called a **kill volume**. This is simply an area that resets the player when they fall into it. Let's use these steps to create it:

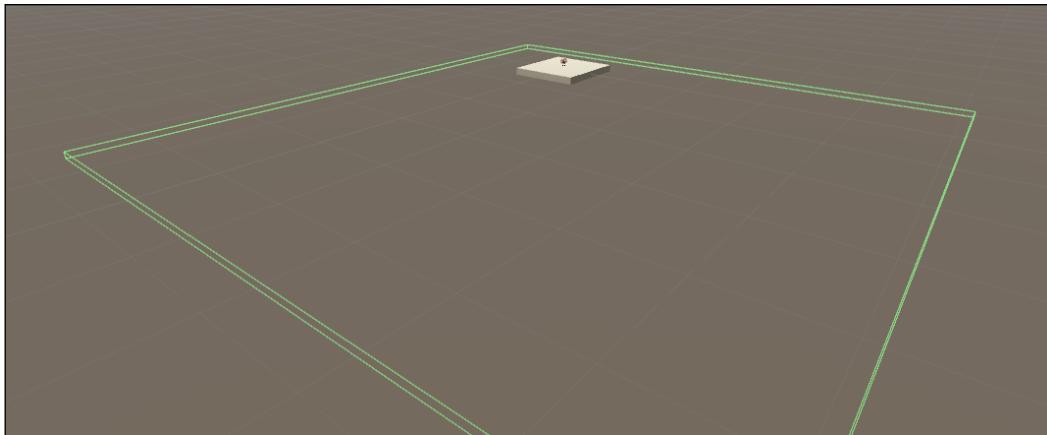
1. First, create a new script and name it `KillVolume`.
2. This script has a single variable. It will keep track of where to put the monkey ball after it has fallen in:

```
public Transform respawnPoint;
```
3. This script also has a single function, `OnTriggerEnter`. This function is called every time an object with a **Rigidbody** component enters a trigger volume. It receives the object that enters as a collider:

```
public void OnTriggerEnter(Collider other) {
```
4. The function simply changes the position of the thing that entered the volume to that of the point where we want to respawn it. The only thing that will be moving around our game will be the monkey ball, so we don't have to worry about any double-checking what has entered. We also set `velocity` to zero so that it doesn't move suddenly when the player regains control:

```
    other.transform.position = respawnPoint.position;
    other.attachedRigidbody.velocity = Vector3.zero;
}
```
5. Next, we need a new empty **GameObject**, named `RespawnPoint`.
6. Position this object at approximately the same location of where our ball starts. This is the point where the ball will be put after it has fallen off the field.
7. Now, create another empty **GameObject** and name it `KillVolume`. This object will catch and reset the game when the player falls in.
8. Set its position to -10 for **Y** and 0 for both **X** and **Z**. This will put it well below where the player is going to be. The important thing for future levels is that this volume is below where the player is normally going to be. If it isn't, they might miss it and fall forever, or suddenly jump back to the beginning, passing through it on their way to an that area they are supposed to be in.

9. We need to give the object a **Box Collider** component and attach our `KillVolume` script.
10. In order to get `OnTriggerEnter` function called by Unity, we need the **Is Trigger** box checked. Otherwise, it will just collide with the volume and appear to the player that they are just floating.
11. Next, we need to make the volume large enough to actually catch our player when they fall in. To do this, set **Size** on the **Box Collider** component to **100** for both the **X** and **Z** axes.
12. Drag the `RespawnPoint` object from the **Hierarchy** window to the **Respawn Point** slot on our `KillVolume` script component in the **Inspector**. Without it, our player will never be able to get back after falling off the map.
13. Finally, delete the `Fence` cubes from our basic environment so that we can test it out. You can move the ball around and when it falls off the ground block, it will hit `KillVolume` and return to `RespawnPoint`.



We now have the ability to reset our players when they fall off the map. The important part is detecting when they are no longer on the map and not interrupting them when they should be. This is why we have made it so large and put it well underneath the main area of our level. However, it would be a bad idea to put the volume too far below the play area, or the player is going to be falling for a long time before the game is reset.

Winning and losing the game

Now that we have the ability to move around and reset if we fall off the map, we just need some way to win or lose the game. This particular type of game is traditionally tracked by how fast you are able to get from one end of the map to the other. If you fail to reach the end before the timer runs out, it is game over. Let's use these steps to create a finish line and a timer for our game:

1. We need a new script named `victoryVolume`.
2. It starts with a pair of variables for tracking the messages for our player. The first will be turned on and shown to the player if they reach the end within the time limit. The second will only display if they run out of time:

```
public GameObject victoryText;  
public GameObject outOfTimeText;
```

3. The next variable will track the `Text` object in the GUI to display the current amount of time left to complete the level:

```
public Text timer;
```

4. This variable is for setting how much time, in seconds, is available for a player to complete the level. When adjusting this in the **Inspector** panel for a larger version of the game, it is a good idea to have several people test the level in order to get a feel of how long it takes to complete:

```
public float timeLimit = 60f;
```

5. Our last variable for the script will simply track whether or not our timer can actually count down. By making it `private` and defaulting it to `true`, the timer will always start counting from the moment the level loads:

```
private bool countDown = true;
```

6. The first function for this script is `Awake`, which is the best location for initialization. The only thing it does is turn off both of the messages. We will turn on the appropriate one later, based on how our player performs:

```
public void Awake() {  
    victoryText.SetActive(false);  
    outOfTimeText.SetActive(false);  
}
```

7. To detect when the player has crossed the finish line, we will be using the same `OnTriggerEnter` function that we used for the `KillVolume` script. Here, however, we will first check to make sure that we are still timing the player. If we are no longer timing them, they must have run out of time and lost. Therefore, we should not let them cross the finish line and win:

```
public void OnTriggerEnter(Collider other) {  
    if(countDown) {
```

8. Next, we turn on the text that tells the player that they have won. We have to let them know at some point, so it might as well be now:

```
    victoryText.SetActive(true);
```

9. The next thing the function does is essentially turn the physics off for the monkey ball to stop it from rolling around. By using `attachedRigidbody`, we gain access to the **Rigidbody** component, which is the part hooking it into Unity's physics engine that is attached to the object. Then, we set its `isKinematic` property to `true`, essentially telling it that it will be controlled by the script and not by the physics engine:

```
    other.attachedRigidbody.isKinematic = true;
```

10. Finally, the function stops counting the player's time:

```
    countDown = false;  
}  
}
```

11. The last function for this script is the `Update` function, which first checks to make sure that the timer is running:

```
public void Update() {  
    if(countDown) {
```

12. It then removes the time since the last frame, from the time remaining to complete the level:

```
    timeLimit -= Time.deltaTime;
```

13. Next, we update the time on screen with the amount of time that remains. The text on the screen must be in the form of a string, or words. A number, such as our remaining time, is not a word, so we use the `ToString` function on it to convert it into the right datatype for it to be displayed. Leaving it at that would have been fine, but it would have displayed a bunch of extra decimal places that the player wouldn't have even cared about. Therefore, `0.00` is passed to the function. We are telling it what format and how many decimal places we want the number to have when it becomes a word. This makes it more meaningful to our players and much easier to read:

```
    timer.text = timeLimit.ToString("0.00");
```

14. After checking to see whether the player is out of time, we turn on the text that tells them that they have lost and turn off the time display. We also stop counting the time. If they are already out of time, what is the point in counting?

```
if(timeLimit <= 0) {  
    outOfTimeText.SetActive(true);  
    timer.gameObject.SetActive(false);  
    countDown = false;  
}  
}  
}
```

15. Now, we need to return to Unity and make this script work. Do this by first creating a new empty **GameObject** and naming it **VictoryPoint**.
16. It is going to need three cubes as children. Remember, you can find them by navigating to **GameObject | 3D Object | Cube**.
17. The first cube should be positioned at 1 for **X**, 1 for **Y**, and 0 for **Z**. In addition, give it a scale of 0.25 for **X**, 2 for **Y**, and 0.25 for **Z**.
18. The second cube should have all of the same settings as the first one, except for having a position of -1 for **X**, which moves it to the opposite side of the object.
19. The last cube needs a position of 0 for **X**, 2.5 for **Y**, and 0 for **Z**. Its scale needs to be set as 2.25 for **X**, 1 for **Y**, and 0.25 for **Z**. Together, these three cubes give us a basic-looking finish line, which will stand out from the rest of the game board.
20. Next, we are going to need some text objects for the GUI. Create three of them, by navigating to **GameObject | UI | Text**.
21. The first should be named **Timer**; this will handle the display, showing how much time remains for the player to reach the finish line. It needs to be anchored to the **top-left** with a position of 80 for **Pos X** and -20 for **Pos Y**. It also needs a value of 130 for **Width** and a value of 30 for **Height**. We can also change the default text to 0.00 so that we have a better idea of how it will look in the game. A **Font Size** value of 20 and **Alignment** of **left-center** will position it well for us.
22. The second text object should be named **victory**; it will display the message shown when the player reaches the finish line. It needs to be anchored in **middle-center** with a position of 0 for **Pos X** and **Pos Y**. It needs a value of 200 for **Width** and a value of 60 for **Height** so that we will have enough space to draw the message. Change the default text to You Win!, increase **Font Size** to 50, and select **middle-center** for **Alignment** so that we get a nice, big message in the center of the screen.

23. The last text object should be named `outOfTime`; it will display the message when the player fails to reach the end, before the timer runs down. It shares all the same settings as the previous one, except it needs a value of `500` for **Width** to fit its larger default text of `You Ran Out Of Time!`.
24. Next, we need to select `VictoryPoint` and give it a **BoxCollider** component, as well as our `VictoryVolume` script.
25. The **BoxCollider** component is going to need the **Is Trigger** box to be checked. It needs a value of `0` for **X**, `1` for **Y**, and `0` for **Z** for **Center**. In addition, **Size** should be `1.75` for **X**, `2` for **Y**, and `0.25` for **Z**.
26. Finally, drag each of the text objects that we just created to the appropriate slot on the `VictoryVolume` script component.



We've just finished putting together a means by which the player can either win or lose the game. If you were to try it out now, you should be able to see the timer tick down in the top-left corner of the screen. When you manage to reach the finish line in time, a nice message is displayed indicating this. If you are not quite as successful in reaching it, a different message is displayed.

This is the entire interface that we will be creating for this game, but it is still awfully bland. Use your skills from what you learned in *Chapter 2, Looking Good – The Graphical Interface* to style the interface. It should look pleasing and exciting, perhaps even monkey-themed. To get extra fancy, you could try to set it up to change colors and size as the remaining time approaches zero, giving the player an indication at a glance of where they stand in terms of the time remaining to complete that level.

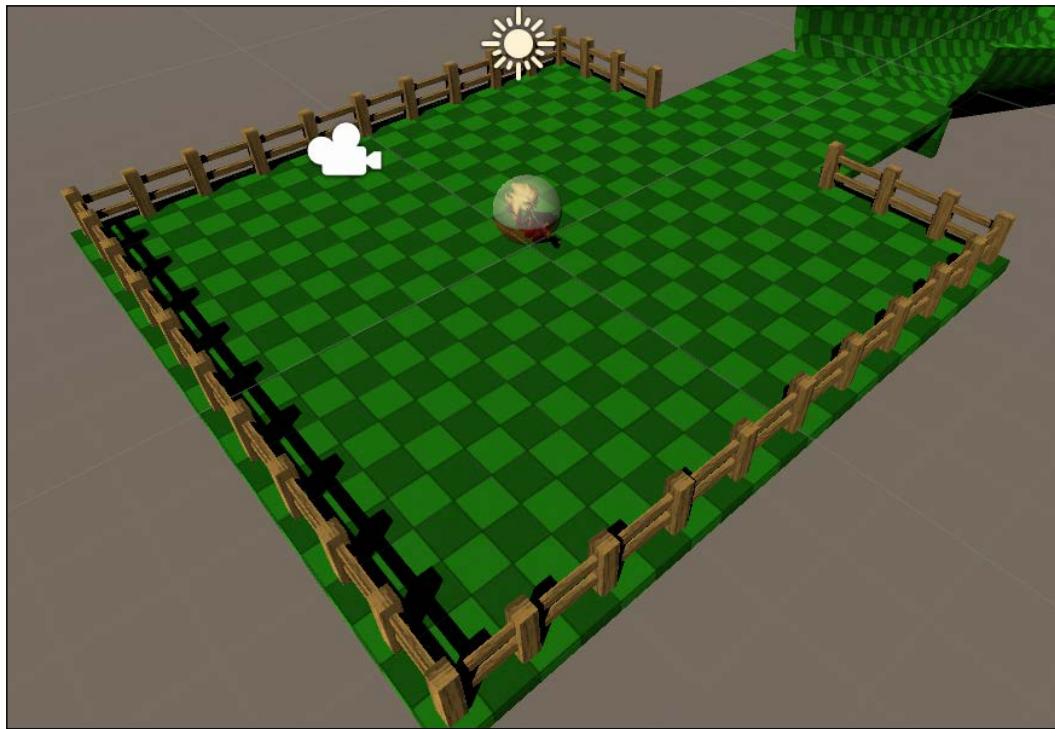
The finish line also looks boring, as it is made only out of cubes. Try your hand at creating a new one. It could have some sort of finish line banner across it, like they have in races. Maybe it could be a little more round-looking. If you wanted to get really fancy, you could look at creating a second timer that would exist at the front of the finish line. It would allow the player to look at the world, where most of their focus will be, and know what their remaining time is.

Putting together the complex environment

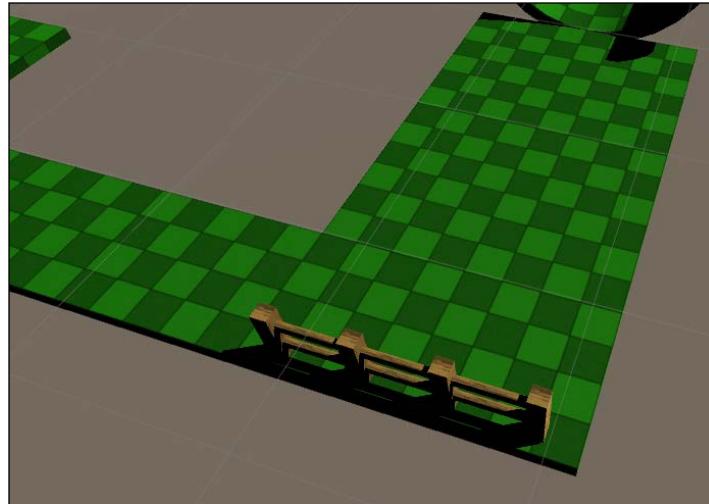
Having a one block map does not make for a very interesting game experience. It works excellently for us to set up the controls, but the players will not find it much fun. So, we need something a little better. Here, we will be setting up a more complex environment with ramps, bridges, and turns. We will also use some fences to help and guide the player. Let's do this all with these steps:

1. Start by adding the `MonkeyBallMap` model to the scene.
2. Set its **Scale** attribute to 100 on each axis and its **Position** attribute to 0 on each axis.
3. If the map appears to be white, then apply the `Grass` texture to it. This map gives us a good starting platform, a half-pipe ramp, a few turns, and a short bridge. Altogether, there are plenty of basic challenges for the player.
4. To enable our ball to actually use the map, it is going to need some colliders to make it physical. Expand `MonkeyBallMap` in the **Hierarchy** window and select both `FlatBits` and `HalfPipe`.
5. On to each of these objects add a **MeshCollider** component, just like we did for some of the parts of our tank city. Remember, it can be found by navigating to **Component | Physics | Mesh Collider**.
6. Next, we have the `Fence` model. With this, we can both help and hinder the player by placing guardrails along the edges or blockages in their path. Start by dragging the `Fence` model into the scene and setting its **Scale** to 100 to keep it sized in proportion to our map.
7. To enable the fences to physically block the player, they need a collider. To both of the children fence objects, add a **BoxCollider** component that can be found by navigating to **Component | Physics | Box Collider**.

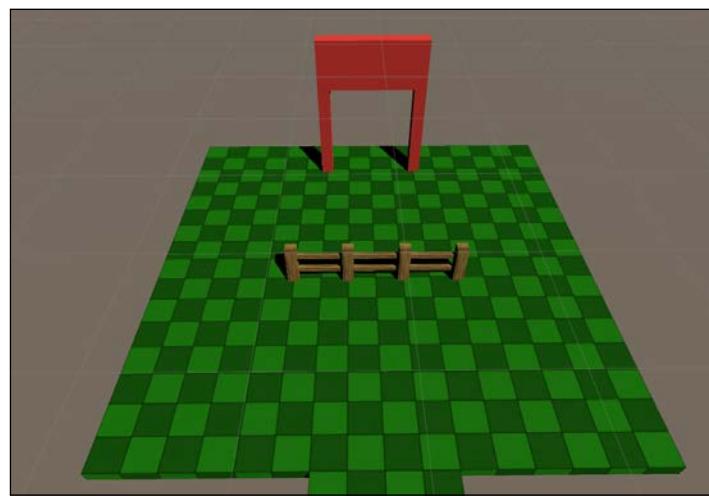
8. In addition, ensure that you apply the wood texture to both of the fence pieces if they appear white in the scene.
9. Create a new empty **GameObject** and name it **Fences**. Then, set its **Position** attribute to 0 on each axis. This object will help us to stay organized because we could end up with a great many pieces of fence.
10. Now, expand the **Fence** model in the **Hierarchy** window and make both **Post** and **PostWithSpokes** children of the **Fences** empty **GameObject**. Then, delete the **Fence** object. By doing this, we break the prefab connection and remove the risk of recreating it. If we had just used the **Fence** object for our organization, there was a risk that we could've ended up deleting all of our work that we put in setting them up in the scene if we made changes to the original model file.
11. We need to position the fences in strategic locations to affect how our player is able to play the game. The first place we might want to put them is around the starting area, giving our player a nice safe beginning to the game. Remember, you can use **Ctrl + D** to duplicate the fence pieces so that you will always have enough fences.



12. The second place to put fences would be after the half pipe, just before the bridge. Here, they would help the player to reorient themselves before they try to go across the small bridge:



13. The last place we could put them would be a hindrance to the player. If we place them in the middle of the final platform, we would force the player to go around and risk falling off before reaching the end.
14. Speaking of the finish line, now that everything else is placed, we need to move it into position. Place it at the end of the lower platform. Here, the player will have to face all of the challenges of the map and risk falling many times before they finally achieve victory by reaching the end.



That is it for setting up our complex environment. We gave the player a chance to orient themselves before forcing them to navigate a handful of challenges and reach the end. Give it a try. Our game is starting to look really nice.

The first challenge here might be quite obvious. Try your hand at making your own map with ramps, bridges, chutes, and blockages. You could perhaps make a big maze out of the fences. Otherwise, you could alter the level so that it actually requires the player to go up some straight paths and ramps, meaning the player would need enough speed to make it. There might even be the need for a few jumps. Make the player go down a ramp to gain speed before jumping across to another platform. Whatever your new level becomes, make sure that `KillVolume` is below it and covers plenty of area underneath. You never quite know how the players will play and where they will manage to get themselves stuck.

The map itself looks pretty good, but the area around it could use some work. Use your skills from the previous chapters—add a skybox to the world, something that looks better than the defaults. While you're at it, work with the lighting. A single **Directional Light** is nice but not very interesting. Create some light source models to place round the map. Then, bake the lightmaps to create some good quality shadows.

Adding bananas

When it comes to a game with monkeys, the most obvious thing for our player to collect is bananas. However, it is never enough to have items in the world for players to collect; we have to show the player that the items are collectible. Usually, this means that the thing is spinning, bouncing, shining, generating sparks, or demonstrating some other special effect. For our game, we are going to make the bananas bounce up and down while spinning in place. Let's do it with these steps:

1. Right off the bat, we are going to need a new script. Create one and name it `BananaBounce`.
2. This script begins with three variables. The first is how fast, in meters per second, the banana will move up and down. The second is how high the banana will go from its starting position. The third is at how many degrees per second the banana will spin in place. Altogether, these variables will let us easily control and tweak the movement of the banana:

```
public float bobSpeed = 1.5f;  
public float bobHeight = 0.75f;  
public float spinSpeed = 180f;
```

3. This next variable will keep track of the actual object that will be moving. By using two objects for the setup and control of the banana, we are able to separate position and rotation and make everything easier:

```
public Transform bobber;
```

4. The function for this script is `Update`. It first checks to make sure that our `bobber` variable has been filled. Without it, we can't do anything to make our banana move:

```
public void Update() {  
    if(bobber != null) {
```

5. Next, we use the `PingPong` function to calculate a new position for our banana. This function bounces a value between zero and the second value you give it. In this case, we are using the current time multiplied by our speed to determine how far the banana might have moved in this game. By giving it a height to it, we end up with a value that moves back and forth from zero to our maximum height. We then multiply it by an up vector and apply it to our `localPosition` so that the banana will move up and down:

```
        float newPos = Mathf.PingPong(Time.time * bobSpeed, bobHeight);  
        bobber.localPosition = Vector3.up * newPos;  
    }
```

6. Finally, we use the same `Rotate` function that we used for rotating turrets to make the banana spin in its place. It will just do this constantly at whatever speed we tell it to:

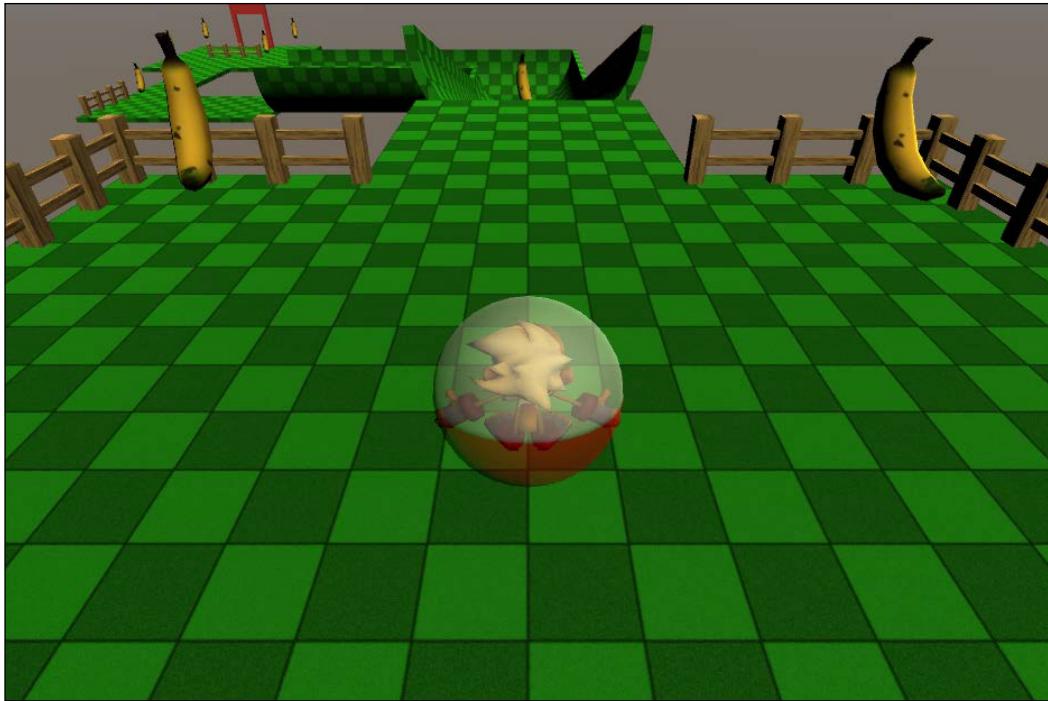
```
        transform.Rotate(Vector3.up * Time.deltaTime * spinSpeed);  
    }
```

7. Next, we need to return to Unity and set these bananas up. To do this, we first need to add the Banana model to the scene. If it is white, be sure to add the Banana texture to it.

8. Add our `BananaBounce` script to the new banana, or else it is just not going to set there.

9. The child object of Banana needs to be put in the **Bobber** slot on our script component.

10. Finally, turn it into a prefab and scatter them about the map: a few at the beginning area, a few near the finish line, and some along the way.



If you try out the game now, you should have several happily bouncing bananas. By using the `Mathf.PingPong` function, our jobs are made very easy for the creation of this effect. Without it, we would have needed to do a bunch of extra calculations to figure out whether we were moving up or down and how far along we were.

Having bananas as collectibles is great, but which game these days has a single type of pickup? Try your hand at making some other pickup models. The most obvious one would be some banana bundles, such as the bunches that you can buy at the grocery store or the huge bunch that actually grows on banana trees. However, you could also go down the route of coins, energy crystals, ancient monkey totems, checkpoints, score multipliers, or anything else that might catch your attention.

Collecting bananas with touch

One of the most obvious features of the modern mobile device is the touch screen. Devices use the electrical conductivity of the user's finger and many tiny contact points to determine the location that is being touched. In order to explore the possibilities of the touch interface for our game, we will be making our players poke the bananas on the screen rather than running into them to collect them. Unity provides us with easy access to the touch inputs. By combining the input with ray casts, as we did for making the tanks fire, we can determine which object in the 3D space was touched by the user. For us, this means we can give the player the ability to touch and collect those bananas. To do it, let's use these steps:

1. First up, we need a new script. Create one and name it `BananaTouch`.
2. The `Update` function is the only function in this script. It starts by checking to see whether the player is touching the screen in any way. The `Input` class provides us with the `touchCount` value, which is simply a counter for how many fingers are currently touching the device's screen. If there are no fingers touching, we don't want to waste our time doing any work, so we exit early with `return`: and are ready to check the next frame again to see whether the player is touching the screen.

```
public void Update() {  
    if (Input.touchCount <= 0) return;
```

3. We next create a `foreach` loop. This is a loop that will check each item in the list of touches, but it will not track the index of that touch. We then check the phase of each touch to see whether it has just started touching the screen. Every touch has five potential phases: **Began**, **Moved**, **Stationary**, **Ended**, and **Canceled**:

```
foreach(Touch next in Input.touches) {  
    if (next.phase == TouchPhase.Began) {
```

Here is the description for each state:

- **Began**: This phase of touch occurs when the user first touches the screen.
- **Moved**: This phase of touch occurs when the user moves his or her finger across the screen.
- **Stationary**: This phase of touch is the opposite of the previous phase; this happens when the user's finger is not moving across the screen.
- **Ended**: This phase of touch occurs when the user's finger is lifted off the screen. This is the normal way for a touch to complete.

- **Canceled:** This phase of touch occurs when an error occurs while tracking the touch. This phase tends to occur most often when a finger is touching the screen but not moving for a lot of time. The touch system is not perfect, so it assumes that it missed the finger being lifted off the screen and just cancels that touch.

4. Next, we create a pair of variables. As with our tanks, the first is a holder for what was hit by our raycast. The second is a Ray, which is just a container for storing a point in space and a directional vector. The ScreenPointToRay function is specially provided by the camera for converting touch positions from the 2D space of the screen to the 3D space of the game world:

```
RaycastHit hit;
Ray touchRay = Camera.main.ScreenPointToRay(next.position);
```

5. The last step for the function is to call the Raycast function. We pass the ray and the tracking variable to the function. If an object is hit, we send it a message to tell it that it has been touched, just like shooting things with our tank. In addition, there are several curly braces that are required to close off the if statements, loop, and function:

```
if(Physics.Raycast(touchRay, out hit)) {
    hit.transform.gameObject.SendMessage("Touched",
    SendMessageOptions.DontRequireReceiver);
}
}
```

6. Before we can try it out, we need to update our BananaBounce script to give it some health and allow it to be destroyed when its health runs out. So, open it now.
7. First, we need a pair of variables. The first is health. Actually, this is just the number of touches that are required to destroy the banana. If we had multiple types of bananas, they could each have a different amount of health. The second variable is a modifier for the banana's speed of movement. Every time the banana loses health, it will slow down, indicating how much health it has left:

```
public int health = 3;
public float divider = 2f;
```

8. Next, we need to add a new function. This Touched function will receive the message from our BananaTouch script. It works similar to how we shot with our tank. The very first thing it does is reduce the remaining health:

```
public void Touched() {  
    health--;
```

9. After some damage has been done, we can slow the movement of the banana by doing a little division. This way it is easy for the player to know whether their touch was successful:

```
    bobSpeed /= divider;  
    spinSpeed /= divider;
```

10. Finally, the function checks to see whether the banana has run out of health. If it has, we use the Destroy function to get rid of it, just like the enemy tanks:

```
    if(health <= 0) {  
        Destroy(gameObject);  
    }  
}
```

11. When you return to Unity, you need to attach our new BananaTouch script to the MonkeyBall object. Due to the way it works, it could technically go on any object, but it is always a good practice to keep player control scripts together and on what they are controlling.

12. Next, add a **Sphere Collider** component to one of your bananas; this can be found by navigating to **Component | Physics | Sphere Collider**. If we make the changes to one and update the prefab, all the bananas in the scene will be updated.

13. Check the **Is Trigger** box so that the bananas do not block the movement of our monkey,. They will still be touchable while allowing our monkey to pass through them.

14. The collider also needs to be positioned in a place where the player will mostly touch when they hit it. So, set the **Center** to 0 for **X**, 0.375 for **Y**, and 0 for **Z**. In addition, make sure the **Radius** is set to 0.5.

15. Finally, be sure to hit the **Apply** button at the top right of the **Inspector** panel to update all the bananas in the scene.



When you try out the game now, you should be able to touch any of the bananas. Initially, all of the bananas will move up and down evenly, like they did earlier. As you touch them, the ones that you touched will move slower, thanks to the bit of division that we made, before they are finally deleted. This lets our player easily see which bananas have been touched and which haven't.

The next step from having collectible objects in a game is to give meaning to the player. This is mostly done by giving them some point value. Try to do that here. It could be done in a very similar manner to the point system we had when we were destroying enemy tanks. If you created some other pickups earlier, you could set each of them up to have different amounts of health. They could also give you different amounts of points as a result. Play around with the numbers and settings until you find something that will be fun for the player to interact with.

Summary

In this chapter, we learned about the specialties of the modern mobile device. We created a Monkey Ball game to try this out. We gained access to the device's gyroscope to detect when it is rotated. This gave our monkey the ability to be directed. After creating a more complex and interesting environment for the player to move around, we created a bunch of bananas that bob up and down while spinning in place. We also made use of the touch screen to give the player the ability to collect the bananas.

In the next chapter, we will be taking a short break from our Monkey Ball game. One of the most popular mobile games on the market, Angry Birds, is a distinct and not uncommon type of game. In order to learn about physics in Unity and the possibility of a 2D-style game, we will be making an Angry Birds clone. We will also take a look at Parallax scrolling to help us create a pleasing background. Before you know it, we will be creating all of the Angry Birds levels that you always wished you could play.

7

Throwing Your Weight Around – Physics and a 2D Camera

In the previous chapter, you learned about the special features of a mobile device and how to create touch and tilt controls. We also created a Monkey Ball game to use these new controls. The steering of the ball was done by tilting the device and collecting bananas by touching the screen. We also gave it some win and lose conditions by creating a timer and finish line.

In this chapter, will we take a short break from the Monkey Ball game to explore Unity's physics engine. We will also take a look at the options available for creating a 2D game experience. To do all of this, we will be recreating one of the most popular mobile games on the market, **Angry Birds**. We will use physics to throw birds and destroy structures. We will also take a look at the creation of a level-selection screen.

In this chapter, we will cover the following topics:

- Unity physics
- Parallax scrolling
- 2D pipelines
- Level selection

We will be creating a new project for this chapter, so start up Unity and let's begin!

2D games in a 3D world

Perhaps the most little-known thing when developing games is the fact that it's possible to create 2D-style games in a 3D game engine, such as Unity. As with everything else, it comes with its own set of advantages and disadvantages, but the choice can be well worth it for generating a pleasing game experience. The foremost advantage is that you can use 3D assets for the game. This allows dynamic lighting and shadows to be easily included. However, when using a 2D engine, any shadow will need to be painted directly into the assets and you will be hard-pressed to make it dynamic. On the side of disadvantages is the use of 2D assets in the 3D world. It is possible to use them, but large file sizes become necessary to achieve the desired detail and to keep it from appearing pixelated. Most 2D engines, however, make use of vector art that will keep the image's lines smooth as it is scaled up and down. Also, one is able to use normal animations for the 3D assets, but frame-by-frame animation is generally required for any 2D asset. Altogether, the advantages have outweighed the disadvantages for many developers, creating a large selection of great looking 2D games that you may never realize were actually made in a 3D game engine.

To address the growing demand from developers for 2D game support, the Unity team has been additionally been working long and hard on creating an optimized 2D pipeline for the 3D engine. When creating your project, you have the option to select 2D defaults, optimizing assets for use in a 2D game. While there is still no direct vector graphics support from Unity, many other features have been optimized to work better in a 2D world. One of the biggest features is the 2D optimization of the physics engine, which we will be focusing on in this chapter. All the principles that we will use will transfer over to 3D physics, which will save some trouble when setting up and working with it.

Setting up the development environment

To explore making a 2D game in a primarily 3D engine, and the use of physics, we will be recreating a highly popular 2D game, Angry Birds. However, before we can dive into the meat of the game, we need to set up our development environment so that we are optimized for 2D game creation. Let's use these steps to do this:

1. To begin with, we need to create a new project in Unity. Naming it `Ch7_AngryBirds` will work well. We also need to select **2D** under **Templates**, so all the defaults are set for our 2D game.
2. We also need to be sure to change the target platform in the **Build Settings** field to **Android** and set **Bundle Identifier** to an appropriate value. We don't want to have to worry about this later.

3. There are a few differences that you will notice right away. First, you can only pan from side to side and up and down when moving around in the scene. This is a setting that can be toggled in the top-middle of the **Scene** view, by clicking on the little **2D** button. Also, if you select the camera in the **Hierarchy** window, you can see that it simply appears as a white box in the **Scene** view. This is because it has been defaulted to use the **Orthographic** mode for its **Projection** setting, which you can see in the **Inspector** panel.



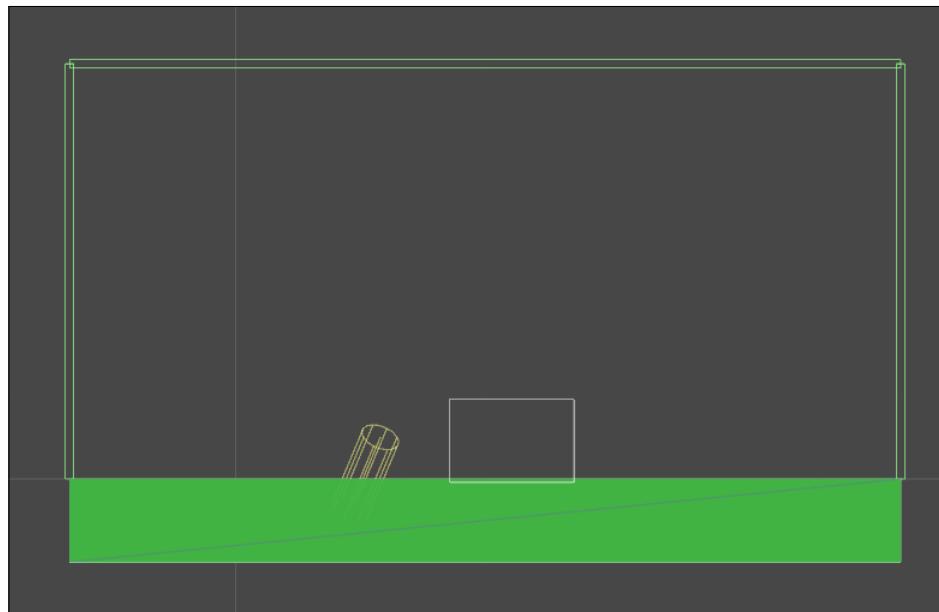
Every camera has two options for how to render the game. A perspective renders everything by utilizing their distance from the camera, imitating the real world; objects that are farther away from the camera are drawn smaller than objects that are closer. An orthographic camera renders everything without this consideration; objects are not scaled based on their distance from the camera.

4. Next, we are going to need a ground. So, go to the menu bar of Unity and navigate to **GameObject** | **3D Object** | **Cube**. This will work well as a simple ground.
5. To make it look a little like a ground, create a green material and apply it to the **Cube** **GameObject**.
6. The ground cube needs to be large enough to cover the whole of our field of play. To do this, set the cube's **Scale** attribute to 100 for the **X** axis, 10 for the **Y** axis, and 5 for the **Z** axis. Also, set its **Position** attribute to 30 for the **X** axis, -5 for the **Y** axis, and 0 for the **Z** axis. Since nothing will be moving along the **x** axis, the ground only needs to be large enough for the other objects that will be in our scene to land on. It does, however, need to be wide and tall enough to keep the camera from seeing the edges.
7. To optimize our ground cube for use in our 2D game, we need to change its collider. Select the **Cube** **GameObject** in the **Hierarchy** window and take a look at it in the **Inspector** panel. Right-click on the **Box Collider** component and select **Remove Component**. Next, at the top of Unity, navigate to **Component** | **Physics 2D** | **Box Collider 2D**. This component works just like a normal **Box Collider** component, except that it does not have limited depth.
8. Right now, the ground looks rather dark due to the lack of light. From the menu bar of Unity, navigate to **GameObject** | **Light** | **Directional Light** in order to add some brightness to the scene.

9. Next, we need to keep all the objects that will be flying around the scene from straying too far and causing problems. To do this, we need to create some trigger volumes. The simplest way to do this is to create three empty **GameObjects** and give each a **Box Collider 2D** component. Be sure to check the **Is Trigger** checkbox in order to change them into trigger volumes.
10. Position one at each end of the ground object and the last GameObject at about 50 units above. Then, scale them to form a box with the ground. Each should be no thicker than a single unit.
11. To make the volumes actually keep objects from straying too far, we need to create a new script. Create a new script and name it `GoneTooFar`.
12. This script has a single, short function, `OnTriggerEnter2D`. We use this function to destroy any object that might enter the volume. This function is used by Unity's physics system to detect when an object has entered a trigger volume. We will go into more detail regarding this later, but for now, know that one of the two objects, either the volume or the object entering it, needs a **Rigidbody** component. In our case, everything that we might want to remove when they enter the trigger will have a **Rigidbody** component:

```
public void OnTriggerEnter2D(Collider2D other) {  
    Destroy(other.gameObject);  
}
```

13. Finally, return to Unity and add the script to the three trigger-volume objects.



We have done the initial setup for our 2D game. By changing the project type from **3D** to **2D**, defaults in Unity are changed to be optimized for 2D game creation. The most immediately noticeable thing is that the camera is now in the **Orthographic** view, making everything appear flattened. We also created a ground and some trigger volumes for our scene. Together, these will keep our birds and anything else from straying too far.

Physics

In Unity, physics simulation primarily focuses on the use of the **Rigidbody** component. When the **Rigidbody** component is attached to any object, it will be taken over by the physics engine. The object will fall with gravity and bump into any object that has a collider. In our scripts, making use of the `OnCollision` group of functions and the `OnTrigger` group of functions requires a **Rigidbody** component to be attached to at least one of the two interacting objects. However, a **Rigidbody** component can interfere with any specific movement we might cause the object to take. But the **Rigidbody** component can be marked as kinematic, which means that the physics engine will not move it, but it will only move when our script moves it. The **CharacterController** component that we used for our tank is a special, modified **Rigidbody**. In this chapter, we will be making heavy use of the **Rigidbody** component to tie all our birds, blocks, and pigs into the physics engine.

Building blocks

For our first physics objects, we will create the blocks that the pig castles are built out of. We will be creating three types of blocks: wood, glass, and rubber. With these few simple blocks, we will be able to easily create a large variety of levels and structures to be smashed with birds.

Each of the blocks we will be creating will be largely similar. So, we will start with the basic one, the wooden plank, and expand upon it to create the others. Let's use these steps to create the blocks:

1. First, we will create the plank of wood. To do this, we need another cube. Rename it `Plank_Wood`.
2. Set the value of the plank's **Scale** to `0.25` for the **X** axis and `2` for both the **Y** and **Z** axes. Its scale on the *x* and *y* axes defines its size as seen by the player. The scale on the *z* axis helps us ensure that it will be hit by other physics objects in the scene.
3. Next, create a new material using the `plank_wood` texture and apply it to the cube.

4. To make this new wooden plank into a physics object suitable for our game, we need to remove the cube's **Box Collider** component and replace it with a **Box Collider 2D** component. Also, add a **Rigidbody** component. Make sure that your plank is selected; go to the menu bar of Unity and navigate to **Component | Physics 2D | Rigidbody 2D**.
5. Next, we need to make the plank function properly within our game; we need to create a new script and name it `Plank`.
6. This script begins with a bunch of variables. The first two variables are used to track the health of the plank. We need to separate the total amount of health from the current health, so that we will be able to detect when the object has been reduced to its half-health. At this point, we will make use of our next three variables to change the object's material to one that shows damage. The last variable is used when the object runs out of health and is destroyed. We will use it to increase the player's score:

```
public float totalHealth = 100f;  
private float health = 100f;  
  
public Material damageMaterial;  
public Renderer plankRenderer;  
private bool didSwap = false;  
  
public int scoreValue = 100;
```

7. For the script's first function, we use `Awake` for initialization. We make sure that the object's current health is the same as its total health and the `didSwap` flag is set to `false`:

```
public void Awake() {  
    health = totalHealth;  
    didSwap = false;  
}
```

8. Next, we make use of the `OnCollisionEnter2D` function, which is just the 2D optimized version of the normal `OnCollisionEnter` function used in 3D. This is a special function, triggered by the **Rigidbody** component, that gives us information about what the object collided with and how. We use this information to find `collision.relativeVelocity.magnitude`. This is the speed at which the objects collided, and we use this as damage in order to reduce the current health. Next, the function checks to see whether the health has been reduced to half and calls the `SwapToDamaged` function if it has. By using the `didSwap` flag, we make sure that the function is only called once.

Finally, the function checks to see whether the health has dropped below zero. If it has, the object is destroyed and we call the `LevelTracker` script, which we will soon be making, to add to the player's score:

```
public void OnCollisionEnter2D(Collision2D collision) {
    health -= collision.relativeVelocity.magnitude;

    if(!didSwap && health < totalHealth / 2f) {
        SwapToDamaged();
    }

    if(health <= 0) {
        Destroy(gameObject);
        LevelTracker.AddScore(scoreValue);
    }
}
```

9. Finally, for the script, we have the `SwapToDamaged` function. It starts by setting the `didSwap` flag to `true`. Next, it checks to make sure that the `plankRenderer` and `damageMaterial` variables have references to other objects. Ultimately, it uses the `plankRenderer.sharedMaterial` value to change the material to the damaged-looking material:

```
public void SwapToDamaged() {
    didSwap = true;
    if(plankRenderer == null) return;

    if(damageMaterial != null) {
        plankRenderer.sharedMaterial = damageMaterial;
    }
}
```

10. Before we can add our `Plank` script to our objects, we need to create the `LevelTracker` script that was mentioned earlier. Create it now.
11. This script is fairly short and starts with a single variable. The variable will track the player's score for the level and is static, so it can easily be changed as objects are destroyed, for points:

```
private static int score = 0;
```

12. Next, we use the `Awake` function to make sure the player starts at zero when beginning a level:

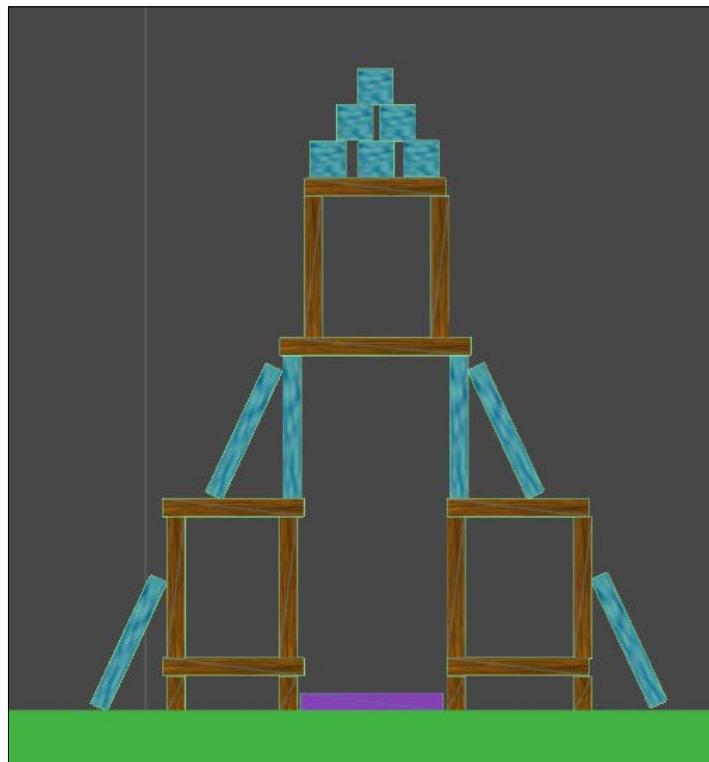
```
public void Awake() {
    score = 0;
}
```

13. Finally, for the script, we add the `AddScore` function. This function simply takes the amount of points passed to it and increases the player's score. It is also static, so it can be called by any object in the scene without needing a reference to the script:

```
public static void AddScore(int amount) {  
    score += amount;  
}
```

14. Back in Unity, we need to create a new material using the `plank_wood_damaged` texture. This will be the material that the script will swap to.
15. We need to add the `Plank` script to our `Plank_Wood` object. Connect the **Damaged Material** reference to the new material and the **Plank Renderer** reference to the object's **Mesh Renderer** component.
16. As we create different types of planks, we can adjust the value of **Total Health** to give them different strengths. A value of 25 works pretty well for the wood planks.
17. Next, create an empty `GameObject` and rename it `LevelTracker`.
18. Add the `LevelTracker` script to the object, and it will begin to track the player's score.
19. If you want to see the wood plank in action, position it above the ground and hit the play button. As soon as the game starts, Unity's physics will take over and drop the plank with gravity. If it started out high enough, you will be able to see it switch textures as it loses health.
20. To make the other two planks that we need, select the `Plank_Wood` object and press `Ctrl + D` twice to duplicate it. Rename one plank to `Plank_Glass` and the other to `Plank_Rubber`.
21. Next, create three new materials. One should be purple in color for the rubber plank, one should use the `plank_glass` texture for the glass plank, and the last material should use the `plank_glass_damaged` texture for when the glass plank is damaged. Apply the new materials to the proper locations for the new planks.
22. As for the health of the new planks, a value of 15 for the glass and 100 for the rubber will work well.

23. Finally, turn your three planks into prefabs and use them to build a structure for you to knock down. Feel free to scale them in order to make differently sized blocks, but leave the z axis alone. Also, all of the blocks should be positioned at 0 on the z axis and your structure should be centered around about 30 on the x axis.



We have created the building blocks we needed for the structures that are going to be knocked down in our game. We used a **Rigidbody** component to tie them into the physics engine. Also, we created a script that keeps track of their health and swaps to damaged materials when it drops below half. For this game, we are sticking to the 2D optimized versions of all the physics components. They work in exactly the same way as the 3D versions, just without the third axis.

Wood and glass work well as basic blocks. However, if we are going to make harder levels, we need something a little stronger. Try your hand at making a stone block. Create two textures and materials for it to show its pristine and damaged states.

Physics materials

Physics materials are special types of materials that specifically tell the physics engine how two objects should interact. This does not affect the appearance of an object. It defines the friction and bounciness of a collider. We will use them to give our rubber plank some bounce and the glass plank some slide. With these few steps, we can quickly implement physics materials to create a pleasing effect:

1. Physics materials are created in the same way as everything else, in the **Project** panel. Right-click inside the **Project** panel and navigate to **Create | Physics2D Material**. Create two physics materials and name one of them **Glass** and the other **Rubber**.
2. Select one of them and take a look at it in the **Inspector** window. The 2D version has only two values (the 3D version has a few extra values, but they are only used in more complex situations):
 - **Friction**: This property controls the amount of movement lost when sliding along a surface. A value of zero denotes no friction, such as ice, and a value of one denotes a lot of friction, such as rubber.
 - **Bounciness**: This property is how much of an object's energy is reflected when it hits something or is hit by something. Zero means none of the energy is reflected, while a value of one means the object will reflect all of it.
3. For the **Glass** material, set the **Friction** value to **0.1** and **Bounciness** to **0**. For the **Rubber** material, set the **Friction** to **1** and **Bounciness** to **0.8**.
4. Next, select your **Plank_Glass** prefab and take a look at its **Box Collider 2D** component. To apply your new physics materials, simply drag and drop them one by one from the **Project** panel to the **Material** slot. Do the same for your **Plank_Rubber** prefab, and any time an object hits one of them, the materials will be used to control their interaction.

We have created a pair of physics materials. They control how two colliders interact when they run into each other. Using these, we are given control over the amount of friction and bounciness that is possessed by any collider.

Characters

Having a bunch of generic blocks is just the beginning of this game. Next, we are going to create a few characters to add some life to the game. We are going to need some evil pigs to destroy and some good birds to throw at them.

Creating the enemy

Our first character will be the enemy pig. On their own, they don't actually do anything. So, they are really just the wooden blocks we made earlier that happen to look like pigs. To make their destruction the goal of the game, however, we are going to expand our `LevelTracker` script to watch them and trigger a **Game Over** event if they are all destroyed. We will also expand the script to update the score on the screen and make it save the score for later use. Unlike our planks, which are cubes that we can only see one side of, pigs are created as flat textures and are used as sprites by Unity's 2D pipeline. Let's get started with these steps to create the pigs for our *Angry Birds* game:

1. The pigs are created in a manner similar to that of the wood planks; however, they use a special 2D object called a sprite. A sprite is really just a flat object that always looks at the screen. Most 2D games are made with just a series of sprites for all the objects. You can create one by navigating to **GameObject | 2D Object | Sprite**. Name it `Pig`.
2. To make your new sprite look like a pig, drag the `pig_fresh` image from the **Project** panel and drop it into the **Sprite** slot of the **Sprite Renderer** component.
3. Next, add a **Circle Collider 2D** component and a **Rigidbody 2D** component. The **Circle Collider 2D** component works just like the **Sphere Collider** components we have used previously but is optimized for working in a 2D game.
4. Before we can use our pigs in the game, we need to update the `Plank` script so that it can handle the changing of sprite images as well as materials. So, we open it up and add a variable at the beginning. This variable simply keeps track of which sprite to change to:

```
public Sprite damageSprite;
```

5. Next, we need to add a small part to the end of our `SwapToDamaged` function. This `if` statement checks whether a sprite is available to change into. If it is, we convert our generic renderer variable into `SpriteRenderer` so that we can get access to the `sprite` variable on it, and update to our new image:

```
if(damageSprite != null) {  
    SpriteRenderer spriteRend = plankRenderer as SpriteRenderer;  
    spriteRend.sprite = damageSprite;  
}
```

6. Add the `Plank` script to the pig and fill in the **Plank Renderer** slot with the **Sprite Renderer** component. Also, put the `pig_damage` image in the **Damage Sprite** slot. By changing this script a little, we will be able to save ourselves a lot of trouble later, when we may perhaps want to track the destruction of more than just pigs.

7. Now, turn the pig into a prefab and add it to your structure. Remember that you need to leave them at zero on the `z` axis, but feel free to adjust their size, health and score values to give them some variety.
8. Next, we need to expand the `LevelTracker` script. Open it up and we can add some more code.
9. First, we need to add a line at the very beginning of the script, so we can edit the text displayed in our GUI. Just like we have done previously, add this line at the very top of the script, where the other two lines that begin with `using` are:

```
using UnityEngine.UI;
```

10. We will next add some more variables at the beginning of the script. The first one, as its name suggests, will hold a list of all the pigs in our scene. The next is a flag for signaling that the game has ended. We also have three `Text` variables, so we can update the player's score while they are playing, tell them why the game ended, and what their final score was. The last variable will allow you to turn on and turn off the final screen, where we tell the player whether or not they won:

```
public Transform[] pigs = new Transform[0];  
  
private gameOver = false;  
  
public Text scoreBox;  
public Text finalMessage;  
public Text finalScore;  
  
public GameObject finalGroup;
```

11. Next, we need to add a line to the `Awake` function. This simply makes sure that the group of GUI objects that tell the player how the game ended are turned off when the game starts:

```
FinalGroup.SetActive(false);
```

12. In the `LateUpdate` function, we first check whether the game has ended. If it hasn't, we call another function to check whether all the pigs have been destroyed. We also update the display of the player's score, both while they are playing and for the game over screen:

```
public void LateUpdate() {  
    if(!gameOver) {  
        CheckPigs();  
  
        scoreBox.text = "Score: " + score;  
        finalScore.text = "Score: " + score;  
    }  
}
```

13. Next, we add the `CheckPigs` function. This function loops through the list of pigs to see whether they are all destroyed. Should it find one that hasn't been destroyed, it exits the function. Otherwise, the game is flagged as being over and the player is given a message. We also turn off the in-game score and turn on the game over a group of GUI objects:

```
private void CheckPigs() {  
    for(int i=0;i<pigs.Length;i++) {  
        if(pigs[i] != null) return;  
    }  
  
    gameOver = true;  
    finalMessage.text = "You destroyed the pigs!";  
  
    scoreBox.gameObject.SetActive(false);  
    finalGroup.SetActive(true);  
}
```

14. The `OutOfBirds` function will be called by the slingshot we are going to create later, when the player runs out of birds to launch at the pigs. If the game has not yet ended, the function ends the game and sets an appropriate message for the player. It also turns off the in-game score and turns on the game over a group of GUI objects, just like the previous function:

```
public void OutOfBirds() {  
    if(gameOver) return;  
  
    gameOver = true;  
    finalMessage.text = "You ran out of birds!";  
  
    scoreBox.gameObject.SetActive(false);  
    finalGroup.SetActive(true);  
}
```

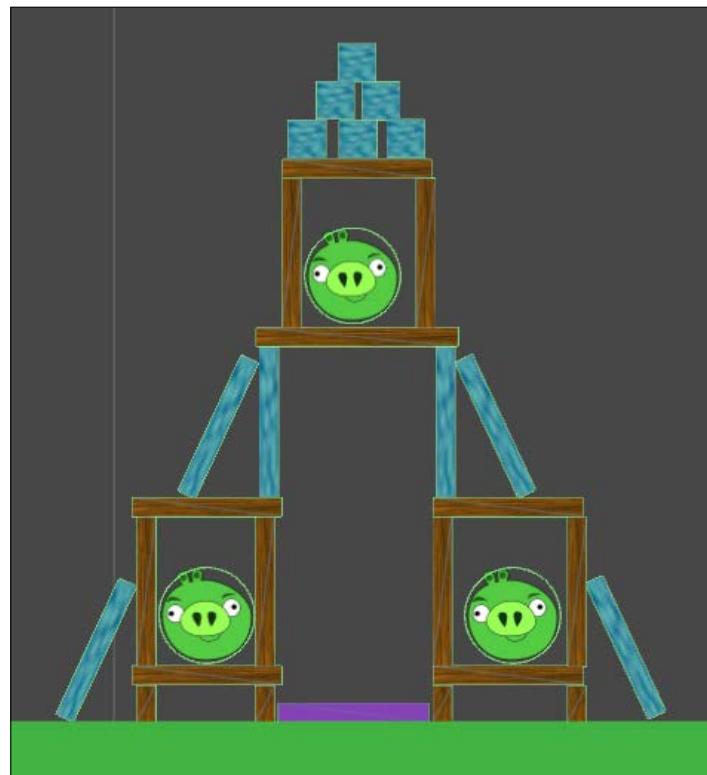
15. Finally, we have the `SaveScore` function. Here, we use the `PlayerPrefs` class. It lets you easily store and retrieve small amounts of data, perfect for our current needs. We just need to provide it with a unique key to save the data under. For this, we use a short string combined with the level's index, as provided by `Application.loadedLevel`. Next, we use `PlayerPrefs.GetInt` to retrieve the last score that was saved. If there isn't one, the zero that we passed to the function is returned as a default value. We compare the new score with the old score and use `PlayerPrefs.SetInt` to save the new score, if it is higher. Finally, the `Application.LoadLevel` function can be used to load any other scene in our game. All the scenes you intend to load have to be added to the **Build Settings** window, found in the **File** menu, and can be loaded by using either their name or their index, as shown here:

```
public void SaveScore() {  
    string key = "LevelScore" + Application.loadedLevel;  
    int previousScore = PlayerPrefs.GetInt(key, 0);  
    if(previousScore < score) {  
        PlayerPrefs.SetInt(key, score);  
    }  
  
    Application.LoadLevel(0);  
}
```

 Note that using `PlayerPrefs` is by far the easiest method of storing saved information in Unity. However, it is not the most secure. If you have experience changing values in the registry of your computer, you can easily find and make changes to these `PlayerPrefs` values from outside the game. This by no means makes it a bad path for storing game information. You should just be aware of it in case you ever make a game and wish to prevent the player from hacking and changing values in their game saves.

16. Next, we need to create some GUI objects so that our player can see how they are doing in the game. Remember that you can find them by navigating to **GameObject** | **UI**. We are going to need three text objects, a button, and a panel.
17. The first text object should be named `Score`. It will display the player's points while the level is in progress. Anchor and position it in the top-left corner of the **Canvas** area.
18. The button needs to be a child of the panel. It should be anchored to the center of the screen and positioned just below it. Also, change the text of the button to something meaningful; `Return to Level Select` will work well here.

19. For **On Click**, we need to click on the plus sign to add a new event. Select the `SaveScore` function of the `LevelTracker` script. Otherwise, we will not be able to record the player's high score and leave the level.
20. The last two text objects should also be made children of the panel. Name one of them `Message`; it will tell our player why the level ended. The other should be named `FinalScore`, displaying the player's score when they are finished. They both need to be anchored to the center of the screen as well. Position the `FinalScore` object above the button, and the message above that.
21. Finally, all the pig objects in our scene need to be added to the `LevelTracker` script's list by dragging and dropping each pig in the `Pigs` value under the **Inspector** window. Also, put each text object into its slot and the panel into the **Final Group** slot.



We created the pigs and updated our `LevelTracker` script to track them. The pigs are really just like the planks of wood, but they are circles instead of boxes. The updated `LevelTracker` script watches for the instance when all the pigs are destroyed and triggers a **Game Over** screen when they are. It also draws the score while the game is being played and saves this score when the level is over.

Our game doesn't quite work yet, but that doesn't mean it has to look like the defaults that Unity provides. Use your skills from the previous chapters to make the interface elements that you have look better. Even just a change in the font will make a world of difference to how our game looks. Perhaps even try changing the background image of Panel, to add that last bit of flare to our game over screen.

Creating the ally

Next, we need something to throw at the pigs and their fortifications. Here, we will create the simplest of birds. The red bird is essentially just a rock. It has no special powers and there is nothing particularly special about its code, besides health. You will also notice that the bird is a 3D model, giving it the shadows that the pigs are missing. Let's use these steps to create the red bird:

1. The red bird is another 3D model, so it is set up in a manner similar to that of the planks. Create an empty **GameObject**, naming it `Bird_Red`, and add the appropriate model from the `birds` model as a child, zeroing out its position and scaling it as needed to make it about a single unit across. The model should be rotated to align it along the `x` axis. If turned a little more toward the camera, the player is able to see the bird's face while still giving the impression of looking down the field of play.
2. Next, give it a **Circle Collider 2D** component and a **Rigidbody 2D** component.
3. Now, we need to create a new script named `Bird`. This script will be a base for all our birds, tracking their health and triggering their special powers when appropriate.
4. The script starts with three variables. The first will keep track of the bird's current health. The second is a flag, so the bird will only use its special power once. It is marked as `protected` so that all our birds can use it while protecting it from interference from outside sources. The last will hold a reference to our **Rigidbody** component:

```
public float health = 50;  
protected bool didSpecial = false;  
public Rigidbody2D body;
```

5. The `Update` function does three checks before activating the bird's special power. First, it checks whether it has already been done and then whether the screen has been touched. We can easily check whether any amount of touching has been done in this frame by checking the left mouse button, which Unity triggers if we touch our screen. Finally, it checks whether the bird has a **Rigidbody** component and whether it is being controlled by another script:

```
public void Update() {  
    if(didSpecial) return;
```

```
if (!Input.GetMouseButtonDown(0)) return;
if (body == null || body.isKinematic) return;

DoSpecial();
}
```

6. In the case of the red bird, the `DoSpecial` function only sets its flag to `true`. It is marked as `virtual` so that we can override the function for the other birds and make them do some fancy things:

```
protected virtual void DoSpecial() {
    didSpecial = true;
}
```

7. The `OnCollisionEnter2D` function works in a similar way to that of the planks, subtracting health based on the strength of the collision and destroying the bird if it runs out of health:

```
public void OnCollisionEnter2D(Collision2D collision) {
    health -= collision.relativeVelocity.magnitude;
    if (health < 0)
        Destroy(gameObject);
}
```

8. Return to Unity and add the script to the `Bird_Red` object.
9. Complete the bird's creation by turning it into a prefab and deleting it from the scene. The slingshot we will be creating next will handle the creation of the birds when the game starts.

We created the red bird. It is set up just like our other physics objects. We also created a script to handle the bird's health. This script will be expanded later when we create the other birds for our game.

Controls

Next, we are going to give the player the ability to interact with the game. First, we will create a slingshot to throw the birds. Following that we will create the camera controls. We will even create a nice background effect to round out the look of our game.

Attacking with a slingshot

To attack the pig fortress, we have our basic bird ammo. We need to create a slingshot to hurl this ammo at the pigs. It will also handle the spawning of the birds at the beginning of the level and automatically reload as birds are used. When the slingshot runs out of birds, it will notify the `LevelTracker` script and the game will end. Finally, we will create a script that will keep the physics simulation from going on for too long. We don't want to force the player to sit and watch a pig slowly roll across the screen. So, the script will, after a little while, start damping the movement of the `Rigidbody` components to make them stop rather than keep rolling. To do all of this, we are going to follow these steps:

1. To start off with the creation of the slingshot, add the slingshot model to the scene and position it at the origin. Scale it, as necessary, to make it about four units tall. Apply a light brown material to the `Fork` model and a dark brown one to the `Pouch` model.
2. Next, we need four empty GameObjects. Make them all the children of the `Slingshot` object.

Name the first GameObject `FocalPoint` and center it between the forked prongs of the slingshot. This will be the point through which we fire all the birds.

The second GameObject is `Pouch`. First, set its rotation to 0 for the `X` axis, 90 for the `Y` axis, and 0 for the `Z` axis, making the blue arrow point forward along our field of play. Next, make the `pouch` model a child of this object, setting its position to 0 on the `X` and `Y` axes and -0.5 on the `Z` axis and its rotation to 270 for `X`, 90 for `Y`, and 0 for `Z`. This will make the pouch appear in front of the current bird without having to make a complete pouch model.

The third GameObject is `BirdPoint`; this will position the bird that is being fired. Make it a child of the `Pouch` point and set its position to 0.3 on the `X` axis and 0 for the `Y` and `Z` axes.

The last GameObject is `WaitPoint`; the birds waiting to be fired will be positioned behind this point. Set its position to -4 for the `X` axis, 0.5 for the `Y` axis, and 0 for the `Z` axis.

3. Next, rotate the `Fork` model so that we can see both prongs of the fork while it appears to be pointing forward. The values of 270 for the `X` axis, 290 for the `Y` axis, and 0 for the `Z` axis will work well.
4. The `slingshot` script will provide most of the interaction for the player. Create it now.

5. We start this script with a group of variables. The first group will keep a reference to the damper that was mentioned earlier. The second group will keep track of the birds that will be used in the level. Next is a group of variables that will track the current bird that is ready to be fired. Fourth, we have some variables to hold references to the points we created a moment ago. The `maxRange` variable is the distance from the focal point to which the player can drag the pouch. The last two variables define how powerfully the bird will be launched:

```
public RigidbodyDamper rigidbodyDamper;

public GameObject[] levelBirds = new GameObject[0];
private Rigidbody2D[] currentBirds;
private int nextIndex = 0;
public Transform waitPoint;
public Rigidbody2D toFireBird;
public bool didFire = false;
public bool isAiming = false;

public Transform pouch;
public Transform focalPoint;
public Transform pouchBirdPoint;

public float maxRange = 3;

public float maxFireStrength = 25;
public float minFireStrength = 5;
```

6. As with the other scripts, we use the `Awake` function for initialization. The `levelBirds` variable will hold references to all the bird prefabs that will be used in the level. We start by creating an instance of each one and storing its **Rigidbody** in the `currentBirds` variable. The `isKinematic` variable is set to true on each bird's **Rigidbody** component so that it does not move when it is not in use. Next, it readies the first bird to be fired and, finally, it positions the remaining birds behind `waitPoint`:

```
public void Awake() {
    currentBirds = new Rigidbody2D[levelBirds.Length];
    for(int i=0;i<levelBirds.Length;i++) {
        GameObject nextBird = Instantiate(levelBirds[i]) as
GameObject;
        currentBirds[i] = nextBird.GetComponent<Rigidbody2D>();
        currentBirds[i].isKinematic = true;
    }
}
```

```
    ReadyNextBird();
    SetWaitPositions();
}
```

7. The `ReadyNextBird` function first checks whether we have run out of birds. If so, it finds the `LevelTracker` script to tell it that there are no birds left to fire. The `nextIndex` variable tracks the current location of the birds in the list to be fired by the player. Next, the function stores the next bird in the `toFireBird` variable and makes it a child of the `BirdPoint` object we created; its position and rotation are zeroed out. Finally, the fire and aim flags are reset:

```
public void ReadyNextBird() {
    if(currentBirds.Length <= nextIndex) {
        LevelTracker tracker = FindObjectOfType(typeof(LevelTracker))
as LevelTracker;
        tracker.OutOfBirds();
        return;
    }

    toFireBird = currentBirds[nextIndex];
    nextIndex++;

    toFireBird.transform.parent = pouchBirdPoint;
    toFireBird.transform.localPosition = Vector3.zero;
    toFireBird.transform.localRotation = Quaternion.identity;

    didFire = false;
    isAiming = false;
}
```

8. The `SetWaitPositions` function uses the position of `waitPoint` to position all the remaining birds behind the slingshot:

```
public void SetWaitPositions() {
    for(int i=nextIndex;i<currentBirds.Length;i++) {
        if(currentBirds[i] == null) continue;
        Vector3 offset = Vector3.right * (i - nextIndex) * 2;
        currentBirds[i].transform.position = waitPoint.position -
offset;
    }
}
```

9. The Update function starts by checking whether the player has fired a bird, and watches the `rigidbodyDamper.allSleeping` variable to see whether all the physics objects have stopped moving. Once they do, the next bird is readied to be fired. If we have not fired, the aiming flag is checked and the `DoAiming` function is called to handle the aiming. If the player is neither aiming nor has just fired a bird, we check for touch input. If the player touches close enough to the focal point, we flag that the player has started aiming:

```
public void Update() {
    if(didFire) {
        if(rigidbodyDamper.allSleeping) {
            ReadyNextBird();
            SetWaitPositions();
        }
        return;
    }
    else if(isAiming) {
        DoAiming();
    }
    else {
        if(Input.touchCount <= 0) return;
        Vector3 touchPoint = GetTouchPoint();
        isAiming = Vector3.Distance(touchPoint, focalPoint.position) <
maxRange / 2f;
    }
}
```

10. The `DoAiming` function checks whether the player has stopped touching the screen and fires the current bird when they have. If they have not, we position the pouch at the current touch point. Finally, the pouch's position is limited to keep it within the maximum range:

```
private void DoAiming() {
    if(Input.touchCount <= 0) {
        FireBird();
        return;
    }

    Vector3 touchPoint = GetTouchPoint();

    pouch.position = touchPoint;
    pouch.LookAt(focalPoint);
```

```
        float distance = Vector3.Distance(focalPoint.position, pouch.  
position);  
        if(distance > maxRange) {  
            pouch.position = focalPoint.position - (pouch.forward *  
maxRange);  
        }  
    }  
}
```

11. The `GetTouchPoint` function uses `ScreenPointToRay` to find out where the player is touching in 3D space. This is similar to when we were touching bananas; however, because this game is 2D, we can just look at the ray's origin and return a zero for its z axis value:

```
private Vector3 GetTouchPoint() {  
    Ray touchRay = Camera.main.ScreenPointToRay(Input.GetTouch(0).  
position);  
    Vector3 touchPoint = touchRay.origin;  
    touchPoint.z = 0;  
    return touchPoint;  
}
```

12. Finally, for this script, we have the `FireBird` function. This function starts by setting our `didFire` flag to `true`. Next, it finds out the direction in which the bird needs to be fired by finding the direction from the pouch's position to `focalPoint`. It also uses the distance between them to determine the power with which the bird needs to be fired, clamping it between our minimum and maximum strengths. Then, it releases the bird by clearing its parent and setting its `isKinematic` flag to `false`, after finding its **Rigidbody** component. To launch it, we use the `AddForce` function and pass it the direction multiplied by the power. `ForceMode2D.Impulse` is also passed to make that the force applied happens once and is immediate. Next, the pouch is positioned at `focalPoint`, as if it were actually under tension. Finally, we call `rigidbodyDamper.ReadyDamp` to start the damping of the **Rigidbody** component's movement:

```
private void FireBird() {  
    didFire = true;  
  
    Vector3 direction = (focalPoint.position - pouch.position).  
normalized;  
    float distance = Vector3.Distance(focalPoint.position, pouch.  
position);  
    float power = distance <= 0 ? 0 : distance / maxRange;  
    power *= maxFireStrength;  
    power = Mathf.Clamp(power, minFireStrength, maxFireStrength);  
}
```

```
    toFireBird.transform.parent = null;
    toFireBird.isKinematic = false;
    toFireBird.AddForce(new Vector2(direction.x, direction.y) *
power, ForceMode2D.Impulse);

    pouch.position = focalPoint.position;

    rigidbodyDamper.ReadyDamp();
}
```

13. Before we can make use of the Slingshot script, we need to create the RigidbodyDamper script.
14. This script starts with the following six variables. The first two define how long you need to wait before damping movement and how much you need to damp it by. The next two track whether damping can be applied and when it will start. The next is a variable that will be filled with a list of all the rigidbodies that are currently in the scene. Finally, it has the allSleeping flag that will be set to true when the movement has stopped:

```
public float dampWaitLength = 10f;
public float dampAmount = 0.9f;
private float dampTime = -1f;
private bool canDamp = false;
private Rigidbody2D[] rigidbodies = new Rigidbody2D[0];

public bool allSleeping = false;
```

15. The ReadyDamp function starts by using FindObjectsOfType to fill the list with all the rigidbodies. The dampTime flag is set when you need to start damping as the sum of the current time and the wait length. It marks that the script can do its damping and resets the allSleeping flag. Finally, it uses StartCoroutine to call the CheckSleepingRigidbodies function. This is a special way of calling functions to make them run in the background without blocking the rest of the game from running:

```
public void ReadyDamp() {
    rigidbodies = FindObjectsOfType(typeof(Rigidbody2D)) as
Rigidbody2D[];
    dampTime = Time.time + dampWaitLength;
    canDamp = true;
    allSleeping = false;

    StartCoroutine(CheckSleepingRigidbodies());
}
```

16. In the `FixedUpdate` function, we first check whether we can damp the movement and whether it is time to do it. If it is, we loop through all the rigidbodies, applying our damp to each one's rotational and linear velocity. Those that are kinematic, controlled by scripts, and already sleeping—meaning that they have stopped moving—are skipped:

```
public void FixedUpdate() {
    if(!canDamp || dampTime > Time.time) return;

    foreach(Rigidbody2D next in rigidbodies) {
        if(next != null && !next.isKinematic && !next.isSleeping()) {
            next.angularVelocity *= dampAmount;
            next.velocity *= dampAmount;
        }
    }
}
```

17. The `CheckSleepingRigidbodies` function is special and will run in the background. This is made possible by the `IEnumerator` flag at the beginning of the function and the `yield return null` line in the middle. Together, these allow the function to pause regularly and keep the rest of the game from freezing while it waits for the function to complete. The function starts by creating a check flag and using it to check whether all the rigidbodies have stopped moving. If one is still found to be moving, the flag is set to `false` and the function pauses until the next frame, when it will try again. When it reaches the end, because all the rigidbodies are sleeping, it sets the `allSleeping` flag to `true` so that the slingshot can be made ready for the next bird. It also stops itself from damping while the player is getting ready to fire the next bird:

```
private IEnumerator CheckSleepingRigidbodies() {
    bool sleepCheck = false;

    while(!sleepCheck) {
        sleepCheck = true;

        foreach(Rigidbody2D next in rigidbodies) {
            if(next != null && !next.isKinematic && !next.IsSleeping())
            {
                sleepCheck = false;
                yield return null;
                break;
            }
        }
    }
}
```

```
    allSleeping = true;
    canDamp = false;
}
```

18. Finally, we have the `AddBodiesToCheck` function. This function will be used by anything that spawns new physics objects after the player has fired the bird. It starts by creating a temporary list and expanding the current one. Next, it adds all the values from the temporary list to the expanded one. Finally, the list of rigidbodies are added after those of the temporary list:

```
public void AddBodiesToCheck(Rigidbody2D[] toAdd) {
    Rigidbody2D[] temp = rigidbodies;
    rigidbodies = new Rigidbody2D[temp.Length + toAdd.Length];

    for(int i=0;i<temp.Length;i++) {
        rigidbodies[i] = temp[i];
    }
    for(int i=0;i<toAdd.Length;i++) {
        rigidbodies[i + temp.Length] = toAdd[i];
    }
}
```

19. Return to Unity and add the two scripts to the `Slingshot` object. In the `Slingshot` script component, connect the references to the `Rigidbody Damper` script component and to each of the points. Also, add as many references to the red bird prefab to the **Level Birds** list as you want for the level.
20. To keep objects from rolling back and through the slingshot, add a **Box Collider 2D** component to `Slingshot` and position it at the stock of the Fork model.
21. To finish off the look of the slingshot, we need to create the elastic bands that tie the pouch to the fork. We will do this by first creating the `SlingshotBand` script.
22. The script starts with two variables, one for the point that the band will end at and one to reference the `LineRenderer` variable that will draw it:

```
public Transform endPoint;
public LineRenderer lineRenderer;
```

23. The `Awake` function ensures that the `lineRenderer` variable has only two points and sets their initial positions:

```
public void Awake() {
    if(lineRenderer == null) return;
    if(endPoint == null) return;
```

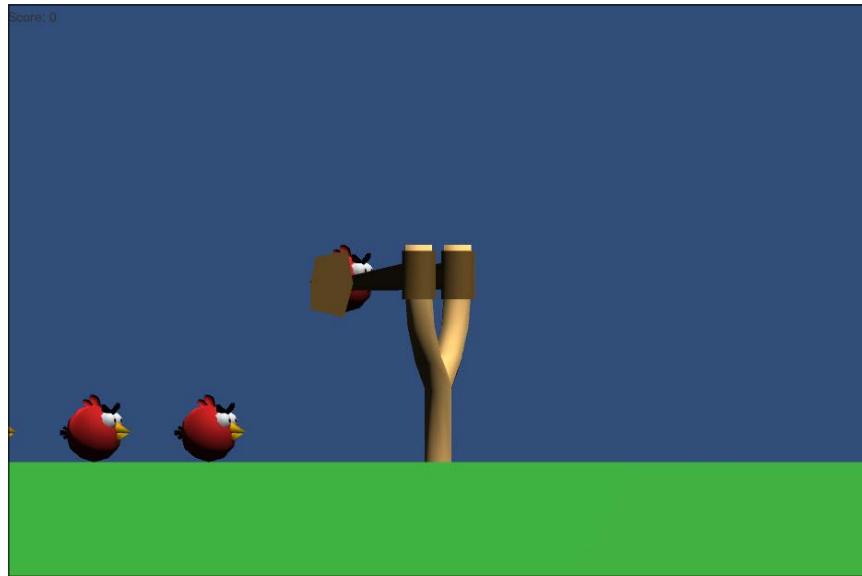
```
    lineRenderer.SetVertexCount(2);  
    lineRenderer.SetPosition(0, transform.position);  
    lineRenderer.SetPosition(1, endPoint.position);  
}
```

24. In the `LateUpdate` function, we set the `lineRenderer` variable's end position to the `endPoint` value. This point will move around with the pouch, so we need to constantly update the renderer:

```
public void LateUpdate() {  
    if(endPoint == null) return;  
    if(lineRenderer == null) return;  
  
    lineRenderer.SetPosition(1, endPoint.position);  
}
```

25. Return to Unity and create an empty **GameObject**. Name it `Band_Near` and make it a child of the `Slingshot` object.
26. As children of this new point, create a cylinder and a second empty **GameObject**, named `Band`.
27. Give the cylinder a brown material and position it around the near prong of the slingshot fork. Be sure to remove the **Capsule Collider** component so that it doesn't get in the way. Also, don't be afraid to scale it in order to make it fit the look of the slingshot better.
28. To the `Band` object, add a **Line Renderer** component found under **Effects** in the **Component** menu. After positioning it in the center of the cylinder, add the `SlingshotBand` script to the object.
29. To the **Line Renderer** component under **Materials**, you can put your brown material in the slot to color the band. Under **Parameters**, set the **Start Width** to `0.5` and the **End Width** to `0.2` in order to set the size of the line.
30. Next, create another empty **GameObject** and name it `BandEnd_Near`. Make it a child of the `Pouch` object and position it inside the pouch.
31. Now, connect the script's references to its line renderer and end point.
32. To make the second band, duplicate the four objects we just created and position them according to the other prong of the fork. The end point for this band can just be moved back along the `z` axis to keep it out of the way of the birds.

33. Finally, turn the whole thing into a prefab so that it can be easily reused in other levels.



We created the slingshot that will be used to fire birds. We used techniques that we learned in the previous chapter to handle touch input and to track the player's finger while they aim and shoot. If you save your scene and position the camera to look at the slingshot, you will notice that it is complete, if not entirely playable. Birds can be fired at the pig fortress, although we can only see the destruction from within Unity's **Scene** view.

Watching with the camera

The game is technically playable at this point, but it is kind of hard to see what is going on. Next, we will create a system to control the camera. The system will allow the player to drag the camera to the left and right, follow the bird when it is launched, and return to the slingshot when everything stops moving. There will also be a set of limits to keep the camera from going too far and viewing things we do not want the player to see, such as beyond the edge of the ground or sky we have created for the level. We will only need one, fairly short, script to control and manage our camera. Let's create it with these steps:

1. To start and to keep everything organized, create a new empty **GameObject** and name it `CameraRig`. Also, to keep it simple, set its position to zero on each axis.

2. Next, create three more empty **GameObjects** and name them **LeftPoint**, **RightPoint**, and **TopPoint**. Set their **Z** axis positions to **-5**. Position the **LeftPoint** object to be in front of the slingshot and **3** on the **Y** axis. The **RightPoint** object needs to be positioned in front of the pig structure you created. The **TopPoint** object can be over the slingshot but needs to be set to **8** on the **Y** axis. These three points will define the limits of where our camera can move when being dragged and following the birds.
3. Make all the three points and the **Main Camera** object children of the **CameraRig** object.
4. Now, we create the **CameraControl** script. This script will control all the movement and interaction with the camera.
5. Our variables for this script start with a reference to the slingshot; we need this so that we can follow the current bird when it is fired. The next are the references to the points we just created. The next group of variables control for how long the camera will sit without input before returning to take a look at the slingshot and how fast it will return. The **dragScale** variable controls how fast the camera actually moves when the player drags their finger across the screen, allowing you to keep the scene moving with the finger. The last group controls whether the camera can follow the current bird and how fast it can do so:

```
public Slingshot slingshot;
public Transform rightPoint;
public Transform leftPoint;
public Transform topPoint;

public float waitTime = 3f;
private float headBackTime = -1f;
private Vector3 waitPosition;
private float headBackDuration = 3f;

public float dragScale = 0.075f;

private bool followBird = false;
private Vector3 followVelocity = Vector3.zero;
public float followSmoothTime = 0.1f;
```

6. In the `Awake` function, we first make certain that the camera is not following a bird and make it wait before heading to take a look at the slingshot. This allows you to initially point the camera to the pig fortress when the level starts and move to the slingshot after giving the player a chance to see what they are up against:

```
public void Awake() {  
    followBird = false;  
    StartWait();  
}
```

7. The `StartWait` function sets the time when it will start to head back to the slingshot and records the position that it is heading back from. This allows you to create a smooth transition:

```
public void StartWait() {  
    headBackTime = Time.time + waitTime;  
    waitPosition = transform.position;  
}
```

8. Next, we have the `Update` function. This function starts by checking whether the slingshot has fired. If it hasn't, it checks whether the player has started aiming, signaling that the bird should be followed and zeroing out the velocity if they have. If they have not started aiming, the `followBird` flag is cleared. Next, the function checks whether it should follow and does so if it should, also calling the `StartWait` function—in case this is the frame in which the bird is destroyed. If it should not follow the bird, it checks for touch input and drags the camera if it finds any. The wait is again started in case the player removes their finger from this frame. Finally, it checks to see whether the slingshot is done firing the current bird and whether it is time to head back. Should both be true, the camera moves back to point at the slingshot:

```
public void Update() {  
    if (!slingshot.didFire) {  
        if (slingshot.isAiming) {  
            followBird = true;  
            followVelocity = Vector3.zero;  
        }  
        else {  
            followBird = false;  
        }  
    }  
  
    if (followBird) {  
        FollowBird();  
    }  
}
```

```
        StartWait();
    }
    else if(Input.touchCount > 0) {
        DragCamera();
        StartWait();
    }

    if(!slingshot.didFire && headBackTime < Time.time) {
        BackToLeft();
    }
}
```

9. The `FollowBird` function starts by making sure that there is a bird to follow, by checking the `toFireBird` variable on the `Slingshot` script, and stops following if a bird is not found. Should there be a bird, the function then determines a new point to move to, which will look directly at the bird. It then uses the `Vector3.SmoothDamp` function to smoothly follow the bird. This function works similar to a spring—the farther away it is from its target position, the faster it moves the object. The `followVelocity` variable is used to keep it moving smoothly. Finally, it calls another function to limit the camera's position within the bounding points we set up earlier:

```
private void FollowBird() {
    if(slingshot.toFireBird == null) {
        followBird = false;
        return;
    }

    Vector3 targetPoint = slingshot.toFireBird.transform.position;
    targetPoint.z = transform.position.z;

    transform.position = Vector3.SmoothDamp(transform.position,
    targetPoint, ref followVelocity, followSmoothTime);
    ClampPosition();
}
```

10. In the `DragCamera` function, we use the `deltaPosition` value of the current touch to determine how far it has moved since the last frame. By scaling this value and subtracting the vector from the camera's position, the function moves the camera as the player drags across the screen. This function also calls upon the `ClampPosition` function to keep the camera's position within the field of play:

```
private void DragCamera() {
    transform.position -= new Vector3(Input.GetTouch(0).
    deltaPosition.x, Input.GetTouch(0).deltaPosition.y, 0) *
```

```
dragScale;  
    ClampPosition();  
}
```

11. The `ClampPosition` function starts by taking the camera's current position. It then clamps the `x` position to be between those of the `leftPoint` and `rightPoint` variables' `x` positions. Next, the `y` position is clamped between the `leftPoint` and `topPoint` variables' `y` positions. Finally, the new position is reapplied to the camera's transform:

```
private void ClampPosition() {  
    Vector3 clamped = transform.position;  
    clamped.x = Mathf.Clamp(clamped.x, leftPoint.position.x,  
    rightPoint.position.x);  
    clamped.y = Mathf.Clamp(clamped.y, leftPoint.position.y,  
    topPoint.position.y);  
    transform.position = clamped;  
}
```

12. Finally, we have the `BackToLeft` function. It starts by using the time and our duration variable to determine how much progress the camera will have made to return to the slingshot. It records the camera's current position and uses `Mathf.SmoothStep` on both the `x` and `y` axes to find a new position that is at an appropriate distance between the `waitPosition` variable and the `leftPoint` variable. Finally, the new position is applied:

```
private void BackToLeft() {  
    float progress = (Time.time - headBackTime) / headBackDuration;  
    Vector3 newPosition = transform.position;  
    newPosition.x = Mathf.SmoothStep(waitPosition.x, leftPoint.  
    position.x, progress);  
    newPosition.y = Mathf.SmoothStep(waitPosition.y, leftPoint.  
    position.y, progress);  
    transform.position = newPosition;  
}
```

13. Next, return to Unity and add the new script to the `Main Camera` object. Connect the references to the slingshot and each of the points to finish it off.
14. Position the camera to point at your pig fortress and turn the whole rig into a prefab.

We created a camera rig that will let the player watch all the action as they play the game. The camera will now follow the birds as they are fired from the slingshot and can now be dragged by the player. By keying off the positions of a few objects, this movement is limited to keep the player from seeing things we don't want them to; if the camera is left idle for long enough, it will also return to look at the slingshot.

Another function of the camera, common to many mobile games, is the pinch-to-zoom gesture. It is such a simple gesture for the user to expect, but it can be complex for us to implement well. Try your hand at implementing it here. You can use `Input.touchCount` to detect whether there are two fingers touching the screen. Then, using the `Vector2.Distance` function, if you have recorded the distance from the last frame, it is possible to determine whether they are moving toward or away from each other. Once you have determined your zoom direction, just change the camera's `orthographicSize` variable to change how much can be seen; be sure to include some limits so that the player can't zoom in or out forever.

Now that we have all the pieces needed to make a complete level, we need some more levels. We need at least two more levels. You can use the blocks and pigs to create any level you might want. It is a good idea to keep structures centered around the same spot as our first level, giving the player an easier time dealing with them. Also, think about the difficulty of the level while making it so that you end up with an easy, medium, and hard level.

Creating the parallax background

A great feature of many 2D games is a parallax scrolling background. This simply means that the background is created in layers that scroll by at different speeds. Think of it as if you are looking out the window of your car. The objects that are far away appear to hardly move, while the ones that are near move by quickly. In a 2D game, it gives the illusion of depth and adds a nice touch to the look of the game. For this background, we will be layering several materials on a single plane. There are several other methods to create this same effect, but ours will make use of a single script that additionally allows you to control the speed of each layer. Let's create it with these steps:

1. We will start this section with the creation of the `ParallaxScroll` script.
2. This script starts with three variables. The first two variables keep track of each material and how fast they should scroll. The third keeps track of the camera's last position, so we can track how far it moves in each frame:

```
public Material[] materials = new Material[0];  
public float[] speeds = new float[0];  
  
private Vector3 lastPosition = Vector3.zero;
```

3. In the `Start` function, we record the camera's beginning position. We use `Start` instead of `Awake` here, in case the camera needs to do any special movement at the beginning of the game:

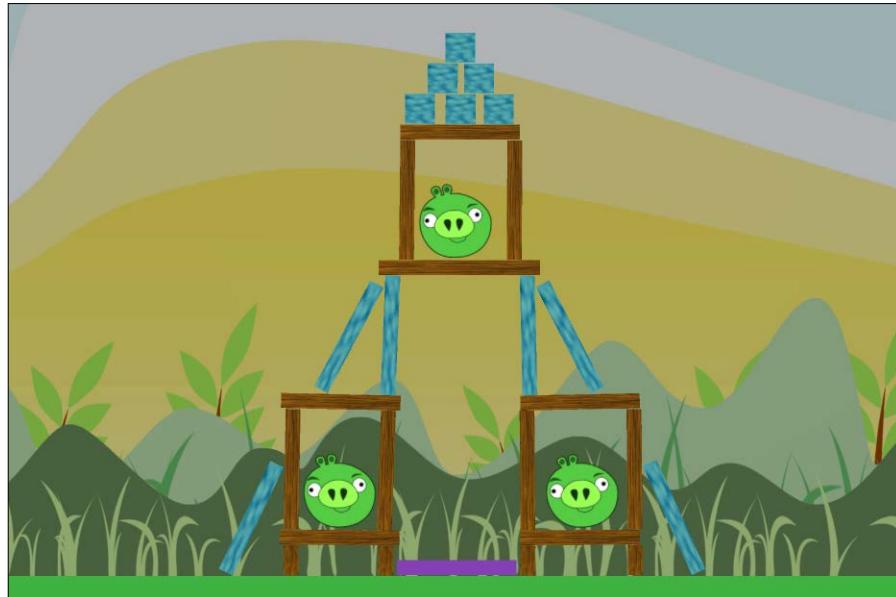
```
public void Start() {  
    lastPosition = Camera.main.transform.position;  
}
```

4. Next, we use the `LateUpdate` function to make changes after the camera has moved about. It starts by finding the camera's new position and comparing the `x` axis values to determine how far it has moved. Next, it loops through the list of materials. The loop first gathers the current offset of its texture using `mainTextureOffset`. Next, the camera's movement multiplied by the material's speed is subtracted from the offset's `x` axis to find a new horizontal position. Then, the new offset is applied to the material. Finally, the function records the camera's last position for the next frame:

```
public void LateUpdate() {  
    Vector3 newPosition = Camera.main.transform.position;  
    float move = newPosition.x - lastPosition.x;  
  
    for(int i=0;i<materials.Length;i++) {  
        Vector2 offset = materials[i].mainTextureOffset;  
        offset.x -= move * speeds[i];  
        materials[i].mainTextureOffset = offset;  
    }  
  
    lastPosition = newPosition;  
}
```

5. Return to Unity and create six new materials. One for each background texture: `sky`, `hills_tall`, `hills_short`, `grass_light`, `grass_dark`, and `fronds`. All the materials, except for `sky`, need to use **Transparent Render Mode**. If they do not, we will not be able to see all the textures when they are layered.
6. Before we can tile the images in the background, we need to adjust their **Import Settings**. Select each in turn and take a look at the **Inspector** window. Because we chose to make a 2D game, Unity imports all the images as sprites by default, which clamps the edges of our images and keeps them from repeating. For all our background images, change the **Texture Type** option to **Texture** and the **Wrap Mode** option to **Repeat**. This will let us use them in a way that makes it look like an infinite scrolling background.

7. We also need to adjust the **Tiling** option for each of these new materials. For all of them, leave the **Y** axis as 1. For the **X** axis, set 5 for the **sky**, 6 for **hills_tall**, 7 for **hills_short**, 8 for **grass_dark**, 9 for **fronds**, and 10 for **grass_light**. This will offset all the features of the textures, so a long pan does not see features regularly lining up.
8. Next, create a new plane. Name it **Background** and remove its **Mesh Collider** component. Also, attach our **ParallaxScroll** script.
9. Position it at 30 on the **X** axis, 7 on the **Y** axis, and 10 on the **Z** axis. Set its rotation to 90 for the **X** axis, 180 for the **Y** axis, and 0 for **Z**. Also, set the scale to 10 for the **X** axis, 1 for the **Y** axis, and 1.5 for the **Z** axis. Altogether, these position the plane to face the camera and fill the background.
10. In the plane's **Mesh Renderer** component, expand the **Materials** list and set the value of **Size** to 6. Add each of our new materials to the list slots in the order of **sky**, **hills_tall**, **hills_short**, **grass_dark**, **fronds**, and **grass_light**. Do the same for the **Materials** list in the **Parallax Scroll** script component.
11. Finally, in the **Parallax Scroll** script component, set the value of **Size** of the **Speeds** list to 6 and input the following values in the order of 0.03, 0.024, 0.018, 0.012, 0.006, and 0. These values will move the materials gently and evenly.
12. At this point, turning the background into a prefab will make it easy to reuse later.



We created a parallax scroll effect. This effect will pan a series of background textures, giving the illusion of depth in our 2D game. To easily see it in action, press the play button and grab the camera in the **Scene** view, moving it from side to side in order to see the background change.

We have two other levels to add backgrounds to. Your challenge here is to create your own background. Use the techniques you learned in this section to create a night-style background. It can include a stationary moon, while everything else scrolls in the shot. For an added trick, create a cloud layer that slowly pans across the screen as well as with the camera and the rest of the background.

Adding more birds

There is one last set of assets that we need to create for our levels: the other birds. We will create three more birds that each have a unique special ability: a yellow bird that accelerates, a blue bird that splits into multiple birds, and a black bird that explodes. With these, our flock will be complete.

To make the creation of these birds easier, we will be making use of a concept called **inheritance**. Inheritance allows a script to expand upon the functions it is inheriting without the need to rewrite them. If used correctly, this can be very powerful and, in our case, will aid in the quick creation of multiple characters that are largely similar.

The yellow bird

First, we will create the yellow bird. Largely, this bird functions exactly as the red bird. However, when the player touches the screen a second time, the bird's special ability is activated and its speed increases. By extending the `Bird` script that we created earlier, this bird's creation becomes quite simple. Because of the power of inheritance, the script we are creating here consists of only a handful of lines of code. Let's create it with these steps:

1. Start by creating the yellow bird in the same way as the red bird, using the `YellowBird` model instead.
2. Instead of using the `Bird` script, we will create the `YellowBird` script.
3. This script needs to extend the `Bird` script, so replace `MonoBehaviour` with `Bird` on line four of our new script. It should look similar to the following code snippet:

```
public class YellowBird : Bird {
```

4. This script adds a single variable that will be used to multiply the bird's current velocity:

```
public float multiplier = 2f;
```

5. Next, we override the `DoSpecial` function and multiply the bird's `body.velocity` variable when it is called:

```
protected override void DoSpecial() {  
    didSpecial = true;  
    body.velocity *= multiplier;  
}
```

6. Return to Unity, add the script to your new bird, connect the **Rigidbody** component reference, and turn it into a prefab. Add some to the list on your slingshot in order to use the bird in your level.

We created the yellow bird. This bird is simple. It directly modifies its velocity to suddenly gain a boost of speed when the player touches the screen. As you will soon see, we use this same style of script to create all our birds.

The blue bird

Next, we will create the blue bird. This bird splits into three birds when the player touches the screen. It will also extend the `Bird` script by using inheritance, reducing the amount of code that needs to be written to create the bird. Let's do it with these steps:

1. Again, start building your blue bird the same way as the previous two birds were built, substituting the appropriate model. You should also adjust the value of **Radius** of the **Circle Collider 2D** component to align appropriately with the small size of this bird.

2. Next, we create the `BlueBird` script.
3. Again, adjust line four so that the script extends `Bird` instead of `MonoBehaviour`:

```
public class BlueBird : Bird {
```

4. This script has three variables. The first variable is a list of prefabs to spawn when the bird splits. The next is the angle difference between each new bird that will be launched. The final variable is a value to spawn the birds a little ahead of their current position in order to keep them from getting stuck inside each other:

```
public GameObject[] splitBirds = new GameObject[0];  
public float launchAngle = 15f;  
public float spawnLead = 0.5f;
```

5. Next, we override the `DoSpecial` function and start, as with the others, by marking that we made our special move. Next, it calculates half of the number of birds to spawn and creates an empty list to store the rigidbodies of the newly spawned birds:

```
protected override void DoSpecial() {  
    didSpecial = true;  
  
    int halfLength = splitBirds.Length / 2;  
    Rigidbody2D[] newBodies = new Rigidbody2D[splitBirds.Length];
```

6. The function continues by looping through the list of birds, skipping the slots that are empty. It spawns the new birds at their position; after trying to store the object's **Rigidbody**, it goes on to the next one if it is missing. The new **Rigidbody** component is then stored in the list:

```
for(int i=0;i<splitBirds.Length;i++) {  
    if(splitBirds[i] == null) continue;  
  
    GameObject next = Instantiate(splitBirds[i], transform.position,  
    transform.rotation) as GameObject;  
  
    Rigidbody2D nextBody = next.GetComponent<Rigidbody2D>();  
    if(nextBody == null) continue;  
  
    newBodies[i] = nextBody;
```

7. Using `Quaternion.Euler`, a new rotation is created that will angle the new bird along a path that is split off from the main path. The new bird's velocity is set to the rotated velocity of the current bird. An offset is calculated and it is then moved forward along its new path, so as to get out of the way of the other birds being spawned:

```
Quaternion rotate = Quaternion.Euler(0, 0, launchAngle * (i -  
halfLength));  
nextBody.velocity = rotate * nextBody.velocity;  
Vector2 offset = nextBody.velocity.normalized * spawnLead;  
next.transform.position += new Vector3(offset.x, offset.y, 0);  
}
```

8. After the loop, the function uses `FindObjectOfType` to find the slingshot that is currently in the scene. If it is found, it is changed to track the first new bird spawned as the one that was fired. The new list of rigidbodies is also set to the `rigidbodyDamper` variable, in order to be added to its list of rigidbodies. Finally, the script destroys the bird it is attached to, completing the illusion that the bird has been split apart:

```
Slingshot slingshot = FindObjectOfType(typeof(Slingshot)) as  
Slingshot;  
if(slingshot != null) {
```

```
    slingshot.toFireBird = newBodies[0];
    slingshot.rigidbodyDamper.AddBodiesToCheck(newBodies);
}

Destroy(gameObject);
}
```

9. Before you add the script to your new bird, we actually need two blue birds: one that splits and one that does not. Duplicate your bird and name one `Bird_Blue_Split` and the other `Bird_Blue_Normal`. To the split bird, add the new script and to the normal bird, add the `Bird` script.
10. Turn both the birds into prefabs and add the normal bird to the other's list of birds to be split into.

We created the blue bird. This bird splits into multiple birds when the user taps the screen. The effect actually requires two birds that look identical, one that does the splitting and another that is split in two but does nothing special.

It is actually possible to add anything that we want to spawn to the blue bird's list of things to split into. Your challenge here is to create a rainbow bird. This bird can split into different types of birds, not just blue ones. Or, perhaps it is a stone bird that splits into stone blocks. For an extended challenge, create a mystery bird that randomly picks a bird from its list when it splits.

The black bird

Finally, we have the black bird. This bird explodes when the player touches the screen. As with all the birds discussed previously, it will extend the `Bird` script; inheriting from the red bird makes the black bird's creation much easier. Let's use these steps to do it:

1. As with the others, this bird is initially created in the same way as the red bird, readjusting the value of `Radius` on your **Circle Collider 2D** component for its increased size.
2. Again, we create a new script to extend the `Bird` script. This time, it is called `BlackBird`.
3. Do not forget to adjust line four to extend the `Bird` script and not `MonoBehaviour`:

```
public class BlackBird : Bird {
```

4. This script has two variables. The first variable is the size of the explosion and the second is its strength:

```
public float radius = 2.5f;
public float power = 25f;
```

5. Once more, we override the `DoSpecial` function, first marking that we did so. Next, we use `Physics2D.OverlapCircleAll` to acquire a list of all the objects that are within the range of the bird's explosion, the 3D version of which is `Physics.OverlapSphere`. Next, we calculate where the explosion is coming from, which is just our bird's position moved down three units. We move it down because explosions that throw debris up are more exciting than the ones that push debris out. The function then loops through the list, skipping any empty slots and those without rigidbodies:

```
protected override void DoSpecial() {
    didSpecial = true;

    Collider2D[] colliders = Physics2D.OverlapCircleAll(transform.
position, radius);

    Vector2 explosionPos = new Vector2(transform.position.x,
transform.position.y) - (Vector2.up * 3);

    foreach(Collider2D hit in colliders) {
        if(hit == null) continue;
        if(hit.attachedRigidbody != null) {
```

6. If the object exists and has a **Rigidbody** component attached, we need to calculate how the explosion is going to affect this object, simulating the way an explosion's strength is reduced the further away you are from it. First, we save ourselves some typing by grabbing the other object's position. Next, we calculate where it is, relative to the position of the explosion. By dividing the magnitude or the length of the relative position by our `radius` variable, we can figure out how much force to apply to the object that was hit. Finally, we use `AddForceAtPosition` to give the object a kick as if the explosion was in a specific spot. The `ForceMode2D.Impulse` variable is used to apply the force immediately:

```
Vector3 hitPos = hit.attachedRigidbody.transform.position;
Vector2 dir = new Vector2(hitPos.x, hitPos.y) - explosionPos;
float wearoff = 1 - (dir.magnitude / radius);
Vector2 force = dir.normalized * power * wearoff;
hit.attachedRigidbody.AddForceAtPosition(force, explosionPos,
ForceMode2D.Impulse);
}
```

7. Finally, the function destroys the exploded bird:

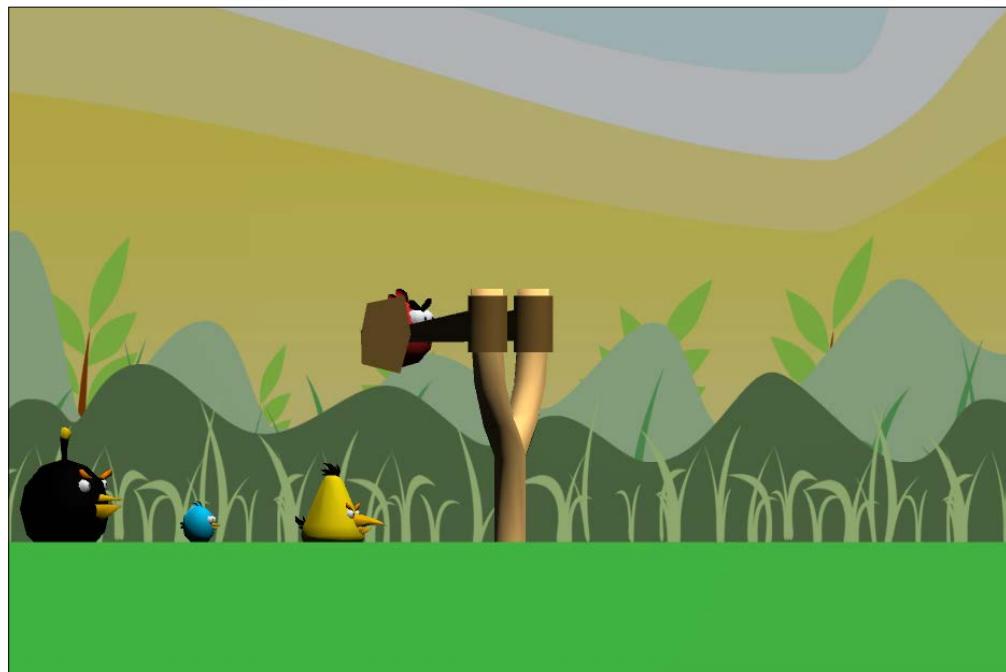
```
Destroy(gameObject);  
}
```

8. As with the last two, apply your new script to your new bird and turn it into a prefab. You now have four birds to choose from when selecting the slingshot arsenal for each level.

We created our fourth and last bird: the black bird. This bird explodes when the user touches the screen, throwing anything that might be near it into the sky. This can be a fun bird to play around with and is effective for destroying your pig forts.

The black bird from the game we are imitating has the additional ability of a timed explosion after it has hit something. Try creating a timer for our black bird to recreate this effect. You will have to override the `OnCollisionEnter` function to start your timer and use `LateUpdate` to count down. Once your timer runs out, you can just use our `DoSpecial` function to actually cause the explosion.

Now that you know how to cause explosions, we have another challenge: create an explosive crate. You need to extend the `Plank` script to make it, and when enough damage is done to the crate, trigger the explosion. For an additional challenge, instead of making the crate explode, configure it to throw out a few bombs that explode when they hit something.



Level selection

Finally, we need to create our level selection screen. From this scene, we will be able to access and start playing all the levels we created earlier. We will also display the current high scores for each level. A new scene and a single script will serve us well in managing our level selection. Let's use these steps to do it:

1. This last section begins by saving our current scene and pressing *Ctrl + N* to create a new one; we will name it `LevelSelect`.
2. For this scene, we need to create a single, short script also named `LevelSelect`.
3. This script is going to work with the buttons in the GUI to tell players about the high scores and load levels. However, before we can do this, we need to add a line at the very beginning of the script, along with the other using lines—just like the other scripts we have created that need to update the GUI:
`using UnityEngine.UI;`
4. The first and only variable is a list of all the button text that we want to update, with the high scores for the levels they are associated with:

```
public Text[] buttonText = new Text[0];
```

5. The first function is the `Awake` function. Here, it loops through all the buttons, finds the high score for it, and updates the text to display it. `PlayerPrefs.GetInt` is the opposite of the `SetInt` function we used earlier to save the high score:

```
public void Awake() {
    for(int i=0;i<buttonText.Length;i++) {
        int levelScore = PlayerPrefs.GetInt("LevelScore" + (i + 1),
0);
        buttonText[i].text = "Level " + (i + 1) + "\nScore: " +
levelScore;
    }
}
```

6. The second and last function for this script is `LoadLevel`. It will receive a number from the GUI button and use it to load the level that the players want to play:

```
public void LoadLevel(int lvl) {
    Application.LoadLevel(lvl);
}
```

7. Return to Unity and add the script to the `Main Camera` object.
8. Next, we need to create three buttons. Without these, our player will not be able to select a level to play. Make each of them of 200 units square and position them in a row in the center of the screen. Also, increase the value of **Font Size** to 25, so that the text is easy to read.
9. Drag each of the buttons' `Text` children to the **Button Texts** list on the `Main Camera` component's **Level Select** script component. The way they are ordered in this list is the order in which they will get their text and high score information changed.
10. Also, each button needs a new **On Click** event. Select `Main Camera` for the object and then navigate to **LevelSelect | LoadLevel (int)** for the function. Then, each button needs a number. The button that has its `Text` child in the **Button Texts** list should have the number 1 since it will display the information for level one. The second has 2, the third has 3, and so on. Each button must have the same number as the order in the list, or they will cause a different level to load than what the player is expecting.
11. Finally, open **Build Settings** and add your scenes to the **Scenes in Build** list. Clicking and dragging on the scenes in the list will let you reorder them. Make sure that your **LevelSelect** scene is first and has an index of zero at the right-hand side. The rest of your scenes can appear in whatever order you desire. However, beware as they will be associated with the buttons in the same order.



We have created a level selection screen. It has a list of buttons associated with the levels in our game. When a button is pressed, `Application.LoadLevel` starts that level. We also made use of `PlayerPrefs.GetInt` to retrieve the high scores for each of the levels.

Here, the challenge is to style the GUI to make the screen look great. A logo and a background will help a lot. Additionally, take a look at the **Scrollbar** GUI object if you have more than three levels. This object will let you create a function that offsets the level buttons when the user scrolls through a list of levels that are far greater in size than can be easily seen on the screen.

Summary

In this chapter, we learned about physics in Unity and recreated the incredibly popular mobile game, *Angry Birds*. Using Unity's physics system, we are able to make all the levels that we will ever want to play. With this game, we also explored Unity's 2D pipeline for creating great 2D games. Our birds and slingshot are 3D assets, giving us the ability to light and shade them. The pigs and background, however, are 2D images, reducing our lighting options but allowing greater detail in the assets. The 2D images were also crucial in the creation of the parallax scrolling effect of the background. Finally, the blocks that make up the levels appear to be 2D but are actually 3D blocks. We also created a level-selection screen. From here, the player can see their high scores and pick any of the levels that we created.

In the next chapter, we return to the Monkey Ball game we started in the previous chapter. We are going to create and add all of the special effects that finish off a game. We will add the bouncing and popping sound effects that every Monkey Ball game needs. We will also add various particle effects. When the bananas are collected, they will create a small explosion, rather than just disappearing.

8

Special Effects – Sound and Particles

In the previous chapter, we took a short break from our Monkey Ball game to learn about physics and 2D games in Unity. We created a clone of *Angry Birds*. The birds utilized physics to fly through the air and destroy the pigs and their structures. We utilized parallax scrolling to make a pleasing background effect. We also created a level selection screen, from which you can load the game's various scenes.

In this chapter, we return to the Monkey Ball game. We are going to add many special effects that will round out the game experience. We will start by learning about the controls that Unity provides when working with audio. We will then move on to add some background music to the game and movement sounds for our monkey. Next, we will learn about particle systems, creating a dust trail for the monkey. Finally, we combine the effects explained in the chapter to create explosions for when the user collects bananas.

In this chapter, we will cover the following important topics:

- Importing audio clips
- Playing SFX
- Understanding 2D and 3D SFX
- Creating particle systems

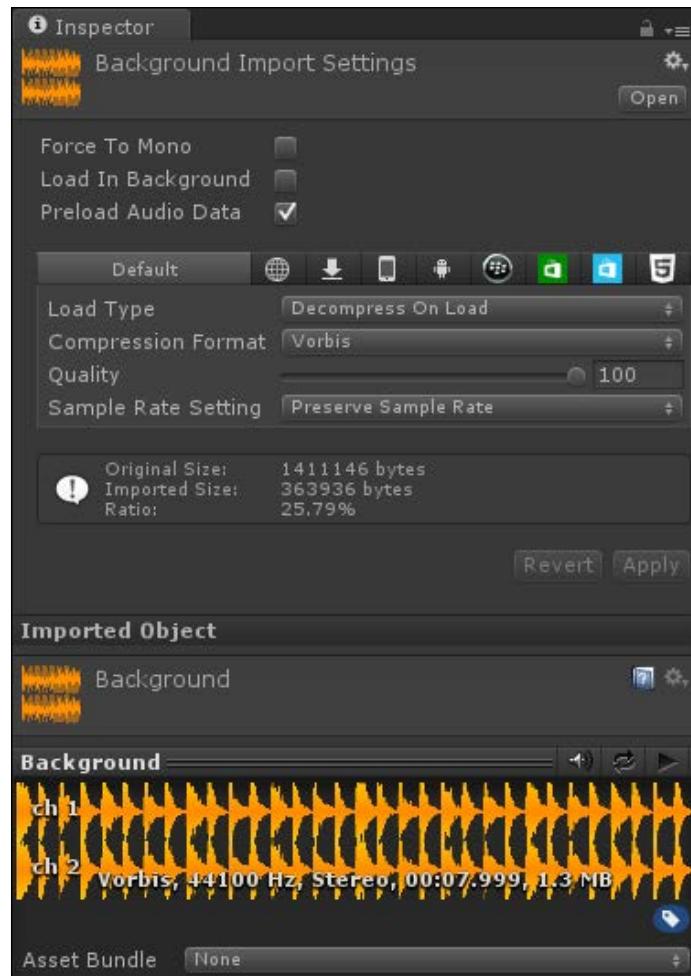
Open up your Monkey Ball project and let's get started.

Understanding audio

As with other assets, the Unity team has worked hard to make working with audio easy and hassle-free. Unity is capable of importing and utilizing a broad range of audio formats, allowing you to keep your files in a format that you can edit in another program.

Import settings

Audio clips have a small assortment of important settings. They let you easily control the type and compression of files. The following screenshot shows the settings that we have to work with while importing audio clips:



The options in the preceding screenshot are as follows:

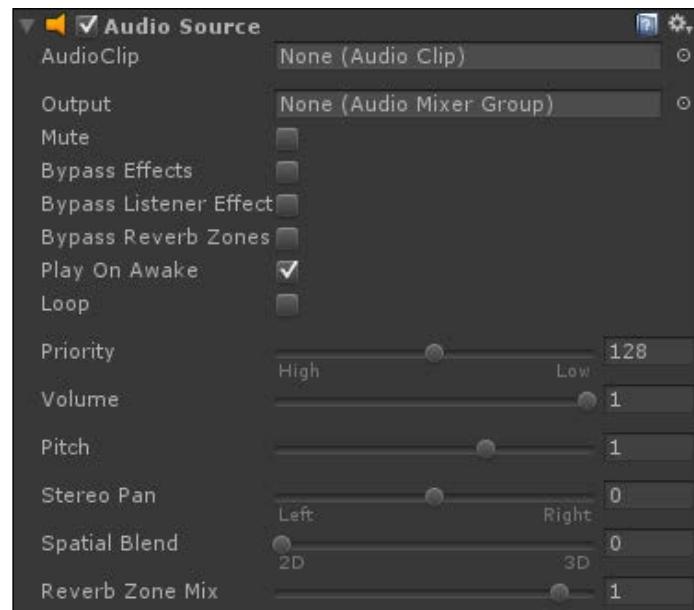
- **Force To Mono:** This checkbox will cause Unity to change a multichannel file to have a single channel of audio data.
- **Load In Background:** This will cause the initial loading of an audio file to not pause the whole game while it is loading the game into memory. It is best to use this for large files that do not need to be used right away.
- **Preload Audio Data:** This will cause the audio information to be loaded as soon as possible. This is best for small files that need to be used almost immediately.
- **Load Type:** This controls how the file is loaded when the game is being played; you can choose from the following three available options:
 - **Decompress On Load:** This removes compression from the file when it is first needed. The overhead for this option makes it a very poor choice for large files. This option is best for short sounds that you often hear, such as gunfire in a shooting game.
 - **Compressed In Memory:** This only decompresses the file as it is being played. When it is just being held in memory, the file remains compressed. This is a good option for short- and medium-length sounds that are not heard often.
 - **Streaming:** This loads in the audio as it is playing, such as streaming music or video from the Web. This option is best for things such as background music.
- **Compression Format:** This allows you to select the kind of compression format to be used for reducing the size of your audio files. The **PCM** format will give you the largest file size and the best audio quality. The **Vorbis** format can give you the smallest file size, but the quality is reduced the smaller you go. The **ADPCM** format adapts to how the audio file is laid out, in order to give you a file size somewhere in the middle.
- **Quality:** This is only used when **Vorbis** is selected as the compression format. A lower value will reduce the final size of the file in your project, but it will also introduce increasing amounts of artifacts to your audio.
- **Sample Rate Setting:** This lets you determine how much detail from your audio files is maintained in Unity. The **Preserve Sample Rate** option will maintain the setting that was used in your original file. The **Optimize Sample Rate** option will allow Unity to choose a setting that works well for your file. The **Override Sample Rate** option will let you access the value of **Sample Rate** and select a specific setting for your audio. A smaller value will reduce the overall file size, at the cost of lowering the quality.

Audio Listener

In order to actually hear anything in the game, every scene needs an **Audio Listener** component in it. By default, the **Main Camera** object (included first in any new scene) and any new camera you might create has an **Audio Listener** component attached. There can only be one **Audio Listener** component in your scene at a time. If there is more than one component or you try to play a sound when there isn't one, Unity will fill your console log with complaints and warnings. The **Audio Listener** component also gives the precise position for any 3D sound effects to key off.

Audio Source

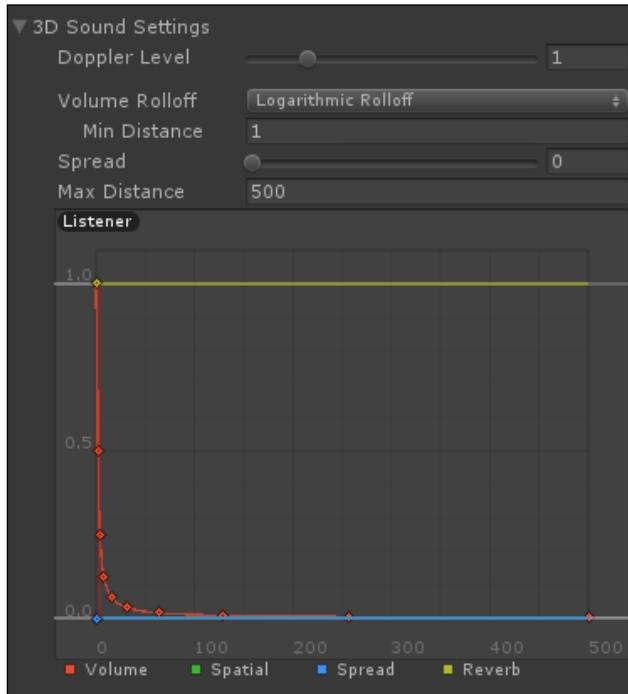
The **Audio Source** component is like a speaker, and it controls the settings used to play any sound effect. If the clip is 3D, the position of this object to the **Audio Listener** component and the mode chosen determine the volume of the clip. The following screenshot shows the various settings for an **Audio Source** component, followed by their explanation:



- **AudioClip:** This is the sound file that this **Audio Source** component will play, by default.

- **Output:** For complex audio effects, one of Unity's new **Audio Mixer** objects can be put here. These allow you to take specific control over the audio and over any effects or blending that might be applied to it, before it is finally played.
- **Mute:** This is a quick way to toggle the volume of the sound that is playing on and off.
- **Bypass Effects:** This allows you to toggle any special filters applied to the **Audio Source** component.
- **Bypass Listener Effect:** This allows the audio to ignore any special effects that might be applied to **Audio Listener**. This is a good setting for background music that should not be warped by the world.
- **Bypass Reverb Zones:** This allows you to control whether **Reverb Zones**, which control the transition areas in ambient audio, affect the sound.
- **Play On Awake:** This will cause the audio clip to immediately start playing when the scene loads or the object is spawned.
- **Loop:** This will cause the playing clip to repeat as it is played.
- **Priority:** This dictates the relative importance of the files being played. The value 0 denotes the most important and best for music, while 256 denotes the least important file. Depending on the system, only so many sounds can be played at once. The list of files to be played starts with the most important and ends when this limit is reached, excluding those with the lowest values if there are more sounds than the limit will allow.
- **Volume:** This decides how loud the clip will be played.
- **Pitch:** This scales the playback speed of the clip.
- **Stereo Pan:** This adjusts how evenly the sound comes out of each speaker, weighting it towards the left or right speaker.
- **Spatial Blend:** This is the percentage of the 3D effects to be applied to the Audio Source component. This affects things such as the falloff and Doppler effects.

- **Reverb Zone Mix:** (Reverb zones are used to create transitions in ambient audio effects.) This setting lets you adjust how much effect these zones will have on the audio from this audio source.



The settings in the preceding screenshot are as follows:

- **3D Sound Settings:** This contains the group of settings that are specific to the playing of 3D audio clips. The **Volume**, **Spatial**, **Spread**, and **Reverb** options can be adjusted by using the graph at the end of the group. This allows you to create more dynamic transitions as the player approaches an **Audio Source** component:
 - **Doppler Level:** This decides how much Doppler effect needs to be applied to moving sounds. Doppler effect is the change in pitch experienced as a source of sound moves closer or further away from you. A classic example is a car blaring its horn as it rushes by.
 - **Volume Rolloff:** This controls how the volume fades with distance. There are three types of rolloffs:
 - **Logarithmic Rolloff:** This is a sudden and rapid falloff of the sound at a short distance from the source's center.

- **Linear Rolloff:** This is an even falloff with distance, the loudest being at the **Min Distance** value and the quietest at the **Max Distance** value.
- **Custom Rolloff:** This allows you to create a custom falloff by adjusting the graph at the end of the group. It is also automatically chosen when the graph is altered.
- If the **Audio Listener** component is closer than the **Min Distance** value, the audio will be played at the current volume level. Outside this distance, the sound will fall off, according to the **Rolloff mode**.
- **Spread:** This adjusts the amount of area in speaker space that the sound covers. It becomes more important when working with more than one or two speakers.
- Beyond the **Max Distance** value, the sound will stop transitioning, based on the graph at the bottom of the group.

Adding background music

Now that we know about the available audio settings, it is time to put that knowledge into action. We will start by adding some background music. This will have to be a 2D sound effect so that we can hear it comfortably, no matter where the **Audio Source** component is. We will also create a short script to fade-in the music, in order to reduce the suddenness with which sound effects bombard the player. We will use the following steps to do this:

1. We will start by creating a new script and name it `FadeIn`.
2. This script begins with four variables. The first variable is the goal volume that the script has to reach. The second is the number of seconds that the transition will take. The third variable is the time when the transition began. The last variable keeps track of the **Audio Source** component attached to the same object as our script, allowing us to update it regularly, as follows:

```
public float maxVolume = 1f;
public float fadeLength = 1f;
private float fadeStartTime = -1f;
private AudioSource source;
```

3. Next, we make use of the `Awake` function. It begins by checking for an attached **Audio Source** component and filling our `source` variable with it. If one cannot be found, the **GameObject** is destroyed and the function is exited:

```
public void Awake() {
    source = gameObject.GetComponent<

```

```
if(source == null) {  
    Destroy(gameObject);  
    return;  
}
```

4. The Awake function ends by setting the audio's volume to 0 and starts playing it if it isn't already playing:

```
source.volume = 0;  
  
if(!source.isPlaying)  
    source.Play();  
}
```

5. To cause the transition over time, we use the Update function. It will check whether the value of the fadeStartTime variable is below zero and set it to the current time if it is. This allows you to avoid the hiccup that can be caused by the initialization of a scene starting:

```
public void Update() {  
    if(fadeStartTime < 0)  
        fadeStartTime = Time.time;
```

6. Next, the function checks whether the transition time has ended. If it has, the **Audio Source** component's volume is set to maxVolume and the script is destroyed in order to free resources:

```
if(fadeStartTime + fadeLength < Time.time) {  
    source.volume = maxVolume;  
    Destroy(this);  
    return;  
}
```

7. Finally, the current progress is calculated by finding the amount of time that has passed since the fade started and dividing it by the length of the transition. The resulting percentage of progress is multiplied by the value of maxVolume and applied to the **Audio Source** component's volume:

```
float progress = (Time.time - fadeStartTime) / fadeLength;  
source.volume = maxVolume * progress;  
}
```

8. Back in Unity, we need to create a new empty **GameObject** and name it **Background**.
9. Add our FadeIn script and an **Audio Source** component to our object; these can be found by navigating to **Component | Audio | Audio Source**.

10. If you have not done so already, create an **Audio** folder in your **Project** panel and import the sound files included in the **Starting Assets** folder for the chapter. Because of the small size of these files and our current game, the default import settings for them will work just fine.
11. Select your **Background** object in the **Hierarchy** window and drag the **Background** sound to the **AudioClip** slot.
12. Make sure that the **Play On Awake** and **Loop** checkboxes are checked in the **Audio Source** component. Both the **Volume** and **Spatial Blend** options also need to be set to 0 to make the file play throughout the game, but make no noise when starting.

We added background music to our game. For the sound to be constant and not directional, we utilized the music as a 2D sound. We also created a script to fade in the music when the game starts. This eases the transition into the game for the player, preventing a sudden onslaught of sound. If your background music ends up being too loud to hear anything else in the game, reduce the **Max Volume** value in the **Inspector** panel of your **Background** object something more pleasing.

Background music adds a lot to a game's experience. A horror scene is not nearly as scary without some scary music. Bosses are much less intimidating without their daunting music. Look for some good background music for your other games. Something light and cheery will work well for *Angry Birds*, while a piece that is more industrial and fast-paced will keep hearts racing through the Tank Battle game.

Poking bananas

To understand 3D audio effects, we are going to add a sound to the bananas, which will be triggered every time the player touches them. This will give the players extra feedback when they have successfully touched one of the bananas, while also giving some indication of the distance and direction of the banana that was touched. Let's use these steps to create this effect:

1. First, we need a new script named `BananaPoke`.
 2. This script has one variable, `source`, to keep track of the **Audio Source** component attached to the object:
- ```
private AudioSource source;
```
3. Just like in our previous script, we use the `Awake` function to find a reference to our **Audio Source** component, saving us a little bit of work in the editor:

```
public void Awake() {
 source = gameObject.GetComponent<();
}
```

4. When the player touches a banana on the screen, a message is sent to the banana that calls the Touched function. We used this function in our BananaBounce script to adjust its health, which we created in *Chapter 6, Specialities of the Mobile Device*. We can again use it here to play our sound effect, if we have an **Audio Source** component. The PlayOneShot function uses an **Audio Source** component's position and settings to play a quick sound effect. Without this, we will be unable to play many sound effects from the same **Audio Source** component in rapid succession. All we need to pass is for the audio clip to be played. In this case, the audio clip is attached to the **Audio Source** component itself:

```
public void Touched() {
 if(source != null)
 source.PlayOneShot(source.clip);
}
```

5. Next, we need to add our new script and an **Audio Source** component to the Banana prefab in our **Project** panel.
6. The BananaPoke sound file needs to be dragged from the Audio folder to the new **Audio Source** component's **AudioClip** slot.
7. So that you don't hear an annoying pop at the very beginning of the game, uncheck the **Play On Awake** box.
8. Next, we want to hear the difference in the distance of the bananas we touch. Set the **Spatial Blend** setting to 1, in order to turn it from a 2D sound effect to a 3D sound effect.
9. Finally, we need to change the value of **Volume Rolloff** to **Linear Rolloff** and set **Max Distance** to 50. This gives us a comfortable and easily heard change in the volume of our sound effect based on distance.

Living in a 3D world, we expect most sounds to come from a specific direction and to fall off with distance. By creating a similar effect in our 3D games, the player is able to easily judge where things are in the game world and how far away they might be. This is especially important for games where the player needs to be able to hear potential enemies, obstacles, or rewards so that they will be able to find or avoid them.

Our Tank Battle game has many enemies that can easily sneak up on us, because they make no noise as they approach. Tanks are not generally recognized as quiet machines. Go find an engine's rumble sound or make it and add it to the enemy tanks. This will give the player some indication of where the enemies might be and how far away they are. Also, different types of tanks have different types of engines. Every engine sounds a little bit different. So, while you're at it, find different engine noises for each type of tank you have, giving the player even more indication of what dangers may lie just around the corner.

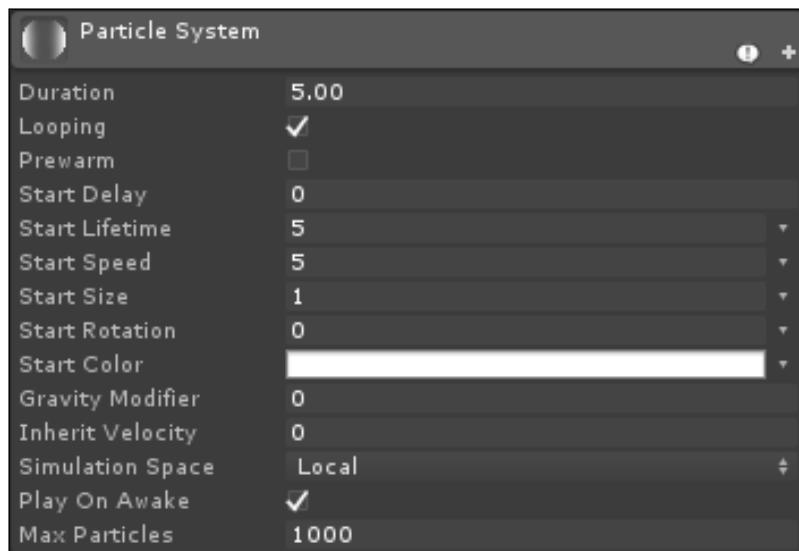
## Understanding particle systems

Particle systems add a lot to the final look of a game. They can take the form of fire, magic waves, rain, or a great many other effects that you can dream up. They are often hard to create well, but are well worth the effort if you do. Keep in mind, especially when working with the mobile platform, that less is more. Larger particles are more effective than a great amount of particles. If your particle system ever contains thousands of particles in a small space or is duplicated on itself to increase the effect, you need to rethink the design and find a more efficient solution.

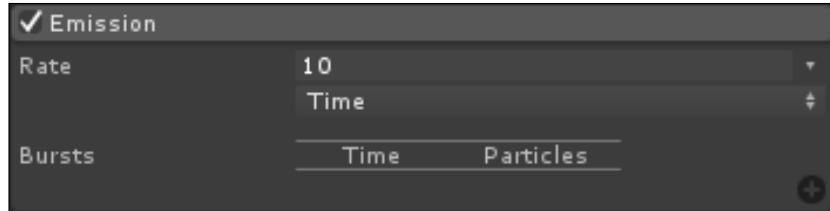
## Particle system settings

Every particle system contains a large variety of components, each with its own settings. Most of the available settings have the option to be **Constant**, **Curve**, **Random Between Two Constants**, and **Random Between Two Curves**. The **Constant** option will be a specific value. The **Curve** option will be a set value that changes along the curve over time. The two random settings select a random value between the respective value types. This may seem confusing at first, but as you work through them, they will become more understandable.

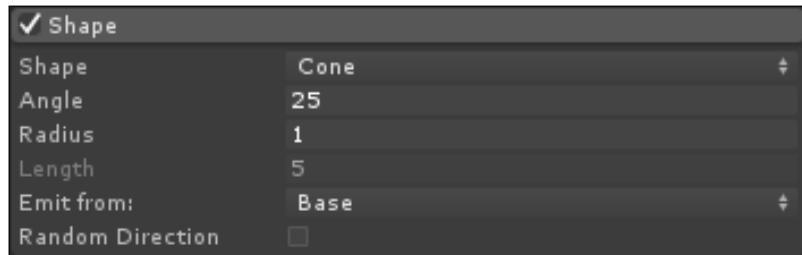
As you will see in the following screenshots and descriptions that follow, we will work through and gain an understanding of each piece of a particle system:



- The first portion, the **Initial** module, of the particle system holds all the settings used by every emitter in Unity:
  - **Duration:** This denotes the time for which an emitter lasts. A looping system will repeat itself after this amount of time. A nonlooping system will stop emitting new particles after this length of time.
  - **Looping:** This checkbox dictates whether or not the system loops.
  - **Prewarm:** This checkbox, if checked, will start a looping system if it has already had a chance to loop for a while. This is useful in the case of torches that should already be lit, not start when the player enters the room.
  - **Start Delay:** This will stop the particle system from emitting for the given number of seconds, when it is initially triggered.
  - **Start Lifetime:** This is the number of seconds for which an individual particle will last.
  - **Start Speed:** This is how fast a particle will initially move when spawned.
  - **Start Size:** This dictates how large a particle is when it is spawned. It is always better to use large particles rather than small and, hence, a greater number of particles.
  - **Start Rotation:** This will rotate the emitted particles.
  - **Start Color:** This is the color tint of the particles when they are spawned.
  - **Gravity Modifier:** This gives the particles a greater or lesser amount of gravity effect.
  - **Inherit Velocity:** This will cause particles to gain a portion of their transform's momentum if it is moving.
  - **Simulation Space:** This determines whether the particles will stay with the game object as it is moved (that is, local) or remain where they are in the world.
  - **Play On Awake:** This checkbox, if checked, will cause the emitter to start emitting as soon as it is spawned or the scene starts.
  - **Max Particles:** This limits the total number of particles that this system supports at a single time. This value only comes into play if the rate at which particles are emitted (or their lifespan) is great enough to overbalance their rate of destruction.

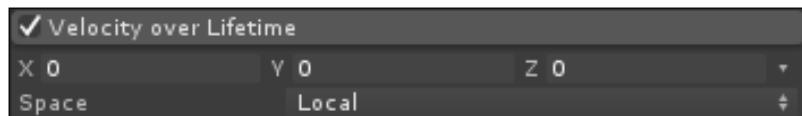


- The **Emission** module controls how fast the particles are emitted:
  - **Rate:** If this is set to **Time**, it denotes the number of particles that are created per second. If this is set to **Distance**, it denotes the number of particles per unit of the distance that the system travels as it moves.
  - **Bursts:** This is only used when the **Rate** option is set to **Time**. It allows you to set points in the system's timeline when a specific number of particles will be emitted.

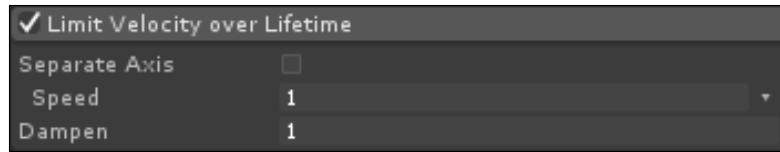


- The **Shape** module, as shown in the preceding screenshot, controls how the system emits particles. It has the following options:
  - **Shape:** This dictates what form the emission point will take. Each option comes with a few more value fields that determine its size.
  - **Sphere:** This is the point from which particles are emitted in all the directions. The **Radius** parameter determines the size of the sphere. The **Emit from Shell** option dictates whether the particles are emitted from the surface of the sphere or from within the volume.
  - **HemiSphere:** This is, as the name suggests, the half of a sphere. The **Radius** parameter and the **Emit from Shell** option work the same here as they do for **Sphere**.

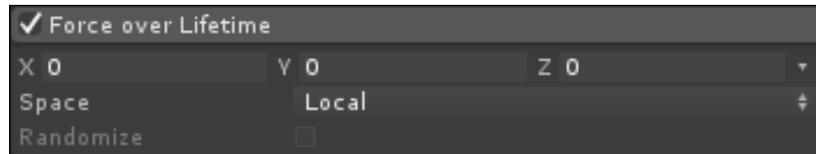
- **Cone:** This emits particles in one direction. The **Angle** parameter determines whether the shape is closer to a cone or cylinder. The **Radius** parameter dictates the size of the emission point of the shape. The **Length** parameter is used when the **Emit from** option is set to **Volume** or **Volume Shell**, to dictate how much space is available for spawning particles. The **Emit from** option will determine where the particles are emitted from. **Base** emits from the base disc of the shape. The **Base Shell** option emits from the base of the cone but around the surface of the shape. **Volume** will emit from anywhere inside the shape, and **Volume Shell** emits from the surface of the shape.
- **Box:** This emits particles from a cube-type shape. The **Box X**, **Box Y**, and **Box Z** options determine the size of the box.
- **Mesh:** This allows you to select a model to use as an emission point. You then have the option of emitting particles from each **Vertex**, **Edge**, or **Triangle** that makes up the **Mesh**.
- **Circle:** This emits particles from a single point along a 2D plane. **Radius** determines the size of the emission, and **Arc** dictates how much of the circle is used. **Emit from Edge** decides whether the particles are emitted from the inner or outer edge of the circle.
- **Edge:** This emits particles in a single direction, out from a line. The **Radius** parameter determines how long the emission area is.
- **Random Direction:** This determines whether a particle's direction is determined by the surface normal of the shape chosen or whether it is chosen at random.



- The **Velocity over Lifetime** module allows you to control the momentum of the particles after they have been spawned:
  - **X, Y, and Z:** These define the number of units per second along each axis of the particle's momentum.
  - **Space:** This dictates whether the velocity is applied locally to the system's transformation or relative to the world.



- The **Limit Velocity over Lifetime** module dampens a particle's movement if it exceeds the specified value:
  - Separate Axis:** This allows you to define a value unique to each axis and whether that value is local or relative to the world.
  - Speed:** This is how fast the particle has to be moving before the damp is applied.
  - Dampen:** This is a percentage of the speed by which the particle is cut. It can be any value between zero and one.



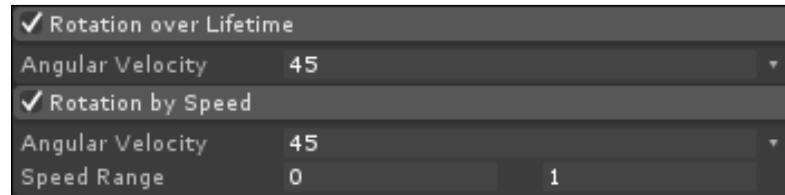
- The **Force over Lifetime** module adds a constant amount of movement to each particle over the course of its life:
  - X, Y, and Z:** These define how much force needs to be applied along each axis.
  - Space:** This dictates whether the force is applied local to the system's transformation or in the world space.
  - Randomize:** If X, Y, and Z are random values, this will cause the amount of force to apply to be randomly picked in each frame, resulting in a statistical averaging of the random values.



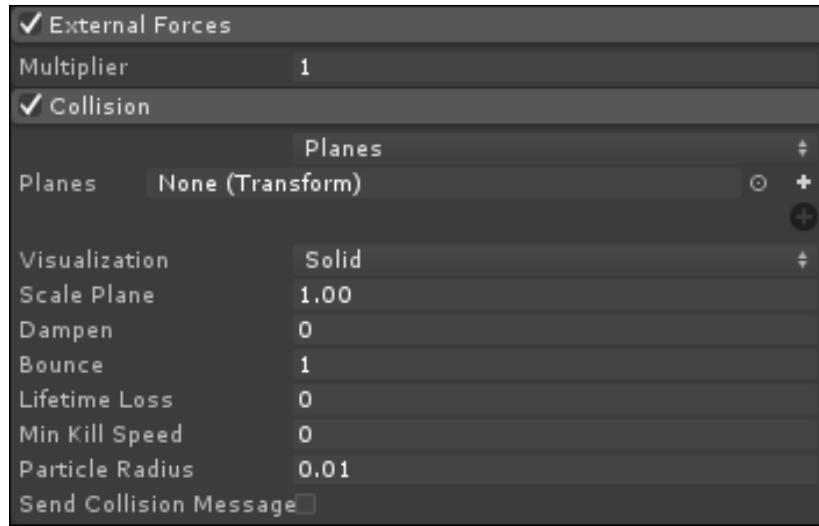
- The **Color over Lifetime** module allows you to define a series of colors for the particle to transition through after it has been spawned.
- The **Color by Speed** module causes the particle to transition through the defined range of colors as its speed changes:
  - **Color:** This is the set of colors to transition through.
  - **Speed Range:** This defines how fast the particle must go, in order to be at the minimum and maximum ends of the **Color** range.



- The **Size over Lifetime** module changes the size of the particle over the course of its life.
- The **Size by Speed** module adjusts the size of each particle, based on how fast it is going, as follows:
  - **Size:** This is the adjustment that the particles transition through.
  - **Speed Range:** This defines the minimum and maximum values for each of the **Size** values.

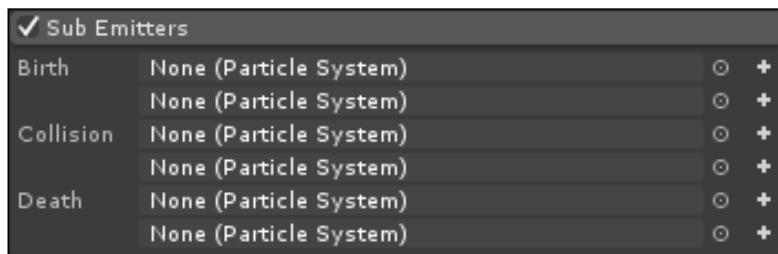


- The **Rotation over Lifetime** module rotates particles over time after they have been spawned.
- The **Rotation by Speed** module rotates particles more as they go faster:
  - **Angular Velocity:** This is the degrees per second speed of the particle's rotation.
  - **Speed Range:** This is the minimum and maximum range for the **Angular Velocity** value if it is not set to **Constant**.



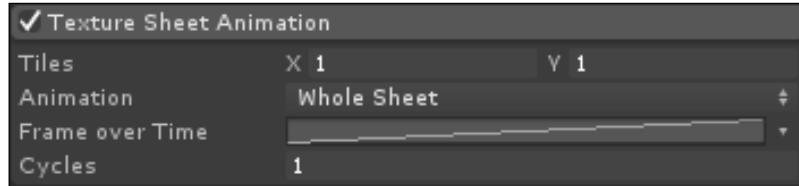
- The **External Forces** module multiplies the effect of wind zone objects. Wind zones simulate the effects of wind on particle systems and on Unity's trees.
- The **Collision** module allows particles to collide and interact with the physical game world:
  - If this is set to **Planes**, you can define a number of flat surfaces for the particles to collide with. This is faster to process than **World** collisions:
    - **Planes:** This is a list of transformations that define the surfaces to collide with. Particles will only collide with the local, positive y side of the transform. Any particles on the other side of the point will be destroyed.
    - **Visualization:** This gives you the option to view the planes as a **Solid** surface or as a **Grid** surface.
    - **Scale Plane:** This adjusts the size of the **Visualization** option. It does not affect the actual size of the surface to collide with.
    - **Particle Radius:** This is used to define the size of the sphere that is used to calculate the particle's collision with the planes.
  - If set to **World**, the particles will collide with every collider in your scene. This can be a lot for the processor to handle.
    - **Collides With:** This defines a list of layers that the particles can collide with. Only the colliders on the layers that are checked in this list will be used for the collision calculation.

- **Collision Quality:** This defines how precise the collision calculations are for this particle system. The **High** option will calculate precisely for every single particle. The **Medium** option will use an approximation and a limited number of new calculations in each frame. The **Low** option just calculates less often than **Medium** does. If **Collision Quality** is set to **Medium** or **Low**, the **Voxel Size** parameter dictates how precisely the system estimates the points of collision.
- **Dampen:** This removes the defined fraction amount of speed from the particle when it collides with a surface.
- **Bounce:** This allows the particle to maintain the defined fraction of its speed, specifically along the normal of the surface that was hit.
- **Lifetime Loss:** This is the percentage of life. When the particle collides, this percentage of life is removed from the particle. When the particle's life drops to zero over time, or through collision, it is removed.
- **Min Kill Speed:** If, after collision, the particle's speed is below this value, the particle is destroyed.
- **Send Collision Messages:** If this checkbox is checked, scripts attached to the particle system and the object that was collided with will be alerted every frame that the collision took place. Only one message is sent per frame, not per particle.

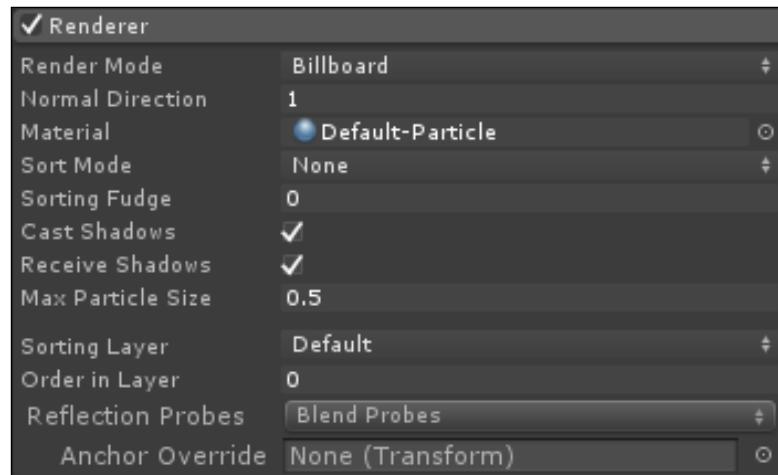


- The **Sub Emitters** module allows additional particle systems to be spawned at points in the life of each particle of this system:
  - Any particle systems in the **Birth** list will be spawned and will follow the particle when it is first created. This can be used to create a fireball or smoke trail.
  - The **Collision** list spawns particle systems when the particle hits something. This can be used for rain drop splashes.

- The **Death** list spawns particles when the particles are destroyed. It can be used to spawn a firework explosion.



- The **Texture Sheet Animation** module causes the particle to flip through a number of particles over the course of its life. The texture used is defined in the **Renderer** module:
  - **Tiles:** This defines the number of rows and columns in the sheet. This will determine the total number of frames available.
  - **Animation:** This gives you the options of **Whole Sheet** and **Single Row**. If this option is set to **Single Row**, the row used can either be chosen at random or specified by using the **Random Row** checkbox and the value of **Row**.
  - **Frame over Time:** This defines how the particle transitions between frames. If set to **Constant**, the system will only use a single frame.
  - **Cycles:** This is the number of times the particle will loop through the animation over the course of its life.



- The **Renderer** module dictates how each particle is drawn on the screen, as follows:
  - **Render Mode:** This defines which method a particle should use in order to orient itself in the game world:
    - **Billboard:** This will always face the camera directly.
    - **Stretched Billboard:** This will face particles at the camera, but it will stretch them based on the speed of the camera, the particle's speed, or by a specific value.
    - **Horizontal Billboard:** This is flat on the XZ plane of the game world.
    - **Vertical Billboard:** This will always face the player but will always stay straight along the Y axis.
    - If set to **Mesh**, you can define a model to be used as a particle, rather than a flat plane.
  - **Normal Direction:** This is used for the lighting and shading of the particles by adjusting the normal of each plane. A value of **1** aims the normals directly at the camera, while a value of **0** aims them toward the center of the screen.
  - **Material:** This defines the material that was used to render the particles.
  - **Sort Mode:** This dictates the order in which the particles should be drawn, by distance or age.
  - **Sorting Fudge:** This causes particle systems to be drawn earlier than normal. The higher the value, the earlier it will be drawn on the screen. This affects whether the system appears in front of or behind other particle systems or partially transparent objects.
  - **Cast Shadows:** This determines whether or not the particles will block light.
  - **Receive Shadows:** This determines whether or not the particles are affected by the shadows cast by other objects.
  - **Max Particle Size:** This is the total amount of screen space that a single particle is allowed to fill. No matter what the real size of the particle is, it will never fill more than this space of the screen.
  - **Sorting Layer and Order in Layer:** These are used when working with a 2D game. These dictate what level it is on and where in that level it should be drawn, respectively.

- **Reflection Probes:** These can also be used to reflect the world rather than just a particle. When the world is reflecting rather than a particle, **Anchor Override** can be used to define a custom position to sample reflections from.

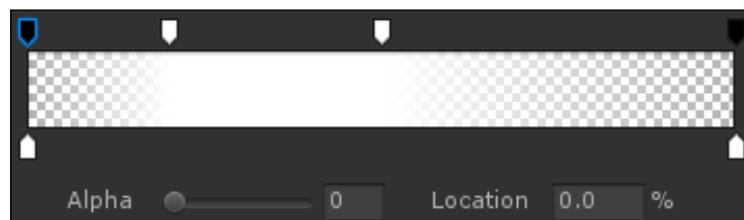
That was a whole lot of information. You will use the **Initial**, **Emission**, and **Shape** modules most often. They control the main features of any particle system. To a slightly lesser degree, you will use the **Renderer** module to change the texture used for the particle system and the **Color over Lifetime** module to adjust the fade. All of these pieces, when used together effectively, will give you some really great effects that round out the look of any game. The absolute best way to learn what all they can do is to just play around with the settings and see what happens. Experimentation and a few tutorials, such as the next few sections, are the best ways to become an expert particle system creator.

## Creating dust trails

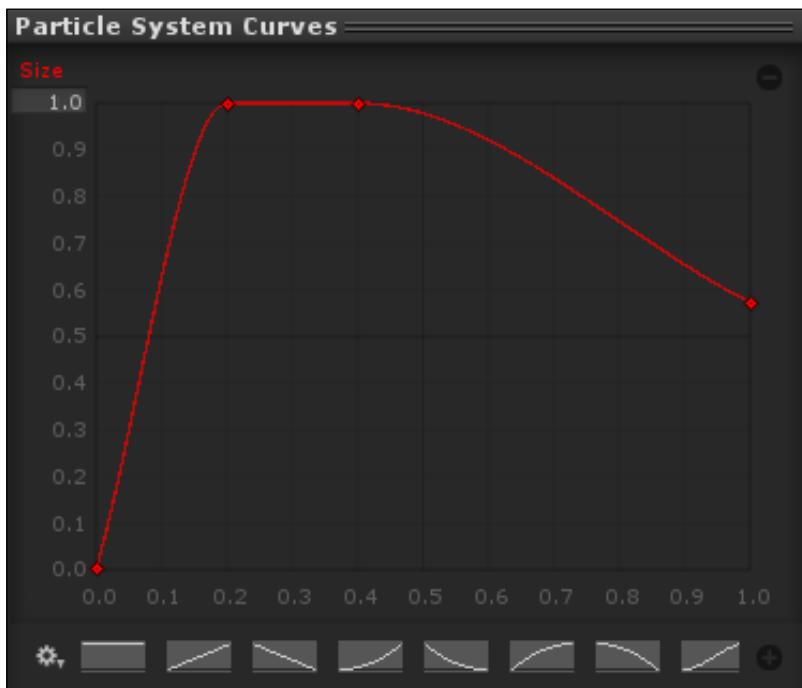
To give our players a better sense that characters are actually in the world and touching it, they are often given the ability to kick up little dust clouds as they move around the environment. It is a small effect but adds a good bit of polish to any game. We are going to give our monkey ball the ability to kick up little dust clouds. Let's use these steps to do this:

1. First, we need to create a new particle system, by navigating to **GameObject | Particle System**. Name it **DustTrail**.
2. By default, particle systems shoot out little white balls in a cone shape. For the dust, we need something a little more interesting. Import the textures from the **Starting Assets** folder for the chapter to a **Particles** folder in your project. These are particle textures, provided by Unity, which were in the older versions of the engine.
3. Next, we need to create a new material in our **Particles** folder. Name it **DustPoof**.
4. Change the new material's **Shader** property by going to **Particles | Alpha Blended** and put the **DustPoof** texture into the **Particle Texture** image slot. This changes the material to be partially transparent and to blend well with both the world and the other particles that are being emitted.
5. To change the look of our **DustPoof** particle system, put the material in the **Material** slot of the **Renderer** module.
6. The particles in the system last too long and go too far, so set **Start Lifetime** to **0.5** and **Start Speed** to **0.2**. This will make the particles just rise up a little from the ground before disappearing.

7. We also need to make the particles more appropriate for the size of our monkey. Set **Start Size** to **0.3** in order to make them appropriately small.
8. It is a little weird to see all the particles in the exact same orientation. To make the orientations different, change **Start Rotation** to be **Random Between Two Constants** by clicking on the small down arrow to the right-hand side of the input field. Then, set the two new input fields to **-180** and **180**, so that all the particles have a random rotation.
9. The brownish color of the particle is alright, but it doesn't always make sense with the color and nature of what our level terrain is made of. Click on the color field next to **Start Color** and use the **Color Picker** window that pops up to pick a new color based on the environment. This will allow the particles to make more sense when being kicked up from the surface of our game field.
10. Lastly, for the **Initial** module, we need to set **Simulation Space** to **World** so that the particles are left behind as our monkey moves, rather than following him.
11. In **Emission**, we need to make sure that there are enough particles to give us a good amount of dust being kicked up. Set **Rate** to **20** for a light dusting.
12. Next, we are going to adjust the **Shape** module so that the particles are emitted under the whole area of the ball. Ensure that the **Shape** is set to **Cone**, the **Angle** to **25**, and the **Radius** to **0.5**.
13. With the **Color over Lifetime** module, we can ease the sudden appearance and disappearance of the particles. Hit the checkbox at the left-hand side of the module's name to activate it. Click on the white bar at the right-hand side of **Color** to open the **Gradient Editor** window. In **Gradient Editor**, clicking just above the colored bar will add a new flag that will control the transparency of the particles over their lifetime. The left-hand side of this bar corresponds to the very beginning of a particle's life, and the right-hand side corresponds to the end of its life. We need a total of four flags. One at the very beginning, with the value of **Alpha** set to **0**, a second flag with a **Location** value of **20** and **Alpha** value of **255**, the third flag at a **Location** of **50** and **Alpha** of **255**, and the last flag at the very end with an **Alpha** value of **0**. This will cause the dust particles to fade in quickly at the beginning and fade out slowly after that, easing their transition into and out of existence.



14. We can further ease the transition by using the **Size over Lifetime** module to make the particles grow and shrink as they come into and out of existence. Be sure to activate it with the checkbox by its name. By clicking on the curve bar at the right-hand side of **Size**, the **Particle System Curves** editor opens in the preview area at the bottom of the **Inspector** panel. Here, we can adjust any of the little diamond-shaped keys to control the size of the particles over the course of its life. Just as in the case of **Gradient Editor**, the left-hand side is the beginning of the particle's life and the right is the end. By right-clicking on it, we can add new keys to control the curve. To create a pop-in effect, put the first key at the bottom in the far left side. The second key should go at the top and correspond with the  $0.2$  value at the bottom. The third will work well at the top and  $0.4$  with the bottom values. The fourth should be at the far right and set at about  $0.6$  with the numbers on the left, which indicate the percentage of its **Start Size** that we set in the **Initial** module, as shown in the following screenshot:



15. Finally, to complete the look of our particle system, we are going to use the **Rotation over Lifetime** module to add a little bit of spin to the particles. Change the value to **Random Between Two Constants** and set the two value fields to  $-45$  and  $45$  to make the particles spin a little over the course of their lives.

16. So that our monkey can use the particle system, make it a child of the `MonkeyPivot` object, and set its position to 0 for `X`, -0.5 for `Y`, and 0 for `Z`. Also, make sure that the rotation is set to 270 for `X`, 0 for `Y`, and 0 for `Z`. This will keep it at the base of our monkey ball and throw particles into the air. Because it is a child of `MonkeyPivot`, it will not spin around with the ball, because we already made the object compensate for the spinning of the ball.
17. Try it out. As our monkey moves around, he leaves a nice little trail of dust in his wake. This effect can be a great bit of polish, especially if we tailor it to the material that the level is made out of, whether it be grass, sand, wood, metal, or anything else.
18. You might notice that the effect continues to play, even as our monkey flies off the edge of our map. We are going to create a new script to toggle the particles based on whether or not our monkey ball is actually touching the ground. Create a new script named `DustTrail` now.
19. The first variable for this script will hold a reference to the particle system we are trying to control. The second will be a flag that indicates whether or not the ball is actually touching the ground:

```
public ParticleSystem dust;
private bool isTouching = false;
```
20. We use the `OnCollisionStay` function to determine whether the ball is touching anything. This function is similar to the `OnCollisionEnter` function we used in the last chapter. While that function was called by Unity the moment one of our birds hit something, this one is called every frame our ball continues to touch another collider. When it is called, we just set our flag to mark that we are touching something:

```
public void OnCollisionStay() {
 isTouching = true;
}
```

21. Because the physics system only changes during the `FixedUpdate` loop, this is the function that we use to update our particle system. Here, we first check whether we are touching something and the particle system is not currently emitting anything, as indicated by its `isPlaying` variable. If the conditions are met, we use the `Play` function to turn the particle system on. However, if the ball is not touching anything and the particle system is currently playing, we use the `Stop` function to turn it off:

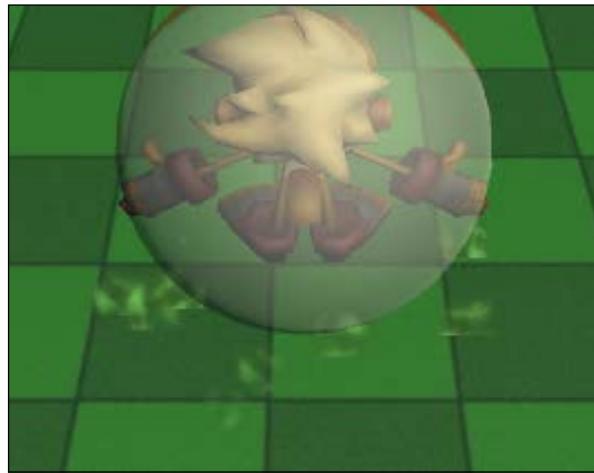
```
public void FixedUpdate() {
 if(isTouching && !dust.isPlaying) {
 dust.Play();
 }
}
```

```
else if(!isTouching && dust.isPlaying) {
 dust.Stop();
}
```

22. At the very end of the `FixedUpdate` function, we set our flag to `false` so that it can be re-updated in the next frame as to whether or not we need to turn the particle system on or off:

```
isTouching = false;
}
```

23. Next, add the new script to the `MonkeyBall` object. As we learned in the previous chapter, if we don't attach it to the same object as the ball's **Rigidbody** component, we will not receive the collision messages we need to make the script work.
24. Finally, drop your `DustTrail` particle system into the **Dust** slot so that your script can actually take control over it.
25. Try it again. Now our monkey can easily move around and create a little dust trail until it falls off the edge of the level, goes off a jump, or otherwise ends up in the air.



We gave our monkey ball the ability to kick up dust. We also made the dust turn on and off, based on whether the ball is actually touching the ground or not. This little effect makes the character feel nice and grounded in the world. It can also give you a sense of the speed of the character, based on the time for which the trail is behind it.

Another good effect for grounding characters that we have previously discussed is shadows. If you haven't done so already, be sure to give your environment some shadow detail. You might notice, though, that due to the partially transparent nature of the ball, real-time shadows do not work on it. That's where the blob shadow we used on the tank will come in.

Our effect also runs constantly, even if the ball is not moving. Try to adjust whether or not the particle system plays based on the velocity of its **Rigidbody** component. We messed around with the velocity of **Rigidbody** components a little bit in the last chapter, if you need a refresher. For an added challenge, take a look at the particle system's `emissionRate` variable. Try to make the effect have more particles as the ball starts going faster.

## Putting it together

So far, we learned about audio effects and particle systems on their own. They each can add a lot to the scene, setting the mood and giving that touch of polish that sets a game apart. However, there are many effects that cannot stand on their own. Explosions, for example, are simply not that impressive, unless you have both the visual and auditory effects.

## Exploding bananas

It is so much more satisfying to destroy things when they explode. It takes both a particle effect and a sound effect to make a proper explosion. We will start by creating an explosion prefab. Then, we will update the bananas to spawn the explosion once they are destroyed. Let's use these steps to create the banana explosions:

1. First, we need to create a new particle system and name it `Explosion`.
2. We want our explosion to actually look something like an explosion. This is where our second particle texture comes in. Create a new material for it, named `Smoke`.
3. This time, set the **Shader** property by going to **Particles | Additive**. This will use an additive-blending method that makes the overall particle look brighter, while still blending the alpha of the particle with the things behind.

4. Be sure to set the new material's **Particle Texture** property to **Smoke**.
5. Also, drop your **Smoke** material into the **Material** slot in the particle system's **Renderer** module.
6. We do not want this explosion to last too long. So, in the **Initial** module, set **Duration** to **0.5** and **Start Lifetime** to **1**, making it all much shorter than what it was.

 When working with things such as explosions that occur in short bursts, it can become hard to see how our changes are affecting the look of the particle system. When we are done with this particle system, we will have to uncheck the **Looping** checkbox, but leaving it on for now makes it much easier to view and work with.

7. Next, so that the particles do not fly excessively far, set **Start Speed** to **0.5**, making the explosion contained and centralized.
8. In order to have enough particles for a proper explosion, set **Rate** to **120** in the **Emission** module.
9. To actually make the explosion seem legitimate, change **Shape** to **Sphere** in the **Shape** module. Also, set **Radius** to **0.5**. If you are interested in changing the size of the explosion, adjust **Radius** and the **Emission Rate**. Increasing both will give you a larger explosion, while decreasing both will give you a smaller one.

 This basic explosion effect is just a visual explosion, as most are. Making an explosion that changes the environment or alters its appearance based on the environment will require extra scripting and model consideration that is beyond the scope of this book.

10. The explosion in our game still isn't colored like an explosion and all of the particles pop out of the existence around the edges. That's where the **Color over Lifetime** module comes in. First, we need to get rid of the particle pop by adding some new flags for the alpha channel. Add two new flags at about 20 percent of the way in from the edges and adjust all the four flags, so that the particles fade in at the beginning and out at the end.

11. The flags along the bottom of the gradient bar of **Gradient Editor** control the colors that the particle will transition through over the course of its life. For a decent explosion, we are going to need two more flags, one that is placed one-third of the way in and one flag at two-thirds, spacing all four of them evenly. Explosions tend to start with a moderately bright color, followed by a bright color at the peak of the explosion's energy, then another medium bright color as the energy starts to dissipate, and finally black when all of the energy is gone. Each color you pick will affect the color of the explosion. For a normal explosion, select yellows and oranges. For a sci-fi space explosion, you can select blues or greens. Or, maybe it is an alien spore cloud with the use of purples. Use your imagination and pick something appropriate for what you want to explode.



12. Now that we have all our settings in place, ensure that **Play On Awake** is checked, so the explosion will start the moment it is created, and uncheck **Looping**, so that it does not play forever. If you want to test your particle system at this point, take a look at the **Stop**, **Simulate**, and **Pause** buttons that appear in the bottom-right of your **Scene** window when any particle system is selected. These buttons work just like the buttons of your music player, controlling the playback of your particle system.
13. If we were to start creating explosions now, they will just sit there in the scene after spawning their initial group of particles, though the player will never see them. That's why we need a new script to get rid of them once they are done. Create a new script and name it **Explosion**.
14. This script has a single variable, that is, to keep track of the particle system that indicates its existence:

```
public ParticleSystem particles;
```

15. It also has a single function. The `Update` function checks every frame to see whether the particle system even exists or if it has stopped playing. In either case, the overall object is destroyed so that we can save resources:

```
public void Update() {
 if(particles == null || !particles.isPlaying)
 Destroy(gameObject);
}
```

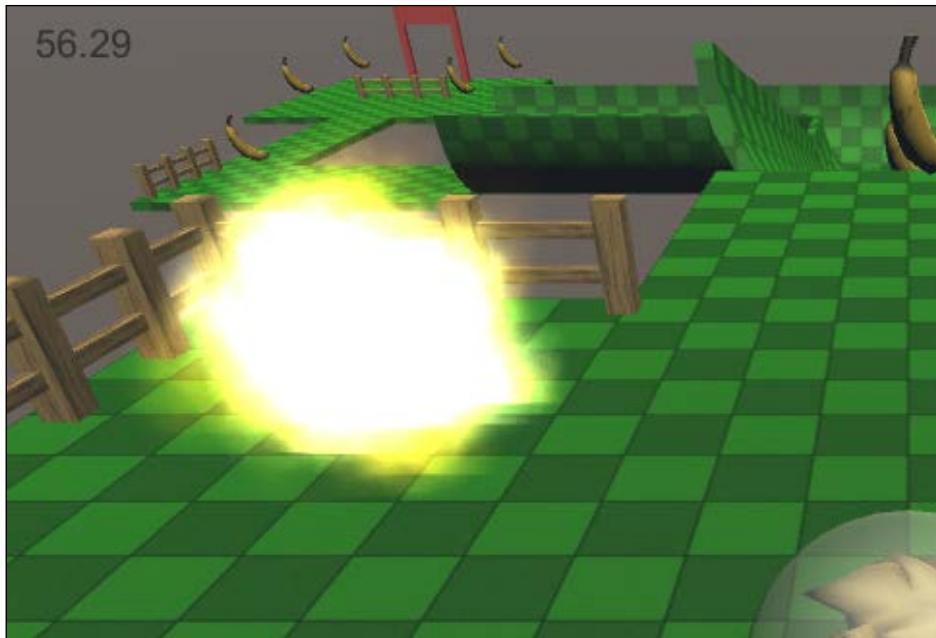
16. Next, we need to add our new script to the `Explosion` object. Also, drag the **Particle System** component to the **Particles** slot in the **Script** component.
17. To make the explosion heard, we need to add an **Audio Source** component to the `Explosion` object as well.
18. Ensure that its **Play On Awake** box is checked. So that the sound makes sense in 3D space, set the **Spatial Blend** property to 1. Also, set it for **Linear Rolloff** and 50 for the **Max Distance**, so that we can hear it.
19. It doesn't make much sense for our bananas to have the same explosion sound that a car has. Instead, we have a nice little popping sound that will differentiate the final touch from those that just reduce the health of the banana. To that end, set the `BananaPop` audio file in the **AudioClip** slot on the **Audio Source** component.
20. With all of our explosive settings in place, create a new prefab out of the `Explosion` object and delete it from the scene.
21. Next, we need to update the `BananaBounce` script to actually spawn the explosion when it has run out of health. Open it up now.
22. First, we add a new variable at the beginning of the script. This will simply keep track of the prefab that we want to spawn after the banana runs out of health:

```
public GameObject explosion;
```

23. Next, we need to add a line to the `Touched` function right after we use the `Destroy` function. This line just creates a new instance of the explosion at the position of the banana:

```
Instantiate(explosion, transform.position, transform.rotation);
```

24. Finally, find your Banana prefab in the **Project** panel and drop the Explosion prefab into the new **Explosion** slot. If you don't, the explosion will never be created and Unity will give you an error every time a banana runs out of health.



As you can see in the preceding screenshot, we have created an explosion. With the help of a few textures from Unity's old particle systems, we made it actually look like an explosion, rather than just the puff of colored balls that it would otherwise be. We also gave the explosion a sound effect. Combining both particle systems and audio sources, we can create many effects, such as our explosion, which would be weak if you just use one or the other. We also updated our bananas so that they spawn the explosions when they are destroyed by the player. Try playing around with the balance of the banana's audio, the volume differences between each touch on the banana, and the explosion itself. The more information we can give the player visually with particle systems and in an auditory manner with audio sources, the better will be the effect.

Bananas aren't the only thing in this world that can explode. In our second game, we were destroying tanks that just disappear. Try adding some new explosions to the Tank Battle game. Every time a tank is destroyed, it should explode in a glorious fashion. Also, shots from a tank tend to explode no matter what they hit. Try spawning an explosion at the point it was shot rather than moving the red sphere around. It will give the player a much better sense and feel of what they are shooting at.

The Angry Birds game can also use some explosions, especially the black bird. Every time something is destroyed, it should throw out some sort of particles and make a little bit of noise. Otherwise, it will continue to look a little weird when things just disappear suddenly.

## **Summary**

In this chapter, we learned about the special effects in Unity, specifically audio and particle systems. We started by understanding how Unity handles audio files. By adding background music and some squeak to the ball, we put into practice what we learned. We moved on to understand particle systems and created a dust trail for the ball. Finally, we put the two skill sets together and created explosions for the bananas when collected. Particle systems and audio effects add a lot to the final polish and look of a game.

In the next chapter, we will complete our gaming experience together by taking a look at optimization in Unity. We will take a look at the tools provided for tracking performance. We will also create our own tool to track specific parts of a script's performance. We will explore asset compression and the other points that we can change to minimize the application footprint. Finally, key points for minimizing the lag while working with games and Unity will be discussed.



# 9

## Optimization

In the previous chapter, we learned about special effects for our games. We added background music to our Monkey Ball game. We also created dust trails for our monkey. By combining both audio effects and particle systems, we created explosions when a player collects a banana. Together, these round out the game experience and give us a very complete-looking game.

In this chapter, we explore our options for optimization. We start by looking at the application footprint and how to reduce it. We then move on to look at the game's performance even further. Finally, we explore some key areas that can cause lag and look at how we can minimize their effects.

In this chapter, we will be covering the following topics:

- Minimizing the application footprint
- Tracking performance
- Minimizing lag
- Occlusion culling

For this chapter, we will be working on both our Monkey Ball and Tank Battle games. Start the chapter by opening the Monkey Ball project.

### Minimizing the application footprint

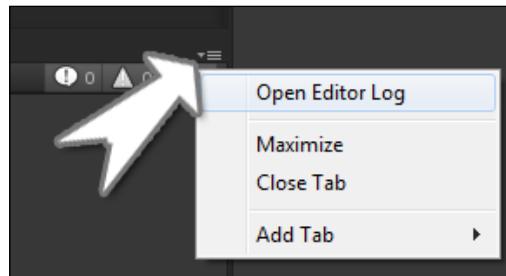
One of the keys to a successful game is the size of the game itself. Many users will quickly uninstall any application that appears to be unnecessarily large. In addition, all of the mobile app stores impose restrictions on how your game will be supplied to users based on the size of the application itself. Becoming familiar with the options that you have for minimizing the size of your game is the key to control how your game will be distributed.

The first thing to note when working to minimize the footprint is how Unity handles assets as it builds the game. Only assets that are used somewhere in one of the scenes for the build are actually included in the game. If it is not in the scene itself or referenced by an asset that is in the scene, it will not be included. This means you could have test versions of assets, or incomplete versions; as long as they are not referenced, they will not affect the final build size of your game.

Unity also allows you to keep your assets in the format that you need for working on them. When the final build is made, all the assets are converted to an appropriate version for their type. This means that you can keep models in the format that are native to your modeling program, which will be converted to FBX files. Otherwise, you can keep your images as Photoshop files, or any other format in which you work, and they will be converted to JPG or PNG appropriately when the game is built.

## Editor log

When you are finally ready to work with the footprint of your game, it is possible to find out exactly what is causing your game to be larger than desired. In the top-right corner of the **Console** window is a drop-down menu button. Inside this menu is **Open Editor Log**:



The Editor log is the location where Unity outputs information while it is running. This file tracks information about the current version of the Unity Editor, performs any checks done for your license, and contains a bit of information about any assets you have imported. The log will also contain detailed information about the file size and assets included in the game, after it has been built. An example of the Editor log is shown in the following screenshot:

|                                                  |          |                                 |
|--------------------------------------------------|----------|---------------------------------|
| Textures                                         | 1.9 mb   | 26.6%                           |
| Meshes                                           | 160.2 kb | 2.2%                            |
| Animations                                       | 0.0 kb   | 0.0%                            |
| Sounds                                           | 0.3 kb   | 0.0%                            |
| Shaders                                          | 834.9 kb | 11.3%                           |
| Other Assets                                     | 10.2 kb  | 0.1%                            |
| Levels                                           | 124.1 kb | 1.7%                            |
| Scripts                                          | 231.8 kb | 3.1%                            |
| Included DLLs                                    | 3.9 mb   | 54.6%                           |
| File headers                                     | 20.8 kb  | 0.3%                            |
| Complete size                                    | 7.2 mb   | 100.0%                          |
| <b>Used Assets, sorted by uncompressed size:</b> |          |                                 |
| 835.0 kb                                         | 11.3%    | Resources/unity_builtin_extra   |
| 682.7 kb                                         | 9.3%     | Assets/Models/Monkey/Monkey.psd |
| 682.7 kb                                         | 9.3%     | Assets/Models/Banana/Banana.psd |
| 170.7 kb                                         | 2.3%     | Assets/Models/Map/Grass.psd     |
| 170.7 kb                                         | 2.3%     | Assets/Models/Fence/wood.psd    |

Here, we can see a breakdown of the aspects of the final build. Every asset category has a size and percentage of the total build size. We are also supplied with a list of every asset that is actually included in the game, organized by their file size before they are added to the build. This information becomes very useful when you are looking for assets that can be made smaller.

## Asset compression

Inside the **Import Settings** window for models, textures, and audio, there are options that affect the size and quality of imported assets. In general, the affected change is a reduction in quality. However, especially when working on a game for the mobile device, asset quality can be reduced well below the levels required for a computer before the difference becomes noticeable on the device. Once you understand the options available for each type of asset, you will be able to make optimal decisions regarding the quality of your game. When working with any of these options, look for a setting that minimizes the size before introduction of undesired artifacts.

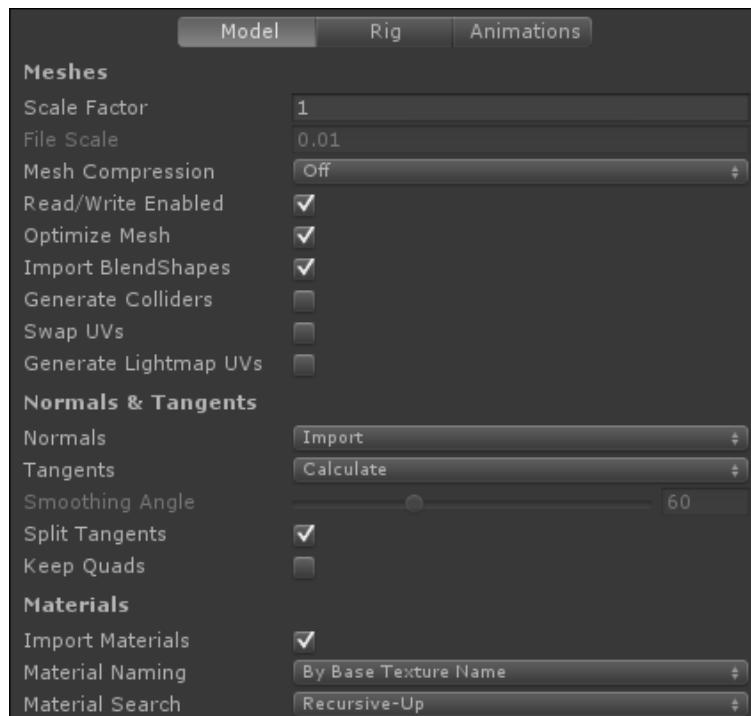
## Models

No matter what program or method you use to create your models, ultimately, there is always a list of vertex positions and triangles, with a few references to textures. Most of the file size of a model comes from the list of vertex positions. To make sure that the models in your game are of the highest quality, start in the modeling program of your choice. Delete any and all extra vertexes, faces, and unused objects. Not only will this result in a smaller file when you build your final game, but it will also reduce the import time when you work in the editor.

The **Import Settings** window for models consists of three pages, resulting in more options to adjust the quality. Each page tab corresponds to the relevant part of the model, allowing you to fine-tune each one of them.

## The Model tab

On the **Model** tab, you can influence how the mesh is imported. When it comes to optimizing your use of the models, there are many options here that are key to your success. Once your game looks and plays the way you want it to, you should always have a good look at these settings to see if you can make them work even better:



The following are the various settings in the **Model** tab:

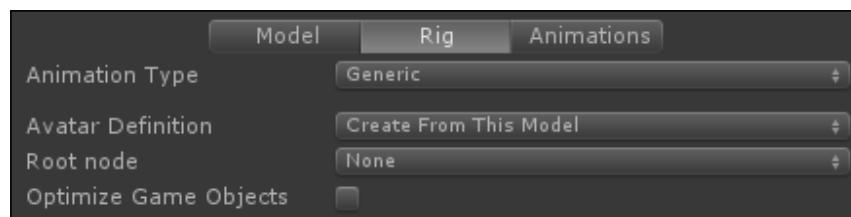
- **Scale Factor** and **File Scale**: These let you control the default visual size of your model. The **File Scale** parameter is how big Unity calculated your model to be when importing it. The **Scale Factor** parameter lets you adjust what additional scaling Unity will apply when it imports your model.

- **Mesh Compression:** This option lets you select how much compression should be applied to the model. The compression effect amounts to combining vertexes to reduce the overall amount of detail that has to be stored for the mesh. This setting is likely to introduce undesired oddities in the mesh, when pushed too far. So, always pick the highest setting that does not introduce any artifacts.
- **Read/Write Enabled:** This option is only useful when you want to manipulate the mesh, through the script, while the game is running. If you never touch the mesh with any of your scripts, uncheck this box. Although this will not affect the final build size, it will affect how much memory is required to run your game.
- **Optimize Mesh:** This option causes Unity to reorder the triangles list that describes the model. This option is always a good one to leave checked. The only reason you might want to uncheck it is if you are manipulating the game or mesh based on the specific order of the triangles.
- **Import BlendShapes:** BlendShapes are similar to keyframes in a normal animation, but they work on the mesh detail itself rather than the positions of bones. By unchecking this box, you can save space in your game and project because Unity will not need to calculate and store them.
- **Generate Colliders:** This option is almost always a candidate to leave unchecked. This option will add **Mesh Collider** components to every mesh in your model. These colliders are relatively expensive to calculate when working with physics in your game. If possible, you should always use a group of significantly simpler **Box Colliders** and **Sphere Colliders**.
- **Swap UVs:** Unity supports models that have two sets of UV coordinates. Generally, the first is for normal texture and the second is for any lightmaps that the object has. If you generate your own lightmap UVs, it is possible for Unity to recognize them in the wrong order. Checking this box then forces Unity to change the order in which they are used.
- **Generate Lightmap UVs:** This option should only be used when you are working with objects that need static shadows. If the object does not need it, this will introduce excess vertex information and bloat the asset.
- **Normals:** This option is used to calculate or import normal information. **Normals** are used by materials for determining direction in which a vertex or triangle faces and how lighting should affect it. If the mesh never uses a material that needs the **Normals** information, be sure to set this to **None**.

- **Tangents:** This option is used to calculate or import tangent information. **Tangents** are used by materials to fake details with bump maps and similar special effects. Just as with the **Normals** setting, if you don't need them, don't import them. If **Normals** is set to **None**, this setting will automatically be grayed out and will no longer be imported.
- **Smoothing Angle:** When calculating normals, this option lets you define how close the angle between two faces needs to be to be shaded smoothly across their shared edge.
- **Split Tangents:** This causes the tangents of your mesh to be recalculated where there are seams in your UVs. This is used for fixing some lighting irregularities in highly-detailed models.
- **Keep Quads:** Unity normally converts all faces to triangles for rendering. If you are using DirectX 11 for rendering, this option will keep your faces as quads for tessellation.
- **Import Materials:** This option lets you control whether or not new materials will be created when you are importing your models. If this is unchecked, no new models will be created when you are importing.
- **Material Naming:** This lets you control the manner in which the models that are imported will name any new materials that are created.
- **Material Search:** Unity can use a variety of methods for finding a material to use on the model that has already been created. The **Local Materials Folder** option will only look in a folder named **Materials** next to where the model is imported. The **Recursive-Up** option will look in the folder of the model, and the root assets folder through parent levels up. The **Project-Wide** option will search your whole project for a material with the right name.

## The Rig tab

As we can see in the following screenshot, there are very few options to adjust for an animation rig:

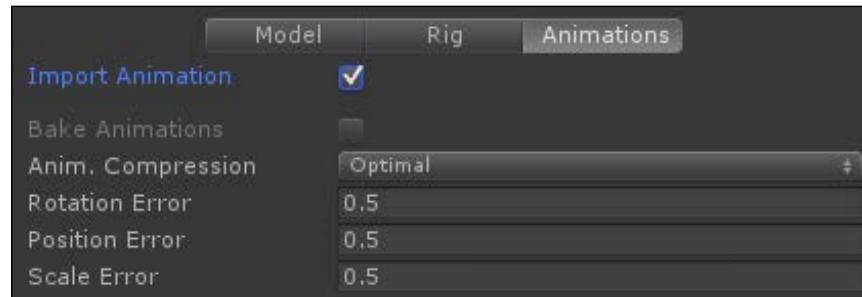


There are really only two things to keep in mind when you are working to optimize your animation rig. The first is, if the asset does not animate, then don't import it. By setting **Animation Type** to **None**, Unity will not try to import the rig or any useless animations. The second thing to keep in mind is to remove any unnecessary bones. Once imported to Unity, delete any and all objects from the rig that do not actually have an effect on the animation or character. Unity can convert any inverse kinematics that you might use for animation into forward kinematics, so the guides used for it can be deleted once Unity is launched.

The **Optimize Game Object** checkbox that is there does not actually help in the overall optimization of the game. It simply hides the extra rig objects in the Hierarchy window, so you don't have to deal with them. This checkbox can also be very helpful when dealing with complex rigs in the editor.

## The Animations tab

As with the **Rig** tab, if the model does not animate, do not import animations. Unchecking the **Import Animation** checkbox when you first import the asset will prevent any extra components from being added to your **GameObject** components in Unity. In addition, if any extra animations get added to your final build accidentally, they can quickly make your application oversized. The Animations tab is highlighted in the following screenshot:

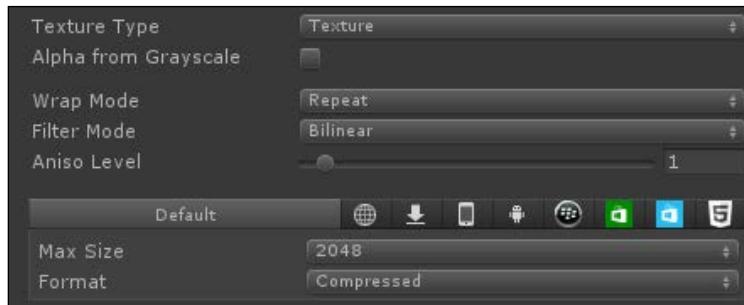


- **Anim.Compression:** This option adjusts how Unity handles excess keyframes in your animations. For most situations, the default option works well. The various options available are as follows:
  - **Off:** This option should only be used if you need a high-precision animation. This is the largest and most costly setting to choose.
  - **Keyframe Reduction:** This option will reduce the number of keyframes used by the animation based on the Error settings that follow. Essentially, if a keyframe does not have a noticeable effect upon the animation, it will be ignored.

- **Optimal:** This option does the same as the previous option, but additionally it compresses the file size of the animations. However, at runtime, the animation will still require the same amount of processor resources for calculation as the previous option.
- **Rotation Error:** This option is the difference of the number of degrees between keyframes that will be ignored when performing keyframe reduction.
- **Position Error:** This option is the movement distance that will be ignored between keyframes when performing keyframe reduction.
- **Scale Error:** This option is the amount of size adjustment in the animation that will be ignored between keyframes when performing keyframe reduction.

## Textures

It is hard to imagine a quality game that does not have a whole bunch of images in it. Textures have a bunch of options to control how much detail will be preserved when they are used in the game. In general, it is best to select the lowest quality settings that do not introduce noticeable artifacts in the image. In addition, it is best to work with texture sizes that are in a power of two to improve processing speed. Moreover, few processors are commonly able to handle textures that are greater than  $1024$  pixels in size. By putting your images in or below this size, you potentially save a lot of memory and space in your final game.



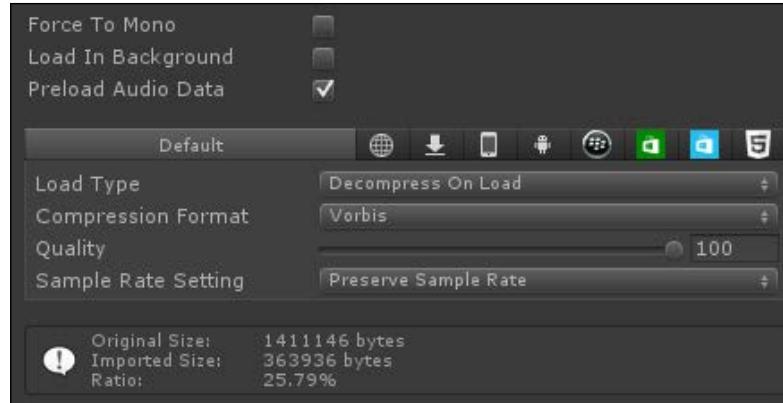
- **Texture Type:** This option affects what type of texture the image will be treated as. It is always best to select the type that is most appropriate for the intended use of the image. The following options show the various types of textures that can be used:
  - **Texture:** This option is the most common and default setting when working with 3D games. This should be used for your normal model textures.

- **Normal Map:** This option is used for special effects such as bump maps. Materials that use this type of texture will also need normal and tangent information from the model's import settings.
  - **Editor GUI and Legacy GUI:** Unless you are working with special editor scripts, or other special cases, you will not use this setting. This is very similar to the **Sprite** setting.
  - **Sprite (2D and UI):** This option is the most common and default setting when working with 2D games. This should always be used for your flat 2D characters and UI elements.
  - **Cursor:** This setting is not particularly relevant to our Android platform. It allows you to create custom mouse pointers that aren't commonly available for most Android devices.
  - **Cubemap:** When you are working with custom reflections or skybox type materials, your images should use this option. This automatically wraps the image around, so it repeats like the edges of a sphere or cube.
  - **Cookie:** These textures are used on lights and they change how the light is emitted from the light object, like the ones we used for our tank's headlights.
  - **Lightmap:** We worked with Unity's lightmapping system in our Tank Battle game. However, this system won't always work for all situations. So, when you are making custom lightmaps outside of Unity, choose this option.
  - **Advanced:** This option gives you full control over all the settings that are concerned with importing images. You will only need this setting if you have a special purpose for your textures or you need precise control over them.
- **Read/Write Enabled:** This checkbox is available when **Texture Type** is set to **Advanced**. This should only be left checked if you plan to manipulate the texture from your scripts while the game is running. If this is unchecked, Unity does not maintain a copy of the data in the CPU, freeing memory for other parts of the game.
  - **Generate Mip Maps:** This option is another **Advanced** setting that lets you control the creation of smaller versions of the texture. These are then used when the texture is small on the screen, reducing the amount of processing needed to draw the texture and the object that is using it on the screen.

- **Filter Mode:** This option is available for all of the texture types. It affects how the image will look when you are very close to it. **Point** will make the image look blocky, while **Bilinear** and **Trilinear** will blur the pixels. In general, **Point** is the fastest mode; **Trilinear** is the slowest mode but gives the best-looking effect.
- **Max Size:** This option adjusts how large the image can be when it is used in the game. This allows you to work with images that are very large but import them to Unity in an appropriately small size. In general, values greater than **1024** are poor choices, not just because of the increased memory requirement but also since most mobile devices simply cannot handle textures that are any larger. In general, 1024-sized textures should be reserved for your main characters and other highly important objects. A size of 256 works well on mobile devices for objects with medium and low importance. For all of your objects, if you can combine their textures to share a 1024 texture, they will have a smaller impact on your game than if they have small separate textures. Choosing the smallest size possible will have a great effect on the footprint size of the textures in your final build.
- **Format:** This option adjusts how the image would be imported and how much detail each pixel can hold. **Compressed** is the smallest format, while **Truecolor** provides the most detail.

## Audio

Giving a game quality sound always adds a lot to the final size of the game. It is one of the assets that a game cannot do without, but it can be hard to include at a suitable level. When working on the sounds in your audio program, keep them as short as possible to minimize their size. In addition, bear in mind that most of your players will not have the same fancy headphones or speakers to listen to your audio, so quality can be reduced to quite an extent before they notice any difference. The audio import settings all have an effect on either their footprint in the build size or the memory required to run the game.



- **Force To Mono:** This setting converts multichannel audio into a single channel. While most devices are technically capable of playing stereo sounds, they do not always have the multiple speakers required for it to make a difference. Checking this box can significantly reduce the file size of the audio by combining all of the channels into a single, smaller one. Multichannel audio files are used to give the illusion of direction based on which speaker the sound is coming from. This essentially requires separate sound files for each speaker. Mono channel audio files use the same sound file for all speakers and thus require much less data and space in your game.
- **Load In Background** and **Preload Audio Data:** These two settings work together to define when the audio information will be loaded and made ready to be played. The **Load In Background** parameter determines whether the game waits until the file is loaded before loading any other game data. Checking this box is a good idea for long or large files, such as background music. The **Preload Audio Data** parameter determines whether the files should be loaded as soon as possible. Any audio clips that you are going to need right away should have this option checked.
- **Load Type:** This setting affects how much of the system's memory will be used, while the game is running, to handle the loading of audio files. The **Decompress on Load** option uses maximum memory and is best for small, short sounds. The **Compressed in Memory** option only decompresses the file while it is playing, using a medium amount of memory, and is best for medium-sized files. The **Streaming** option means that only the part of the file that is currently being played is stored in the runtime memory. This is like streaming video or music from the Internet. This option is best for large files but should only be used by a few at one time.

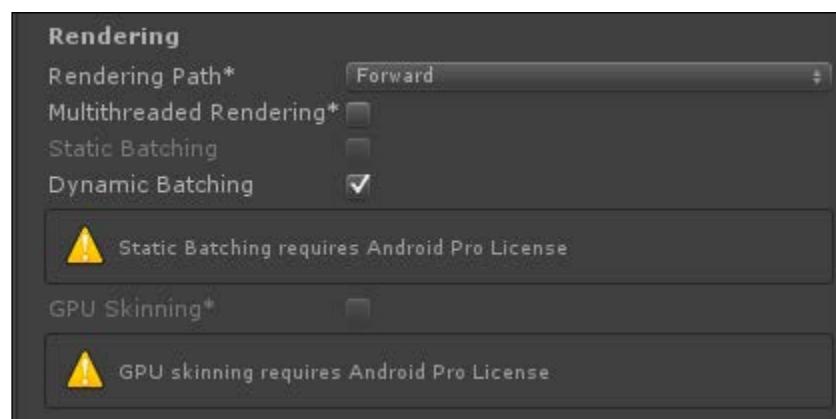
- **Compression Format:** This determines what sort of data reduction to apply to the audio file to make it small enough to include in the game. The **PCM** format is going to preserve most of the original audio and will be the largest file size as a result. The **ADPCM** format will give you a medium level of compression, but it will also reduce some of the quality as a result. The **Vorbis** format can give you the smallest possible file size, but at the cost of maximum reduction in quality.
- **Quality and Sample Rate Setting:** These control the amount of detail that will be preserved when you apply compression from the previous option. If the file size is still too large, you can reduce the overall quality to bring it within acceptable limits. However, reducing quality here comes at the cost of the sound quality. Always seek the lowest setting possible before artifacts are introduced and audible on your target device.

## Player settings

Open your game's **Player Settings** window by going to Unity's toolbar and navigating to **Edit | Project Settings | Player**. In the platform-specific settings for Android, we have another few options under **Other Settings** that will affect the final size and speed of our game.

## Rendering

The **Rendering** group of settings control how your game will handle drawing your game on the screen. This controls the kind of lighting and shadow calculations that are used. It also allows you to optimize the number of calculations needed to draw the many objects that make up your game's scene. The **Rendering** window can be seen in the following screenshot:



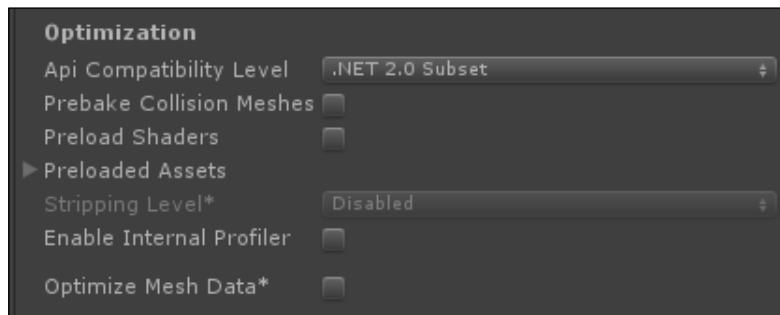
The settings that are seen in the **Rendering** window are as follows:

- **Rendering Path:** This set of options primarily controls the quality with which lights and shadows are rendered. The options under **Rendering Path** are as follows:
  - **Forward:** This is going to be your most common setting. It supports real-time shadows from a single directional light. This option is your normal baseline for rendering light in Unity.
  - **Deferred:** This is going to give you the highest quality lighting and shadow, but it will cost the most for the system to process it. Not every system is going to be able to support it, and it happens to be a Unity Pro-only feature.
  - **Legacy Vertex Lit:** This rendering method is part of the old system. It is also the cheapest method to process. There are no real-time shadows with this method, and the lighting calculations are highly simplified. Older machines and mobile devices will default to this mode.
  - **Legacy Deferred (light prepass):** This method is also part of the old system. The newer **Deferred** method is highly improved over this one and in general, this should not be used. You would only need to select this method if you have a special case or need to support a specific platform.
- **Multithreaded Rendering:** The process and series of steps used to run a program is called a thread. It is possible to start many of these threads and make them work on different parts of the program at the same time. Unity has utilized this feature of programming to increase the speed and quality of the rendering system. However, it requires a more powerful processor to run effectively.
- **Static Batching:** This is a Unity Pro feature that allows Unity to significantly speed up rendering times by grouping identical objects that have been marked as static in the **Inspector**. For each group, it then renders one object in multiple places rather than rendering each object individually. Potentially, this setting can add some extra girth to your final build size because Unity will need to save extra information about your static objects to make this work.
- **Dynamic Batching:** This works in the same manner as **Static Batching**, but with two major differences. First, it is available to both Unity Pro and Basic users. Second, it groups objects that are not marked as static.

- **GPU Skinning:** This setting is less applicable for older mobile devices, and it is used more for the newest of mobile devices and other systems that have both a CPU and GPU. This allows the calculations that are normally done on meshes, that animate and are deformed by bones, to take place on the GPU instead of the CPU. This will free up resources for processing other parts of your game and giving your players the best experience possible.

## Optimization

The **Optimization** group of settings allows you to adjust how Unity will compile your project and the assets involved. Each setting should be given careful consideration when you approach the final build of your game. Altogether, these settings have the potential to make a huge difference in how well your game runs. The **Optimization** window is shown in the following screenshot:



- **Api Compatibility Level:** This setting determines which set of the .NET functions to include in the final build. The **.Net 2.0** option will include all of the available functions, making the largest footprint. The **.Net 2.0 Subset** option is a smaller portion of the functions and includes only the functions that your programming is most likely to use. Unless you need some special functionality, the **.Net 2.0 Subset** option should always be the option that you choose.
- **Prebake Collision Meshes:** This box saves time when you load your levels, by moving the physics calculations from the scene load to the game build. It means your build size will be large but the processing speed will be reduced.
- **Preload Shaders:** When a mesh uses a new shader that has not yet been used in the game scene, the system needs to process and calculate how that shader will render objects. This box will process that information when the scene starts and avoid potential stalls in your game while it tries to do the calculations.

- **Preloaded Assets:** This option does the same thing as the previous one, but for assets and prefabs instead of shaders. When you first instantiate an object, it needs to be loaded into memory. This will change it so that all of the assets in this list are loaded when the scene starts.
- **Stripping Level:** This setting is a Unity Pro only feature. It allows you to reduce the size of your final build by removing all of the excess code before compiling it. System functions are grouped into what are called libraries for easy reference. The **Strip Assemblies** option removes the unused libraries from the final build. The **Use micro mscorelib** option performs the same function as the previous option but utilizes a minimized form of the libraries. While significantly smaller, this library possesses fewer functions for your code to use. However, unless your game is complex, this should not make a difference.
- **Enable Internal Profiler:** This option lets you retrieve information about how your game is running on a device. It does introduce a little bit of overhead to process the information while your game is running, but the effect is less than what Unity Editor introduces. The information can be retrieved using the `adb logcat` command in the command prompt.
- **Optimize Mesh Data:** This setting will remove extra information from all of your meshes that are not being used by any materials that are applied to them. This includes **Normals**, **Tangents**, and a few other bits of information. It also causes the triangle data that makes up the mesh to be reordered for optimal processing and rendering. Unless you have a very special case, this is a good box to always check.

## Tracking performance

Unity provides us with many tools that allow us to determine how well our game is running. The first tool that we will be covering is readily available for both Unity Pro and Basic users. However, the information is rather limited, though it is still useful. The second tool is only available to Unity Pro users. It provides significantly more detail and information on performance. Finally, we will create our own tool, allowing us to view the performance of our scripts in detail.

## Editor statistics

In the top-right corner of the **Game** window, there is a button labeled **Stats**. Clicking on this button will open a window that will give us information about how the game is running and how long it will take to process. Most of the information in this window concerns how well the game is being rendered, largely amounting to how many objects are currently on the screen, how many are animating, and how much memory they take up. Additionally, there is also some information about the sound in the game and any network traffic that might be occurring. The **Stats** tab is displayed in the following screenshot:



- The **Audio** section concerns the various audio clips that are playing in your scene. It contains information about how loud your game is and how much memory is required to process it all. The **Audio** section consists of the following details:
  - **Level:** This is how loud your game is in decibels. It is really just a special form of volume measurement and represents a total for every audio clip that is playing in your game.
  - **DSP load:** This is the cost in processing the digital audio clips in your scene. It is represented as a percentage of the memory used by your game.
  - **Clipping:** This is the percentage of your audio files that are simply not played because of the overload on the system. Based on the power of your device's processor, the device can only play a limited number of audio clips at one time. Any extra audio clips are ignored based on the priority setting in the **Inspector** panel of the **Audio Source** component.

- **Stream load:** This is the cost involved in processing any audio that must be streamed as it is being played. It is again a percentage of the memory used.
- The **Graphics** section is concerned with the rendering of your game and the memory required in doing this. It contains information about how fast the game is running, how many objects are being rendered, and how detailed the objects are. Most of the time, when using the **Stats** window, you will be looking at this section. The **FPS** value to the right of this group heading is an excellent estimation of how fast your game is running. This is the number of frames being processed in one second, followed by the number of milliseconds it takes to process a single frame of your game. The **Graphics** section consists of the following details:
  - **CPU:** This section breaks down into two pieces. The **main** piece is how long it takes to process the code that is used to run your game. The **render thread** piece is how long it takes to draw all the parts of your game on the screen. Together, you can get a good idea of what is taking the most time to run in your game.
  - **Batches:** When using **Static** or **Dynamic Batching**, found in the **Rendering** group of **Player Settings**, the first number denotes how many groups were created for the batch rendering pass and the **Saved by batching** value is the number of draw calls that were avoided because of the batching process. The more saved means that less work was done to draw your game on the screen.
  - **Tris:** Ultimately, every model in 3D graphics is made from a series of triangles. This value is the total number of triangles that are seen and are being rendered by the cameras in your scene. Fewer triangles means that the graphics process has to do less work to draw a model on the screen.
  - **Verts:** Most of the information in a model file is concerned with the world position, normal orientation, and texture position of each vertex. This value is the total number of vertexes seen and rendered by the camera. The lower the number of vertexes for each model, the faster it will be calculated for rendering.
  - **Screen:** This is the current width and height, in pixels, of the **Game** window. It also displays the amount of memory that is needed for rendering at that size. A smaller size results in less detail for your game, but it also makes the game easier to render.

- **SetPass calls:** This is essentially the number of times the different parts of a shader need to be called to draw everything in your scene on the screen. It is based more on the number of different materials in your scene than the number of objects.
- **Shadow casters:** This statistic is used when you make use of real-time shadows. Real-time shadows are expensive. If possible, they should not be used on mobile devices. However, if you have to have them, minimize the number of objects that cast shadows. Limit it to move objects that are large enough for the user to see the shadow. Small, static objects especially do not need to cast shadows.
- **Visible skinned meshes:** This is the total number of rigged objects that are currently in the view of the camera. Skinned meshes are often going to be your characters and just about anything else that animates. They are more expensive to render than static meshes because of the extra calculations that are needed to make them move and change with animation.
- **Animations:** This is simply the total number of animations playing in the scene.
- The **Network** group of statistics only becomes visible when it is connected to other players in a multiplayer game. The information generally amounts to how many people the game is connected to and how fast those connections are.

## The Profiler

The **Profiler** window, found in Unity's toolbar by navigating to **Window | Profiler**, is a great tool for analyzing how your game is running. It gives us a colorful breakdown of each part of our system and how much work it is doing. The only really unfortunate part of this tool is that it is only available to Unity Pro users. The **Profiler** window is shown in the following screenshot:



By first opening the **Profiler** window, we can then play our game in the window and watch the tool give us a fairly detailed breakdown of what is going on. We can click on any point and see detailed information about that frame at the bottom of the window. The information provided is specific to the point, such as **CPU Usage**, **Rendering**, **Memory**, and so on, that you clicked on.

The **CPU Usage** information is particularly useful when we are trying to find parts of our game that are taking too long to process. Spikes in processing cost stand out pretty easily. By clicking on a spike, we can see the breakdown of what each part of the game played in making that frame expensive. For most of these parts, we can dig down to the exact object or function that is causing the issue. However, we can only get down to the function level. Just because we know where an issue in the code generally is, the **Profiler** window will not tell us exactly which part of that function is causing the issue.

In order to actually work, the Profiler needs to hook into every part of your game. This introduces a little extra cost in the speed of your game. Therefore, when analyzing the information provided, it is best to consider the relative costs rather than hold each cost as an exact value.

## Tracking script performance

All of these tools that Unity provides are great, but they are not always the right solution. The Unity Basic user does not have access to the **Profiler** window. In addition, both **Profiler** and **Editor Statistics** are fairly generalized. We can get a little more detail with the **Profiler**, but the information is not always enough without you having to dig through a bunch of menus. In this next part, we will create a special script that is capable of tracking the performance of specific parts of any script. It should definitely become a regular piece of your developer kit. Let's use these steps to create the script in our Monkey Ball game:

1. First, we will need a special class that will keep track of our performance statistics. To do this, create a new script and name it `TrackerStat`.
2. To begin this script, we need to enable the ability to interact with the various GUI elements. Go to the very top of the script and add this line next to the other lines that begin with `using`:  
`using UnityEngine.UI;`
3. Next, we need to change the class definition line. We do not want or need to extend the `MonoBehaviour` class. So, find the following line of code:

```
public class TrackerStat : MonoBehaviour {
```

And, change it to the following code:

```
public class TrackerStat {
```

4. This script starts with four variables. The first variable will be used as an ID, allowing us to track multiple scripts at once by supplying different key values. The second variable will keep track of the average amount of time that the tracked bits of code will take. The third variable is just the total number of times the tracked code has been called. The fourth variable is the longest time the code has taken to execute:

```
public string key = "";
public float averageTime = 0;
public int totalCalls = 0;
public float longestCall = 0;
```

5. Next, we have two more variables. These will do the work of actually tracking how long the script takes to execute. The first variable includes the time when the tracking starts. The second variable is a flag that marks that tracking has started.

```
public float openTime = 0;
public bool isOpen = false;
```

6. The third and last batch of variables for this script is used to store references to the Text objects that will actually display our stat information:

```
private Text averageLabel;
private Text totalLabel;
private Text longestLabel;
```

7. The first function for this script is `Open`. This function is called when we want to start tracking a bit of code. It first checks to see whether the code is already being tracked. If it is, then it uses `Debug.LogWarning` to send a warning to the **Console** window. Next, it sets the flag marking that the code is being tracked. Finally, the function tracks the time it was called by using `Time.realtimeSinceStartup`, which contains the actual number of seconds since the game started.

```
public void Open() {
 if(isOpen) {
 Debug.LogWarning("Tracking is already open. Key: " + key);
 }

 isOpen = true;
 openTime = Time.realtimeSinceStartup;
}
```

8. The next function, `Close`, acts as the opposite of the previous one. It is called when we have reached the end of the code that we want to track. The time when the tracking should stop is passed to it. This is done to minimize excess code from being executed. As with the previous function, it checks to see whether tracking is being done and sends out another warning and exits early if the function is not being tracked. Next, the `isOpen` flag is cleared by setting it to `false`. Finally, the amount of time since tracking was opened is calculated and the `AddValue` function is called.

```
public void Close(float closeTime) {
 if(!isOpen) {
 Debug.LogWarning("Tracking is already closed. Key: " + key);
 return;
 }

 isOpen = false;
 AddValue(closeTime - openTime);
}
```

9. The `AddValue` function is passed `callLength`, which is the length of time that the tracked bit of code took. It then uses some calculations to add the value to `averageTime`. Next, the function compares the current `longestCall` with the new value and updates it, if the new one is greater than the current one. The function then increments `totalCalls` before finally updating the text on screen with the new values.

```
public void AddValue(float callLength) {
 float totalTime = averageTime * totalCalls;
 averageTime = (totalTime + callLength) / (totalCalls + 1);

 if(longestCall < callLength) {
 longestCall = callLength;
 }

 totalCalls++;

 averageLabel.text = averageTime.ToString();
 totalLabel.text = totalCalls.ToString();
 longestLabel.text = longestCall.ToString();
}
```

10. The last function for our script, `CreateTexts`, is called when we first create an instance of this class to track some bit of code. It first calculates the vertical position of the GUI elements. By using the `ScriptTracker.NewLabel` function, which we will create in our next script, we can save ourselves some work; it automatically handles the creation and basic setup of the Text objects that are needed to display the stat information. We just need to pass a name to it to use it in the **Hierarchy** window and set the position and size when it gives us the new object.

```
public void CreateTexts(int position) {
 float yPos = -45 - (30 * position);

 Text keyLabel = ScriptTracker.NewLabel(key + ":Key");
 keyLabel.text = key;
 keyLabel.rectTransform.anchoredPosition = new Vector2(75, yPos);
 keyLabel.rectTransform.sizeDelta = new Vector2(150, 30);

 averageLabel = ScriptTracker.NewLabel(key + ":Average");
 averageLabel.rectTransform.anchoredPosition = new Vector2(200, yPos);
 averageLabel.rectTransform.sizeDelta = new Vector2(100, 30);

 totalLabel = ScriptTracker.NewLabel(key + ":Total");
```

```

 totalLabel.rectTransform.anchoredPosition = new Vector2(200,
yPos);
 totalLabel.rectTransform.sizeDelta = new Vector2(100, 30);

 longestLabel = ScriptTracker.NewLabel(key + ":Longest");
 longestLabel.rectTransform.anchoredPosition = new Vector2(200,
yPos);
 longestLabel.rectTransform.sizeDelta = new Vector2(100, 30);
 }
}

```

11. Next, we need to create another new script and name it `ScriptTracker`. This script will allow us to do actual performance tracking.
12. Just like we did for the previous script, we need to add a new line at the very top of the script next to the other using lines so that the script can create and interact with GUI objects:

```
using UnityEngine.UI;
```

13. This script starts off with a single variable. This variable maintains all of the stats that are currently being tracked. Note the use of `static` here; it allows us to easily update the list from anywhere in the game:

```
private static TrackerStat[] stats = new TrackerStat[0];
```

14. The first function for this script, `Open`, allows us to start tracking the code's execution. It uses the `static` flag, so the function can be called easily by any script. A key value is passed to the function, allowing us to group track calls. The function starts by creating a variable to hold the index of the stat to start tracking. Next, it loops through the current set of stats to find a matching `key` value. If one is found, the `index` variable is updated with the value and the loop is exited.

```

public static void Open(string key) {
 int index = -1;

 for(int i=0;i<stats.Length;i++) {
 if(stats[i].key == key) {
 index = I;
 break;
 }
 }
}

```

15. The open function continues by checking whether a stat was found. The index variable will be less than zero only if we make it through the whole loop of current stats and are unable to find a matching key. If one is not found, we first check to see whether the list of stats is empty and if it is empty, we create some display labels by calling the CreateLabels function. We then call AddNewStat to set up the new stat for tracking. We will create both of these functions shortly. The index is then set to that of the new stat. Finally, the stat is triggered to start tracking by using the stat's Open function.

```
if(index < 0) {
 if(stats.Length <= 0) {
 CreateLabels();
 }

 AddNewStat(key);
 index = stats.Length - 1;
}

stats[index].Open();
}
```

16. The AddNewStat function is passed the key of the stat that is to be created. It starts by storing the list of stats in a temporary variable and increasing the size of the stats list by one. Each value is then transferred from the temp list to the larger stats list. Finally, a new stat is created, and it is assigned to the last slot in the stats list. Then, the key is set and its CreateTexts function is called so that it can display on screen.

```
private static void AddNewStat(string key) {
 TrackerStat[] temp = stats;
 stats = new TrackerStat[temp.Length + 1];

 for(int i=0;i<temp.Length;i++) {
 stats[i] = temp[i];
 }

 stats[stats.Length - 1] = new TrackerStat();
 stats[stats.Length - 1].key = key;
 stats[stats.Length - 1].CreateTexts(stats.Length - 1);
}
```

17. Next, we have the `Close` function. This function is passed the key value of the stat to be closed. It starts by finding the time when the function was called, minimizing the amount of excess code that will be tracked. It continues by looping through the list of stats to find a matching key. If one is found, the stat's `Close` function is called and the function is exited. If a match is not found, `Debug.LogError` is called to send an error message to the **Console** window.

```
public static void Close(string key) {
 float closeTime = Time.realtimeSinceStartup;

 for(int i=0;i<stats.Length;i++) {
 if(stats[i].key == key) {
 stats[i].Close(closeTime);
 return;
 }
 }

 Debug.LogError("Tracking stat not found. Key: " + key);
}
```

18. The `CreateLabels` function handles the creation of text labels on the screen, so we can easily tell what each bit of displayed information means. Just like our previous script, it uses the `NewLabel` function to handle the basic creation of the text objects, passing it a name to be displayed in the **Hierarchy** window. It then sets the text to be displayed on the screen, positions it along the top-left corner of the screen, and sets its size.

```
private static void CreateLabels() {
 Text keyLabel = NewLabel("TrackerLabel:Key");
 keyLabel.text = "Key";
 keyLabel.rectTransform.anchoredPosition = new Vector2(75, -15);
 keyLabel.rectTransform.sizeDelta = new Vector2(150, 30);

 Text averageLabel = NewLabel("TrackerLabel:Average");
 averageLabel.text = "Average";
 averageLabel.rectTransform.anchoredPosition = new Vector2(200,
 -15);
 averageLabel.rectTransform.sizeDelta = new Vector2(100, 30);

 Text totalLabel = NewLabel("TrackerLabel:Total");
 totalLabel.text = "Total";
 totalLabel.rectTransform.anchoredPosition = new Vector2(275,
 -15);
 totalLabel.rectTransform.sizeDelta = new Vector2(50, 30);
```

```
 Text longestLabel = NewLabel("TrackerLabel:Longest");
 longestLabel.text = "Longest";
 longestLabel.rectTransform.anchoredPosition = new Vector2(350,
-15);
 longestLabel.rectTransform.sizeDelta = new Vector2(100, 30);
}
```

19. The last static function for this script is the `NewLabel` function. It handles the basic creation of each text object that we are using in the rest of the script. It first tries to find the canvas object and creates a new one if it can't be found. To use our text objects, we need the canvas so that they can actually be drawn.

```
public static Text NewLabel(string labelName) {
 Canvas canvas = GameObject.FindObjectOfType<Canvas>();
 if(canvas == null) {
 GameObject go = new GameObject("Canvas");
 go.AddComponent<RectTransform>();
 canvas = go.AddComponent<Canvas>();
 }
}
```

20. Next, the `NewLabel` function creates a new **GameObject** by using the name that was passed to it and making it a child of the canvas. It then adds the `RectTransform` component so that it can position itself in 2D space and anchors it to the top-left corner. The text object is then given a `CanvasRenderer` component so that it can actually draw on the screen and a `Text` component so it is actually a text object. We then use the `Resources.GetBuiltinResource` function to grab Unity's default `Arial` font for the text object before returning it to the caller of the function.

```
GameObject label = new GameObject(labelName);
label.transform.parent = canvas.transform;

RectTransform labelTrans = label.AddComponent<RectTransform>();
labelTrans.anchorMin = Vector2.up;
labelTrans.anchorMax = Vector2.up;

label.AddComponent<CanvasRenderer>();
Text textComp = label.AddComponent<Text>();
textComp.font = Resources.GetBuiltinResource(typeof(Font), "Arial.
ttf") as Font;
return textComp;
}
```

21. To test these scripts, open your `BananaBounce` script. At the beginning of the `Update` function, add the following line to start tracking how long it takes to run:

```
ScriptTracker.Open("BananaBounce Update");
```

22. At the end of the `Update` function, we need to call the `Close` function with the same key:

```
ScriptTracker.Close("BananaBounce Update");
```

23. Finally, start the game and take a look at the results (shown in the following screenshot):



We created a tool for testing specific parts of code. By wrapping any bit of code in calls to the functions and sending a unique ID, we can determine how long it takes to execute the code. By averaging out the calls to the script and wrapping different parts of code, we can determine exactly which parts of a script are taking the longest time to complete. We can also find out whether the parts of code have been called too many times. Both cases are ideal points to look at for minimizing processing and lag.

Be sure to remove any references to this tool before you deploy your game. If it is left in the final levels, it can add an unnecessary amount of load on the CPU. This adverse effect on the game could make the game unplayable. Always remember to clear out any uses of tools that are exclusively for Editor debugging.

## Minimizing lag

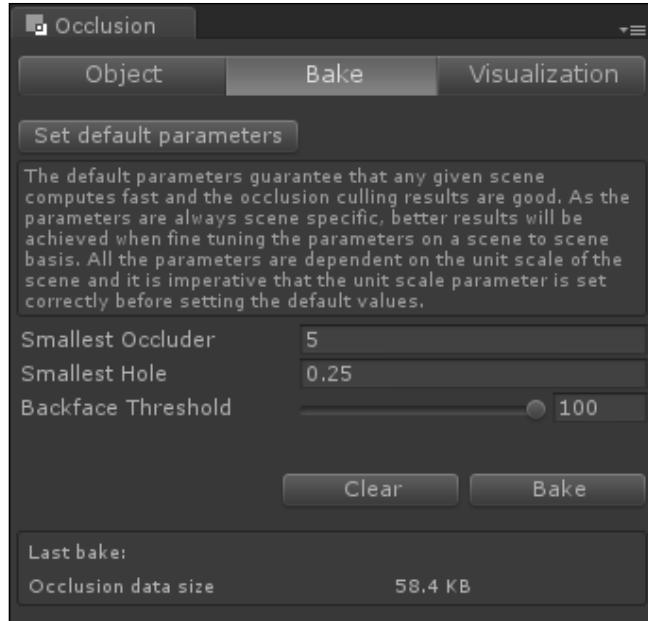
Lag is one of those nebulous ideas used to describe an application that is slower than expected. It is most commonly seen and recognized in an application's frame rate. Most games run at about 60 FPS and are considered to be lagging if they drop to 30 FPS or less. However, lag and its issues run deeper and include issues such as input responsiveness, internet connectivity, and file reading/writing. As developers, we constantly fight against providing the highest quality experience we can, while maintaining the speeds and responsiveness that users expect. It essentially amounts to whether or not the processor on the user's device can handle the cost of providing the game experience. A few, simple objects in your game will result in fast processing, but several complex objects will require the most processing.

## Occlusion

Occlusion is great for games with a lot of objects. In its basic form, anything that is at the sides or behind the camera is not seen and therefore not drawn. In Unity Pro, we are able to set up occlusion culling. This will calculate what can actually be seen by the camera and not draw anything that is blocked from view. There is a balance that has to be achieved when using these tools. The cost of calculating what cannot be seen needs to be less than the cost of just drawing the objects. There are no solid numbers for how long a scene might take to render. It all depends on the render settings that you have chosen and the detail in your models and textures. As a rule of thumb, if you have many small objects that are regularly blocked from view by larger objects, occlusion culling is the right choice.

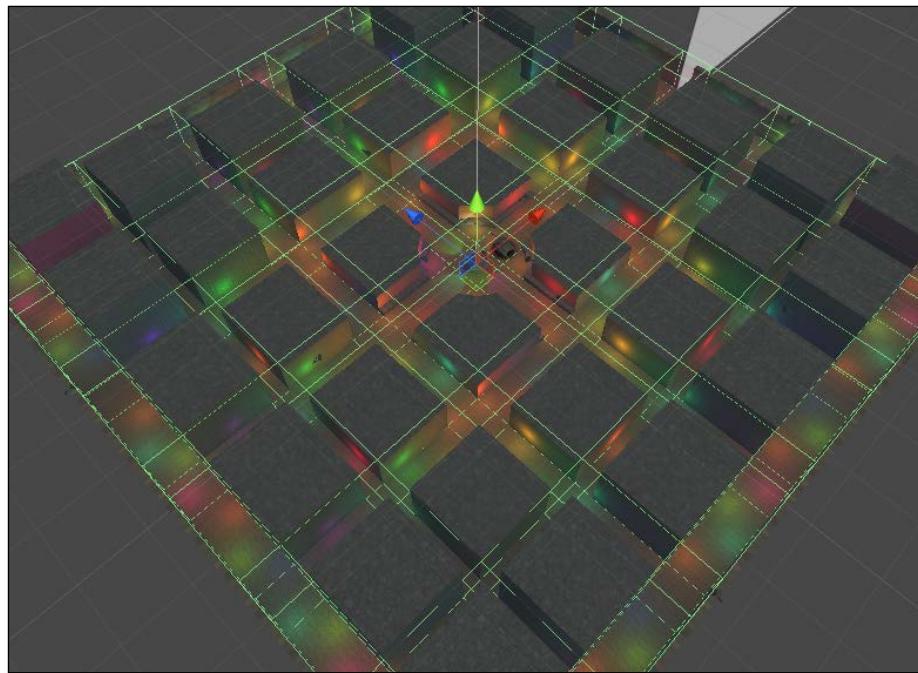
We will add occlusion culling to the Tank Battle game because it is the only one with anything large enough to regularly block objects from view. Let's set it up with these steps:

1. Open up the Tank Battle game now. If you completed the challenges and added the extra debris and obstacles, this section will be particularly effective for you.
2. Open the Occlusion window by going to Unity's toolbar and navigate to **Window | Occlusion Culling**. This window is your primary point of access for modifying the various settings associated with occlusion in your game. Unfortunately, this is a Unity Pro only feature. If you try to open the window while in Unity Basic, this will result in nothing more than an error message in the **Console**.
3. Switch to the **Bake** page and we can take a look at the options associated with occlusion culling:



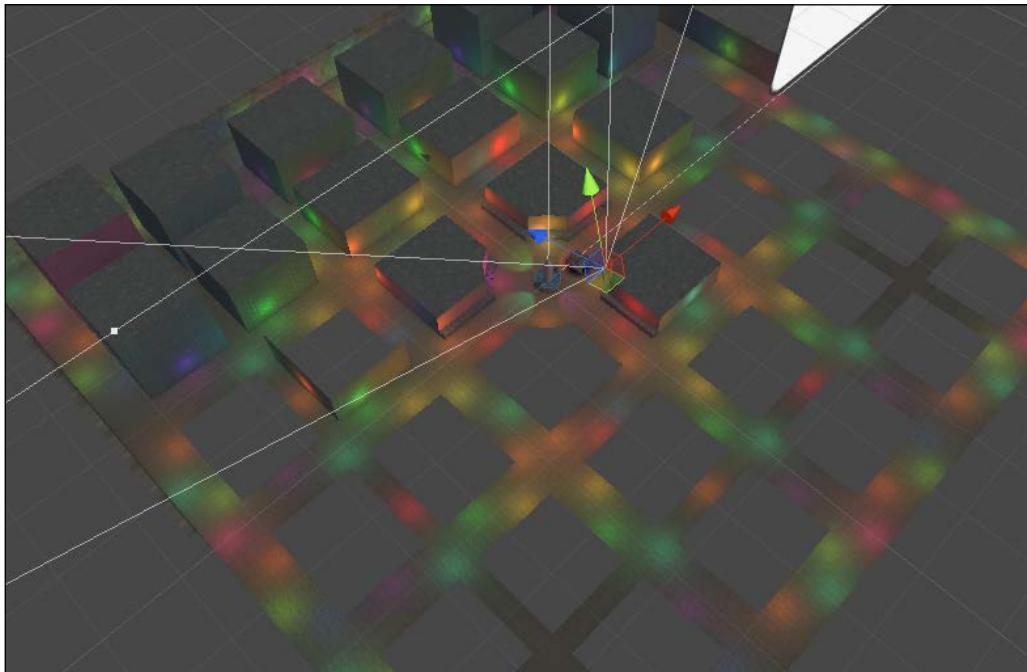
- **Smallest Occluder:** This should be set to the size of the smallest thing that can block other things from view. Things like large boulders and houses make good occluders. Smaller things like furniture or books are generally too small to block anything of significance from view.
  - **Smallest Hole:** This is the smallest gap in your scene through which you can see other objects. Smaller values require more detailed calculations. Larger values are less costly but are more likely to cause objects to flicker in and out of view as the player moves around.
  - **Backface Threshold:** This setting causes the system to do extra checks for objects that might be inside others. A value of 100 means no checks will be done and the calculation time will be saved. A value of 5 will require a bunch of extra calculations to figure out where everything is in relation to each other.
4. For our purposes at this point, the defaults will work fine. You ideally want to find settings that are balanced between the reduced cost of rendering and the cost of calculating what should be rendered.
  5. In order to make the occlusion system work with dynamic objects, we need to set up a number of occlusion areas. To create them, create an empty **GameObject** and add an **Occlusion Area** component that can be found in Unity's toolbar by navigating to **Component | Rendering | Occlusion Area**.

6. You will need to create and manipulate several of these objects. They need to cover the whole area where any dynamic objects and the camera can be located. To this end, create and position enough areas to cover the streets of our game. Their size can be edited just as when working with the **Box Collider** components. You can also use the little cylinders on each side of the area to manipulate the field. Be sure to make them tall enough to cover all of your targets (as you can see in the following screenshot):



7. Next, hit **Bake** at the bottom of the **Occlusion** window. A progress bar will appear at the bottom-right corner of the Unity Editor, which will tell you how much longer the calculations will take. This process usually takes a good amount of time, especially as your game becomes more and more complex. For our simple Tank Battle game, this process will not take particularly long. Our simple scene with very little in it should only take a couple of seconds to be processed. A large level that is full of detail could easily take all day to be processed.

8. When the baking process has completed, the **Occlusion** window would switch to the **Visualization** tab and, if it can be found, the camera should be selected in your **Scene** window. If it is not, select it now. In the **Scene** view, Unity will give us a preview of how occlusion culling is working. Only those objects that can be seen will be visible while the rest will be turned off (as seen in the following screenshot):



We went through the basic process to set up occlusion culling. We took a look at the **Occlusion** window and learned about the settings available there. Occlusion culling is great for reducing the number of draw calls in a scene. However, that reduction needs to be balanced against the cost of storing and retrieving the occlusion calculations. This balance is achieved by selecting a proper technique and an appropriate **View Cell Size**. Play around with the different values now to find a cell size that gives the appropriate amount of detail without supplying too much information.

## Tips for minimizing lag

The following is a list of tips for dealing with and avoiding lag in your games. Not all of them will apply to every game that you make, but they are good to keep in mind for every project:

- Avoid using transparency, if possible, when creating your materials. They are more expensive to render than a normal opaque material. In addition, you can save yourself a world of headaches of dealing with depth sorting, if you avoid them.
- Use one material per object. The greater the number of draw calls in your game, the longer each frame will take to render. Every mesh is drawn once per material on it, even if the material doesn't appear to do anything. By keeping to one material per object, especially on mobile platforms, you minimize the number of draw calls and maximize your rendering speed.
- Combine textures when possible. Not every texture you make will utilize the whole of the image. Whenever possible, combine the textures of objects that are in the same scene. This maximizes your efficient use of images, while reducing the final build size and amount of memory that is needed to utilize the textures.
- Group objects in your **Hierarchy** window using empty **GameObject** components. Though this is not specific to minimizing lag, it will make your project easier to work with. Especially with large and complex levels, you will be able to spend less time searching through the objects in your scene and more time making a great game.
- The **Console** window is your friend. Before worrying about your game not working, first take a look at the **Console** window or the bar at the bottom in Unity. Both will display any complaints that Unity might have about the way your game is currently setup. The messages here are great for pointing you in the right direction to fix any problems. If you are ever unsure what the messages are trying to tell you, perform a Google search for the message and you should be able to easily find a solution from one of the many Unity users. If your code ever appears to be not working and Unity isn't complaining about it, use the `Debug.Log` function to print messages to the Console. This will let you find places where your code might be exiting unexpectedly or find values that are not what they should be.
- Device testing is important. Working in the Editor is great, but there is nothing quite like testing on the target device. You can get a much better feel for how your game is performing when it is on the device. The Editor always introduces a small amount of additional processing overhead. In addition, the computer you are working on will always be more powerful than the mobile devices on which you might intend to deploy your game on.

## Summary

In this chapter, we learned about our options for optimization in Unity. We first took a look at the various settings for the assets used in our games that are used to keep their file size down while maintaining quality. Next, we learned about some settings that affect the overall game. After that, we explored options for tracking the performance of the game. We first looked at some tools provided by Unity for tracking that performance. Then, we created a tool of our own for tracking script performance in detail. We then took a look at some options for minimizing lag in our games, including utilizing occlusion culling. Now that we know about all of these tools and options, go through the games that we created and optimize them. Make them the best that they can be.

In this book, we learned a whole lot. We started with learning about Unity, Android, and how to make them work together. Our journey continued with an exploration of Unity's GUI system and the creation of a Tic-tac-toe game. We then learned about the basic assets needed for any game while we started the creation of a Tank Battle game. Our Tank Battle game then expanded with the addition of a few special camera effects and some lighting. We concluded the creation of the Tank Battle game by introducing some enemies and making them chase the player. The creation of our Monkey Ball game taught us about the touch and tilt controls that we can utilize in our game. A short break from that game saw the creation of an Angry Birds' clone while learning about physics and the options for working with Unity's 2D pipeline. We then returned to the Monkey Ball game to polish it with the addition of sound and particle effects. Finally, our journey concluded by learning about optimizing our games. Thank you for reading this book. We hope that you enjoy your experiences with Unity and Android while creating the awesome games that you always dreamed about.



# Index

## Symbols

### 2D games, in 3D world

- about 202
- development environment,
  - setting up 202-205
- parallax background, creating 232-235

## A

### Android

- about 1
- features 4
- working, with Unity 4, 5

### Android Debug Bridge (ADB) 13

### Android SDK

- installing 13-15

### Angry Birds game

- birds, adding 235
- black bird, adding 238-240
- blocks, building 205-209
- blue bird, adding 236-238
- evil pigs, creating 211-216
- level selection 241-243
- parallax background, creating 232-235
- physics materials, implementing 210
- red bird, creating 216
- yellow bird, adding 235, 236

### animations, in Unity

- about 94
- controlling, with state machine 101
- prefab, creating 112, 113
- settings, for importing 94-99
- target's animations, setting up 100

### application footprint

- asset compression 279
- Editor log 278
- minimizing 277, 278
- Player settings 288

### Artificial Intelligence (AI) 148

### asset compression

- about 279
- Animations tab 283, 284
- audio 286-288
- models 279, 280
- Model tab settings 280-282
- Rig tab 282, 283
- textures 284-286

### audio

- about 246
- background music, adding 251, 252
- import settings 246, 247

### Audio Listener component 248

### Audio Source component

- about 248
- settings 248-251

## B

### background music

- adding, to Monkey Ball game 251, 252

### bananas, Monkey Ball game

- adding 193-195
- collecting, with touch 196-199

### Blender

- about 65
- URL 65

### blob shadows 143-146

## C

### **camera effects**

- about 120
- distance fog 120-122
- skybox 120-122
- target indicator 122
- turbo boost 129-131

### **characters, 2D games**

- about 210
- ally, creating 216, 217
- enemy, creating 211-216

### **city, Tank Battle game**

- creating 86-88
  - Main Maps section 88, 89
  - Secondary Maps section 89-93
- cocos2d** 3
- controls, 2D games**
- about 217
  - camera, watching with 227-232
  - slingshot, attacking with 218-227
- cookies** 141, 142

## D

### **development environment**

- Android SDK, installing 13-15
  - JDK, installing 12
  - optional code editor 19
  - setting up 12
  - Unity 3D, installing 16-18
- device connection** 19-22
- distance fog** 120-122

## E

### **Editor log** 278, 279

## G

### **Graphical User Interface (GUI)** 33

## H

### **Hello World application**

- building 23-30

## I

### **import settings, audio** 246, 247

### **inheritance** 235

### **Inverse Kinematics (IK)** 103

## J

### **JDK**

- download link 12
- installing 12, 13

## K

### **kill volume, Monkey Ball game** 184

## L

### **lag**

- about 303
  - minimizing 303
  - tips, for minimizing 308
- license comparison, Unity**
- 100,000 dollar turnover 7
  - about 6
  - audio filter 6
  - crowd simulation 6
  - custom splash screen 8
  - dark skin 12
  - deferred rendering 10
  - fullscreen post-processing effects 10
  - fully-fledged streaming, with asset bundles 7

GPU profiling 11

GPU skinning 11

HDR 9

lightmapping, with global illumination and area lights 9

light probes 9

LOD support 6

Mecanim (IK Rigs) 7

Mecanim (sync layers and additional curves) 8

native code plugins' support 11

Navmesh 11

occlusion culling 10

pathfinding 6  
profiler 11  
real-time spot/point 8  
render-to-texture effects 10  
script access, to asset pipeline 11  
soft shadows 8  
static batching 10  
stencil buffer access 11  
tone mapping 9  
URL 6  
video playback 7  
video streaming 7  
**lightmaps** 136-140  
**lights**  
about 132  
adding 133-135  
Area Light 132  
cookies 141, 142  
Directional Light 132  
lightmaps 136-140  
Point Light 132  
Spotlight 132

## M

**materials, Tank Battle game**  
city, creating 86-88  
creating 86  
treads, moving 93, 94  
**Mecanim** 94  
**Monkey Ball game**  
audio effects, implementing 270  
bananas, adding 193-195  
bananas, collecting with touch 196-199  
bananas, exploding 270-275  
bananas, poking 253, 254  
camera, following with 179, 180  
complex environment, merging 190-193  
development environment,  
    setting up 174, 175  
gyroscope 176  
kill volume 184, 185  
losing 186-189  
monkey, adding 181-183  
special effects, adding 245

tilt, controlling with 176-179  
winning 186-189  
**MonoDevelop** 19

## N

**NavMesh**  
generating 148-154  
**NavMeshAgent component**  
about 154  
creating 154-157  
**NotePad++**  
URL 19

## O

**occludee** 150  
**occluder** 149  
**occlusion** 149, 304  
**occlusion culling**  
    adding, to Tank Battle game 304-307  
**optional code editor** 19

## P

**particle systems**  
about 255  
Collision module 261, 262  
Color by Speed module 260  
Color over Lifetime module 260  
dust trails, creating 265-270  
Emission module 257  
External Forces module 261  
Force over Lifetime module 259  
Initial module 256  
Limit Velocity over Lifetime module 259  
Renderer module 264  
Rotation by Speed module 260  
Rotation over Lifetime module 260  
settings 255, 256  
Shape module 257, 258  
Size by Speed module 260  
Size over Lifetime module 260  
Sub Emitters module 262  
Texture Sheet Animation module 263  
Velocity over Lifetime module 258  
**pathfinding** 148

**performance tracking**  
about 291  
editor statistics 292-294  
Profiler window 294, 295  
script performance, tracking 296-303

**physics simulation**  
about 205  
blocks, building 205-209  
physics materials 210  
Rigidbody component 205

**Player Settings window**  
audio 288  
Optimization group of settings 290  
Optimization window 290, 291  
Rendering group of settings 288  
Rendering window 288, 289

## R

**ray tracing to shooting**  
script, creating 114-116

## S

**skybox 120-122**

**state machine**  
about 101  
creating 101  
target, controlling 101-108  
target, scripting 109-111

## T

**Tank Battle game**  
chassis, controlling 78-80  
enemy, attacking 166-168  
enemy tanks, spawning 168-171  
materials, creating 86  
meshes, importing 65, 66  
occlusion culling, adding to 304-307  
player, attacking 161-166  
player, chasing 157-160  
player's location, revealing 157, 158  
ray tracing to shooting 114  
repeat button, creating 76, 77

score, keeping track 75  
scripts, implementing 82-85  
setting up 64, 65  
tank import settings 66  
tank, setting up 72-74  
turret, controlling 80-82

**tank import settings**  
about 66, 67  
Materials section 70  
Meshes section 68, 69  
Normals & Tangents section 69, 70  
Revert and Apply buttons 71, 72

**target indicator**  
about 122  
controlling 124-126  
pointer, creating 122-124  
second camera, working 126-129

**Tic-tac-toe game**  
building, to device 59  
code, adding 54-58  
controlling 40-45  
creating 34  
devices, rotating 48-51  
elements, setting up 52-54  
fonts, working with 45-47  
game board, creating 34-39  
game squares 39, 40  
implementing 58  
menus 51  
playing, on Android device 59, 60  
user preferences 35  
victory 51

**turbo boost 129-131**

## U

**Unity**  
about 1  
animations 94  
features 2, 3  
URL 16  
working, with Android 5

**Unity 3D**  
installing 16-18

**Unity Asset Store**

about 22  
reference 22

**Unity Basic**

URL 5

**Unity Pro**

URL 5

**Unity Remote**

about 5, 22  
reference 22

**Unreal Engine** 3

**V**

**Visual Studio** 19





## Thank you for buying Learning Unity Android Game Development

### About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at [www.packtpub.com](http://www.packtpub.com).

### Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

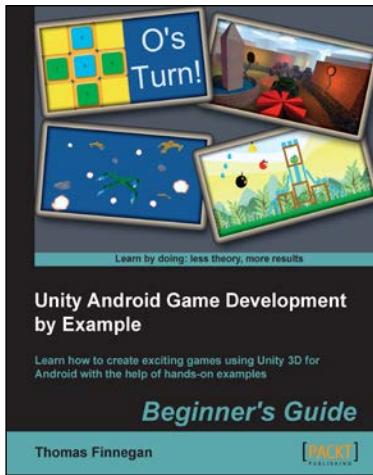


## Learning Unity 2D Game Development by Example

ISBN: 978-1-78355-904-6      Paperback: 266 pages

Create your own line of successful 2D games with Unity!

1. Dive into 2D game development with no previous experience.
2. Learn how to use the new Unity 2D toolset.
3. Create and deploy your very own 2D game with confidence.



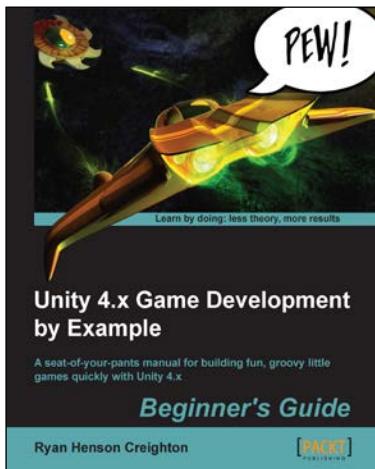
## Unity Android Game Development by Example Beginner's Guide

ISBN: 978-1-84969-201-4      Paperback: 320 pages

Learn how to create exciting games using Unity 3D for Android with the help of hands-on examples

1. Enter the increasingly popular mobile market and create games using Unity 3D and Android.
2. Learn optimization techniques for efficient mobile games.
3. Clear, step-by-step instructions for creating a complete mobile game experience.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

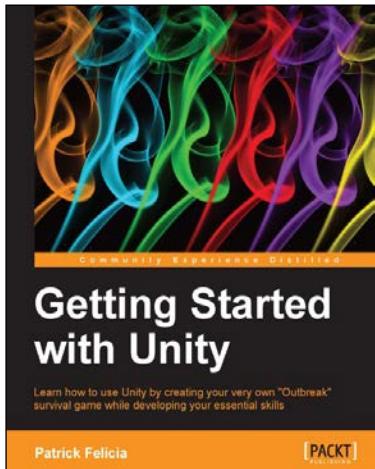


## Unity 4.x Game Development by Example: Beginner's Guide

ISBN: 978-1-84969-526-8      Paperback: 572 pages

A seat-of-your-pants manual for building fun, groovy little games quickly with Unity 4.x

1. Learn the basics of the Unity 3D game engine by building five small, functional game projects.
2. Explore simplification and iteration techniques that will make you more successful as a game developer.
3. Take Unity for a spin with a refreshingly humorous approach to technical manuals.



## Getting Started with Unity

ISBN: 978-1-84969-584-8      Paperback: 170 pages

Learn how to use Unity by creating your very own "Outbreak" survival game while developing your essential skills

1. Use basic AI techniques to bring your game to life.
2. Learn how to use Mecanim; create states and manage them through scripting.
3. Use scripting to manage the graphical interface, collisions, animations, persistent data, or transitions between scenes.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles