

Community Experience Distilled

Building an FPS Game with Unity

Create a high-quality first person shooter game using the Unity game engine and the popular UFPS and Probuilder frameworks

Foreword by Calle Lundgren, Creator of UFPS

John P. Doran

[PACKT] PUBLISHING

Building an FPS Game with Unity

Create a high-quality first person shooter game using the Unity game engine and the popular UFPS and Probuilder frameworks

John P. Doran

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Building an FPS Game with Unity

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: October 2015

Production reference: 1271015

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78217-480-6

www.packtpub.com

Cover image by John P. Doran

Credits

Author
John P. Doran

Reviewers
Schuyler L. Acosta
Alex Madsen
Tony Pai
Chittersu Raghu Vamsi
Nevin Vu

Commissioning Editor
Edward Bowkett

Acquisition Editor
Vivek Anantharaman

Content Development Editor
Riddhi Tuljapurkar

Technical Editor
Shivani Kiran Mistry

Copy Editor
Akshata Lobo

Project Coordinator
Sanchita Mandal

Proofreader
Safis Editiing

Indexer
Hemangini Bari

Graphics
Disha Haria

Production Coordinator
Shantanu N. Zagade

Cover Work
Shantanu N. Zagade

Foreword

My first love affair with an FPS game was in 1995. I was an intern at a local radio station and someone had installed the shareware version of *DOOM* on the CD database computer. A fast, sprawling ballet of violence unfolded before my eyes. This was what a computer game was supposed to be like! Running around 3D dungeons, guns blazing, blood splattering, and demons growling and scaring the bejeezus out of me before they were being blown to bits. We didn't get a whole lot of work done that summer. And my fate was sealed; I was going to be a game developer.

It took me a good 4 years of modding, scripting, and 3D modeling to land a job at a small startup game studio. Thrilled, I found myself working on a real multiplayer FPS game as a part of a team of 15 people. Coming from a hobbyist, do-it-all-by-yourself mindset, I remember my jaw hitting the floor as the project manager told me some numbers over lunch. He estimated that for one single person to create the whole game, it would take 65 years. 65 years!

In the following months, Moore's law and a relentless push for realism saw budgets and team sizes skyrocket. Soon it wasn't uncommon for an FPS project to have a head count in the hundreds. The would-be-indie developer inside me mourned these figures as I pondered my secret indie ambitions and sensed those already impossible 65 years stretching into 650.

Of course, back then, everybody was building their own game engine from scratch. Game programming books would explain in great detail how to construct your code from the bare metal up, going into hardware specifics, the basics of rasterizing polygons, brutal 3D math, and communicating with different brands of audio cards. You could license a game engine, but it would set you back hundreds of thousands of dollars. Besides, game coders loved to do everything from scratch back then (and as a result rarely got around to finishing their games).

Then, something happened. During the early 2000s, affordable and free engines such as Torque, Auran Jet, Crystal Space, and Ogre started popping up. Around the same time, the idea of "gap games" revitalized the indie movement. They were of real high quality, but were limited in scope; not your multimillion dollar production, but no scrawny "bedroom programmer" games either. They were fantastic looking games that could realistically be created by a small team with a good off-the-shelf engine in a reasonable amount of time. The dream was revived.

The Unity engine was first built for the Mac game *GooBall*. As the story goes, the team realized that, in the end, their game didn't show as much promise as their game engine, and Unity3d was announced at the 2005 Apple Worldwide Developers Conference. Initially, what Unity had going for it was the ability to run high-definition 3D games on a web browser. When the iOS and Android support was added, it became the engine of choice for mobile game development, and everything just exploded. Today, Unity3d is a free, extremely popular, powerful, and multiplatform AAA game engine. It has triggered an incredible surge in indie game development and spawned untold indie game successes. The addition of the Unity Asset Store allows thousands of pros and hobbyists to share and trade high-quality scripting, art, sound, design, and services.

UFPS started out as my side project dubbed by Ultimate FPS Camera. I released it as a small script pack in the Asset Store just to see what would happen. The response was overwhelming. Three years later, the system has grown into a full blown FPS solution. My team has assisted many hundreds of indies in pursuing their game ideas. We've seen many awesome and original games take shape; some released to critical acclaim. I've also had the privilege of working with the authors of several amazing Unity assets, including Gabriel and Karl, the developers of ProBuilder, two incredibly dedicated and talented guys who have put innumerable hours of hard work into their tool suite (so you won't have to). It's with a sense of joy and excitement that I learned of this book being written and featuring ProBuilder along with UFPS.

In this book, John has summarized not only how to take advantage of the awesome power of Unity and Asset Store. In a casual and direct way, he explains how to arrive at a small, complete FPS in the shortest amount of steps possible. He doesn't go into the nitty gritty details of programming camera systems, level editors, or a combat AI from scratch. Instead, he helps you free up time for the core activities that make your game fun with creative game- and level-design. If you're prototyping a game or just starting out as a game developer, the power available to you through this book, Unity, and its Asset Store would have been unthinkable just a few years ago.

Good luck with your dream game!

Calle Lundgren
Creator of UFPS

About the Author

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

He previously worked at LucasArts on *Star Wars 1313* as a game design intern. He later graduated from DigiPen Institute of Technology in Redmond, WA, with a bachelor of science in game design.

John is currently a designer in DigiPen's research and development branch in Singapore. He is also the lead instructor of the DigiPen-Ubisoft Campus Game Programming Program, instructing graduate-level students in an intensive, advanced-level game programming curriculum. In addition to this, he also tutors and assists students on various subjects while giving lectures on C#, C++, Unreal, Unity, game design, and more.

In addition to this title, he is the author of *Unreal Engine Game Development Cookbook*, *Unity Game Development Blueprints*, *Getting Started with UDK*, *UDK Game Development*, and *Mastering UDK Game Development*, and also the co-author of *UDK iOS Game Development Beginner's Guide*, all by Packt Publishing. More information on him can be found on his website (`http://johnpdoran.com/`).

Acknowledgment

Firstly, I would like to thank my family for being patient with me while I took yet another challenge that reduced the amount of time I could spend with them. Especially Hien, my wife, who has taken a big part of that sacrifice, and also Chris, my brother, who encourages me in his particular way.

This book also couldn't have been written at all without the amazing support from the game development community. Most of all, I'd like to thank Calle Lundgren at VisionPunk for being so supportive of the project and providing his very valuable insights for it. I'd also want to thank Gabriel Williams from ProCore3D and Nick Canafax at Rival Theory for their assistance.

In addition, the reason the projects in the book look so good is due to the artistic talents of the guys at *GameTextures.com*, who provided some amazing textures to work with as well as Paul Blackham for letting me use his awesome gun model.

On this same note, I also want to thank Samir Abou Samra and Elie Hosry for their support and encouragement while working on this book, as well as the rest of the DigiPen Singapore staff.

Thanks again to the wonderful guys at Packt, who were a pleasure to work with, including Vivek Anantharaman, who approached me about the project in the first place, and Riddhi Tuljapurkar, who worked with me as the book was being written.

Last, but not least, I want to thank my parents Sandra and Joseph Doran, who took me seriously when I told them that I wanted to make games for a living.

About the Reviewers

Schuyler L. Acosta is a graduate from the Art Institute of California, San Francisco, with a bachelor of science in game art and design. In his free time, he enjoys working on 2D/3D portfolio projects. This is his first book review. His other passion besides art is music, playing the piano and electric/acoustic guitar.

I would like to thank Pooja Mhapsekar, Sanchita Mandal, and Riddhi Tuljipurkar for their feedback and support in reviewing this book. I'd also like to thank my parents, Bonnie and Edwin Acosta, and my sister, Angel Acosta, for their continued support and encouragement.

Alex Madsen is a Gameplay Programmer at Pure Arts Ltd., Shanghai. He is a tech enthusiast that loves finagling with anything that is based on computers: Arduino, Linux, and the like. He started his career programming Excel spreadsheets for a rural Alberta tree farm, and got an opportunity to work in Shanghai.

He works at Pure Arts Ltd., Shanghai.

I would love to thank my parents and beloved friends, and Stark, the big dumb dog.

Tony Pai is an indie game developer from Taiwan. He learned Unity for about a year and a half and is now working with his friends to make games. They released two games to date, *The Guys* and *Elpis*. This is his first book review.

I want to thank Sanchita Madal, project coordinator, and everyone at Packt for their help in producing this book.

Chittersu Raghu Vamsi is a professional programmer, game developer, analyst, and designer. He has a computer science background, which he pursued at one of the most reputed colleges in India, BITS Pilani. He has an experience of over 3 years in the field of game development. He has done projects on machine learning and artificial intelligence. His other interests include reading novels, making short films, and writing articles.

I would like to thank my family and friends for their encouragement and support. I also thank Packt Publishing for providing me such a great opportunity.

Nevin Vu graduated with a diploma in game design. While he was out to complete his diploma, he gained valuable experience by interning at Panasonic Avionics, Singapore. He is currently pursuing a degree in computer science to ensue his passion in programming. Upon graduation, he hopes to work in the video game industry to contribute to the gaming development community with his knowledge in programming. He has developed various games from platforms to FPS games, which are available on his website, `http://www.n3evin.com`.

I would like to thank my parents for giving me this opportunity and being supportive of my interest in programming. Moreover, I am thankful to the rest of my family and friends for encouraging me to pursue my passion.

---

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 **v**

Chapter 1: Getting Started on an FPS **1**

Prerequisites **2**
Project creation **4**
Getting started with the Asset Store **6**
Installing UFPS 9
Installing Prototype **12**
File organization **15**
Customizing Unity's layout **16**
Summary **17**

Chapter 2: Building Custom Weapons **19**

Prerequisites **20**
Setting up a testbed **20**
Getting models/sounds for weapons **22**
Building our weapon – the mesh **24**
Creating a UnitBank **29**
Creating the weapon **30**
Customizing our weapon's properties **34**
Summary **39**

Chapter 3: Prototyping Levels with Prototype **41**

Prerequisites **42**
Level design 101 – planning 42
Creating the architectural overview 44
3D modeling software 44
Constructing geometry with brushes 44
Modular tilesets 45
Creating geometry **46**
Building a doorway **60**

Duplicating rooms / creating a hallway **68**
Preventing falls - collision **73**
Adding stairways **76**
Coloring your world **84**
Summary **86**
Chapter 4: Creating Exterior Environments **87**
Prerequisites **87**
Introduction to Terrain **88**
Height maps 88
Hand sculpting 90
Creating the Terrain **91**
Adding color to our Terrain – textures **97**
Adding water **106**
Adding trees **108**
Adding details – grass **112**
Building the atmosphere – Skyboxes and Fog **114**
Summary **120**
Chapter 5: Building Encounters **121**
Prerequisites **121**
Adding a simple turret enemy **122**
Integrating an AI system – RAIN **129**
Integrating an AI system – Shooter AI **153**
Spawning enemies with the help of a trigger **164**
Spawning multiple enemies at once **179**
Cleaning up dead AI **184**
Placing healthpacks/ammo **186**
Summary **188**
Chapter 6: Breathing Life into Levels **189**
Prerequisites **189**
Building an explosive barrel **190**
Using triggers for doors **198**
Creating an elevator **208**
Summary **214**
Chapter 7: Adding Polish with ProBuilder **215**
Prerequisites **215**
Upgrading from Prototype to ProBuilder **216**
Creating material **223**
Working with ProBuilder – placing materials **227**
Meshing your levels **249**
Summary **253**

Chapter 8: Creating a Custom GUI **255**
Prerequisites **255**
Creating a main menu: part 1 – adding text **256**
Creating a main menu: part 2 – adding buttons **261**
Creating a main menu: part 3 – button functionality **265**
Replacing the default UFPS HUD **270**
Summary **278**
Chapter 9: Finalizing Our Project **279**
Prerequisites **279**
Building the game in Unity **279**
Building an installer for Windows **283**
Building an installer for Windows **287**
Summary **296**
Index **297**

Preface

Unity, available in free and pro versions, is one of the most popular third-party game engines available. It is a cross-platform game engine, making it easy to write your game once and then port it to PCs, consoles, and even the Web, making it a great choice for both indie and AAA developers.

Building an FPS Game with Unity takes readers on an exploration of how to use Unity to create a 3D first-person shooter (FPS) title. Over the course of the book, you will learn how to work with Unity's own tools while also leveraging the powerful UFPS framework by VisionPunk and Prototype/ProBuilder 2.0 by ProCore3D. In addition, readers will learn how to create AI characters using both RAIN and Shooter AI.

After setting up the computer, you will start by learning how to create custom weapons, prototype levels, create exterior and interior environments, and breathe life into your levels. You will then polish the levels. Finally, you will create a custom GUI and menus for your title to create a complete package.

What this book covers

Chapter 1, *Getting Started on an FPS*, will give readers a brief overview, from a beginner's point of view, of the exciting world of Unity development for the creation of a first-person shooter (FPS) title. We will start off by creating a project and then set up our project by installing UFPS and Prototype. We'll also see how we can customize Unity's layout and organize our files effectively.

Chapter 2, *Building Custom Weapons*, will talk about one of the most important things in the FPS game: the weapons. In this chapter, we will create a testbed to work on and learn how to build a weapon in UFPS from a model while also learning about UFPS features such as UnitBanks and various weapon properties to give your weapons a feel "just right" for you.

Chapter 3, *Prototyping Levels with Prototype*, will help us take on the role of a level designer, who has been tasked to create a level prototype to prove that our gameplay is solid. We will make use of the free tool Prototype to help in this endeavor. In addition, you will also learn some beginning level designing.

Chapter 4, *Creating Exterior Environments*, explores the various ways to add more organic-feeling areas to your levels, making use of Unity's different terrain tools while also adding water, trees, grass, custom skyboxes, and fog to create a complete environment that can be used in your project.

Chapter 5, *Building Encounters*, will show how to create various types of encounters that players may experience to create effective gameplay scenarios starting with a simple turret, then creating a melee enemy using the free RAIN AI Engine as well as a ranged enemy using Shooter AI. You will then learn how to place enemies as well as spawn them into our scene. Lastly, you will learn how to place ammo and healthpacks in the level to guide players through the level.

Chapter 6, *Breathing Life into Levels*, will explore some of the ways we can breathe life into our levels with moving objects and more things that the player can interact with, such as exploding barrels, moving doors, and elevators.

Chapter 7, *Adding Polish with ProBuilder*, will take all of the pieces we created in the previous chapters and put them together to create a finished level. You will learn how to upgrade Prototype levels to using ProBuilder, gaining the ability to add materials to wall faces and create custom UVs. You will also learn how to create materials and how to mesh your levels to create a complete environment.

Chapter 8, *Creating a Custom GUI*, will demonstrate how to use Unity's new GUI system to create a custom GUI to replace the one given to us by UFPS to help our project stand out.

Chapter 9, *Finalizing Our Project*, will focus on exporting our game from Unity and then creating an Installer so that we can give it to all of our friends, family, and prospective customers.

What you need for this book

Throughout this book, we will be working within the Unity 3D game engine that you can download from `http://unity3d.com/unity/download/`. The projects were created using version 5.0.1, but the project should work with minimal changes.

For simplicity's sake, we will assume that you are working on a Windows-powered computer. Though Unity allows you to code in either C# or UnityScript, for this book, we will be using C#.

We will also be using assets that are available from the Asset Store in Unity, most notably UFPS which costs money ($95 normally, but it also has sales where it can be gotten for cheaper). Aside from UFPS, all of the other topics will cover how to use a free asset to accomplish things while also discussing an easier to use asset which does cost. Readers will use ProCore3D's free Prototype tool for the creation of levels while also learning how you may upgrade your project to using ProBuilder in *Chapter 9*, *Finalizing Our Project*, for advanced functionality, but it also costs money. For enemies, you will also learn how to use the free RAIN toolkit for melee enemies and how to use Shooter AI for ranged enemies, which costs money.

Who this book is for

This book is for those who want to create an FPS game in Unity and gain knowledge on how to customize it to be their very own. If you are familiar with the basics of Unity, you will have an easier time, but it should make it possible for someone with no prior experience to learn Unity at an accelerated pace.

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: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
 void OnTriggerEnter(Collider other)
{
  //If the player touches the trigger, and if it hasn't
        //been triggered before
  if(other.tag == "Player" && hasTriggered == false)
  {
    // Spawn a new enemy using the properties from the
    // spawnPoint object
    GameObject newEnemy = Instantiate(enemy,
                                      spawnPoint.position,
                                      spawnPoint.rotation)
                            as GameObject;
    // We only want this to happen once.
    hasTriggered = true;
  }
}
```

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

```
 void OnTriggerEnter(Collider other)
{
  //If the player touches the trigger, and if it hasn't
         //been triggered before
  if(other.tag == "Player" && hasTriggered == false)
  {
    // Spawn a new enemy using the properties from the
    // spawnPoint object
    GameObject newEnemy = Instantiate(enemy,
                                      spawnPoint.position,
                                      spawnPoint.rotation)
                            as GameObject;

    //Tell enemy to go to the player's position
    newEnemy.GetComponent<NavMeshAgent>().SetDestination(other.
    transform.position);

    // We only want this to happen once.
    hasTriggered = true;
  }
}
```

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: "Once completed, select **Create project**."



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/Building_FPS_Games_with_Unity.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
Getting Started on an FPS

Welcome to *Building an FPS Game with Unity*! This chapter is dedicated to offer a brief overview, from a beginner's point of view, of the exciting world of Unity development for the creation of a **First Person Shooter (FPS)** title, leveraging the powerful **Ultimate First Person Shooter** (**UFPS**) framework by VisionPunk and Prototype/ProBuilder 2.0 by ProCore3D. But, before we get started, we first need to get all of the resources we'll need and set up our project for success.

Over the course of this book, we will be creating a 3D FPS game similar to the popular games in the market such as *Call of Duty: Black Ops III* and *Halo 5: Guardians*.

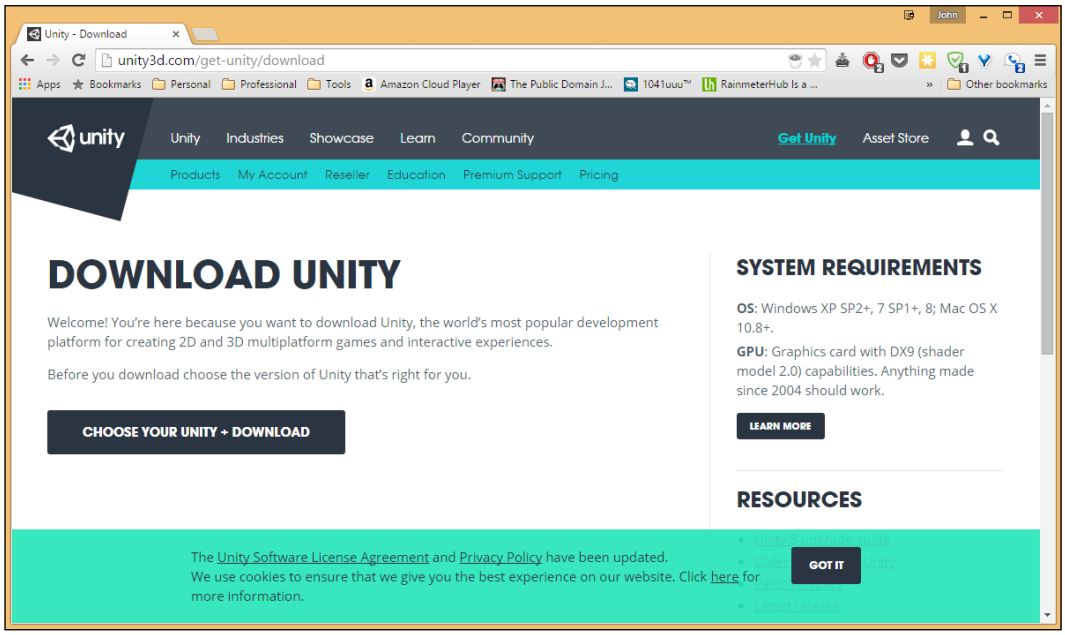
We will learn how we can create custom weapons of our own as well as how we can create interior and exterior environments. After creating our environments, we will populate them with different combat encounters for players to fight as well as include intractable objects such as exploding barrels. We'll then customize our user interface using Unity's new GUI system before we package our game and create an installer to get the game out into the world!

This project will be split into a number of tasks. It will be a simple step-by-step process from the beginning to the end. Here is the outline of our tasks:

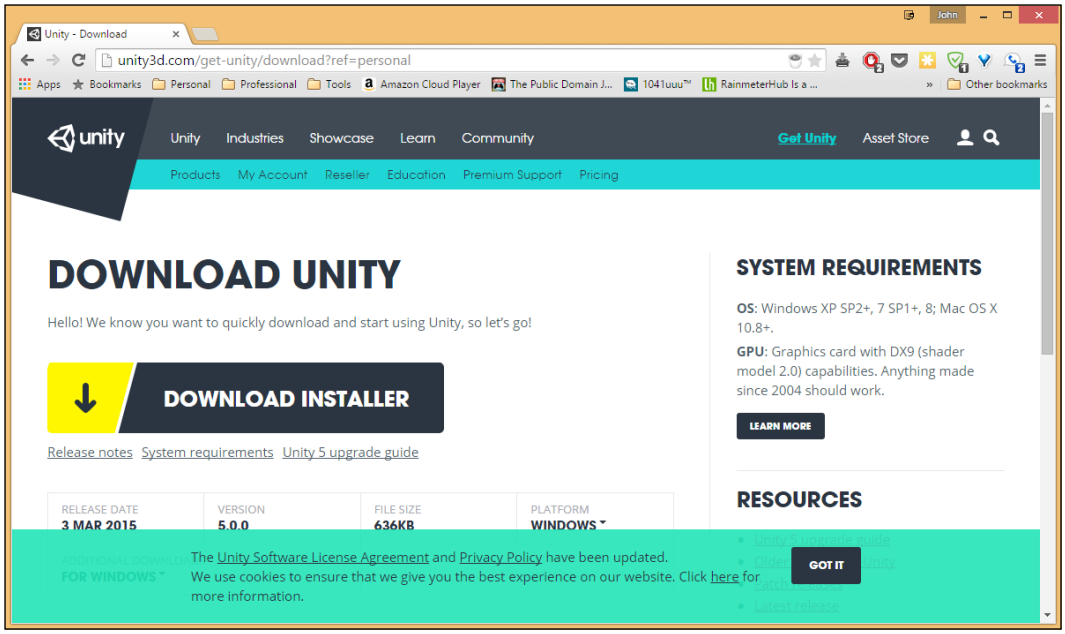
- Project creation
- Getting started with Unity's Asset Store
- Installing UFPS
- Installing Prototype
- File organization
- Customizing Unity's layout

Prerequisites

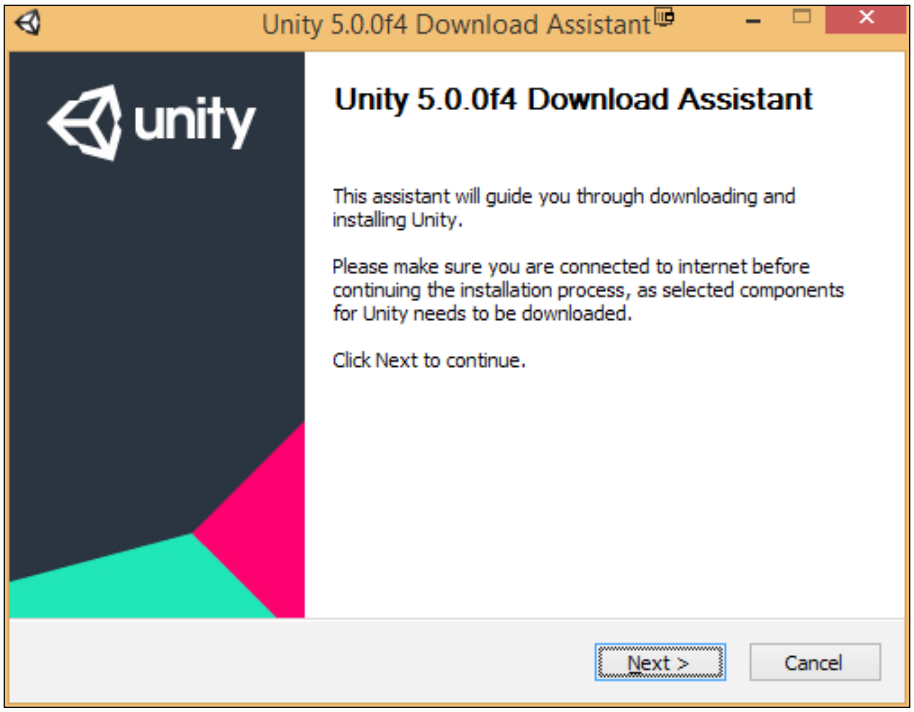
Before we start, we will need to get the latest Unity version. You can always download it from `http://unity3d.com/unity/download/`. At the time of writing this book, the page looks like this:



Once you get to this page, click on the **CHOOSE YOUR UNITY + DOWNLOAD** button (this page onward, I will be using the **PERSONAL EDITION** version). Then, click on the **DOWNLOAD INSTALLER** option.



The download assistant should be installed from your default downloads directory. Once it is installed, double-click on the file to open it.



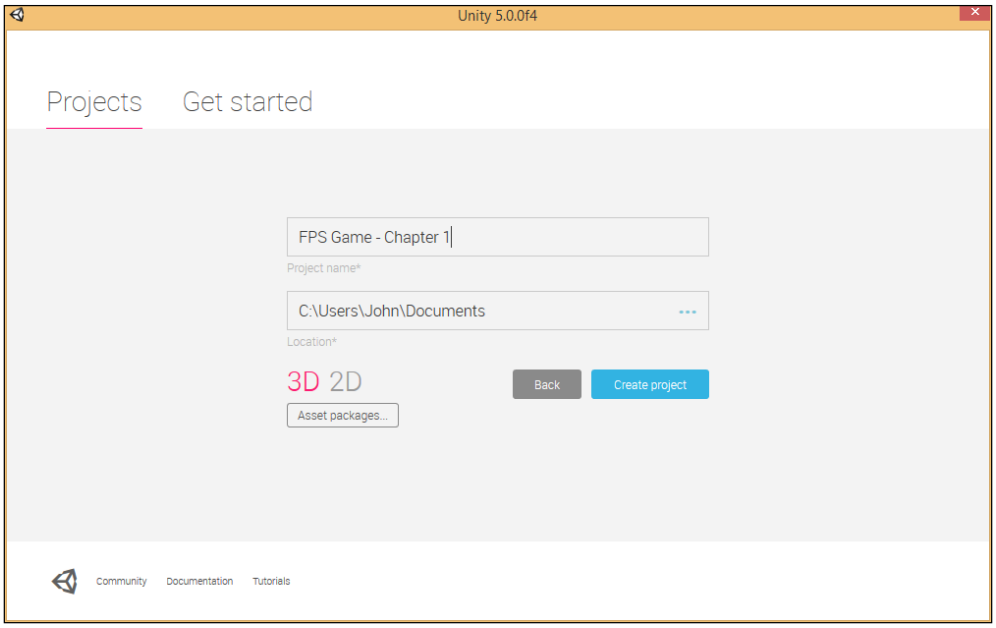
Once there, go through the installation using the default properties. At the time of writing, we are using **Unity 5.0.0f4**, but most things should work with minimal changes but be sure to check out the book's website to check for any errata.

Since it needs to download Unity, this process may take some time.

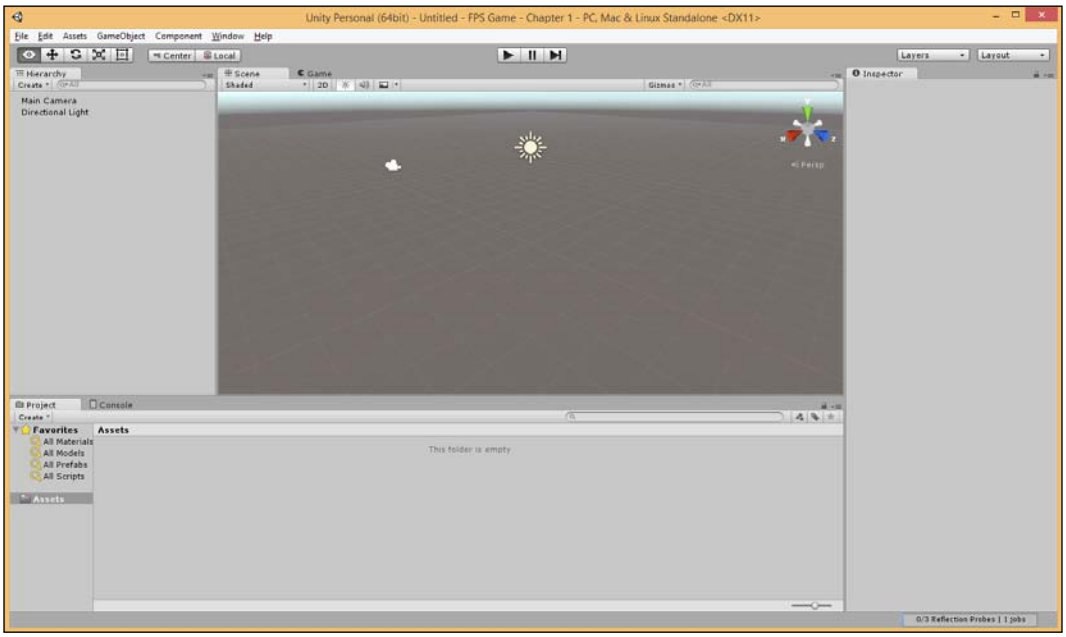
Project creation

At this point, I assume that you have freshly installed Unity and have started it up. Follow these steps to create a new project in Unity:

1. With Unity started, go to **New Project**. Enter a **Project Name** value such as `FPS Game - Chapter 1` as that's what we are making and the chapter we're making it for, or whatever you want to call your project. Select **Location** of your choice somewhere on your hard drive and ensure that you have your game set to `3D`. Once completed, select **Create project**. At this point, we do not need to import any packages as we'll be doing it manually.



2. Here on, if you see the **Welcome to Unity** pop up, feel free to close it as we won't be using it. At this point, you will be brought to the general Unity layout, which should look as follows:





I'm assuming you have some familiarity with Unity before you read this book. If you want more information on the interface, please visit `http://docs.unity3d.com/Documentation/Manual/LearningtheInterface.html`.

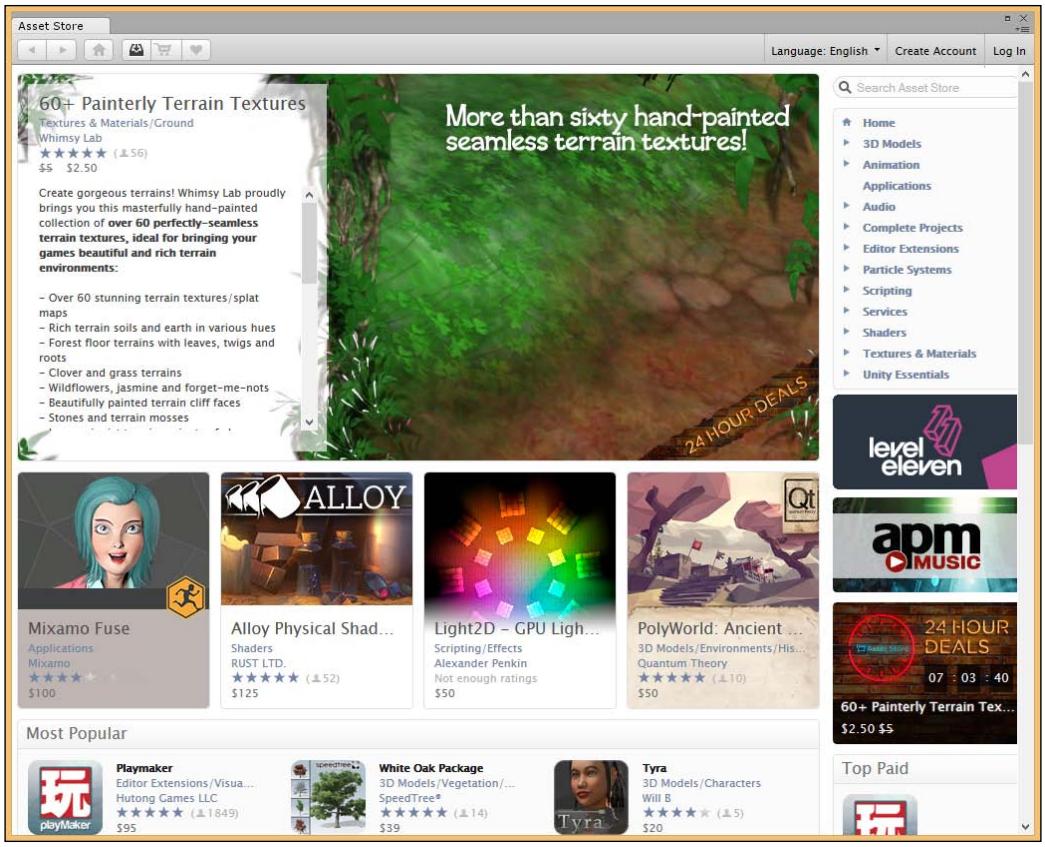
Getting started with the Asset Store

Since Unity 3, the Asset Store has been similar to Apple's/Android's app stores, except, instead of apps, you can buy prebuilt assets that can be imported directly into your project. We will be using this in our project; but, before we do so, we will need to have an account, that can be created using the following steps:

1. To open the **Asset Store** via Unity, we can go to **Window | Asset Store** from the top toolbar.

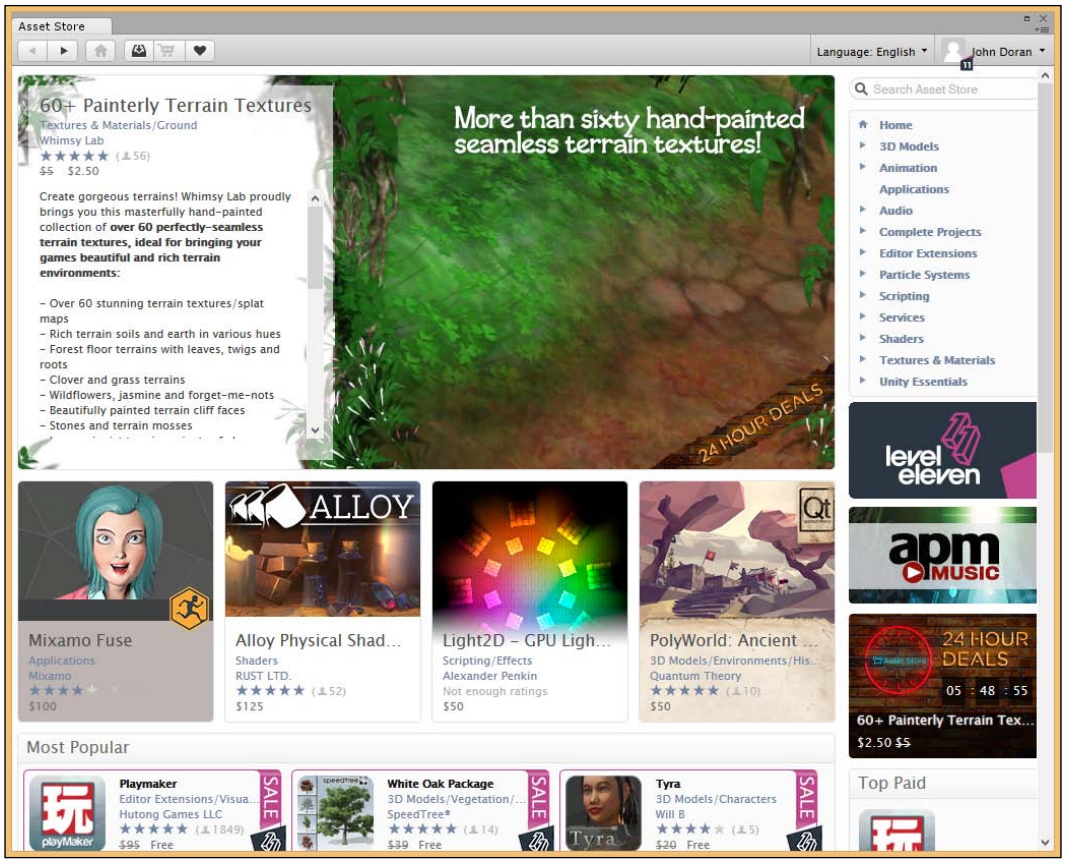


You may also open the panel by pressing *Ctrl + 9* (*command + 9* on Mac). If nothing happens, make sure Unity is focused on window by clicking on the program and trying again. If you prefer working outside Unity, you may also go to the **Asset Store** website at `https://www.assetstore.unity3d.com/`. However, you'll need to download assets from Unity via the **Download Manager**, which you can learn about from the link given later in this chapter.



2. Next, go to the top-right corner of the panel and click on the **Create Account** button. Fill out your information and click on **Create account** at the bottom of the screen.
3. You should receive an activation e-mail shortly after you submit the form. Open it and click on the activation link provided to verify your account.

4. Once this is done, go to **Log In** from the top-right corner of the panel and enter the information you put in when you created the account. If all goes well, the top-right corner of the window will change to reflect that you are logged in.



With this, we can download and purchase assets from the Asset Store!



Do not be concerned if you do not see **11** by your account, it just shows that I have a Unity Pro account.

For more information on the Asset Store and how to navigate using it, visit `http://docs.unity3d.com/Manual/AccessNavigation.html`.

Installing UFPS

Creating a game completely from scratch takes a considerable amount of time. Creating a first-person shooter relies on having a lot of knowledge as well as problem solving with things such as physics, mathematics, graphics, and programming. Now, books of each of these subjects could have been written based on it; but we are assuming that you want results in the fastest amount of time possible.

Rather than taking the tens of thousands of hours it would take to create it from scratch, we will leverage the very popular Unity add-on UFPS (Ultimate FPS) from VisionPunk. It will give us a solid foundation to create our own project with hundreds of parameters, which we can customize to create a project exactly the way we want it to be.

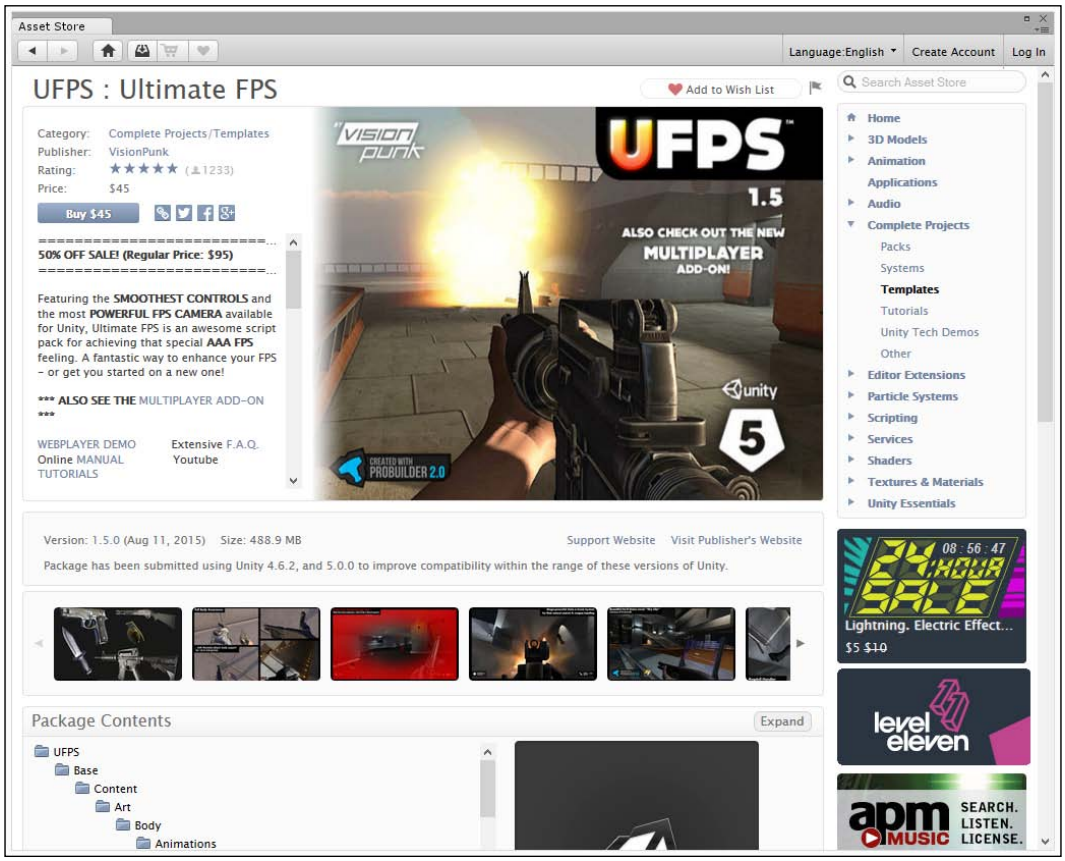
It's important to note that UFPS does cost money. But considering that a solid game programmer generally makes $50 or more an hour, the amount of time saved really makes it a worthwhile investment.

Now that we are logged into the Asset Store, let's purchase and install UFPS:

1. Now, from the search bar directly below the login information, search for `UFPS`. You should see an icon that looks similar to the following:

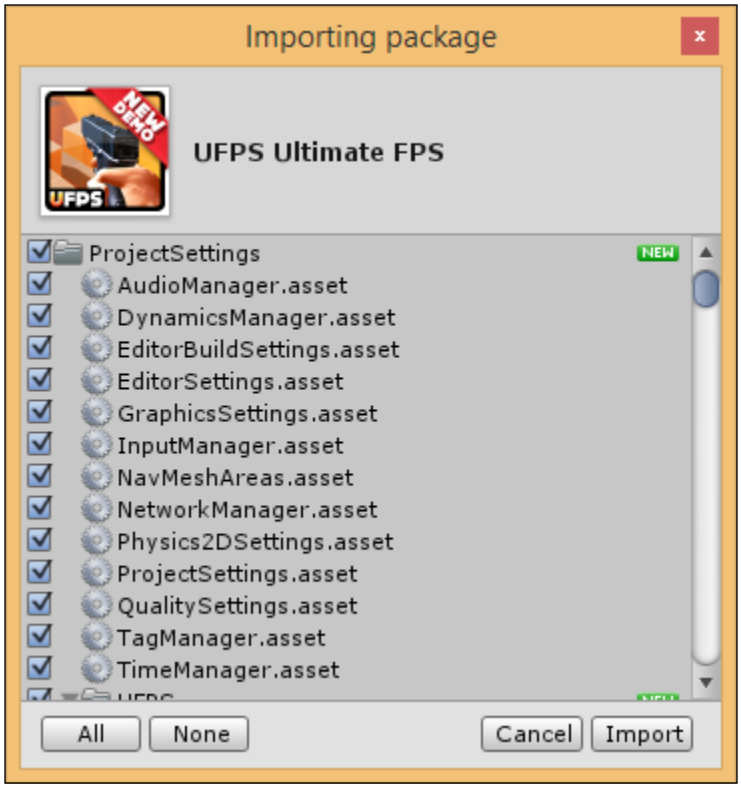


2. Click on the first option, which will bring you to the following page:



3. Now, click on the **Buy** button and purchase the asset. Once it is purchased, the screen will change the **Buy** button to a **Download** button. You can click on it to download the asset and bring it into your project.

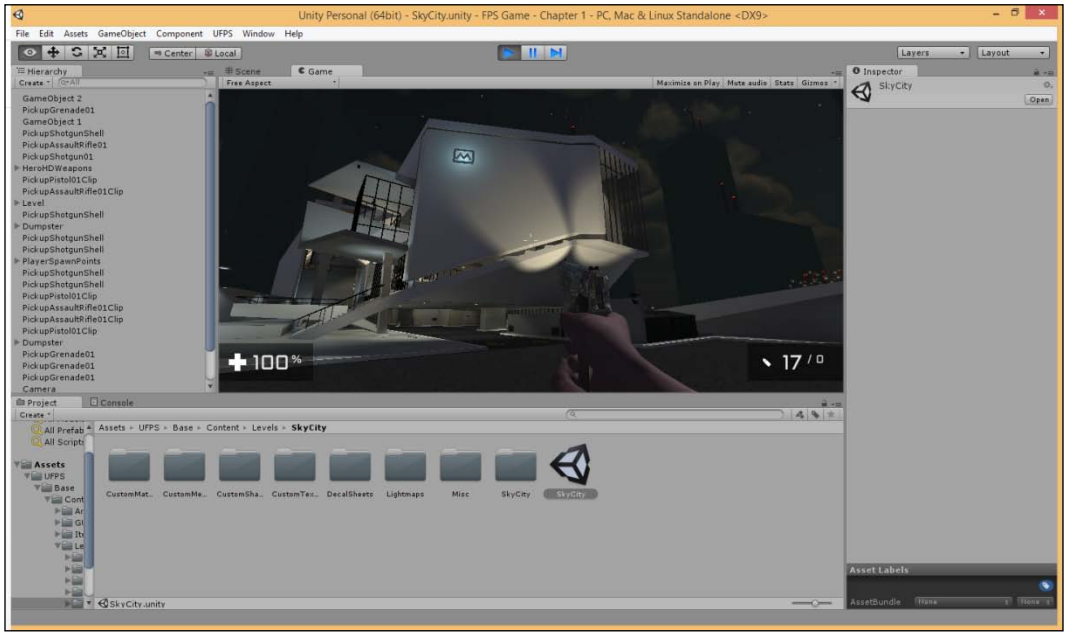
4. Once it is downloaded, an **Importing package** window will open, giving you the option to choose which parts of the package you wish to import. We want everything, so just click on the **Import** button.





If, for some reason, you get an error (the bottom bar, or Console, in the Unity interface) while importing that says something like `'Fatal error! getManagerFromContext:..."`. This is a bug in Unity due to the large file size UFPS has. Just **Quit** and continue with the import once again, when you restart Unity.

5. Once the project is imported, close the **Asset Store** window and go back into the Unity Editor. From there, go down to the **Project** tab in the bottom-left corner of the screen and double-click on the `UFPS` folder. Go to `Base\Content\Levels\SkyCity` and double-click on the **SkyCity** file to open an example level. Then, press the Play button at the top-center of the project to start the project!



With this, we know that UFPS is installed correctly!

Installing Prototype

In general, creating levels in Unity is a painful experience. You have to type in the values for each piece that you want to add or you have to create everything inside a modeling program and import it, which will require you to have 3D modeling skills.

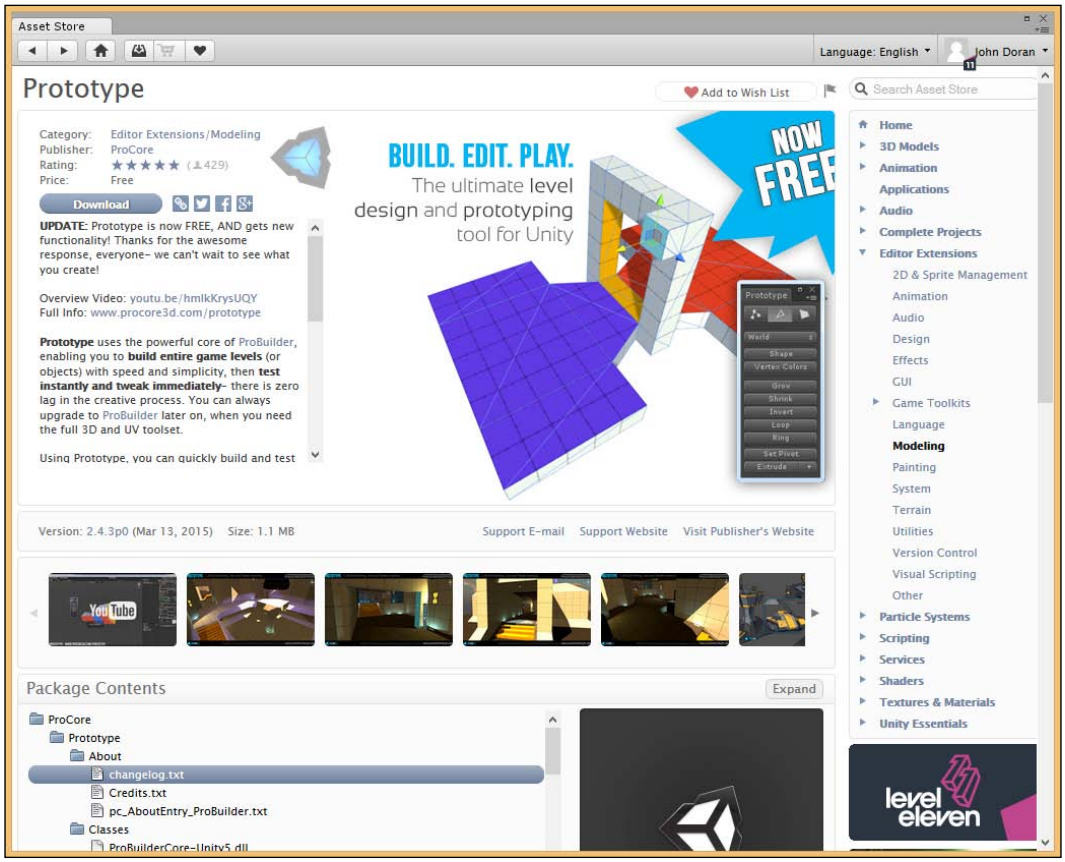
Other game engines have tools like **Binary Space Partitioning** (**BSP**) or **Constructive Solid Geometry** (**CSG**), which allow you to build geometry from scratch and apply materials to it to create areas for play. Filling in the gap that Unity has, ProCore3D has created a toolkit that allows for in-editor construction.

We will later on use ProBuilder to polish up our final product; but, in the meantime, we will use Prototype, their free version to build the basic structure and flow of our levels without wasting time, thus making things visually appealing until we polish it. Perform the following steps to install **Prototype**:

1. Open the **Asset Store** once again by going to the toolbar, searching for `Prototype`, and looking for the following icon:



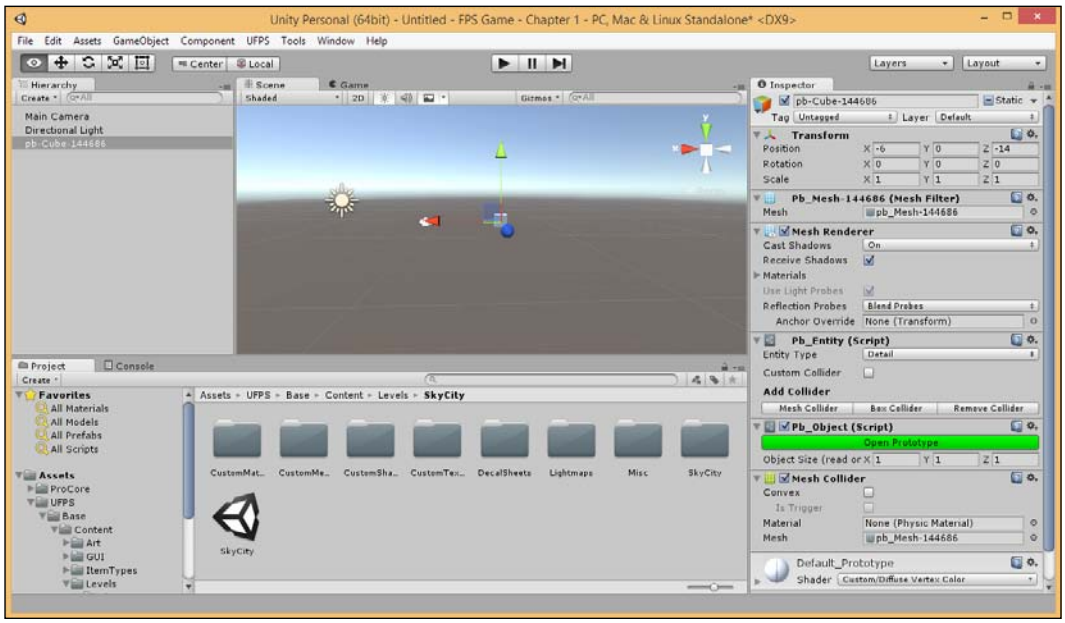
2. Click on the first option in the top-left corner and you should be brought to a page like the following one:





After the writing of this book, Prototype will in the near future be replaced by ProBuilder Basic which is still 100% free but has additional features. I've talked with the creators about this, and they've confirmed with me after reading the book that everything discussed here should still work with the new version. For more information on this, check out `http://www.protoolsforunity3d.com/probuilder/`.

3. Click on the **Download** button and wait for it to finish. Then, import the entire package by clicking on **Import**.
4. We will now test it out to see whether everything is imported correctly. Create a new scene inside Unity by going to **File | New Scene**. Once there, press *Ctrl + K* to create a new cube using Prototype.



With this, we know that it was installed correctly! It may look pretty simple now, but we will be diving even more into using these tools later on.



For more information on Prototype, check out `http://www.protoolsforunity3d.com/prototype/`.

File organization

Keeping your Unity project organized is incredibly important. As your project moves from a small prototype to a full game, more and more files will be introduced to your project. If you don't start organizing it from the beginning and keep planning to tidy it up later on, things may get quite out of hand as the deadlines keep coming.

Setting up a project structure at the start and sticking to it will save you countless minutes in the long run. It will only take a few seconds and is what we'll be doing now.

1. Go to the `Assets` folder from the **Project** tab in the bottom-left corner of the screen. Once there, click on the **Create** drop-down menu. Click on **Folder** and you'll notice that a new folder has been created inside your Assets folder.



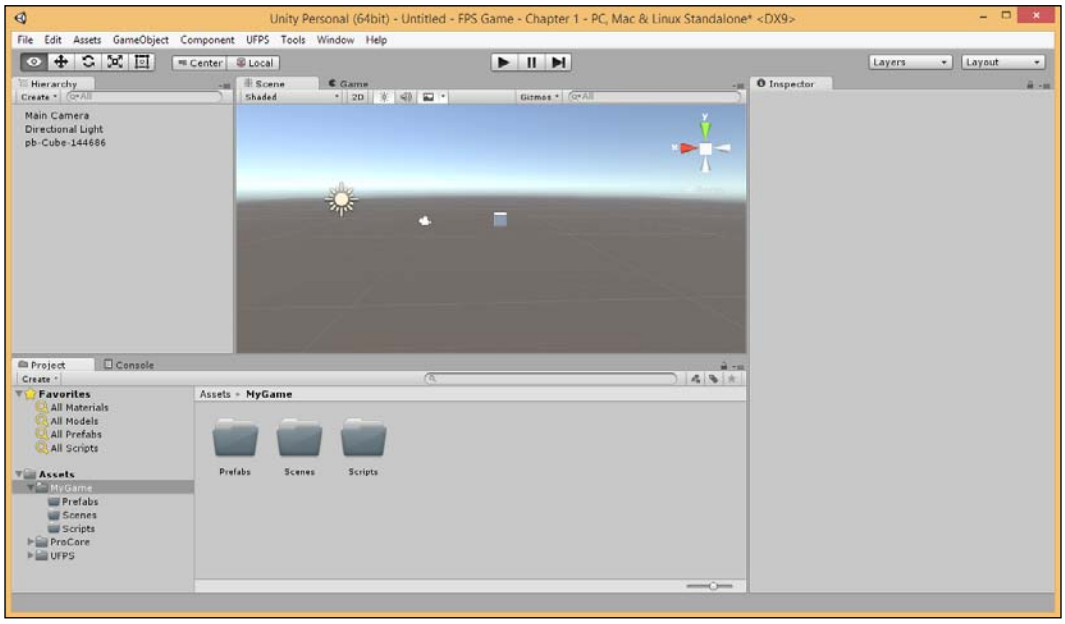
Similarly, you can right-click on the negative space within the **Project** window to create a folder.

2. After the folder is created, you can type in the name of your folder. Once it is named, press *Enter*. Let's now create a folder called `MyGame`. We also need to create folders for the following directories inside the `MyGame` folder:
    - `Prefabs`
    - `Scenes`
    - `Scripts`



If you happen to create a folder inside another folder, you can simply drag and drop it from the left toolbar. If you need to rename a folder, just simply click on it once and wait, you'll be able to edit it again. Alternatively, on the keyboard you can press *F2*.

Your project should now look like this:



Customizing Unity's layout

While working with Unity in this book, I will mostly be using its default layout. If, for some reason, your layout does not look like the earlier screenshot, you can reset it by going to the top-right corner of the window and selecting **Layout** | **Default**.

You can customize the layout by clicking and dragging any tab to wherever you want it to be. There are also additional options such as making the **Project** tab use a one-column layout by right-clicking on the tab and selecting **One Column Layout**. Sometimes, I like to split the center with the **Scene** tab on one side and the **Game** tab on the other, so I can see how things change from a different angle. For those of you with multiple monitors, you may use a monitor just for the game. It's all up to your preferences, but keep your changes in mind, as I'll assume that you'll be using **Default**.



For more information on customizing your Unity layout, check out `http://docs.unity3d.com/Manual/CustomizingYourWorkspace.html`.

Summary

Hopefully, you've enjoyed taking the first few steps toward becoming an FPS game developer with Unity! In this chapter, we learned how to create a project inside Unity 5. We then learned how to create an account and navigate the Unity Asset Store. After this, we learned about UFPS and Prototype and installed them. Finally, we touched upon file organization and customized Unity's layout.

In the next chapter, we will delve deeper into using Unity Editor and UFPS by creating our own custom weapons!

2
Building Custom Weapons

Now that we have everything installed, let's start working on one of the most important things that a first-person shooter game needs—the weapons. Often, it is the weapons of the games that become characters in their own right and tweaking them to be "just right" is something that many will try to do.

This project will be split into a number of tasks. It will be a simple step-by-step process from the beginning to the end. Here is the outline of our tasks:

- Importing new models/sound effects
- Replacing the default weapon
- Creating a new weapon template
- Customizing weapon position/behavior
- Recoil/Muzzle Flash
- Changing firerate
- Creating new ammo type
- Creating new ammo pickup
- Customize sound effects
- Determining behavior

Prerequisites

Before we start, we need to have a project created that already has UFPS installed. If you do not have it already, follow the steps described in the previous chapter.

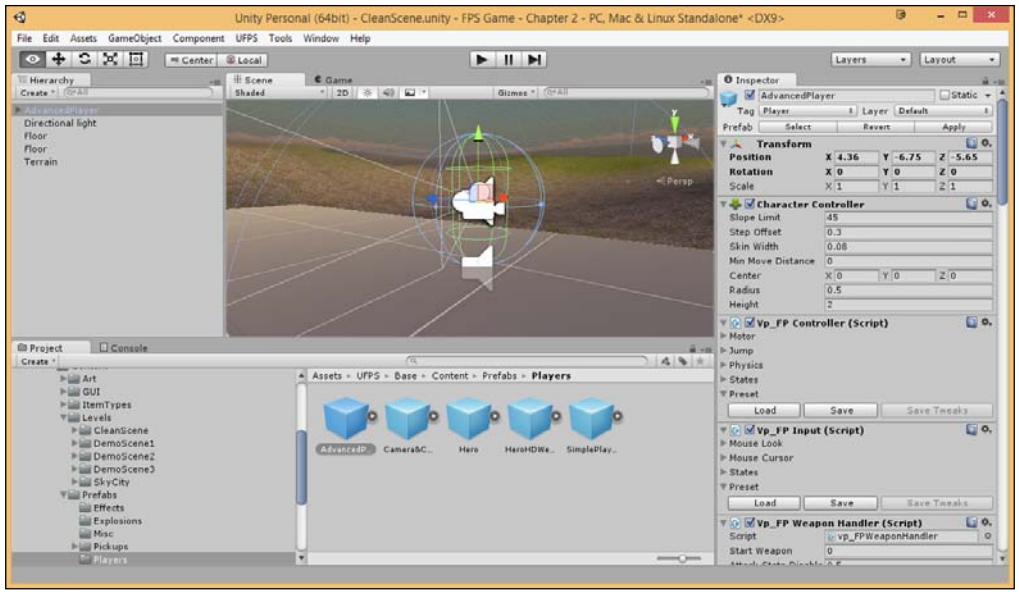
Setting up a testbed

Now, before we get started, let's create an area that we can use to test our new weapons.

1. Open up the project and a scene for us to work with. We can create a new level but, in this instance, UFPS comes with a nice scene for us to start with. From the **Project** tab, go to the `UFPS/Base/Content/Levels/CleanScene` folder and double-click on the **CleanScene** file.

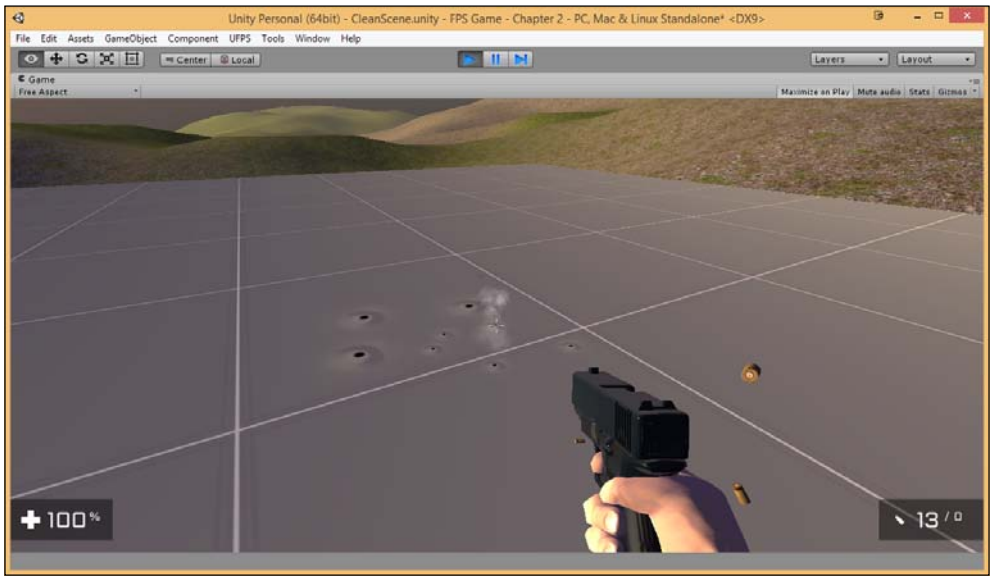
   This scene (or level) is a simple terrain with a basic first-person camera and controller intended for prototyping, which is perfect for us, except we want our player to be already set up. To do this, we will remove this simple camera and replace it with a built-in one.

2. From the **Hierarchy** tab, select the **Camera** object by clicking on it and delete it by pressing the *Delete* key.
3. Next, go to the `UFPS/Base/Content/Prefabs/Players` folder and drag and drop an **AdvancedPlayer** prefab into your game world. To see the object, double-click on it in **Hierarchy** (or press the *F* hotkey with it selected) and it'll zoom directly to its position.
4. You may notice that there are quite a few things attached to this player, but the most important thing to notice right now is the light green capsule shape, or half capsule shape, depending on how it's positioned. Move this object up on the *Y* axis until the entire capsule is above the ground plane by grabbing on to the green arrow and dragging it upward. (If you do not see the arrows, press the *W* hotkey to select the Translation tool. Alternatively, you can set the **Transform** component's **Position Y** property to `-7` using **Inspector**.**)**



This capsule is where the player will collide with the world, so it's important that we're up, on the top of the ground or else we may fall through the ground into the void below!

5. Now, click on the Play button. You should note that we can now control a player.



If you press keys from *1* to *4*, you should be able to shoot using various weapons.

Here are the various other controls you can use:

Key	**Action**
WASD	Move
C	Crouch
Space	Jump
Shift	Sprint
R	Reload
Right Mouse Button	Aim Down Sights / Zoom
ESC	Quit app (if offline standalone player)
Enter	Toggle menu

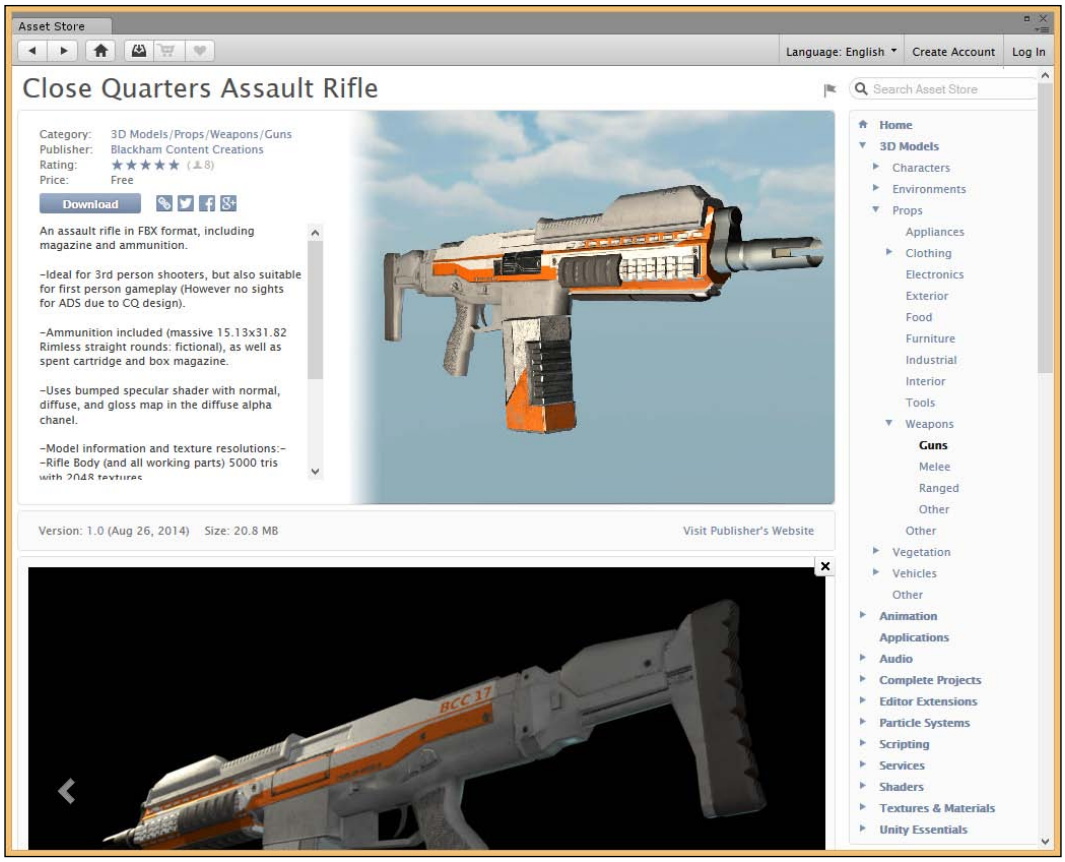
Getting models/sounds for weapons

Before we start creating our custom weapons, we will first require the models and sound effects that are needed for our project. Of course, you could model them yourselves, but it would require modeling knowledge (and the knowledge of using 3D modeling software such as Blender, Autodesk's Maya, or 3ds Studio Max), which is out of the scope of this book. For simplicity, we are going to use some free assets that you can obtain from the Asset Store.



If you would like to build a gun of your very own, some information on making the model set up for UFPS can be found at `http://www.visionpunk.com/hub/assets/ufps/manual/weapons`.

1. To open the **Asset Store** inside Unity, we can go to **Window** | **Asset Store** from the top toolbar.
2. If you aren't already logged in, go to **Log In** from the top-right corner of the panel and enter the information that you put in when you created the account.
3. From the right-hand side menu, go to **3D Models** | **Props** | **Weapons** | **Guns** and select a weapon that you'd like. I, personally, would choose **Close Quarters Assault Rifle** that you can see next:





The link to this content is `https://www.assetstore.unity3d.com/en/#!/content/21025`.

This rifle was created by Paul Blackham, whose portfolio can be seen at `https://www.behance.net/PaulBlackham`.

4. Next, download Asset and import it in your project.

Building our weapon – the mesh

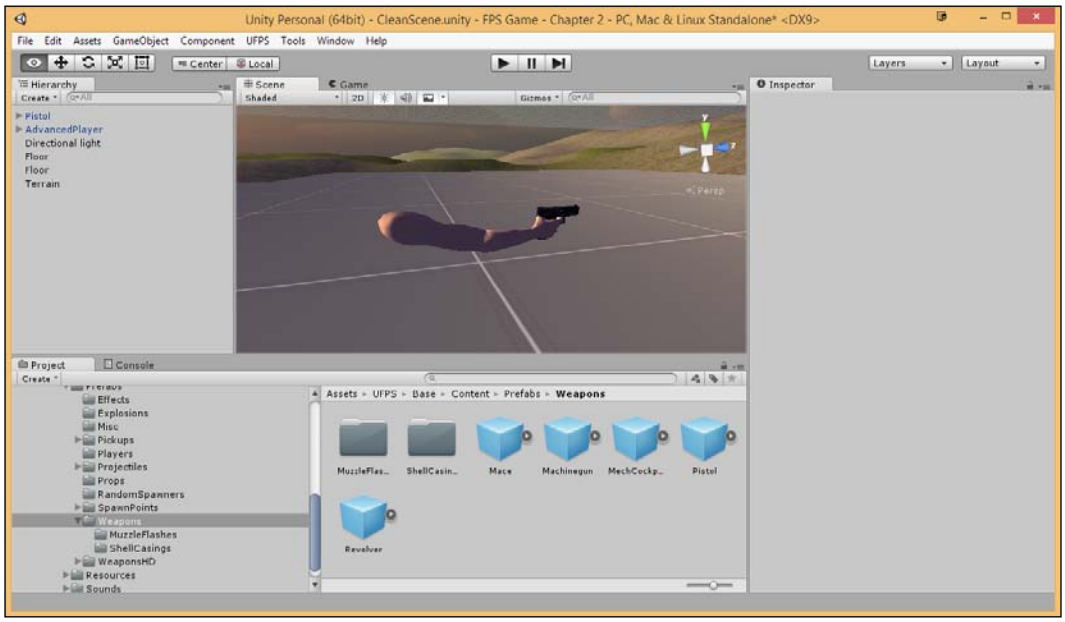
Now that we have the resources to create the weapon, let's first get the mesh ready with a hand to hold it like in a normal FPS game.

1. To start off, go to the `UFPS/Base/Content/Prefabs/Weapons` folder and drag the **Pistol** prefab out into the world. Move the object and camera until you get to the point where you can see it clearly.



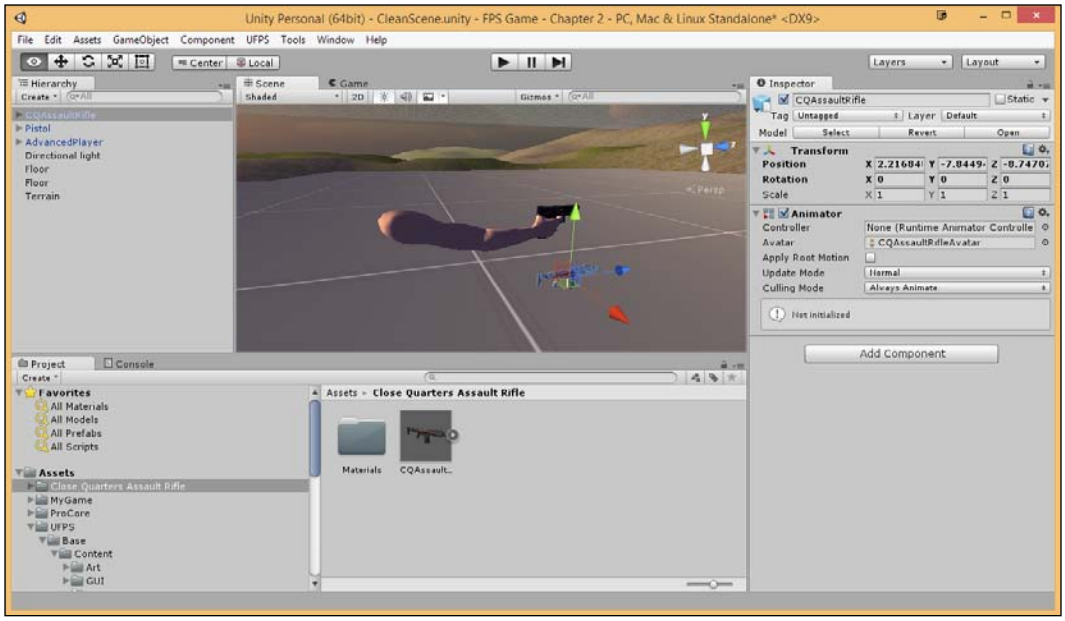
In case you need a refresher on the camera controls, here are the most commonly used ones:

- **Zoom In/Out**: *Alt* + the right mouse button/scroll the middle mouse button up/down
- **Rotate Around Selected Object**: *Alt* + the left mouse button
- **Panning**: *Alt* + the middle mouse button



Now, this is the fully created pistol we used previously that is already attached to a hand. We are going to use this pistol as a base to create our very own weapon by placing our new weapon where the old one is.

2. From the **Project** tab, drag out the **CQAssaultRifle** file that we downloaded earlier (`Assets/Close Quarters Assault Rifle/CQAssaultRifle`) and bring it into the world.



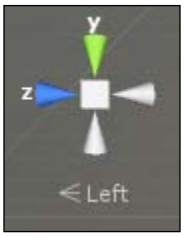
3. If the object placed in the scene is not scaled in comparison to the pistol object, look underneath the **Inspector** window. You will be able to increase/decrease the scale values (**X**, **Y**, and **Z**) equally. In the case of the **CQAssaultRife** mesh, it is very tiny, so we first need to scale it to an appropriate size. In this case, we'll go to the **Inspector** tab with the selected object and, from the **Transform** component, we will change the **Scale** value to `3,3,3`.



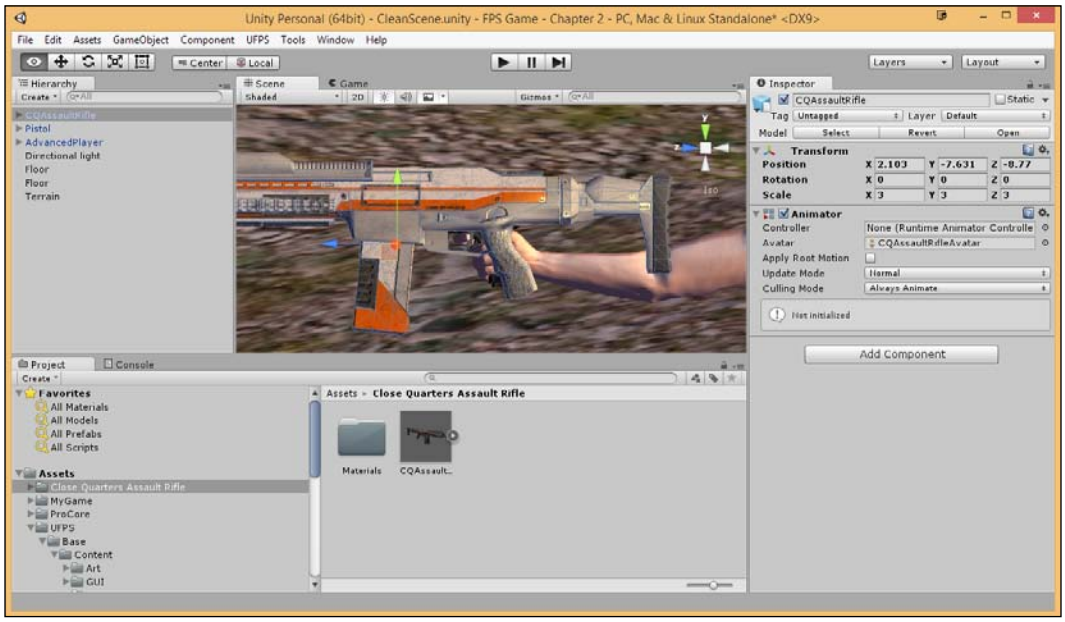
Alternatively, we can press the *R* button and drag it from the center box to scale it up and down as needed.

After this, we have to place the weapon in the player's hand in the exact same position as the trigger hand.

4. To help with this, use the camera gizmo at the top-right of the **Scene** tab. Rotate it to face the hand by clocking on the right or left edges and then clicking on the center cube to switch to toggle to an isometric camera. This will make it really easy to see exactly where it needs to go.



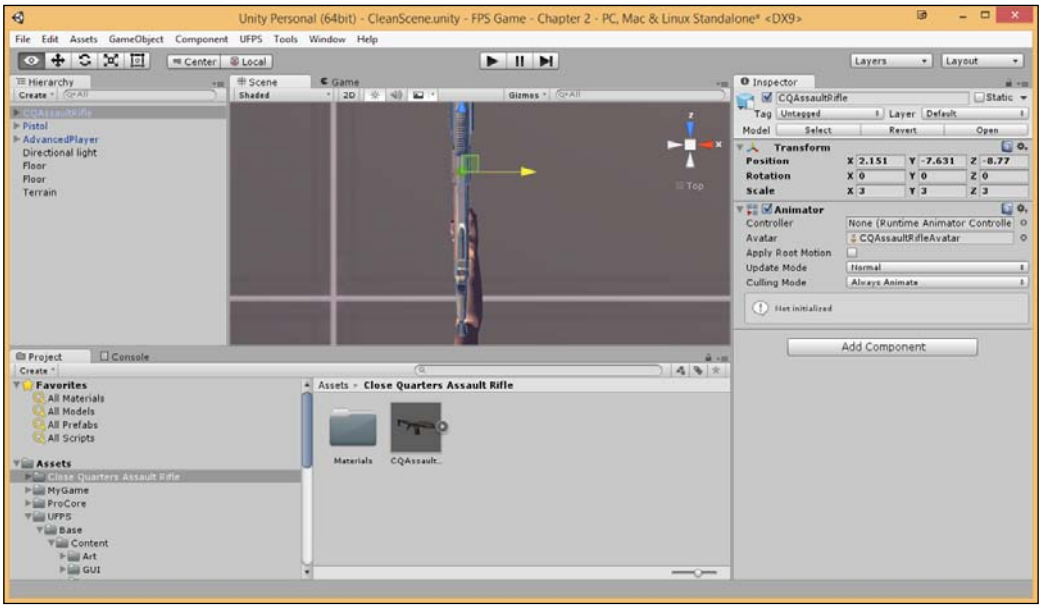
5. Once there, move the object until it fits your trigger finger, like the following:



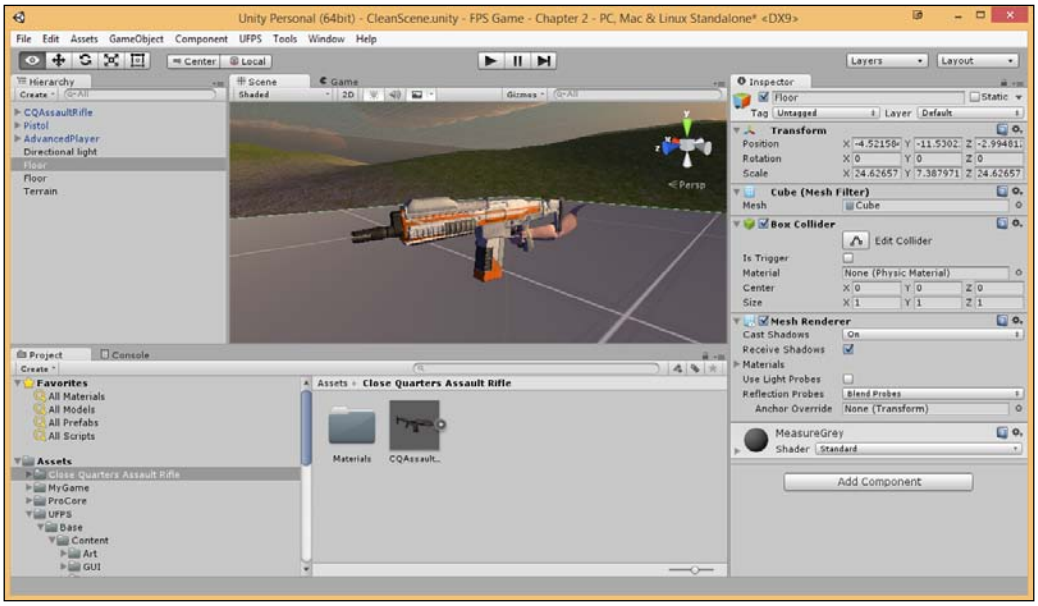


Alternatively, we can press *F* on the keyboard to focus. We can also hold the right-click to rotate the camera in the direction we want and hold the middle mouse button to move the camera as needed.

6. After completing the side view, move on to the **Top** camera view. Let it be centered to the hand as well:

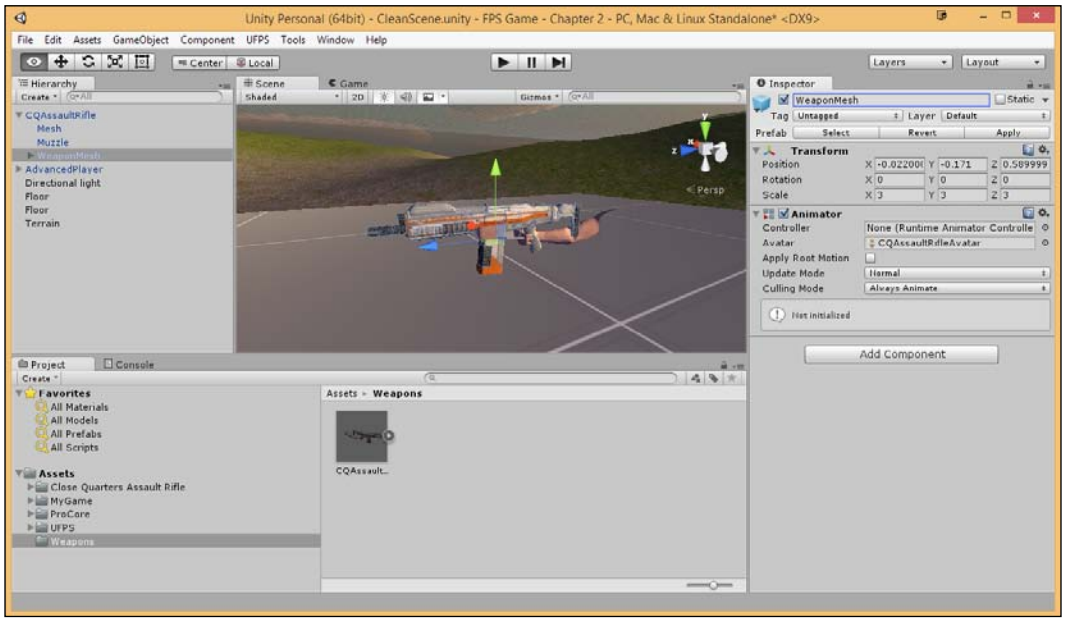


Keep at it, making adjustments via the **Translate** tool until it looks perfect to you.



Once you're satisfied with the weapon's placement, you need to remove the original pistol. Thankfully, it's very easy to do this.

7. Open the **Hierarchy** of the **Pistol** by clicking on the arrow to the left of its name and select the **Mesh** actor. You'll notice that the mesh has three **Materials** on it, one of which is the hand/arm and the others are the gun. From the **Mesh Renderer** component, expand the **Materials** tab and change the **Size** value to `1`, eliminating the others and giving us an empty hand to work with.
8. Now, from the **Hierarchy** tab, drag and drop the `CQAssaultRifle` object from our level on top of the `Pistol` prefab to make it a part of the group.
9. Rename `CQAssaultRifle` to `WeaponMesh` and the `Pistol` object to `CQAssaultRifle`.
10. Once this is done, from the **Project** tab, go to the `Assets` folder and create a new folder called `Weapons`. Drag and drop the `CQAssaultRifle` object there, creating a prefab in the process.

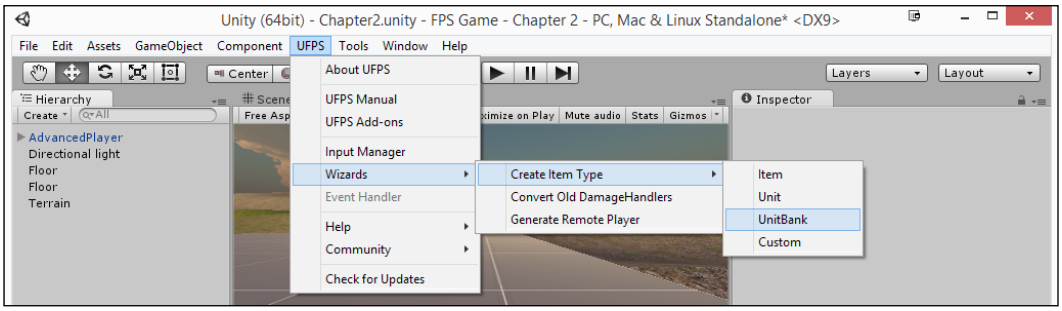


11. With this, we can delete `CQAssaultRifle` inside the **Hierarchy** tab (not the one from the **Project** tab).

Creating a UnitBank

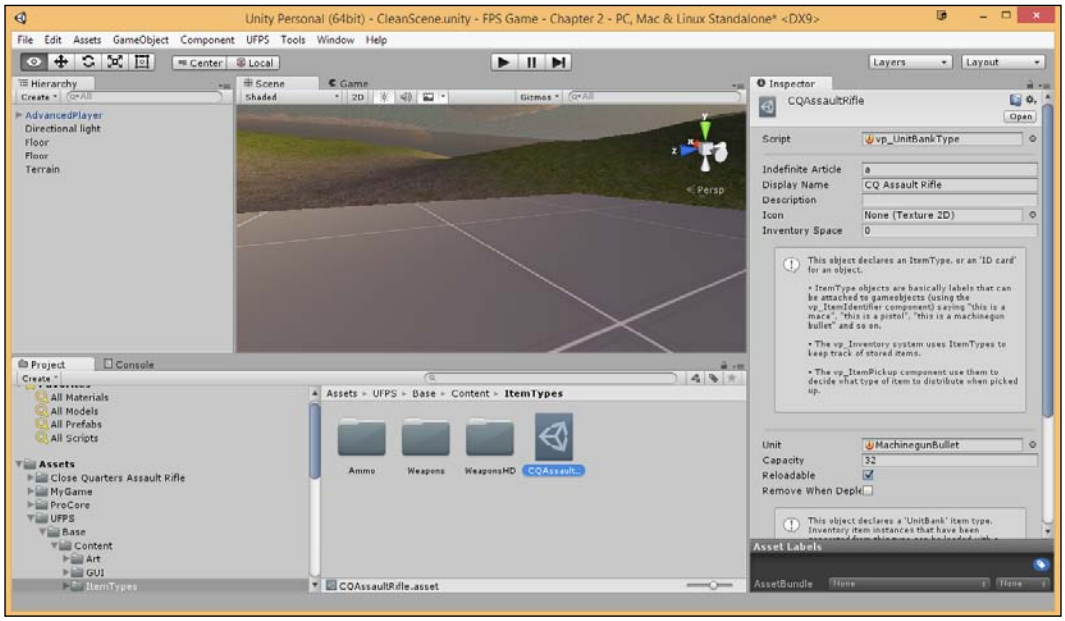
The next thing we need to do is to create a weapon that will associate with the mesh we just created. In UFPS, any kind of weapon that uses some kind of resource is referred to as a **UnitBank**. These UnitBanks use **Units** to fire. To create a UnitBank perform the following steps:

1. Let's create a **UnitBank** for our weapon by going to the top menu and selecting **UFPS** | **Wizards** | **Create Item Type** | **UnitBank**.



2. Once created, go over to the **Project** tab and change the name of the `New UnitBank Type` file to `CQAssaultRifle` by clicking on the file and renaming it (or by pressing *F2*).
3. From the **Inspector** tab, under **Display Name**, change the value to `CQ Assault Rifle` and, under **Unit**, select the type of bullet you want to use. In this case, I will use `MachinegunBullet`.

4. Next, select the **Capacity** value to `32`. This means that the weapon can fire `32` shots before it needs to be reloaded.



5. Finally, we will drag and drop the file into the `Weapons` folder.

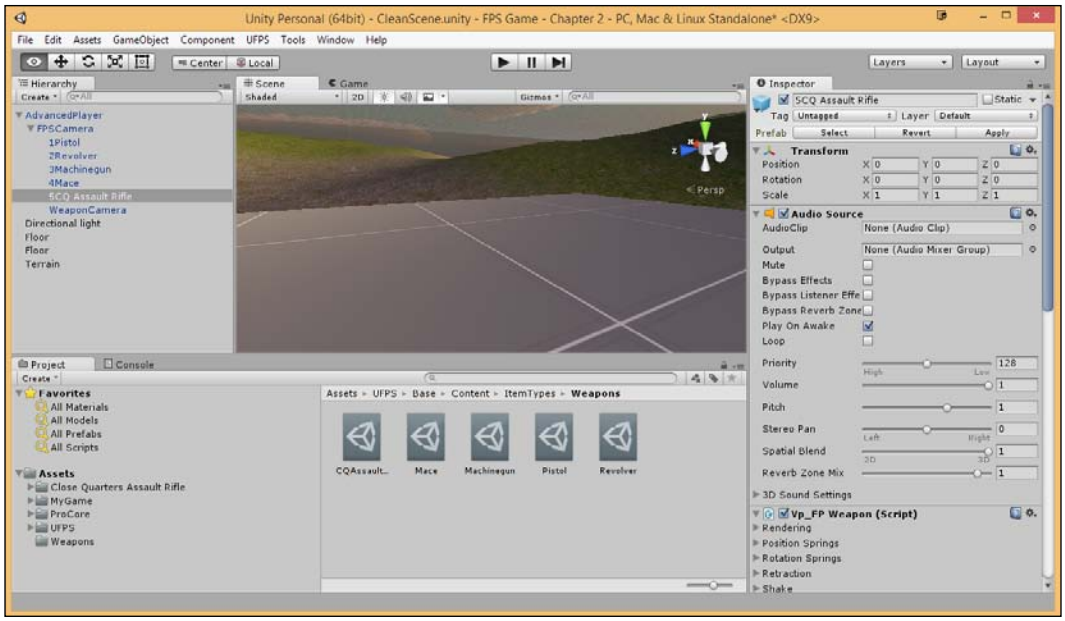


More information on UnitBanks can be found in the **Item Types** section at `http://www.visionpunk.com/hub/assets/ufps/manual/inventory`.

Creating the weapon

Next, let's add the weapon by performing the following steps so that our player can use it:

1. From the **Hierarchy** tab, go to the **AdvancedPlayer** object and expand it. Expand the **FPSCamera** object to show all of the weapons that the player can use. From there, click on the `1Pistol` object and press *Ctrl* + *D* to duplicate the object (alternatively, use **Edit** | **Duplicate**).
2. Move the newly created object below the `4Mace` object and rename it to `5CQ Assault Rifle`.

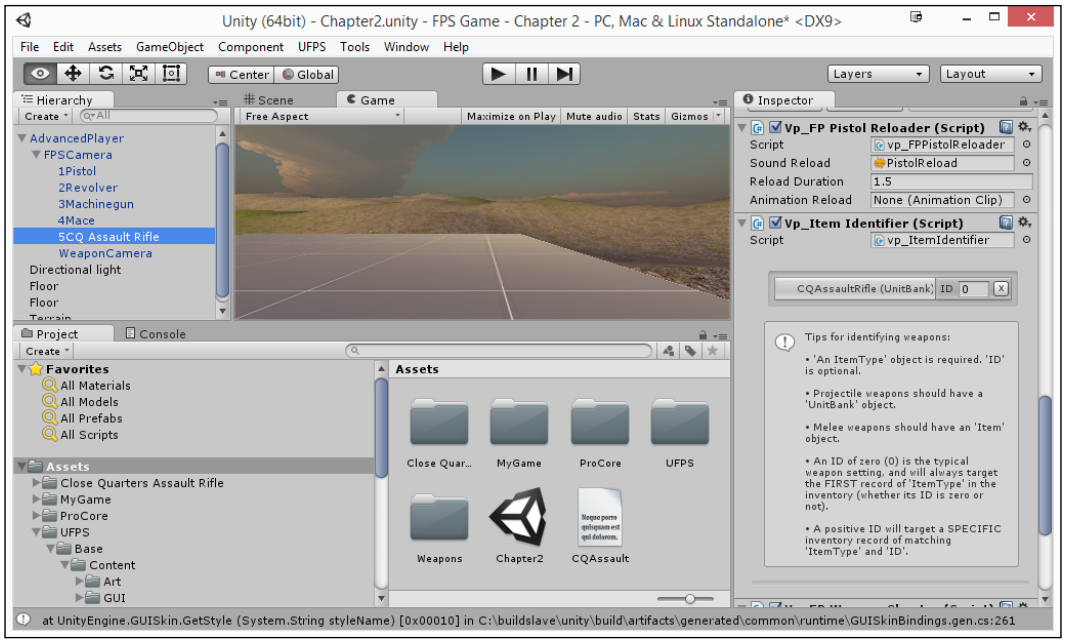


If we were to play the game right now, we could press *5* and a second pistol would be displayed. Let's make it so that it will display our new weapon by replacing the weapon that is being rendered in the **Vp_FP Weapon (Script)** component. The **Vp_FP Weapon (Script)** component is used to animate a weapon using its procedural motion properties. This means that the component will manipulate the weapon transform's position and rotation using springs, bob, and perlin noise.

3. With the `5CQ Assault Rifle` object selected in **Hierarchy**, go to the **Inspector** tab, scroll down to the **Vp_FP Weapon (Script)** component, and extend the **Rendering** options. From there, set **1st Person Weapon** to the `CQAssaultRifle` prefab we created earlier in the `Assets/Weapons` folder with the hand by dragging and dropping it in the slot. This will replace the mesh to use the model of our rifle instead of the pistol.

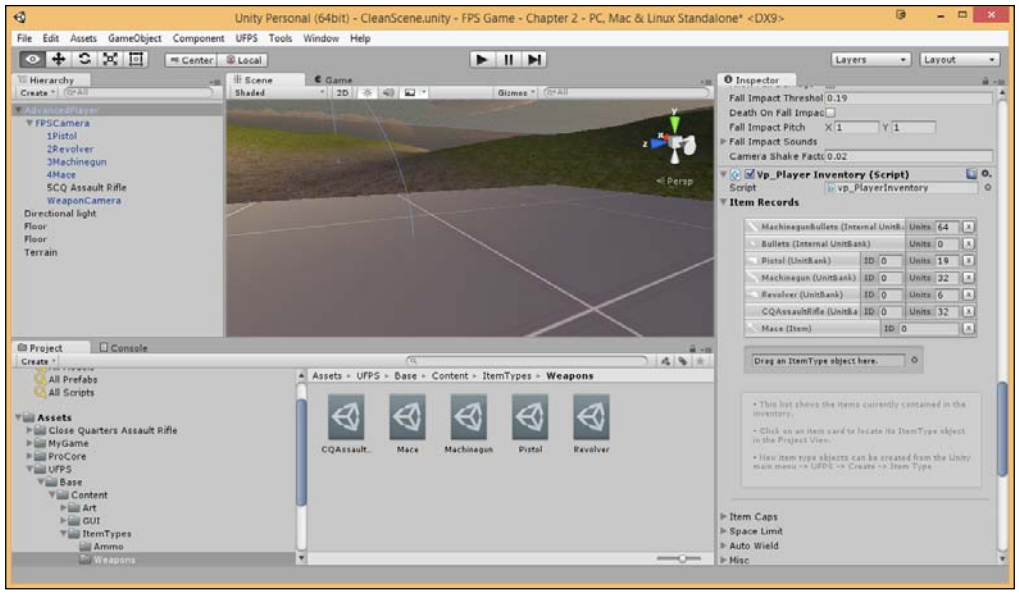
   Next, we need to say that this is an Assault Rifle. To do this, we need to modify the **Vp_ItemIdentifier (Script)** class. The item identifier can be slapped onto any game object to basically say that "this is an item of type *X*." It is what allows the weapon handler to associate a particular first-person weapon object with the information from the inventory.

4. Scroll all the way down in the **Inspector** tab to the **Vp_Item Identifier (Script)**. Click on **X** to the right of the `Pistol (UnitBank)` identifier. Then, go to the `Assets/UFPS/Base/Content/ItemTypes/Weapons` folder and drag and drop the `CQAssaultRifle` UnitBank in the slot.



Now, if we were to play the game, we would not be able to use the *5* key anymore. This is because even though we created the blueprint of the weapon, the player does not have it in their inventory (it was using the earlier pistol, which is included by default).

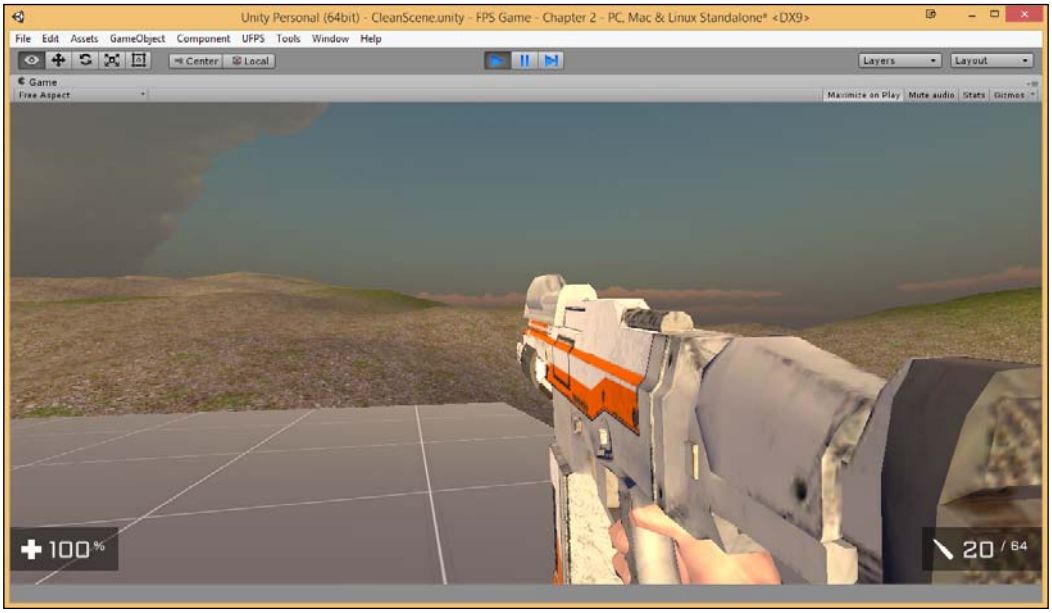
5. Next, select the `AdvancedPlayer` object, go to the **Vp_Player Inventory (Script)** component, and open **Item Records**. Click on the circle to the right of the **Drag an ItemType object here** selection and select the **CQAssaultRifle** file.





For more information on the Inventory class, check out `http://www.visionpunk.com/hub/assets/ufps/manual/inventory`.

6. Now, let's play the game and press the *5* key.



With this, our weapon is all set to be used!

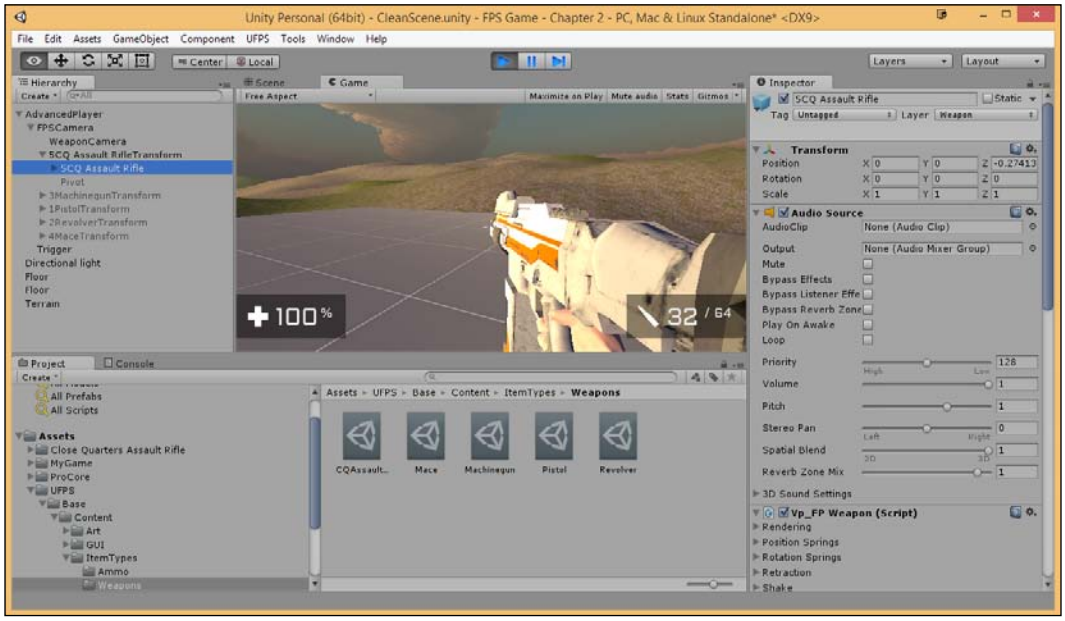
Customizing our weapon's properties

Now, it's great that the weapon looks like our new weapon, but it fires and moves exactly like a pistol. In this section, we are going to modify the properties of our weapon to give it the behavior we want by using the following steps:

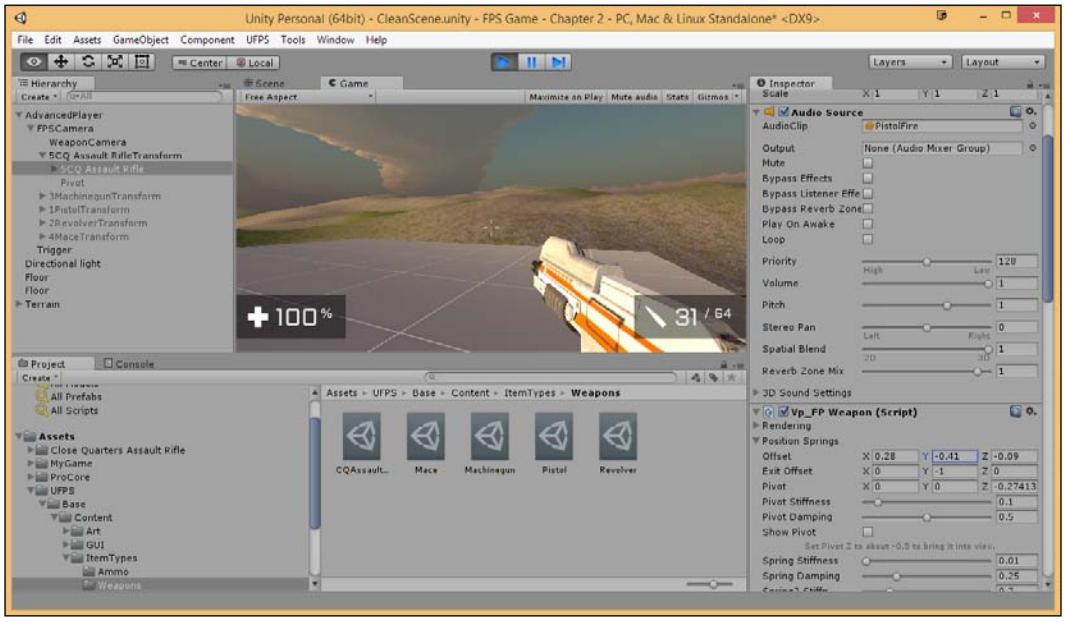
1. Now, let's start off by switching over to the **Game** tab and unchecking the **Maximize on Play** option if it is currently enabled so we can still access the Inspector while the game is being played. We will play the game and press *5* to bring out our weapon.

   It's important to note that while the game is being played, any changes we've made would be undone when we exist; but we can get around this by copy/pasting the component values.

2. Press *Enter* to return the control to your mouse. Then, from the **Hierarchy** tab, select the weapon, which can be found in **AdvancedPlayer** | **FPSCamera** | **5CQ AssaultRifleTransform** | **5CQ Assault Rifle**.



3. Now, with the object selected, move down to the **Vp_FP Weapon (Script)** component and extend the **Position Springs** property. Generally, these heavier weapons tend to be lower and closer to the players. So, click and drag on the **Offset** and set **X**, **Y**, and **Z** properties till you get them to where you want your weapon to be. I, personally, used `0.28`, `-0.41`, and `-0.09`.



This **Offset** property is where the weapon "wants to be" and it will move toward this target position if anything causes it to change, such as recoil.



Information on the **Position Springs** properties can be found at `http://www.visionpunk.com/hub/assets/ufps/manual/camera`.

The next thing we are going to change is the recoil. Right now, it is reacting very violently to an occurring shot. To alter this, we will need to modify the `Spring2 Stiffness` and `Spring 2 Damping` variables.

In layman's terms, stiffness determines how loosely or rigidly the weapon spring will behave. Damping makes the spring velocity wear off as it approaches its rest state.

In addition to modifying the **Position Springs** properties, we will also need to modify the **Rotation Springs** properties to higher numbers so the gun doesn't fly into the air on being fired.

4. Under the **Positional Springs**, set **Spring2 Stiffn** to `.8` and **Spring2 Damp.** to `.9`.
5. Next, under **Rotation Springs**, change **Spring2 Stiffn.** to `.5` and **Spring2 Damp.** to `1`.
6. Finally, right-click on the **Vp_FP Weapon (Script)** component and select **Copy Component**. Quit the game; notice that the values have changed to what they were before. Now, right-click on your component and select **Paste Component Values**. Perfect!

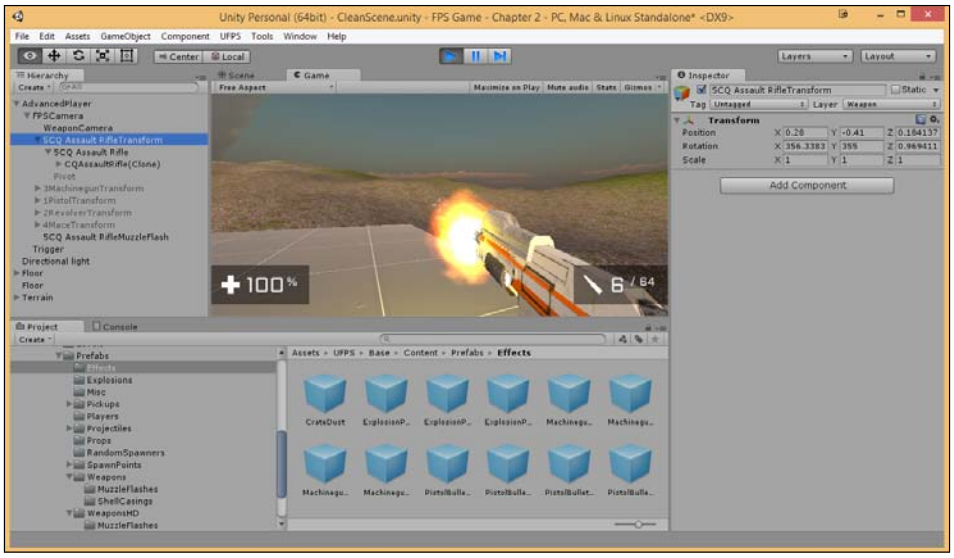


Information on the **Rotation Springs** properties can be found at `http://www.visionpunk.com/hub/assets/ufps/manual/weapons`.

The next thing we are going to adjust is the muzzle flash because, right now, the position doesn't really reflect where the bullets should be coming out.

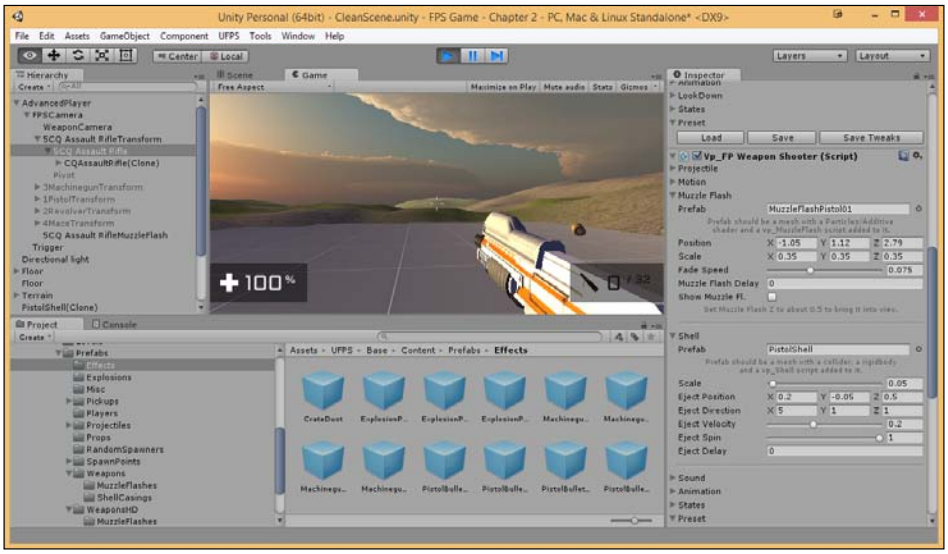
To do this, we will need to access the **Vp_FPWeapon Shooter (Script)** component. The **Vp_FPWeapon Shooter (Script)** class adds firearm properties to a weapon, which allows us to manipulate the recoil and manages the firing rate, accuracy, sounds, muzzle flashes, and spawning of projectiles and shell casings. It also has basic ammo and reloading features.

7. Next, go to the **Vp_FPWeapon Shooter (Script)** component and open the **Muzzle Flash** option. Inside **Prefab**, click on the circle and, from the search bar, select the **MuzzleFlashPistol01** prefab to give us a muzzle flash when we shoot.
8. Check the **Show Muzzle Fl.** option that allows us to see the muzzle to easily adjust and modify the position until it's at the correct position.



Now, we can adjust the shell. Currently, it pops out from the top of our gun but, for this one in particular, it comes out from the side.

9. Inside the **Vp_FP Weapon Shooter (Scripts)** component, expand the **Shell** property. Then, change **Eject Position** to `.2,-0.05,.5`.
10. After this, change **Eject Direction** to `5,1,1`. This means that it will move five times in the *X* direction and once in the *Y* and *Z* directions modified by **Eject Velocity**.



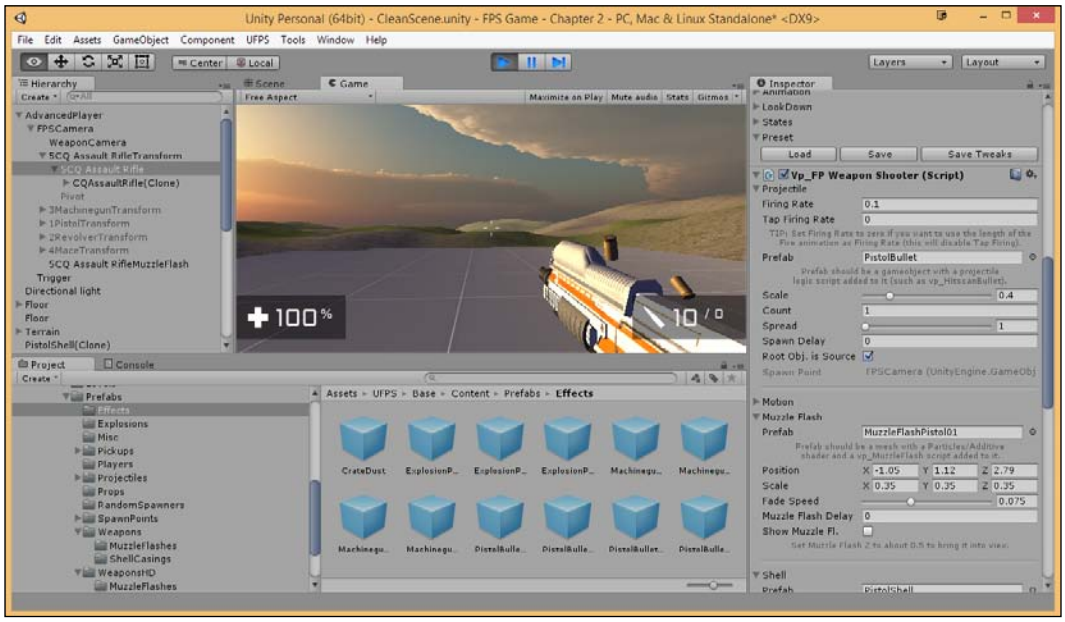
With this, the shell flies out of our weapon correctly!



More information on the **Shell** and its properties can be found at `http://www.visionpunk.com/hub/assets/ufps/manual/shells`.

The weapon currently fires very slowly, because it's using the pistol's speed. Our rifle is going to shoot a lot quicker.

11. Under the **Vp_FPWeapon Shooter (Scripts)** component, open the **Projectile** section and change **Firing Rate** to `.10`. This means that the bullets will fire once in every `.10` seconds.



I'm sure you can now tell that it's already shooting a lot quicker. Depending on the weapon, you may want it to take a longer or shorter distance. You may also want the player to shoot faster if they are tapping instead of just holding down the mouse button, this is what the **Tap Firing Rate** property is for.

We can use the current sound for this weapon but, for sake of completion, I'm going to show how we can use our own sound as well.

12. Open the **Sound** section and set the **Fire** property to `MachinegunFire`. Then, change **Fire Pitch** to `1.5`, `1.5`.

    Fire is the sound that plays when you fire the weapon, while **Dry Fire** plays when you are out of ammo and try to fire. The **Fire Pitch** property is used to alter the pitch of the sound for some variations, which may be useful to differentiate your weapons.



For more information on the **Sound** properties, please visit `http://www.visionpunk.com/hub/assets/ufps/manual/shooters`.

13. Once you are finished with the **Vp_FWeapon Shooter (Script)** component, click on the **Save** button at the bottom. Name the file as `CQAssault` and continue. Once you exit the game, click on the **Load** button and bring the properties in!
14. Select the `AdvancedPlayer` object and, from the **Inspector** tab at the top, click on the **Apply** button in the **Prefab** section. This will save all of the newly created data for any other advanced players we wish to create in the future.
15. Finally, we are going to save our progress and our scene by going to **File** | **Save Scene**.

Summary

With this, you've hopefully gotten a taste of all of the various toys working with UFPS gives you, allowing you to only be limited, in terms of your imagination, to what you want to do with your weapons. Specifically, in this chapter, you learned how to create a gameplay testbed. We imported assets for a new weapon, created a mesh for the weapon, added it to our player, added it to the inventory, and then customized it to suit what we wanted.

In the next chapter, we will delve into level creation by prototyping some levels, making use of the Prototype tool.

3
Prototyping Levels with Prototype

In the game industry, while there are many different roles in the process of developing games, there are two main roles in level creation, the **environment artist** and the **level designer**.

An environment artist is the person who builds the assets to go into the environment. They use tools such as 3ds Max or Maya to create the model and then use other tools like Photoshop to create textures and normal maps.

The level designer is responsible for assembling the assets that an environment artist creates in an environment for players to enjoy. They design the gameplay elements, create the scripted events, and test the gameplay. Typically, a level designer will create environments through a combination of scripting and use some tools that may be in development. In our case, that tool is Unity.



One thing that is important to note is that most companies have their own definition for different roles. In some companies, a level designer may need to create assets and an environment artist may need to create a level layout. There are also some places that hire someone just for lighting or to place meshes (called a **mesher**) because they're so good at it.

In this chapter, we take on the role of a level designer who has been tasked to create a level prototype to prove that our gameplay is solid. We will use the free tool Prototype to help in this endeavor. In addition, you will also learn some of the basics of level designing.

This project will be split into a number of tasks. It will be a simple step-by-step process from the beginning to the end. Here is the outline of our tasks:

- Creating geometry—building a room
- Building a doorway
- Duplicating rooms / creating a hallway
- Preventing falls—collision
- Adding stairways
- Coloring your world

Prerequisites

Before we start, we will need to have a project created that already has UFPS and Prototype installed. If you do not have these, follow the steps described in *Chapter 1*, *Getting Started on an FPS*.

Level design 101 – planning

Now, because we are going to be diving straight into Unity, I feel it's important to talk a little more about how level designing is done in the game industry. While you may think a level designer will just jump into the editor and start playing, the truth is you normally need to do tons of planning ahead of time before you even open your tool.

Generally, a level begins with an idea. This can come from anything; maybe you saw a really cool building or a photo on the Internet gave you a certain feeling; maybe you want to teach the player a new mechanic. Turning this idea into a level is what a level designer does. Taking all of these ideas, the level designer will create a level design document that will outline exactly what you're trying to achieve with the entire level from start to end.

A level design document will describe everything inside the level, listing all of the possible encounters, puzzles, so on and so forth that the player will need to complete as well as any side quests that the player will be able to achieve. To prepare for this, you should include as many references as you can with maps, images, and movies similar to what you're trying to achieve. If you're working with a team, making this document available on a website or Wiki will be a great asset so that you know exactly what is being done in the level, what the team can use in their levels, and how difficult their encounters can be. Generally, you'll also want a top-down layout of your level done either on a computer or graph paper with a line showing the player's general route through the level with the encounters and missions planned out.

Of course, you don't want to be too tied down to your design document. It will change as you playtest and work on the level, but the documentation process will help solidify your ideas and give you a firm basis to work on.

For those of you interested in seeing some level design documents, feel free to check out Adam Reynolds (*Level Designer on Homefront* and *Call of Duty: World at War*) at `http://wiki.modsrepository.com/index.php?title=Level_Design:_Level_Design_Document_Example`.



If you want to learn more about level designing, check out *Beginning Game Level Design*, *Marc Scattergood* and *John Feil* (previously my teacher), Cengage Learning PTR. For more of an introduction to all of game designing from scratch, check out *Level Up!: The Guide to Great Video Game Design*, *Scott Rogers*, Wiley and *The Art of Game Design*, *Jesse Schell*, CRC Press.

For some online resources, Scott has a neat GDC talk called *Everything I Learned About Level Design I Learned from Disneyland* that can be found at `http://mrbossdesign.blogspot.com/2009/03/everything-i-learned-about-game-design.html`, and *World of Level Design* (`http://worldofleveldesign.com/`) is a good source for learning about level designing, though it does not talk about Unity specifically.

In addition to a level design document, you can also create a **game design document** (**GDD**) that goes beyond the scope of just levels and includes story, characters, objectives, dialogue, concept art, level layouts, and notes about the game's content. But it is something that you have to do on your own.

Creating the architectural overview

As a level designer, one of the most time-consuming tasks of your job will be to create environments. There are many different ways out there to create levels. By default, Unity gives us some default meshes such as a Box, Sphere, and Cylinder. While it's technically possible to build a level in this manner, it could get tedious very quickly. Next, I'm going to go through the most popular options to build levels for the games made in Unity before we jump into building a level of our own.

3D modeling software

Opening up a 3D modeling software package and building architectures this way is what professional game studios often do. This gives you maximum freedom to create your environment and allows you to do exactly what you'd like to do; but this requires you to be proficient in that tool, be it Maya, 3ds Max, Blender (that can be downloaded for free at `http://www.blender.org/`), or some other tool. Then, you just need to export your models and import them in Unity.

Unity supports a lot of different formats for 3D models (the most commonly used are `.obj` and `.fbx`), but there are a lot of issues to consider. For some best practices when it comes to creating art assets, please visit `http://blogs.unity3d.com/2011/09/02/art-assets-best-practice-guide/`.

Constructing geometry with brushes

Using **Constructive Solid Geometry** (**CSG**), commonly referred to as brushes, is a tool artists/designers use to quickly block out pieces of a level from scratch. Using brushes inside the in-game level editor has been a common approach for artists/designers to create levels. Unreal Engine 4, Hammer, Radiant, and other professional game engines make use of this building structure, making it quite easy for people to create and iterate through levels quickly through a process called whiteboxing as it's very easy to make changes to simple shapes. However, just like learning a modeling software tool, there can be higher barriers in creating complex geometry using a 3D application, but using CSG brushes will provide a quick solution to create shapes with ease.

Unity does not support building things like using brushes (CSG) by default, but there are several tools in the Unity Asset Store that allow you to do so. For example, sixbyseven studio has an extension called **ProBuilder** that can add this functionality to Unity, making it very easy to build our levels. The only possible downside is the fact that it costs money, though it is worth every penny. However, sixbyseven has kindly released a free version of their tools called **Prototype** that we installed earlier, which contains everything we would need for this chapter. But it does not allow us to add custom textures and some of the more advanced tools. We will be using ProBuilder later on in the book to polish the entire product. You can find more information on ProBuilder at `http://www.protoolsforunity3d.com/probuilder/`.

Modular tilesets

Another way to generate architecture is through the use of **tiles** that are created by artists to build levels. Similar to using Lego pieces, we can use these tiles to snap together walls and other objects to create a building. With the creative use of tiles, you can create a lot of content with minimal assets. This is probably the easiest way to create a level at the expense of not being able to create unique-looking buildings, since you only have a few pieces to work with. Titles such as *Skyrim* use this to great extents to create their large world environments.

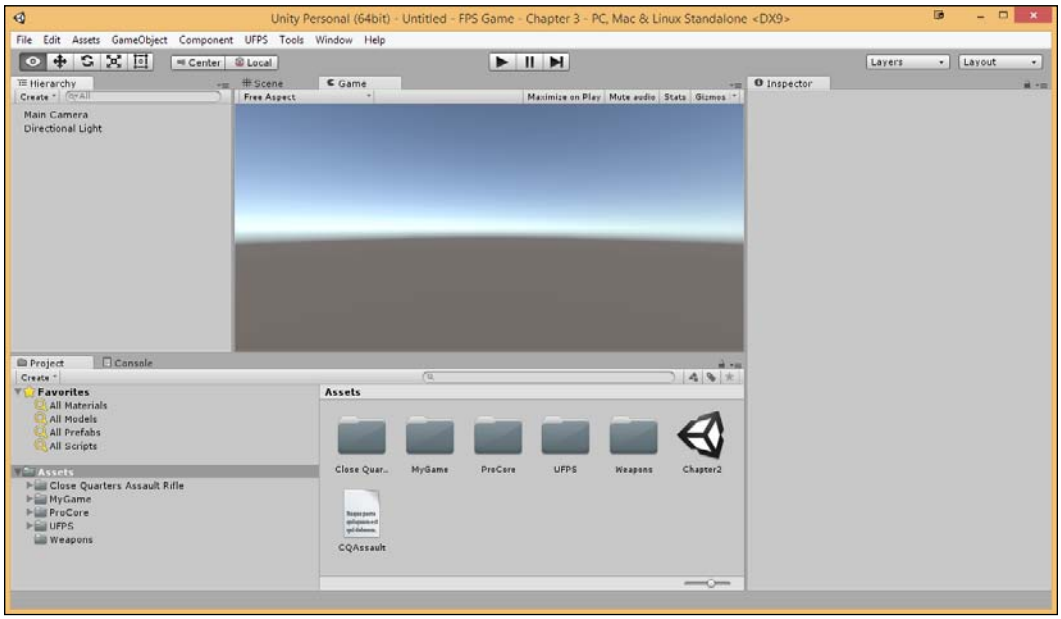
Mix and match

Of course, it's also possible to use a mixture of the previously mentioned tools to use the advantages of doing things in certain ways. For example, you could use brushes to block out an area and then use a group of tiles called a **tileset** to replace the boxes with highly detailed models, which is what a lot of AAA studios do. In addition, we could also place brushes initially to test our gameplay and then add in props to break up the repetitiveness of the levels.

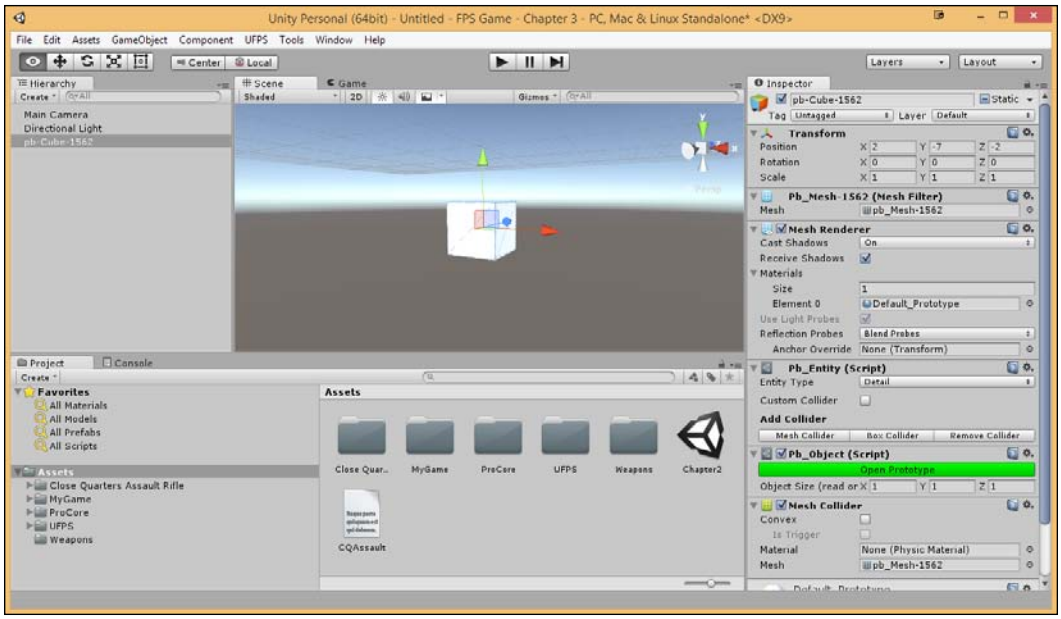
Creating geometry

The first thing we are going to do is to learn how to create geometry as follows:

1. From the top menu, go to **File | New Scene**. This will give us a fresh start to build our project.



2. Next, because we already have Prototype installed, let's create a cube by hitting *Ctrl* + *K*.

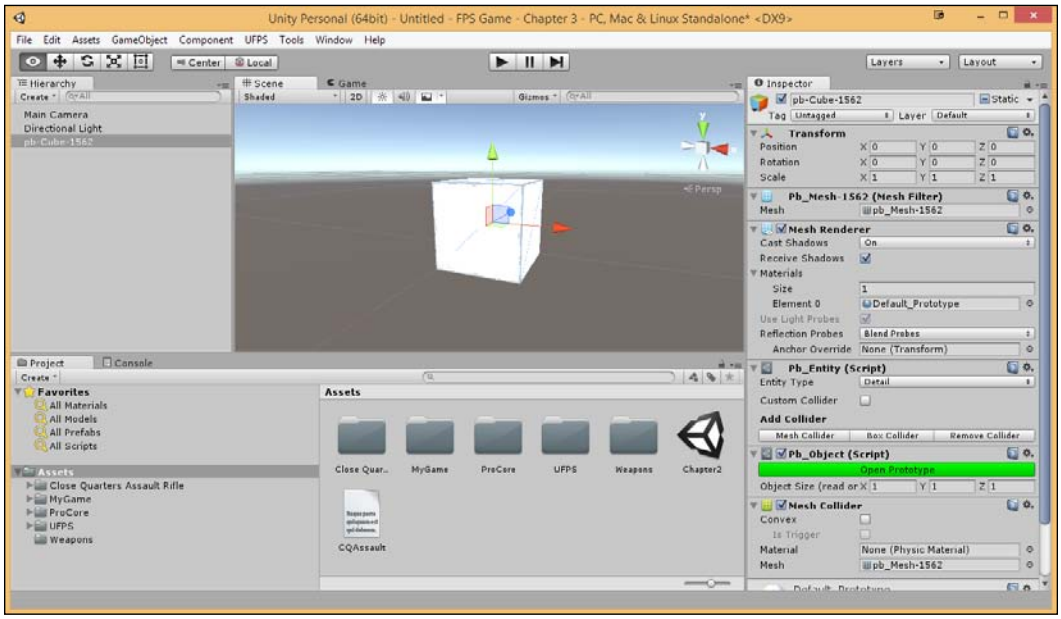


3. Right now, the **Position** value of our cube (with its name as `pb-Cube-1562` or something similar) is `2, -7, -2`. But for simplicity's sake, I'm going to place it in the middle of the world. We can do this by typing in `0, 0, 0` by going over to the **Inspector** tab, going to the **Transform** component and then clicking on the **X** position. Notice that the cursor is now automatically in the **Y** slot. Type in `0` and press *Tab* again. Then, in the **Z** slot, press `0`.



Alternatively, you can right-click on the **Transform** component and select **Reset Position**.

4. Next, to center the camera back on our `Cube` object, we will go over to the **Hierarchy** tab and double-click on the `Cube` object (or select it and press *F*).



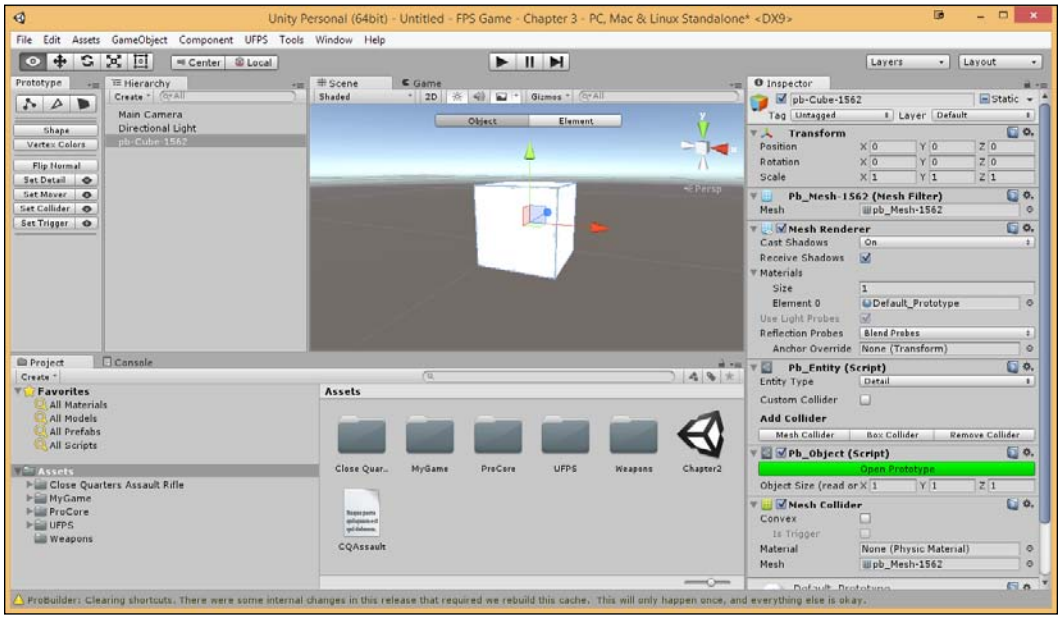
5. Now, to actually modify this cube, we are going to open Prototype. We can do this by first selecting our `Cube` object, going to the **Pb_Object (Script)** component, and then clicking on the green **Open Prototype** button.



Alternatively, you can also go to **Tools** | **Prototype** | **Prototype Window**.



This is going to bring up a window much like the one I have displayed in the preceding screenshot. This new **Prototype** tab can be detached from the main Unity window, or if you drag it from the tab over to Unity, it can be "hooked" into place elsewhere, like the following screenshot shows by dragging and dropping it to the right of the **Hierarchy** tab.



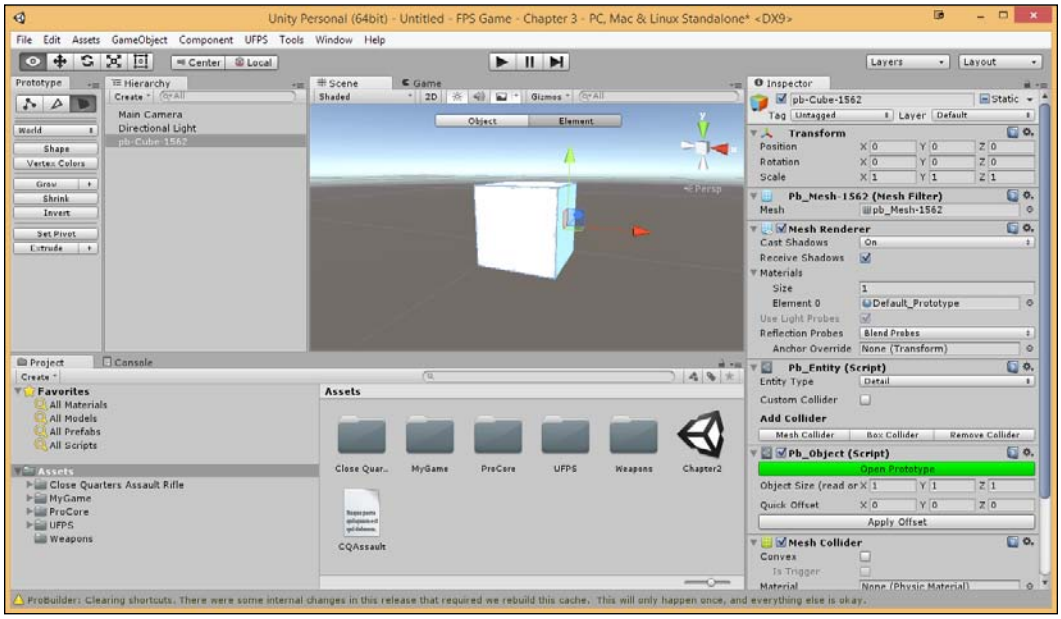
6. Next, select the **Scene** tab at the center of the screen and press the *G* key to toggle us into the Object/Geometry mode. Alternatively, you can also click on the **Element** button in the **Scene** tab. Unlike the default Object/Top Level mode, this will allow us to modify the cube directly to build on it.



For more information on the different modes, check out the modes and elements section at `http://www.protoolsforunity3d.com/docs/probuilder/#buildingAndEditingGeometry`.

You'll notice that the top of the **Prototype** tab has three buttons. These stand for the selection type you currently want to use. The default is the Vertex or Point mode, which will allow us to select individual parts to be modified. Next are Edge and Face. Face is a good standard to use at this stage, because we only want to extend things out.

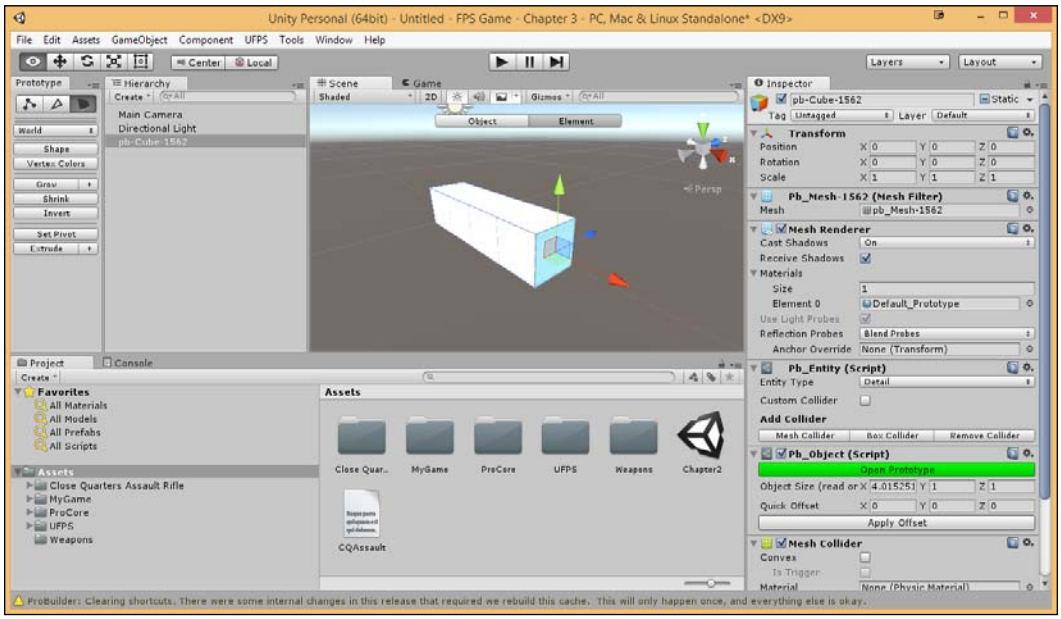
7. Select the Face mode by either clicking on the button or pressing the *H* key twice till it says **Editing Faces** on the screen. Afterward, select the box's right side.





For a list of keyword shortcuts included with Prototype/ProBuilder, check out `http://www.protoolsforunity3d.com/docs/probuilder/#keyboardShortcuts`.

8. Now, pull the red handle to extend the brush outward.



Easy enough. Note that, by default, while pulling things out, it is done with an increment of `.1`. This is nice when we are polishing our levels and trying to make things exactly where we want them to be; but right now, we are just prototyping, so it is paramount to get it out as quickly as possible to test if it's enjoyable. To help with this, we can use a feature of Unity called **Unit Snapping**.

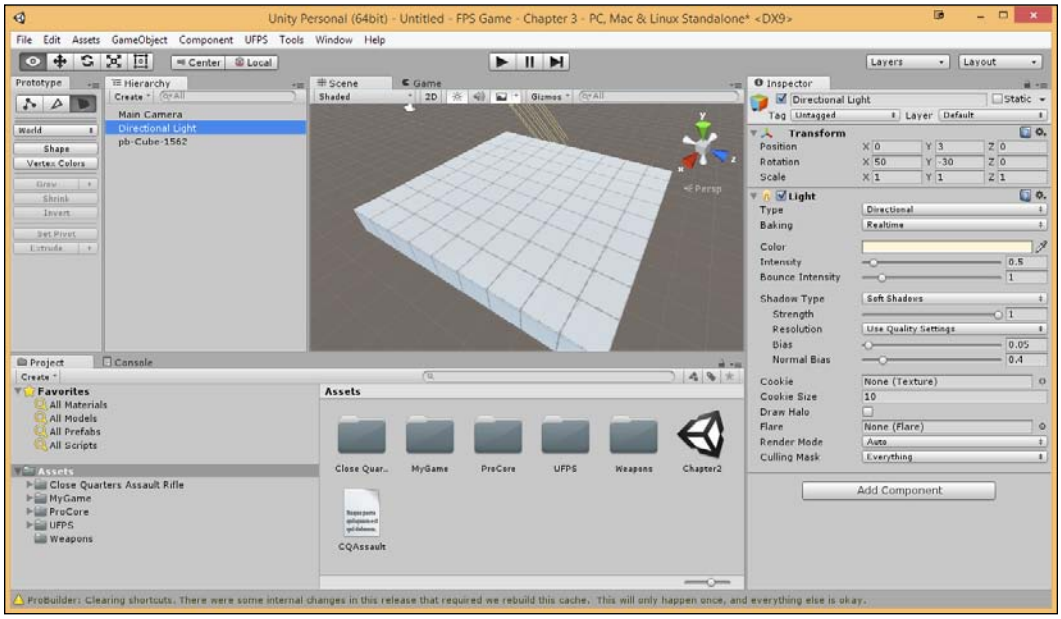
9. Undo the previous change we made by pressing *Ctrl* + *Z*, move the camera over to the other side, and select our longer face. Drag it out to be `9` units by holding down the *Ctrl* key (*command* on Mac).



ProCore3D also has another tool out called **ProGrids** that has some advanced unit-snapping functionality, but we are not going to be using it. For more information on it, check out `http://www.protoolsforunity3d.com/progrids/`.

If you'd like to change the distance travelled while using unit-snapping, set it using the **Edit** | **Snap Settings...** menu.

10. Next, drag both the sides out till they are 9 x 9 wide. To make things easier to see, select the **Directional Light** object in the scene via the **Hierarchy** tab and reduce the **Light** component's **Intensity** value to `.5`.

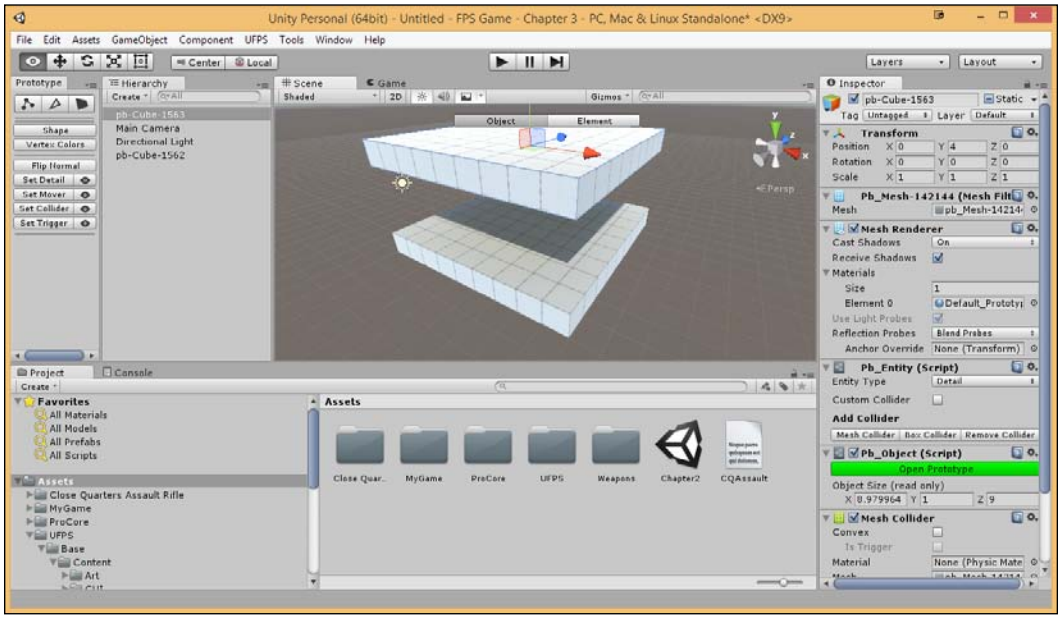


So, at this point, we have a nice looking floor, but to create our room, we first need to create the ceiling.

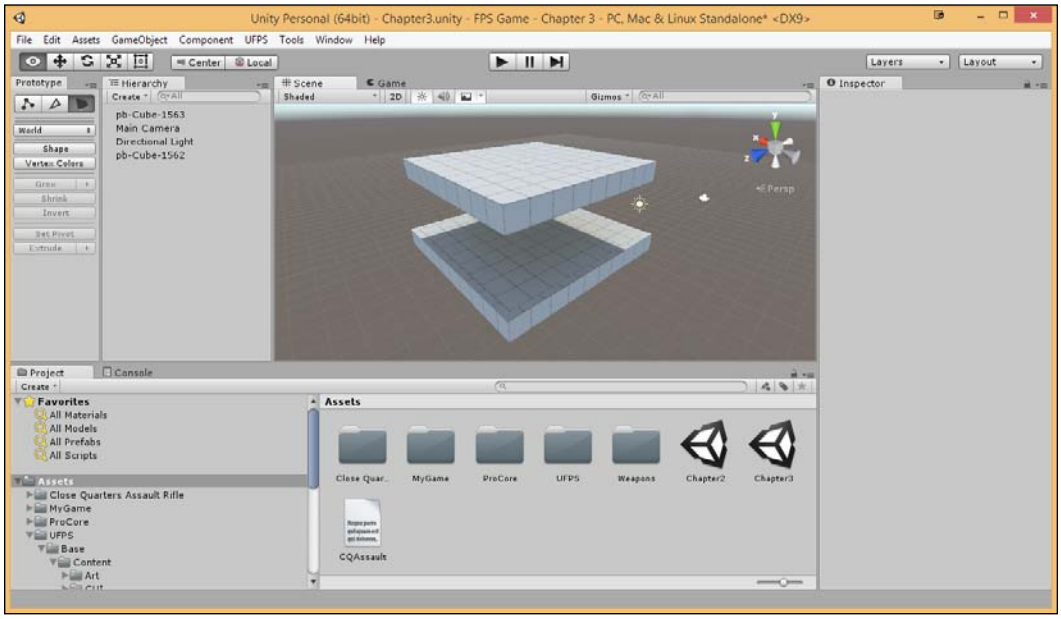
11. Select the floor we have created and then press *Ctrl* + *D* to duplicate the brush. Once completed, go back to the Object/Top Level editing mode and move the brush so that its **Position** value is at `0, 4, 0`.



Alternatively, you can click on the duplicated object and, in the **Inspector** tab, change the **Position** value of **Y** to `4`.

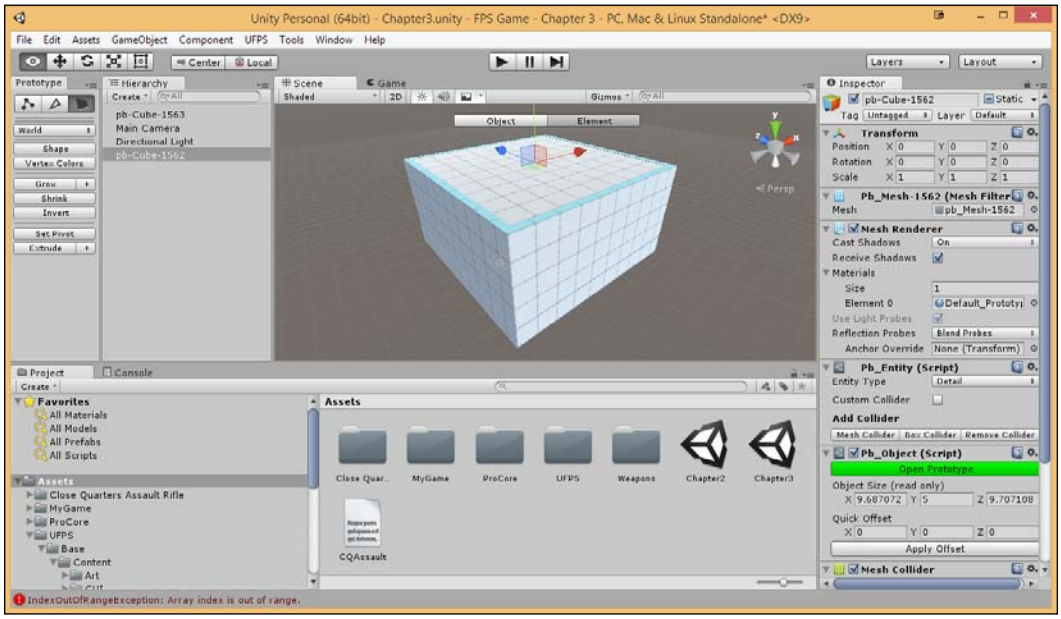


12. Go back to the sub-selection mode by hitting *H* to go back to the **Faces** mode. Hold down *Ctrl* and select all of the edges of our floor. Then, click on the **Extrude** button in the **Prototype** panel.



This creates a new part on each of the four edges that is, by default, `.5` wide (change it by clicking on the **+** button at the edge). This adds additional edges and/or faces to our object.

13. Next, we are going to extrude again. But rather than doing it from the menu, we do it manually by selecting the top of our newly created edges, holding down the *Shift* button, and dragging it up along the **Y** (green) axis. We then hold down *Ctrl* after we start the extrusion to have it snap appropriately to fit around our ceiling.



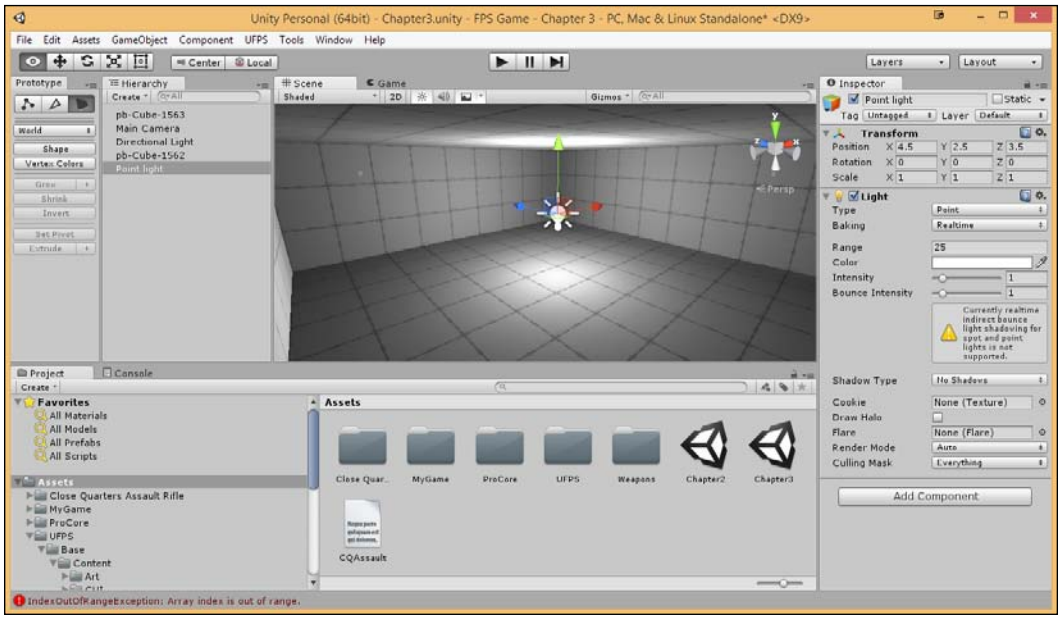


Note that the box may not look like this as soon as you let go, as Prototype needs time to compute the lighting and materials, which it will mention in the bottom-right corner of Unity.

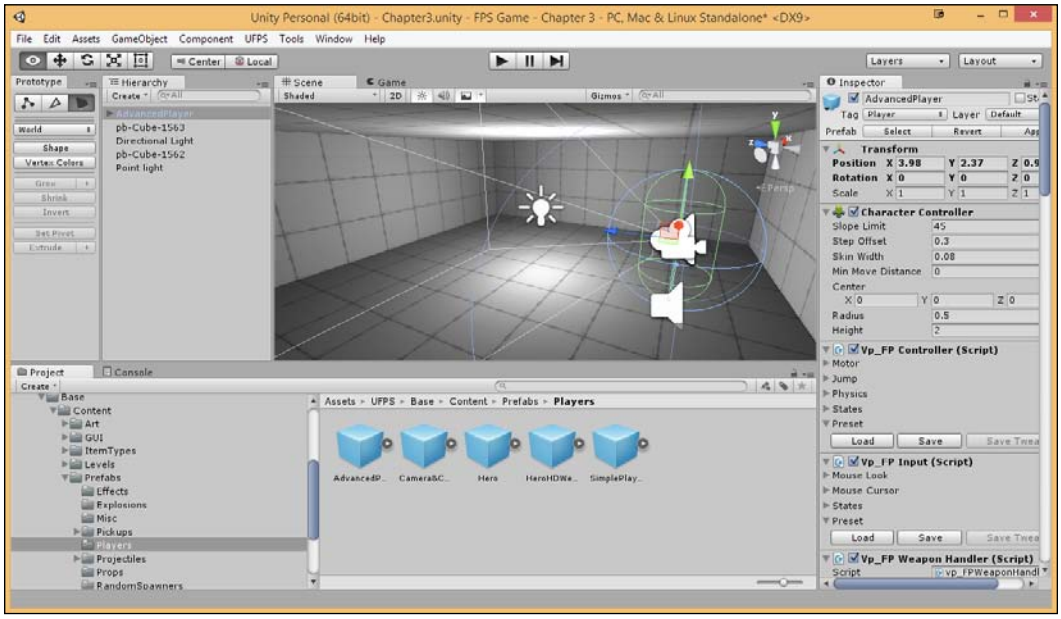
14. Next, select **Main Camera** in the **Hierarchy** tab, hit *W* to switch to the Translate mode and *F* to center-align the selection, and move our camera into the room.

    You'll notice that it's completely dark because of the ceiling, but we can add a light to the world to fix that!

15. Let's add in a point light by going to **GameObject** | **Light** | **Point Light** and positioning it at the center of the room toward the ceiling (in my case, it is at `4.5`, `2.5`, `3.5`). Then, up the **Range** value to `25` so that it hits the entire room.



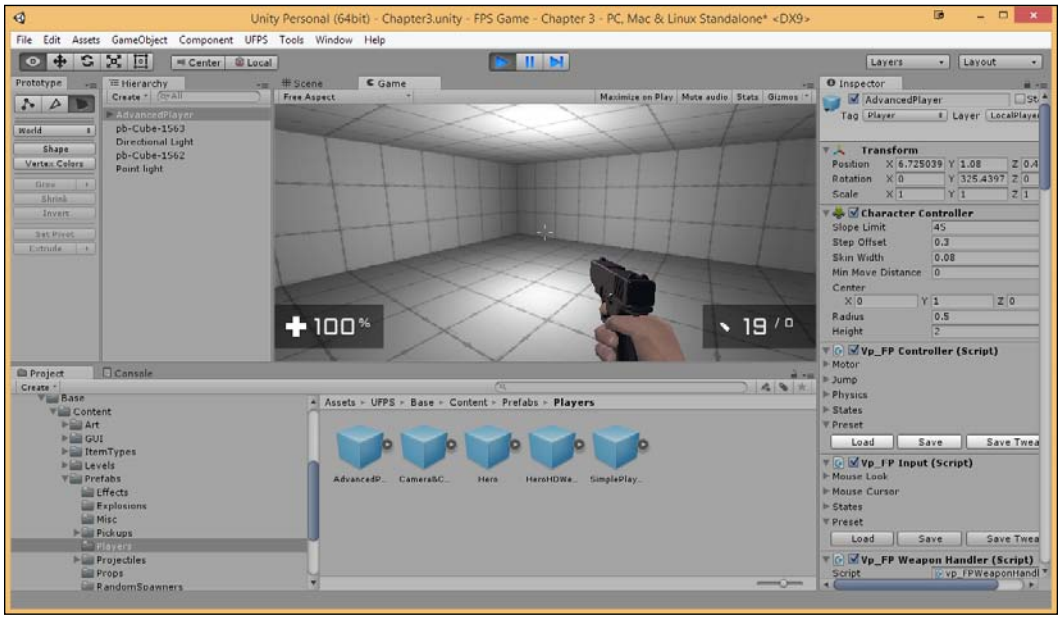
16. Finally, add in a player to see how he interacts. Delete the **Main Camera** object from **Hierarchy** as we won't need it. Then, go to the **Project** tab and open the `Assets\UFPS\Base\Content\Prefabs\Players` folder. Drag and drop the `AdvancedPlayer` prefab in such that it doesn't collide with the walls, floors, or ceiling. Place it a little higher than the ground, as shown in the following screenshot:



17. Next, save the level (`Chapter 3_1_CreatingGeometry`) and hit the Play button.



It may be a good idea for you to save your levels, so you can go back and see what was covered in each section for each chapter, making things easier to find in the future.





Again, remember that we can pull a weapon out by pressing the *1* to *5* keys.

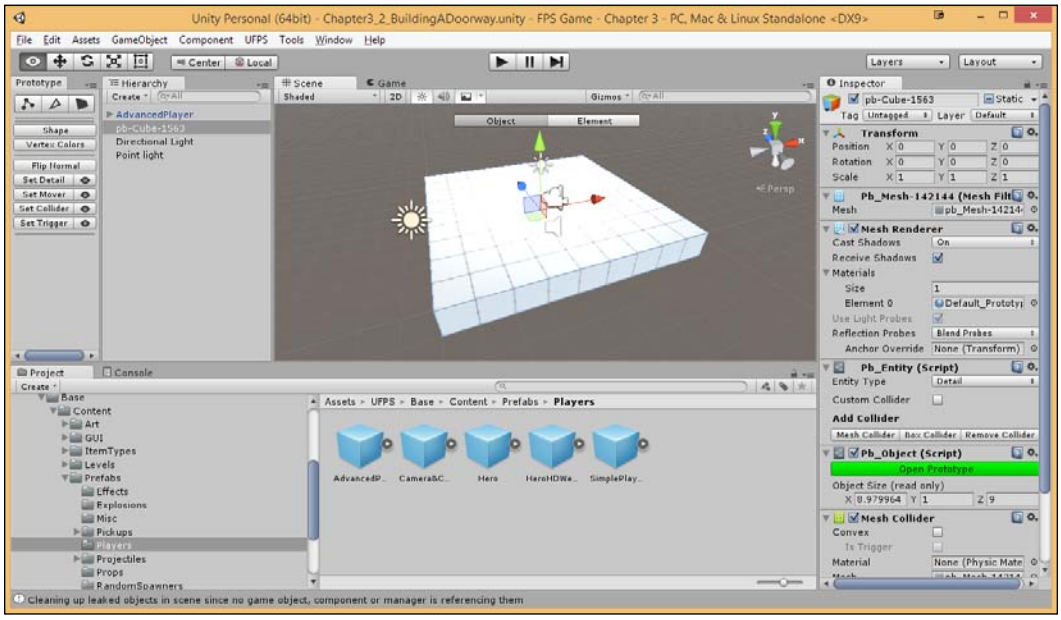
With this, we now have a simple room that we can interact with.

Building a doorway

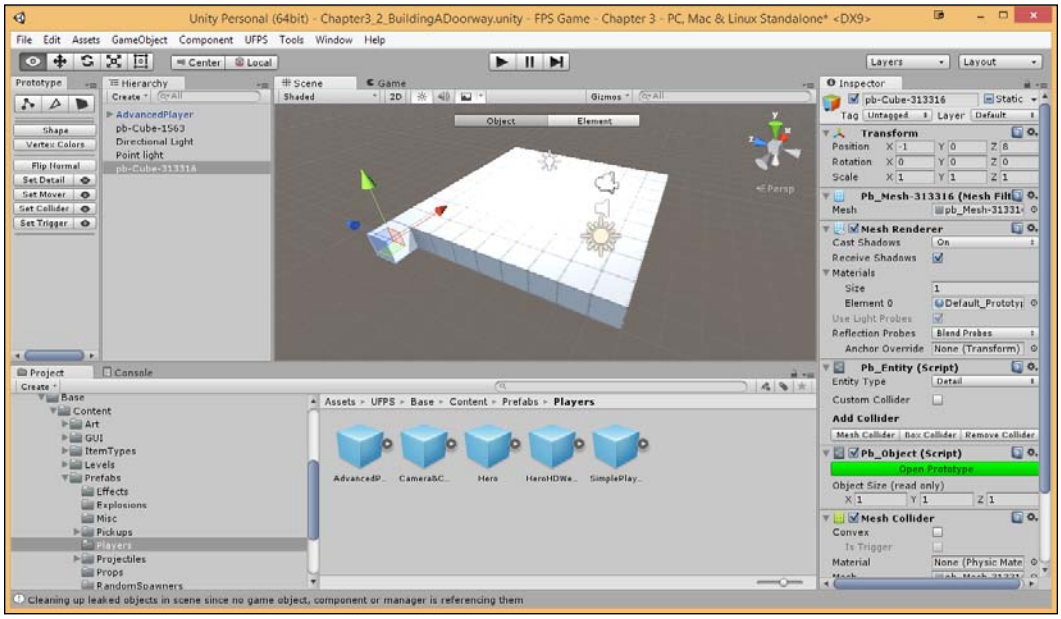
As awesome as it is to have a room, our levels are actually going to be much larger than this. Let's open an opening to the rest of the levels by creating a doorway.

Now, if we had ProBuilder, we could use the subdivide tool and modify the faces to open up a doorway. But in our case, we will rebuild our room, taking the door into consideration:

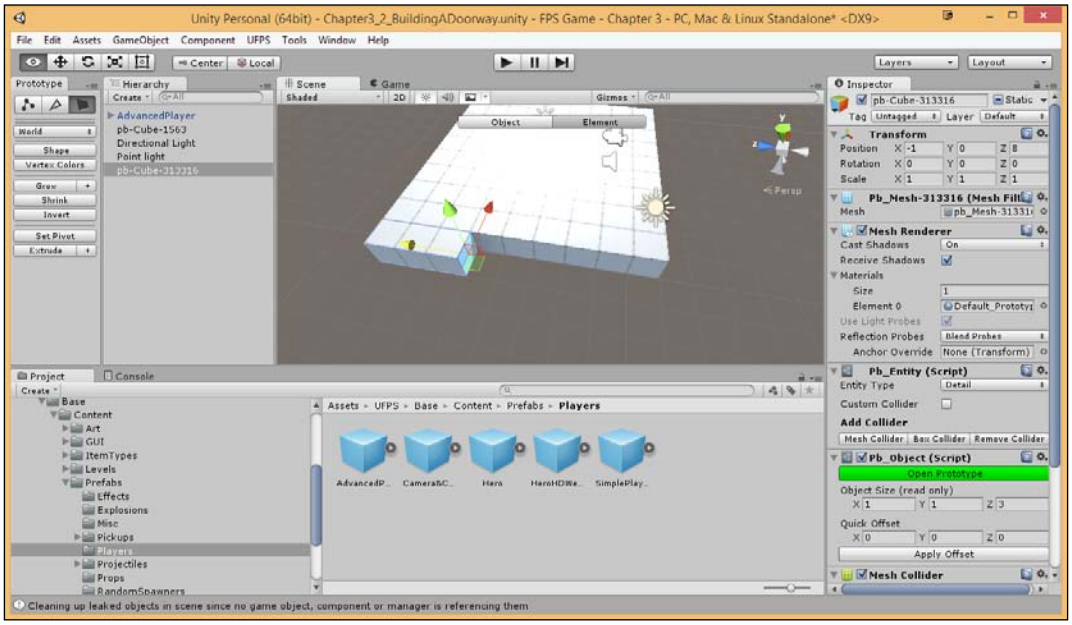
1. Let's go ahead and delete our floor/walls object and move our original ceiling back down to `0, 0, 0`.



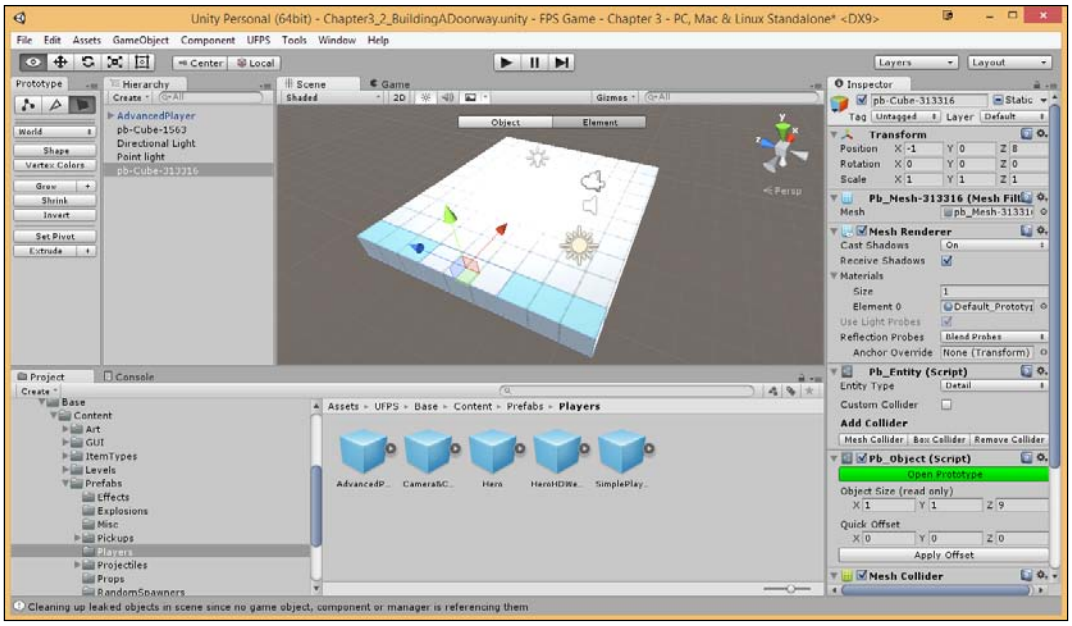
2. Let's press *Ctrl* + *K* to spawn a new brick and bring it into our scene. Move it so that it snaps to the edge of our room, making sure that the Prototype subselection is turned off before the movement.



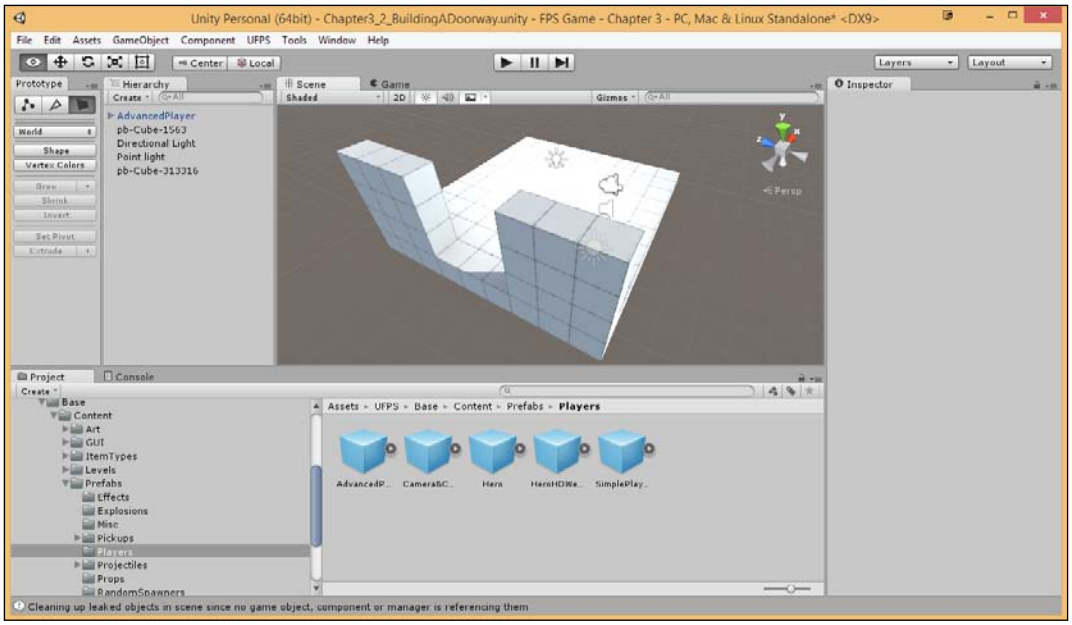
3. In the **Prototype** tab, switch back to the Element/Geometry mode and select the face nearest to you. Move it so that it is `3` units wide, leaving room for our doorway in the center. Hold down *Ctrl* while you do so for it to be even.



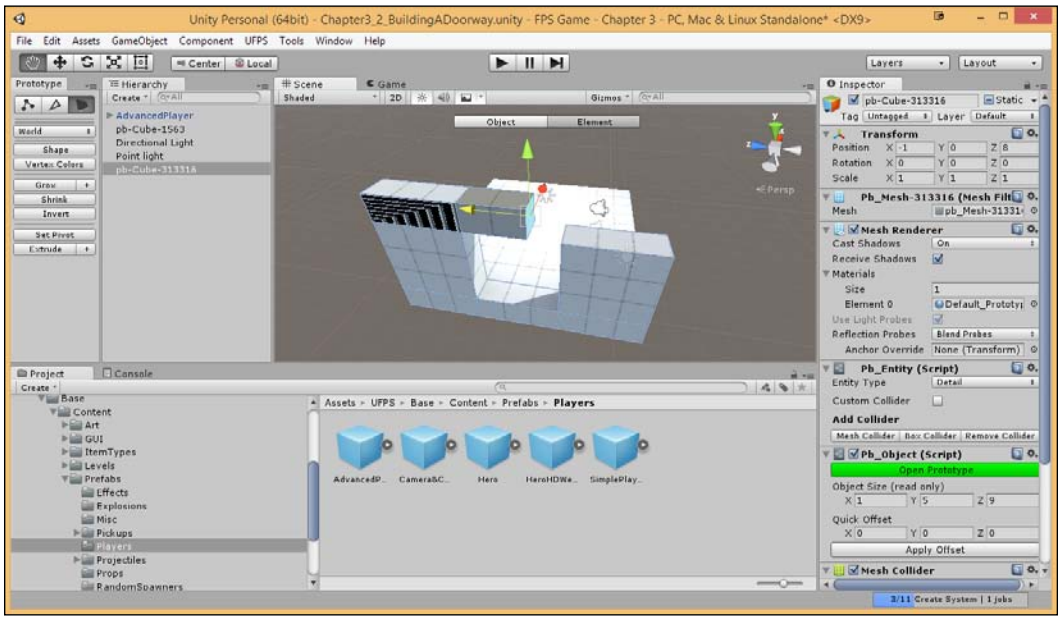
4. Hold down the *Shift* key to extrude and pull it forward a bit. Then, hold down *Ctrl* to start snapping again and pull it so it is `3` units wide again. Finally, perform the same steps again to reach the end of the area. If all goes well, you should have it subdivided into three separate pieces that you can select individually.



5. Select the two sides of the wall and pull it up in the same manner used previously, extruding out again to be 3 units tall.

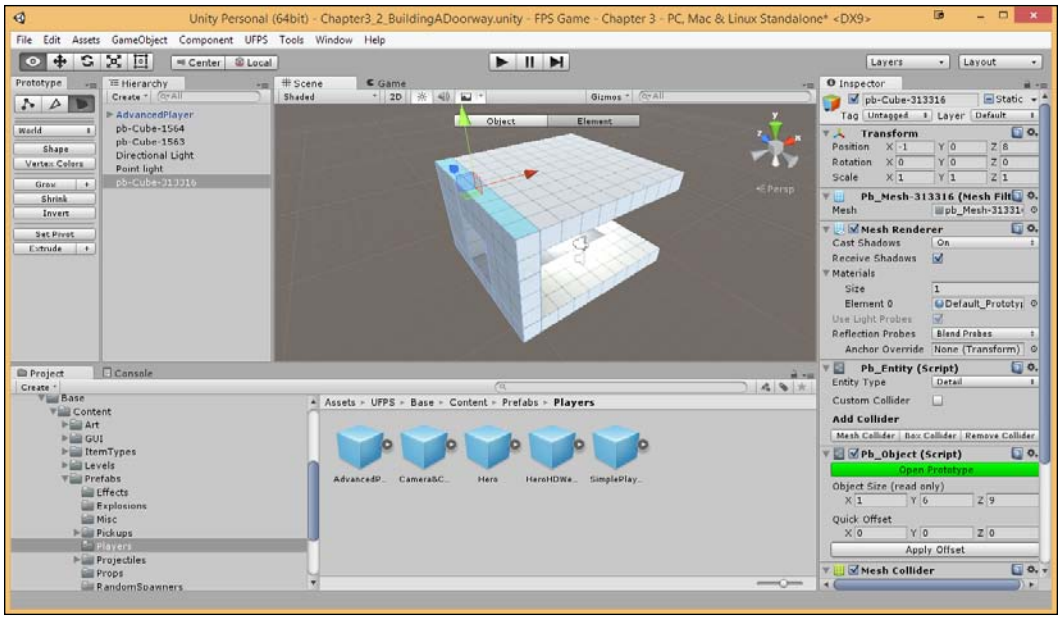


6. Our doorway isn't going to take up the entire wall, so let's extrude one of the edges again and use its side to fill up the rest of the wall to be `4` units tall.
7. Using the center face of the ceiling, extrude using *Shift* + the left mouse button. Drag and then hold *Ctrl* to snap and fill in the top of the door frame.



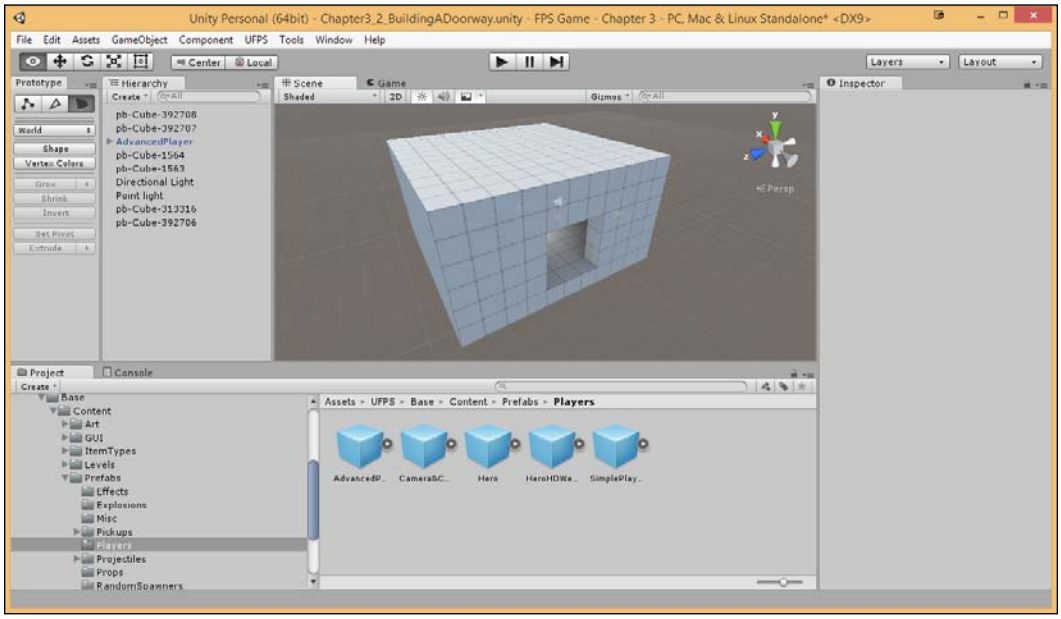
Once you've created this, raise up the top part once again to make it flush with the ceiling we are going to create in the future.

8. Now, let's create a ceiling like the last time by duplicating the floor and moving it to `5` on the **Y** axis, making it `6` units tall.



9. Now that we have the door part of the doorway ready, let's add the rest of the walls to our `room`. We could create a block using *Ctrl* + *K* and then resize it to fit the box, but let's make use of the rotation tool instead.
10. Select the floor object and duplicate it. Hit *E* to go to the Rotation mode. Rotate it on the **Z** axis while holding *Ctrl* to rotate it by 90 degrees with snapping to make it easier to get there.
11. Then, snap the height of the top face down by `4` units to flush it with the ceiling.

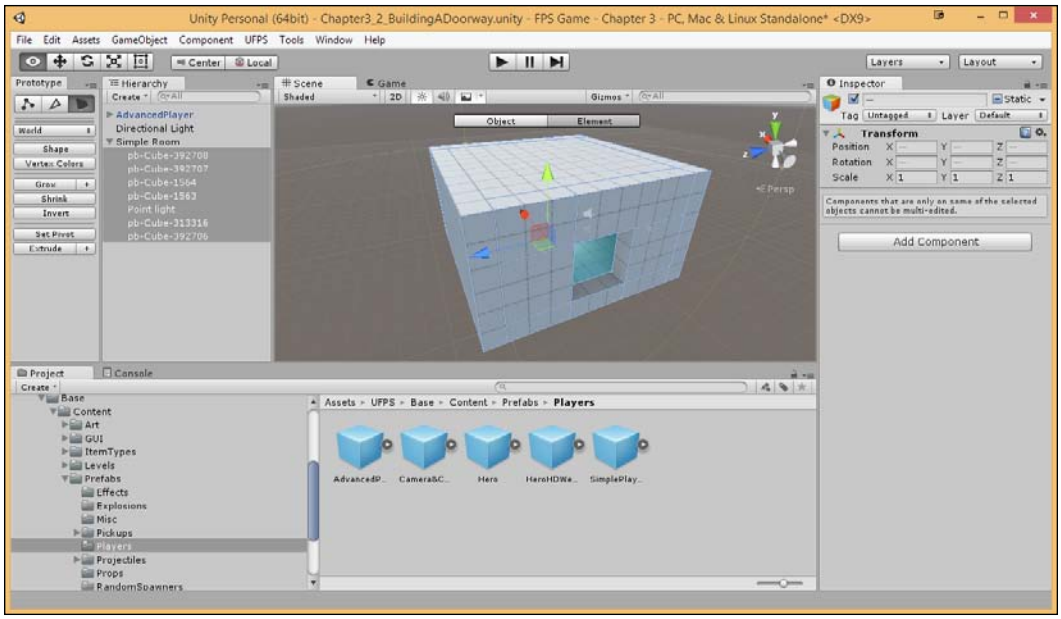
12. Duplicate the newly created wall (*Ctrl* + *D*) and translate (*W*) it to the other side. Finally, do another rotation (*E*) of 90/-90 on the **Y** axis and use the Prototype subselection to pull out the face by `1` unit to flush it with the wall.



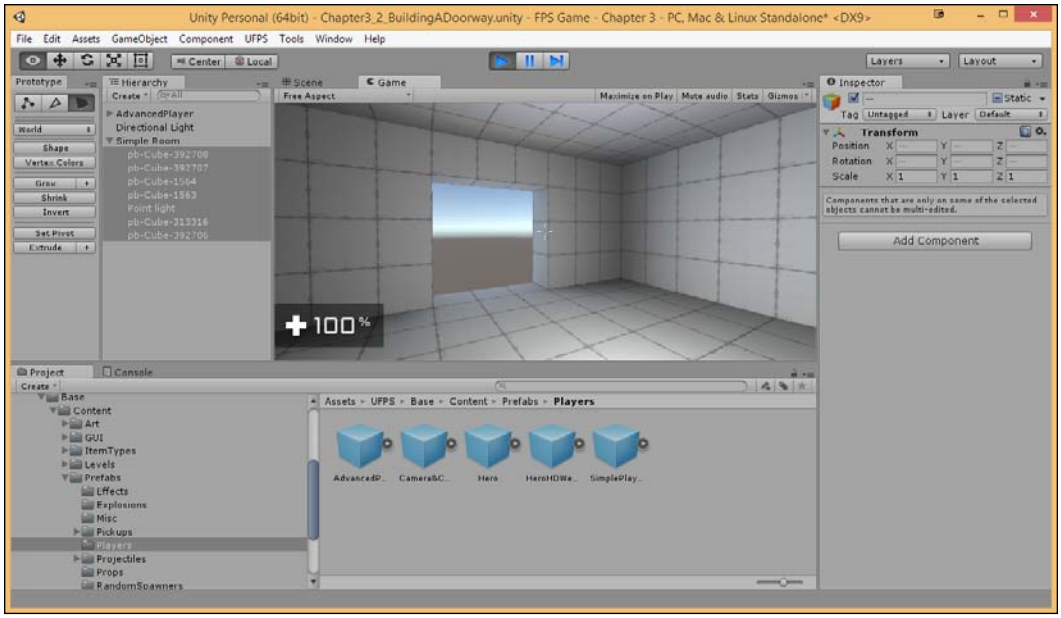
13. Next, for the sake of cleanliness, let's clean up **Hierarchy**. Create an empty game object by going to **GameObject** | **Create Empty**. Reset its position and change the name to `Simple Room`. Then, drag and drop all of the `Cube` objects and `Point light` into it as a child.



For the sake of a clean workspace, you can also rename the child objects to `wall_l`, `wall_r`, `wall_back`, `wall_floor`, `wall_ceiling`, and `wall_doorway`, respectively.



14. Finally, save the level and play the game.



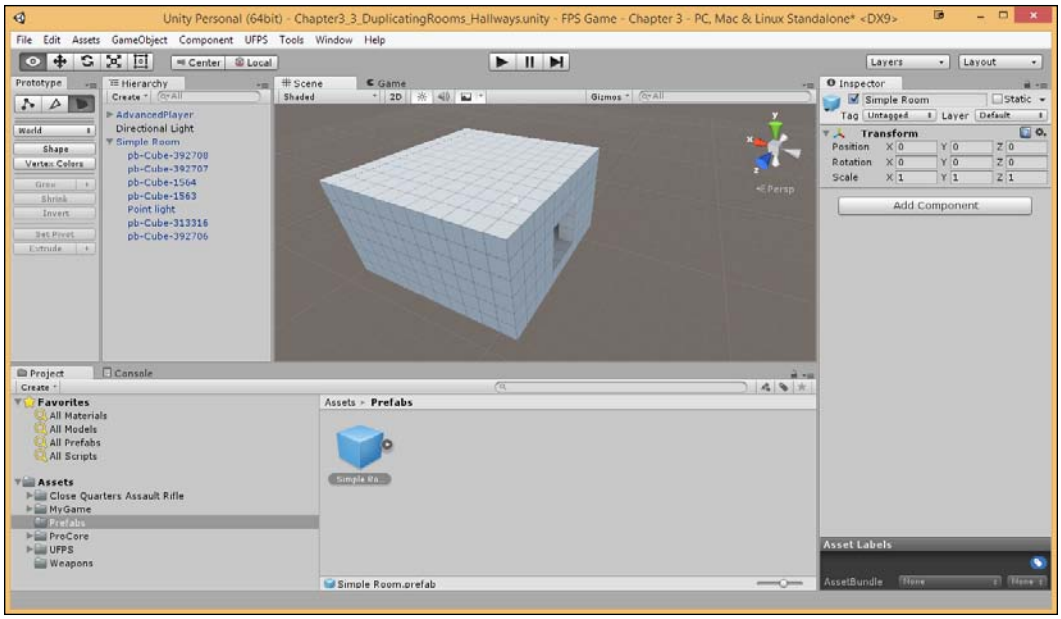
With this, we now have a door.

Duplicating rooms / creating a hallway

Now that we've created a single room, let's see how easy it is to create an additional one.

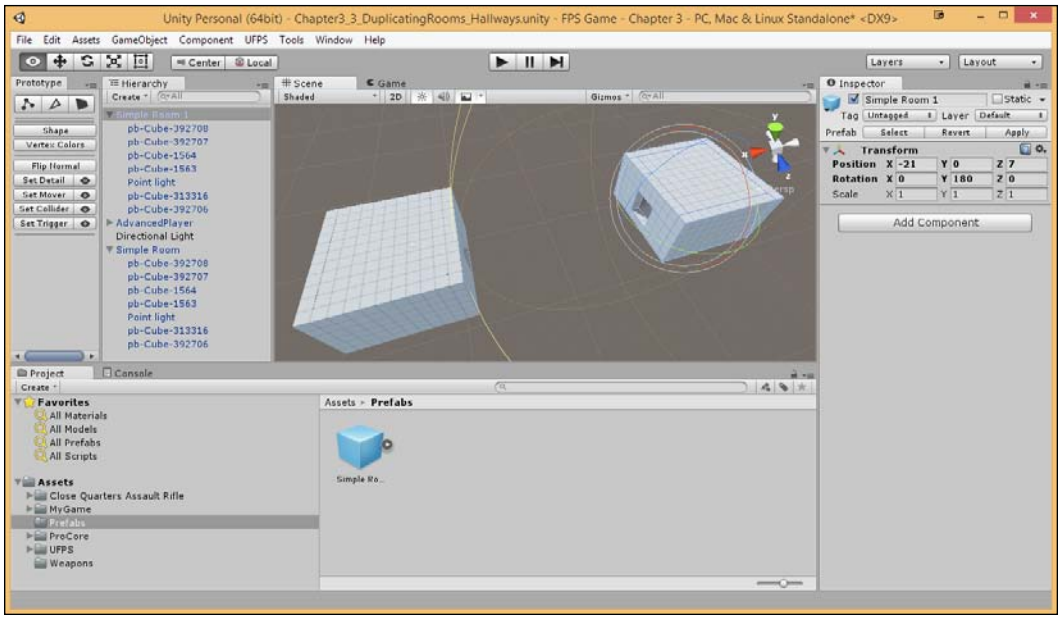
1. In the **Project** tab, select **Create | Folder** and name the new folder `Prefabs`. Move the newly created folder to the `MyGame` folder we created previously and double-click on the folder to enter it.
2. Make sure you are in the Object/Top Level mode and then select the **Simple Room** object from the **Hierarchy** tab. Drag and drop it into our `Prefabs` folder from the **Project** tab. You'll notice that the object in the **Hierarchy** tab will turn blue to indicate that it is a prefab.

   Prefabs or the prefabricated objects are the objects we set aside to make copies of during runtime, such as `AdvancedPlayer` we used previously from UFPS. With this blueprint, we can create as many as we want. When you add a `Prefab` folder to a scene, you will create an instance of it. All of these instances are clones of the object located in our Assets. Whenever you change something in the prefab located in the `Prefab` folder, the changes are applied to all of the objects that are already inside the scene. For example, if you add a new component to a Prefab, all of the other objects we have in the scene will instantly contain the component as well. However, it is also possible to change the properties of a single instance while keeping the link intact. Simply change any property of a prefab instance inside your Scene and that particular value will be bolded to show that the value is overridden; they will not be affected by the changes in the source Prefab. This allows you to modify the Prefab instances to make them unique from their source Prefabs without breaking the Prefab link.

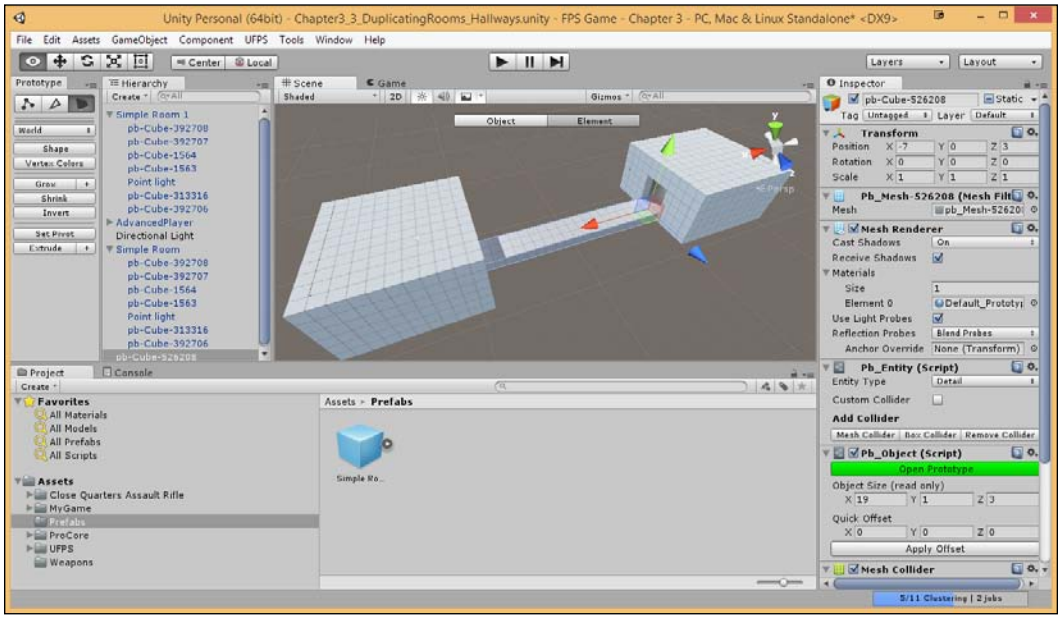


3. After this, drag and drop one of the prefabs into the level, setting the *X*, *Y*, and *Z* values to be similar to the original room to make it easier for us to line it up later.

4. Drag the object over to be away from the old room (I used a **Position** value of `-21, 0, 7`). Switch to the **Rotation** tool and rotate it by 180 degrees on the **Y** axis, making sure that the pivot near the top-left corner of the menu is set to `Center`.



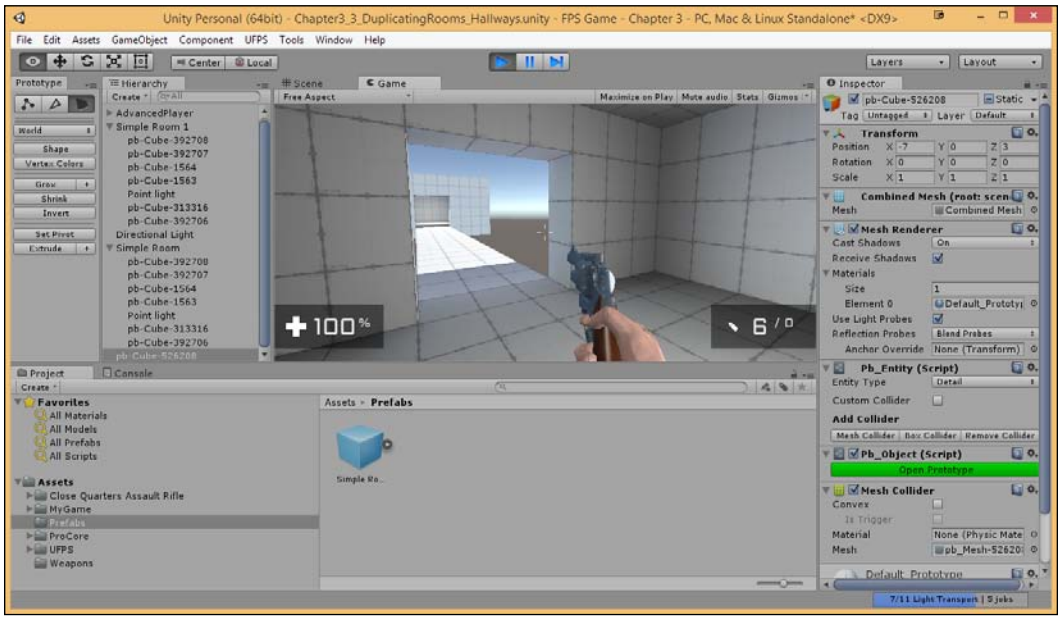
5. Next, in order to connect the rooms together, switch to the **Translate** tool and then switch back to the Geometry/Element mode. Create a new `Cube` and connect them together.





You could also extrude from the bottom of our doorway, but it prevents our ability to modify in the future as needed.

6. Save the level and hit the Play button.



There we go! Now, we have two rooms and a connection between them.

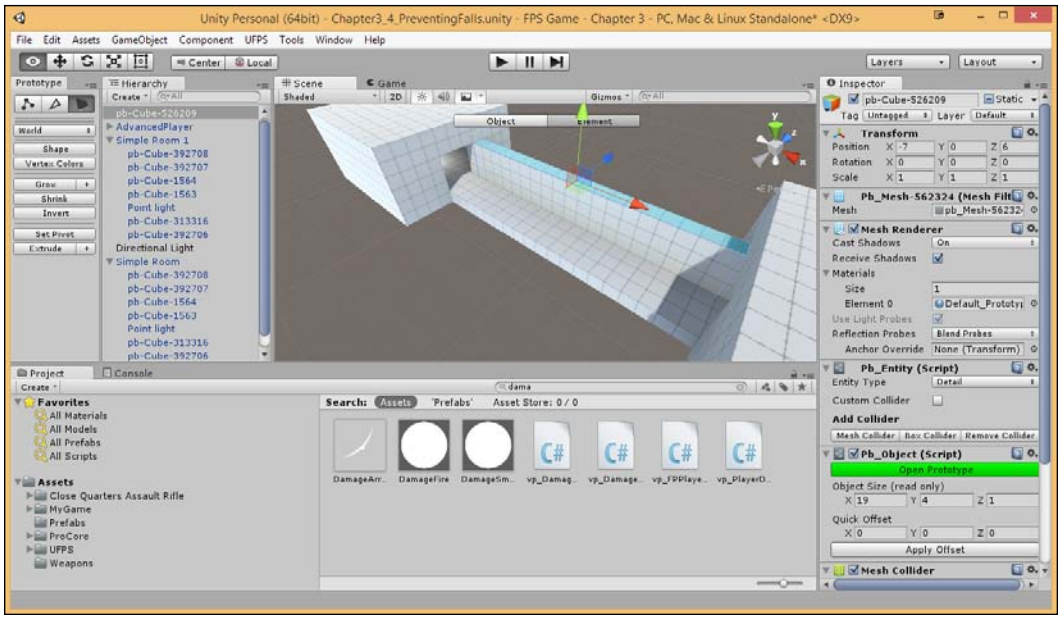


Depending on if its still building or not, in-game rendering might not display all the objects appropriately. If this happens, pause the game, try saving it again, and run it again.

Preventing falls - collision

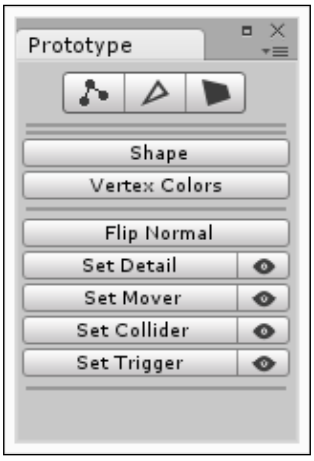
We don't want to have players always be stuck in walls, but we also want to make sure they don't fall down into the void. We can fix this problem with the addition of colliders:

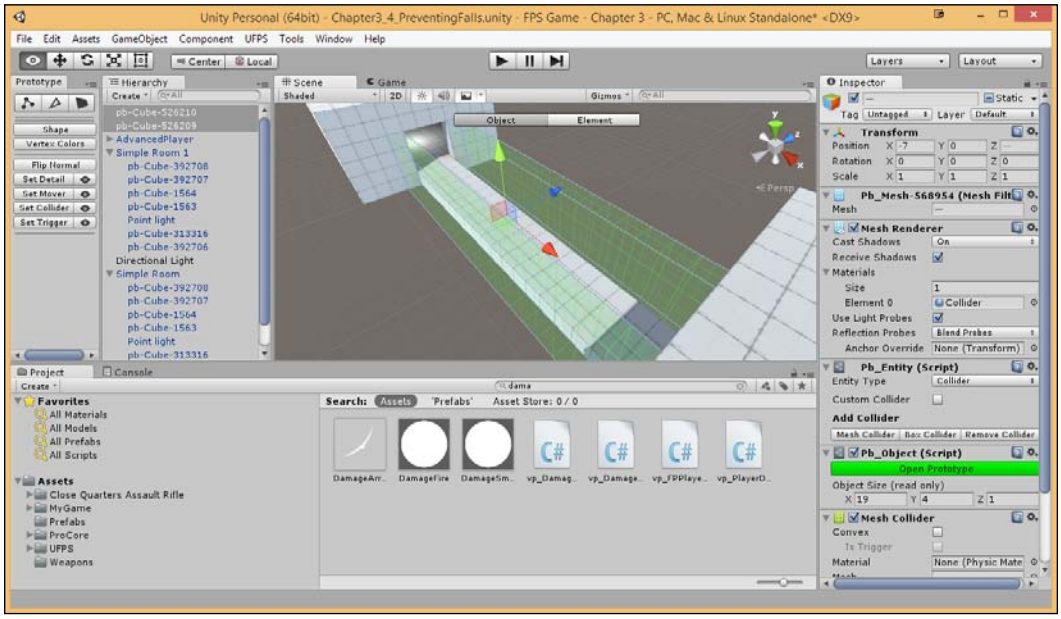
1. Select our "bridge" between the two areas and duplicate the object by hitting *Ctrl* + *D*. Reduce the newly created object's width to `1` unit and have it go up to be `3` units tall.



2. Switch back to the **Object** mode, duplicate the object, and then move it to the other side.

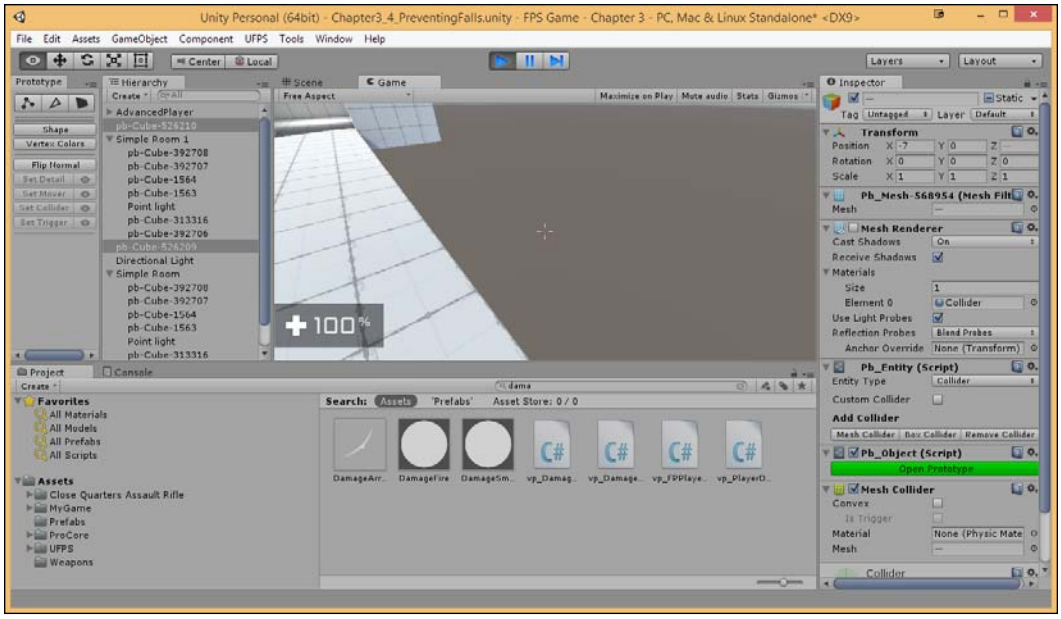
3. Select both the objects and, in the **Prototype** panel, click on the **Set Collider** button.





You'll notice that there is now some nice semitransparent boxes in the areas where our full boxes were. These are colliders or rather volumes that will block the player upon colliding.

4. Save the level and play the game.

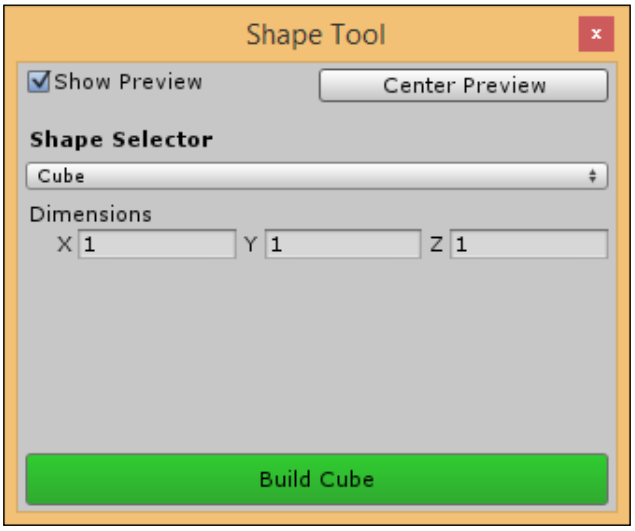


You'll notice that the colliders aren't visible. They, in fact, block us from going out of the level. Perfect!

Adding stairways

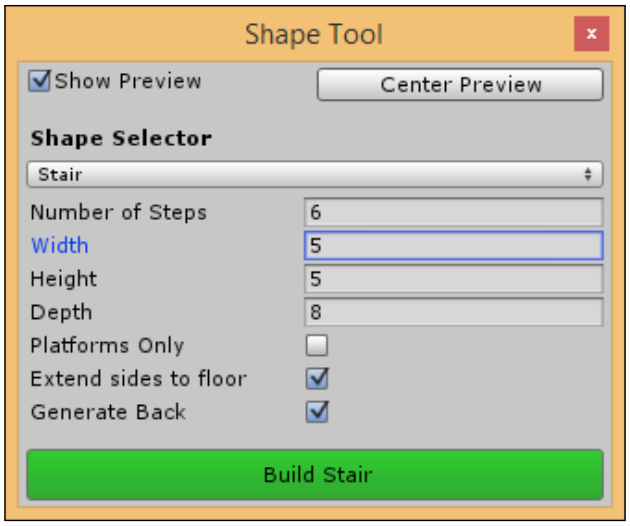
One of the benefits of creating a 3D game is that we can have a vertical element to play, adding in areas for players to snipe or high ground that players would intuitively go to. To help players traverse in these areas, we can use stairways. However, instead of creating the brush from scratch, we can use another tool of Prototype, **Shape Tool**.

1. In the **Prototype** tab, click on the **Shape** button to bring up the **Shape Tool** dialog box.

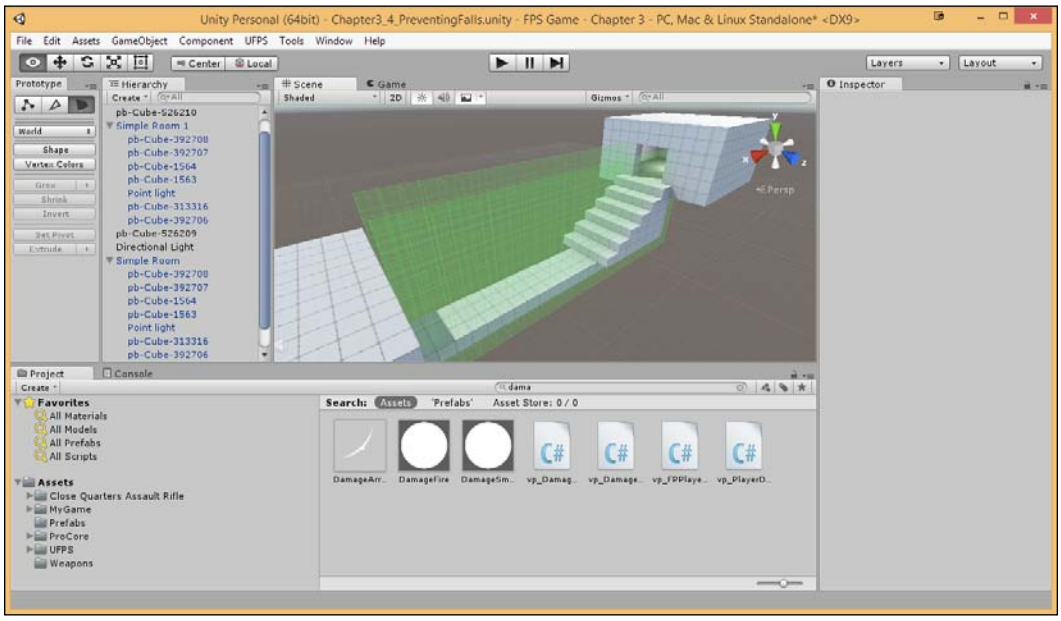


   This menu will allow us to create some commonly used objects in games that we can use to make the building process much simpler.

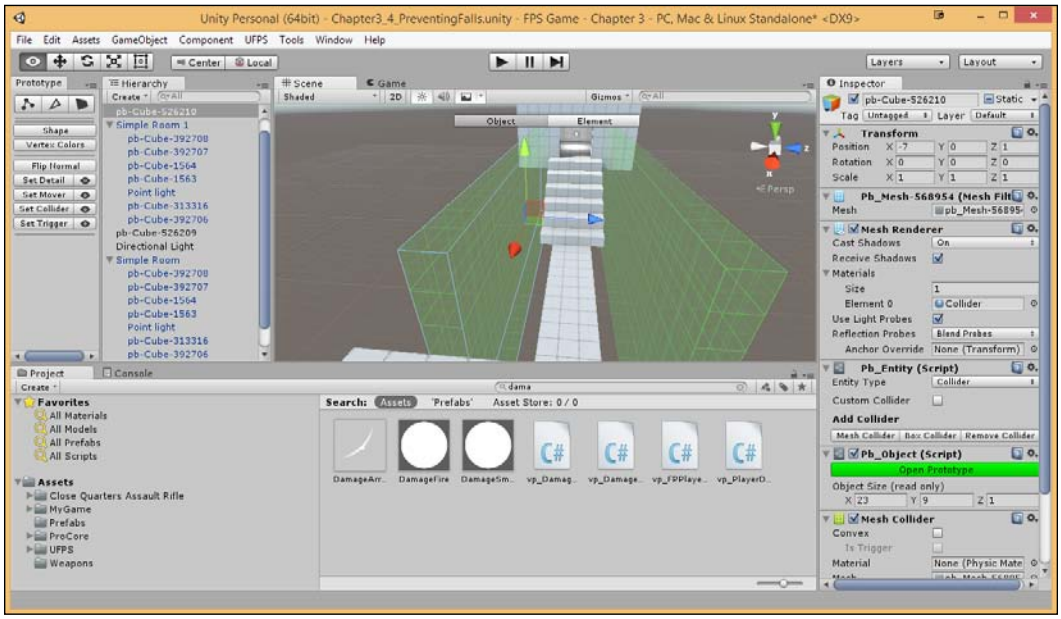
2. In the **Shape Tool** dialog, select the **Shape Selector** dropdown and then select **Stair**. Change the **Width** value to `5` and click on **Build Stair**.



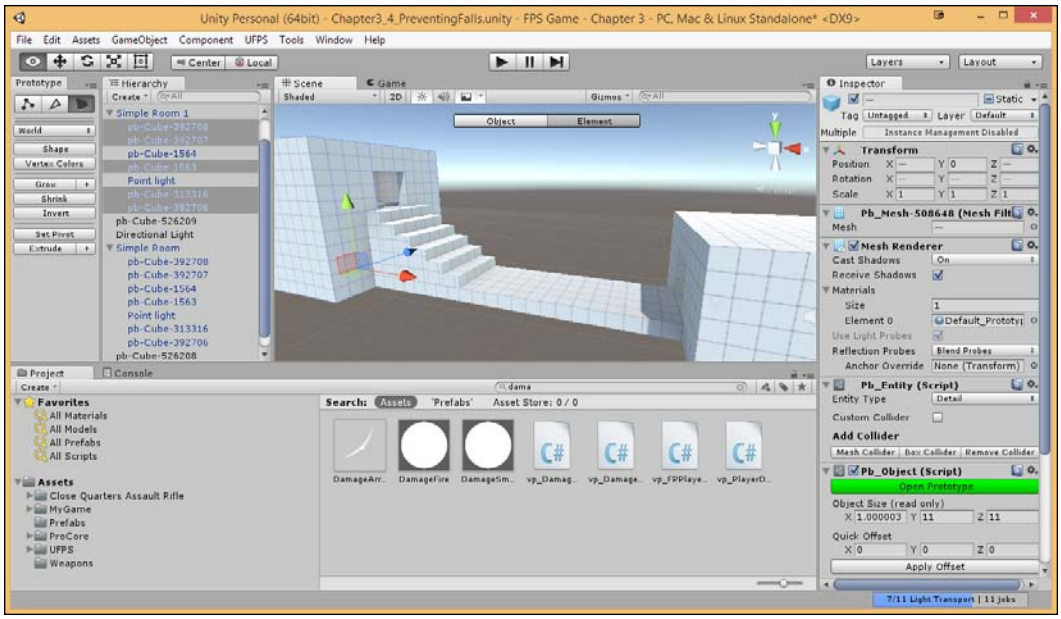
3. After this, click on **X** to close **Shape Tool** and move the stairs to face the first room.
4. Next, raise the other room to fit the stairs and extend the colliders to fit our newly added ground.



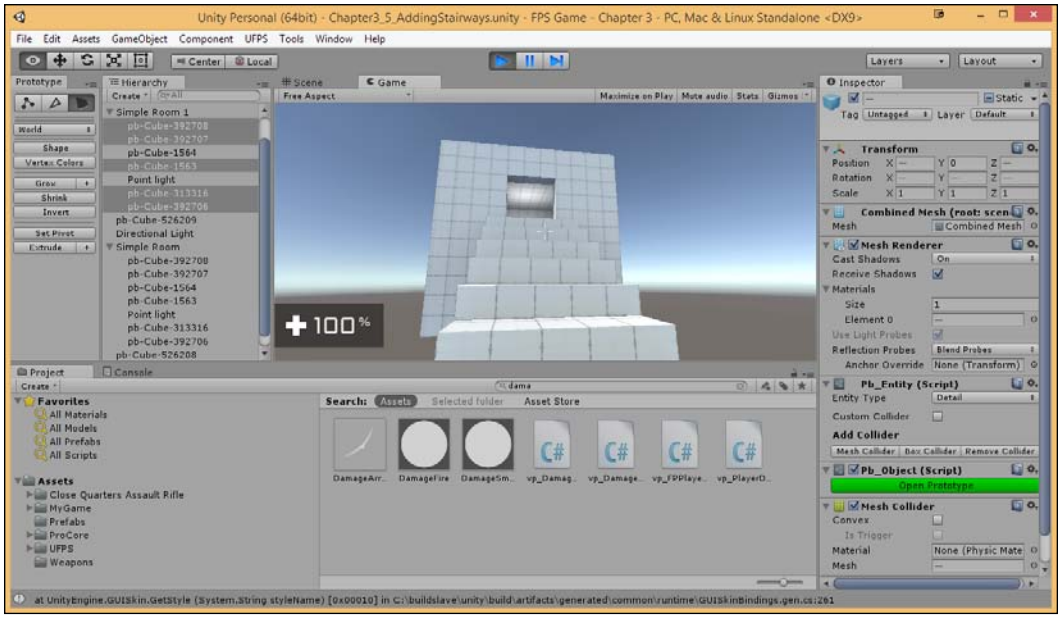
5. Next, switch back to the **Top Level Editing** mode and move the colliders `1` unit away on both sides.



6. Then, select one of the colliders and click on the eye icon to the right of the **Set Collider** button to toggle its visibility. The colliders will still be active; this is just an option to keep them out of the way while we're building the geometry.
7. With this finished, go to the hallway's edges and extend them out. Do the same for the edges in the top room.

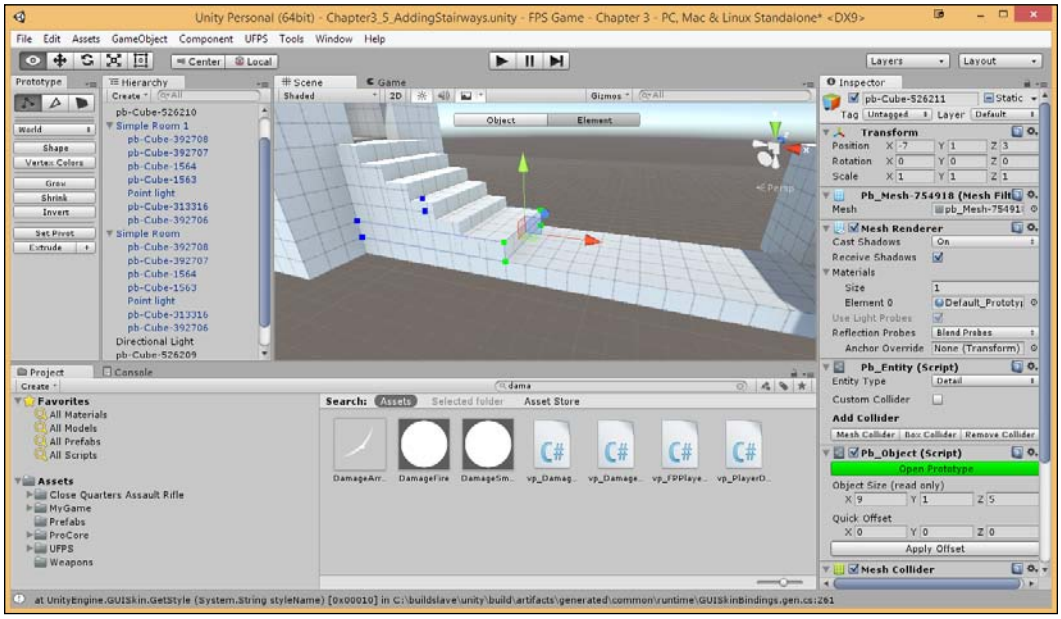


This looks pretty good, but when we go and play the game…



The stairs block us till we jump. Now, we could just rebuild the stairs with more steps; but even this will be jagged, unless there are a lot of steps. Thankfully, we can use colliders in another way.

8. Go back to the **Top Level Editing** mode and duplicate the hallway brush. Move it up by `1` unit on the *Y* axis and reduce its size on the *X* axis by dragging it to start at a distance of 1 unit from the front of the stairs.

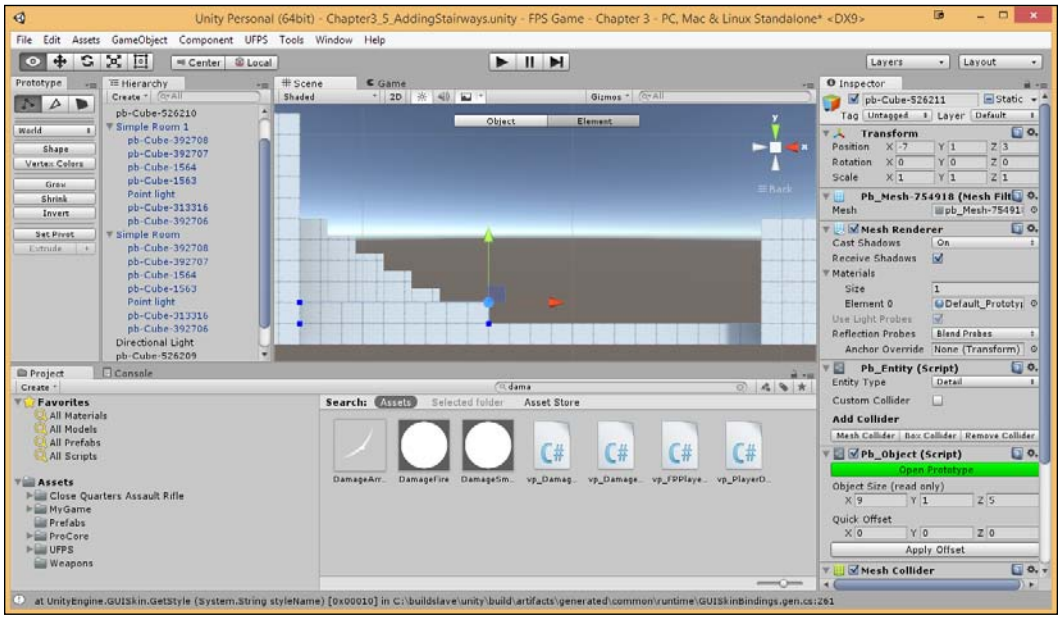


9. From the camera widget in the top-right corner of the **Scene** tab, click on the edge that is closest to the backside of the staircase. To help with the movement of multiple vertices, click on the middle button of our camera widget (also known as the **Scene Gizmo**) to change our camera to the **Isometric** mode.



The Isometric mode allows us to view objects "straight on" so that the pieces on the same spot via a grid overlap rather than be slightly askew like in the Perspective mode. Alternatively, think of it as if you are making a 2D game with all the game being on a plane. While in the Isometric mode, you can hold down the right mouse button and drag to orbit the camera around. You can also hold down *Alt* and the middle mouse button and drag to pan the camera.

For information on the Scene Gizmo, check out `http://docs.unity3d.com/Manual/SceneViewNavigation.html`.

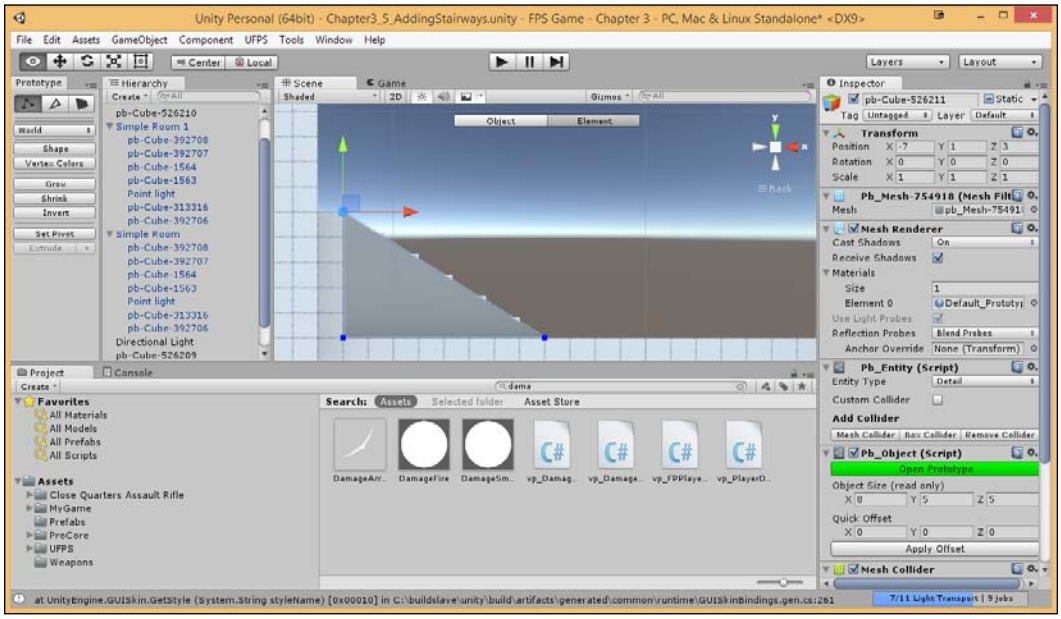


10. From there, do a marquee selection from the top-right vertices of our box. Then, use the **Translate** tool to move it `1` unit down to be flushed with the other vertices.



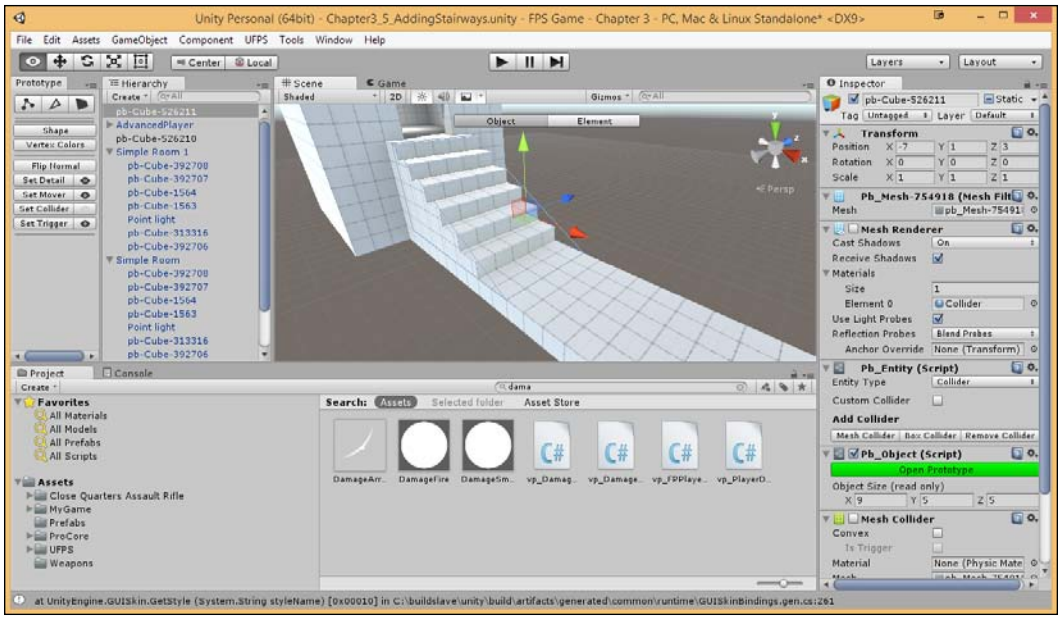
A **marquee selection** is a quick way to select or deselect a group of actors within a certain area in the viewport. This type of selection involves clicking and dragging the mouse to define a box. All the vertices within the box will be selected, including those behind the one on the front.

11. From the left side, select the two vertices on the left side and then move them to the right `1` unit. Next, select the top-left vertices and move them up to flush them with the top stair to create a ramp.

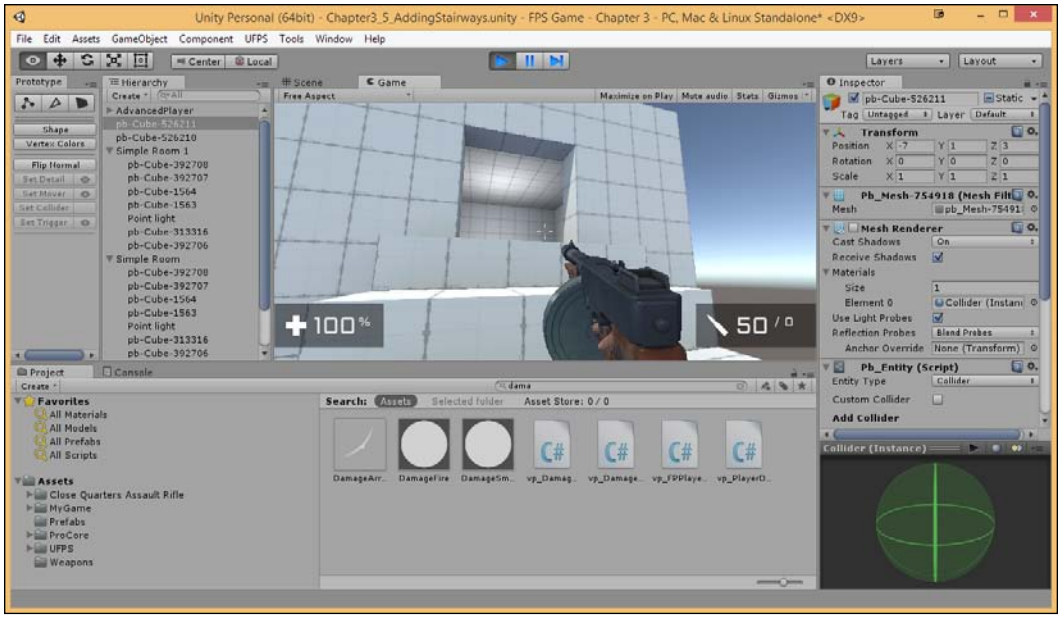


This is starting to look like a nice ramp, but you can still notice the ridges of the steps. This is important to note because, if we play the game now, the player will have to stop at these points and jump to continue to move up.

12. Drag the right side over until all of the edges are gone. When you are finished, click on the camera gizmo again to switch back to **Perspective**. Switch back to the **Object** mode and then, with the ramp selected, click on **Set Collider**.



13. Save your level and play the game.



Coloring your world

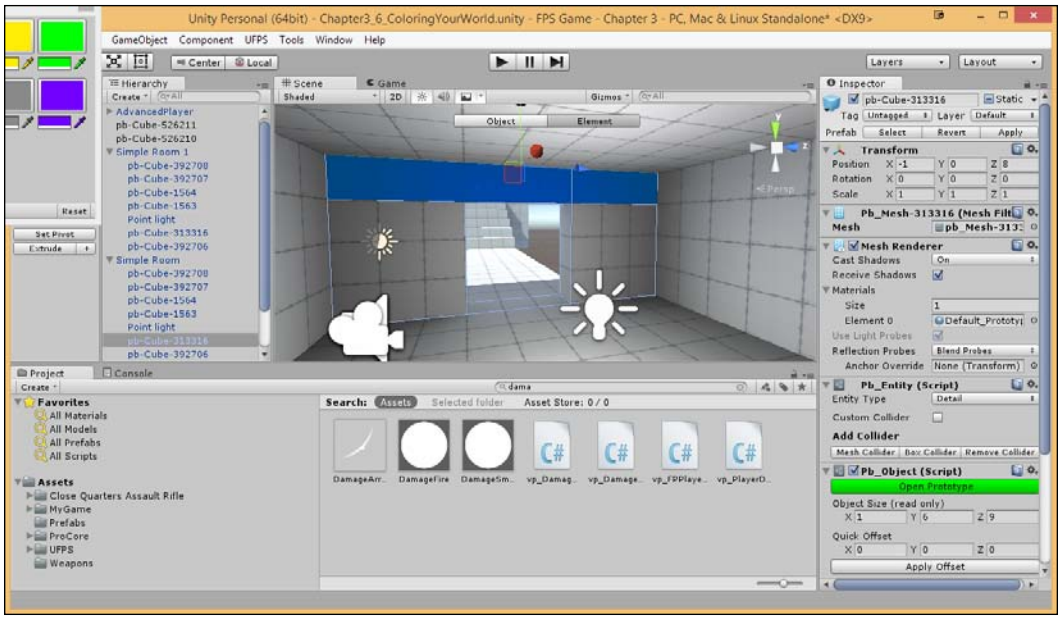
This is looking good. We've got a nice foundation to build upon, but it's pretty barren. We can't use textures yet, as it's a ProBuilder-only tool. But we can apply colors to our walls to help differentiate the areas. Let's see how we can do this now.

1. Go back to our first room and select the top section of our doorway walls. Then, click on the **Vertex Colors** button from the **Prototype** tab.



   This will bring up an example number of colors, but we can customize them by clicking on the bottom section.

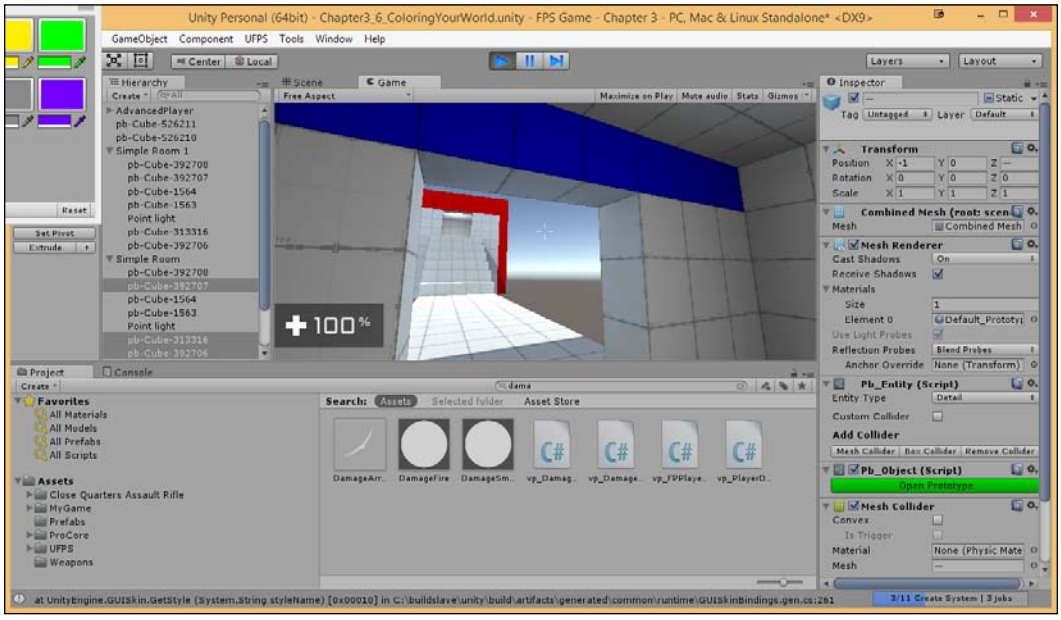
2. Now, with the faces selected, click on the blue button.



3. With this, we have some color. Let's add some color to the other room from the outside. Select the sides and top of the room and click on the red button.



4. Then, do the same to the other sides for both of the changes we made.
5. Save the level and start the game.



There, we have it! This was a simple example; as you build your levels in the future, this can be a good way to help players mark certain areas or help break issues with only using white. However, you'll need to extrude the faces you want to color, unless you want the entire wall to be colored!

Summary

We've now gone over most of the features of Prototype, giving a good basis on which you should be able to create a first-person shooter level using interior spaces.

For those wanting to see more of ProBuilder/Prototype in action, here is a video series involving Gabriel Williams creating the famous E1M1 map from the original *Doom* game. `https://www.youtube.com/watch?v=f2ia28kSiLs&list=PLrJfHfcFkLM9dWnj31b8XTdePDLIOWk7e`

In the next chapter, we will delve into creating exterior areas, making use of Unity's terrain features.

4
Creating Exterior Environments

In the previous chapter, you learned how to create areas quickly, making use of Prototype. Now, this is a great tool when you're trying to create something man-made, such as houses, streets, and space stations. But for more organic things, such as hills, this can lead to issues. In this chapter, we are going to explore ways to add more organic-feeling areas to our level.

This project will be split into a number of tasks. It will be a simple step-by-step process from the beginning to the end. Here is the outline of our tasks:

- How to create a Terrain
- How to add textures to our Terrain
- Adding reflective water
- Adding trees
- Adding grass for details
- How to build the atmosphere of our levels

Prerequisites

Before we start, we need to have a project created that already has UFPS and Prototype installed. If you do not have these already, follow the steps described in *Chapter 1, Getting Started on an FPS*.

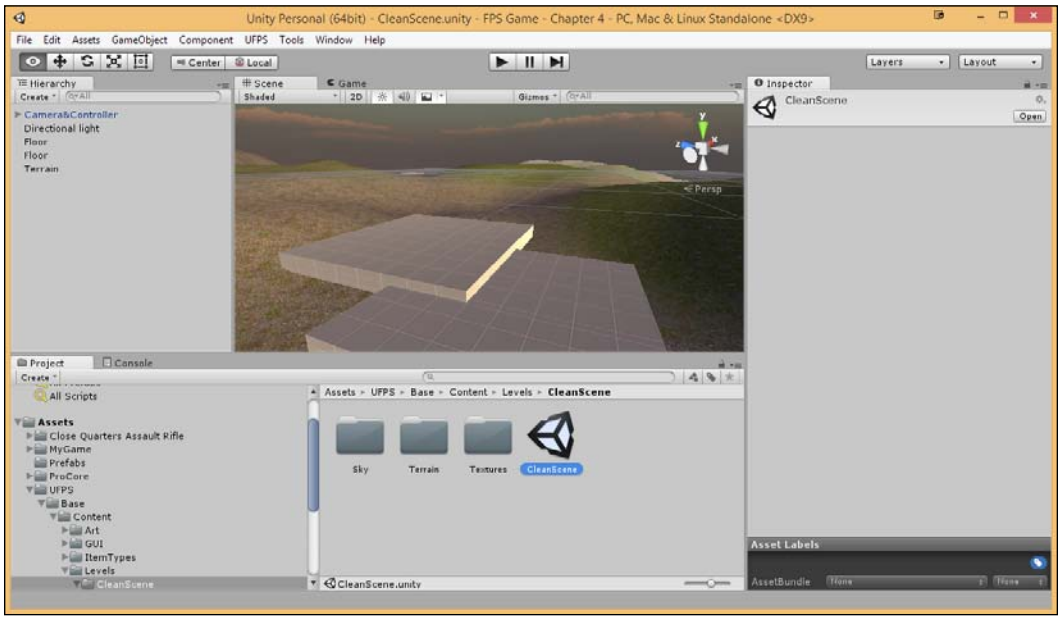
Introduction to Terrain

Terrain is basically anything that is nonman-made such as hills, deserts, mountains, and so on. Unity's way of dealing with Terrains is different than what most engines do in the fact that there are two ways to make them, one being to use a height map and the other being to sculpt from scratch.

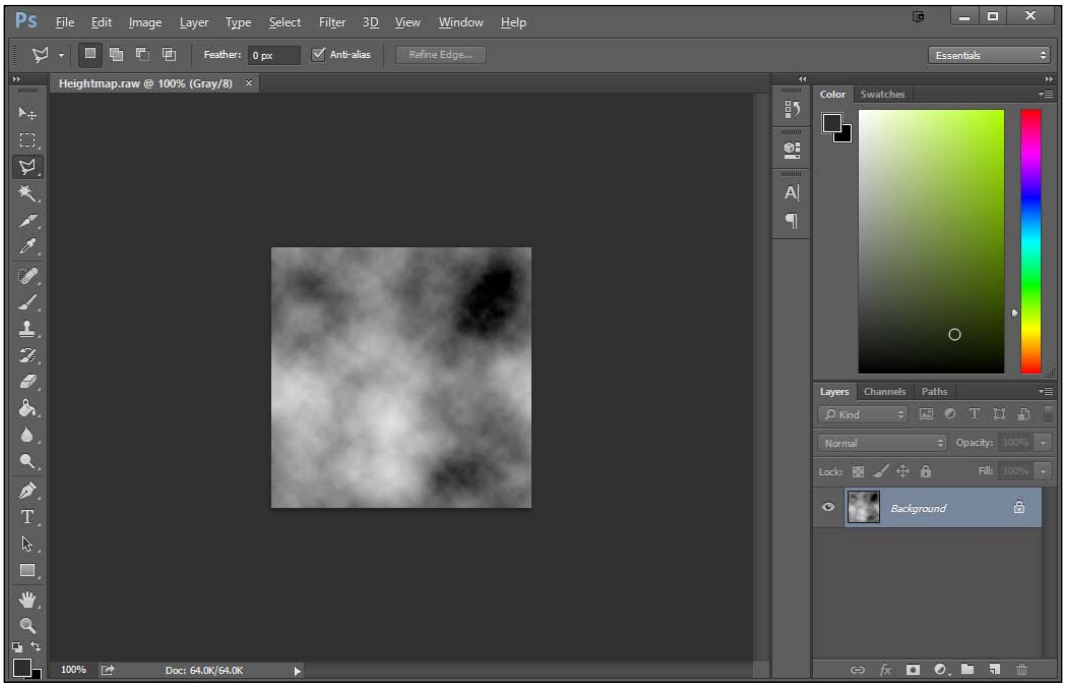
Height maps

Height maps are a common way for game engines to support Terrain. Rather than creating tools to build Terrain within the level, the artist/designer can use a graphic designing software (such as Adobe Photoshop) to create a greyscale image. These values are then interpreted by the software to calculate its dimensional height based on lightness or darkness. We can translate the image into a Terrain by using this height map.

We've already seen an example of a height map in `CleanScene` that we used in *Chapter 2*, *Building Custom Weapons*. You'll notice that there is an object called `Terrain` in the **Hierarchy** tab. This object encompasses all the bumps and hills within the scene.



In the `UFPS/Base/Content/Levels/CleanScene/Terrain` folder, you'll notice that there is a file called `Heightmap`, which is of the RAW type. Unity can load this data in and produce the following hills scene. If we were to open this file in Adobe Photoshop, it would appear like the following:

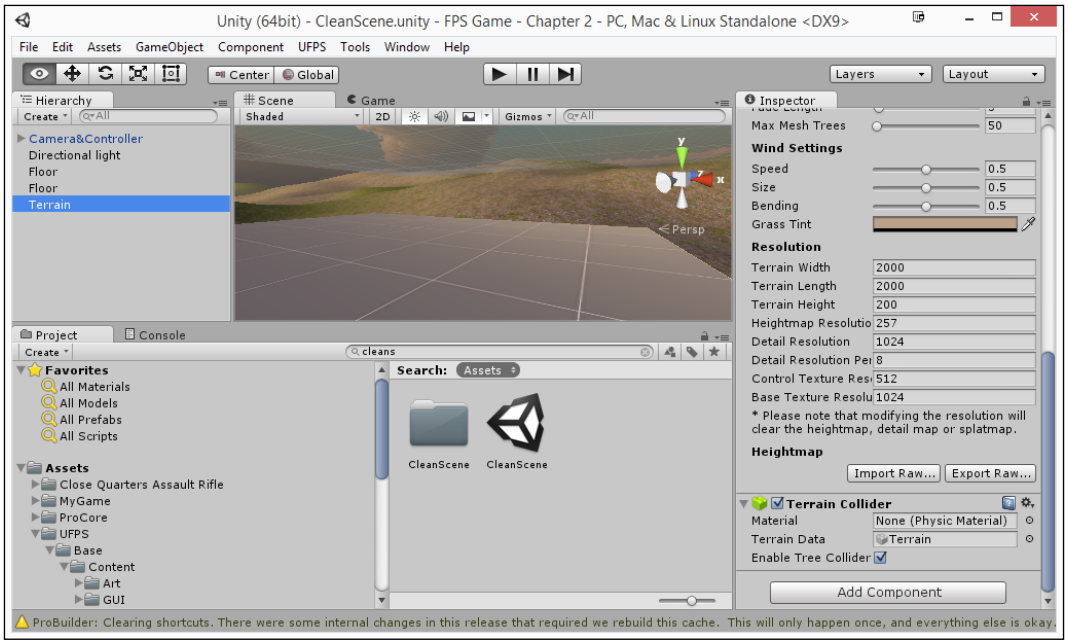


The lighter values of the colors within **HeightMap** are calculated to make the Terrain taller in 3D space, and dark grey or black values are calculated to push the Terrain lower in 3D space. Generally speaking, black is the lowest and white is the highest in terms of creating the altitude of our Terrain.



The **Terrain Height** property sets how high white actually is in comparison to black.

In order to apply a height map to a Terrain object, inside the **Terrain** component, click on the **Settings** button and scroll down to **Import Raw…**.





For more information on Unity's Height tools, check out `http://docs.unity3d.com/Manual/terrain-Height.html`.

If you want to learn more about creating your own HeightMaps using Photoshop, check out `http://worldofleveldesign.com/categories/udk/udk-landscape-heightmaps-photoshop-clouds-filter.php`. While this tutorial is for UDK, the area in Photoshop is the same.

Others also use software such as Terragen to create heightmaps. More information on this is available at `http://planetside.co.uk/products/terragen3`.

Hand sculpting

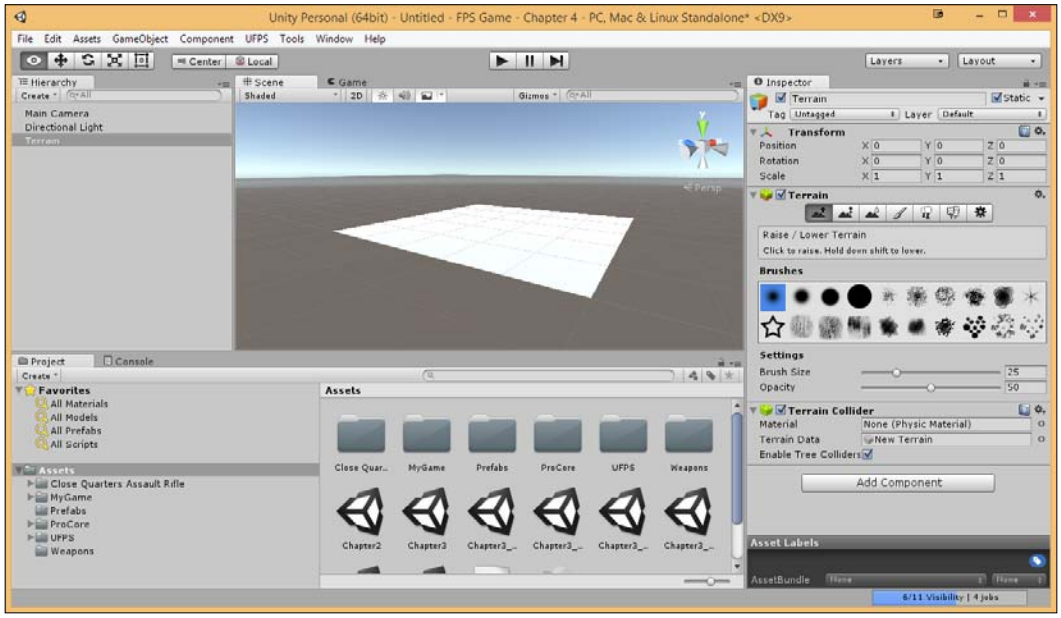
The other way of creating the Terrain is to do so by hand. This allows us to have everything exactly as we want it, and is the way we will be doing it in this chapter.

Creating the Terrain

In Unity, the best tool to use to create a natural landscape is the terrain tool. Unity's terrain system allows us to sculpt and shape the landscape of our level. This tool is frequently used to create outdoor environments, because the ground is never completely flat in nature. This tool will help the artist create organic and asymmetrical details as well as a realistic ground plane for your player to walk on. After the Terrain is sculpted and formed, we can complete our environment by adding in bushes, trees, and fading materials.

To see how easy it is to use the tool, let's get started with creating the Terrain:

1. First, let's create a new scene from scratch by going to **File | New Scene**.
2. Next, we need to actually create the Terrain we'll be placing for the world. Let's first create a Terrain by selecting **GameObject | 3D Object | Terrain**.



At this point, you should see the Terrain.



If for some reason you have problems seeing the Terrain object, go to the **Hierarchy** tab and double-click on the terrain object to focus your camera on it and move in as needed.

Right now, it's just a flat plane, but we'll be doing a lot to it to make it shine. If you look to the right with the terrain object selected, you'll see the Terrain editing tools that do the following (from left to right):

- **Raise/Lower height**: This will allow us to raise or lower the height of our Terrain in a certain radius to create hills, rivers, and more.
- **Paint height**: If you already know the exact height a part of your Terrain needs to be, this tool will allow you to paint a spot at that location.
- **Smooth height**: Averages the area that it is in and attempts to smoothen the areas and reduce the appearance of abrupt changes.
- **Paint texture**: Allows us to add textures to the surface of our Terrain. One of its nice features is the ability to lay multiple textures on top of each other.
- **Place trees**: Allows us to paint objects in our environment that will appear on the surface. Unity attempts to optimize these objects by billboarding distant trees, so we can have dense forests without having a horrible frame rate.
- **Paint details**: In addition to trees, you can also have small things such as rocks or grass covering the surface of your environment by using 2D images to represent individual clumps and using a bit of randomization to make it appear more natural.
- **Terrain settings**: Settings that will affect the overall properties of the particular Terrain; options such as the size of the Terrain and wind can be found here.

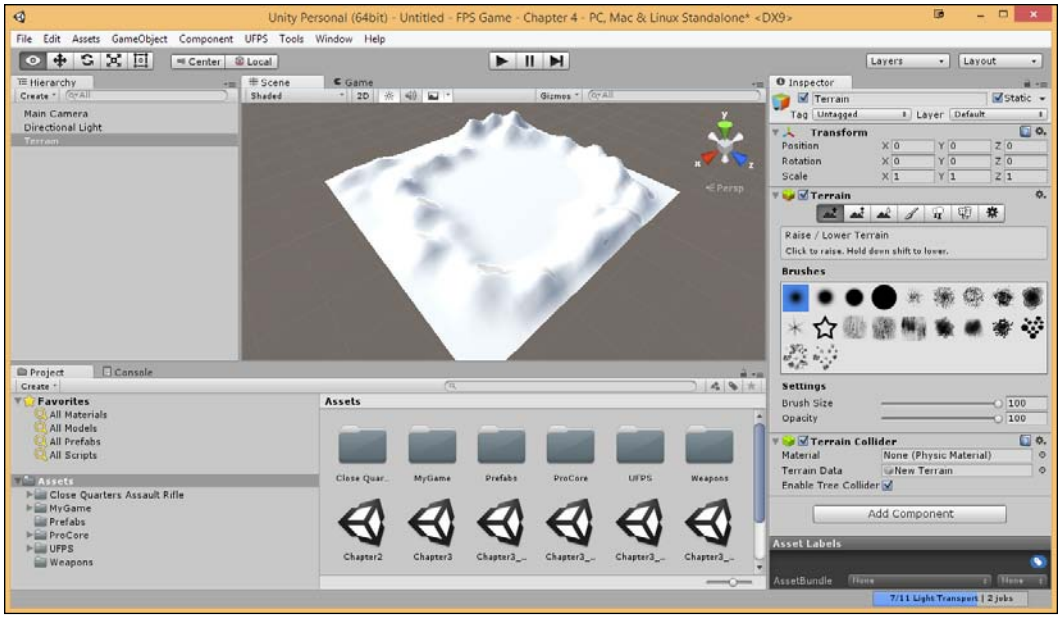
By default, the entire Terrain is set to be at the bottom. But we want to have ground above and below us, so we can add in things such as lakes.

3. With the Terrain object selected, click on the second button to the left of the Terrain component (the Paint height mode).



4. From there, set the **Height** value under **Settings** to `100` and then press the **Flatten** button. At this point, you should see the plane moving up, so now everything is above ground level by default.
5. Next, we are going to create some interesting shapes in our world with some hills by painting on the surface. With the Terrain object selected, click on the first button to the left of our Terrain component (the **Raise/Lower Terrain** mode). Once this is completed, you should see a number of different brushes and shapes that you can select from.

   Our use of Terrain is to create hills in the background of our scene, so it does not seem like the world is completely flat.
6. Under **Settings**, change the **Brush Size** and **Opacity** value of your brush to `100` and click around the edges of the world to create some hills. You can increase the height of the current hills by clicking on the top of the previous hills.





While creating hills, it's a good idea to look at them from multiple angles, so you can make sure that none are too high or too short. Generally, having tall hills around the outside of the Terrain prevents the player from seeing where the Terrain cuts off, which is a good thing.

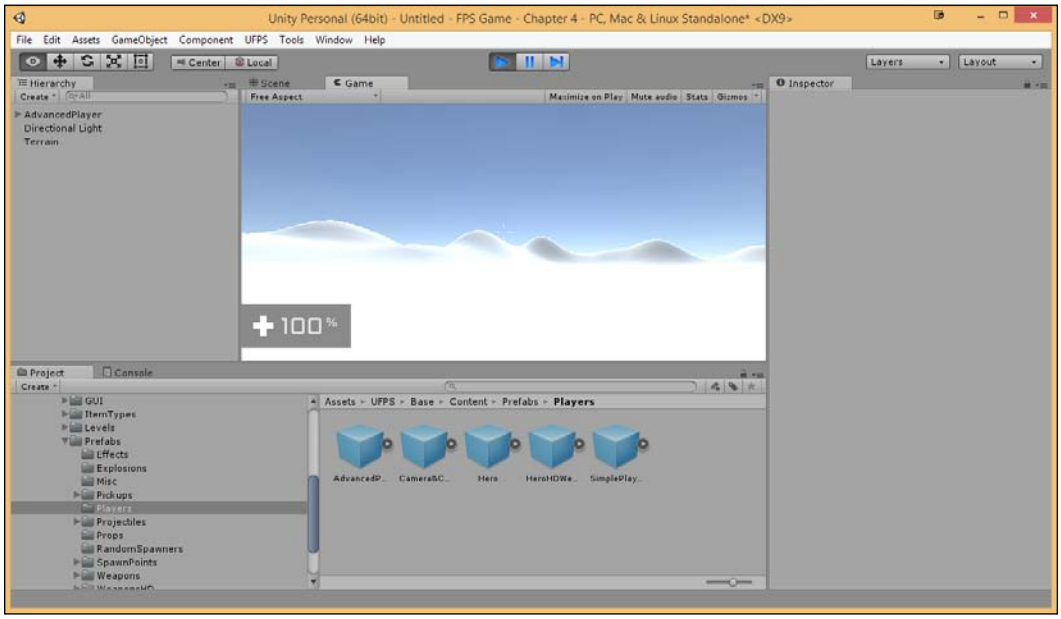
In the **Scene** view, to move your camera around, you can use the toolbar at the top-right or right-click and drag the mouse in the direction you want the camera to move in, pressing the **WASD** keys to pan. In addition, you can hold down the middle button and drag the mouse to move the camera around. The mouse wheel can be scrolled to zoom in and out from where the camera is.

Even though you should plan the level ahead of time on something like a graph paper to plan encounters and so on, you should avoid making the level entirely from above, as when the player is actually in the game, they will not see it that way at all (most likely). Referencing the map from the same perspective of your character will help ensure that the map looks great.

To see many different angles at the same time, you can use a layout with multiple views of the scene such as *4 Split*, which you can go to by clicking on the **Layout** dropdown menu at the top-right of the screen and then selecting it.

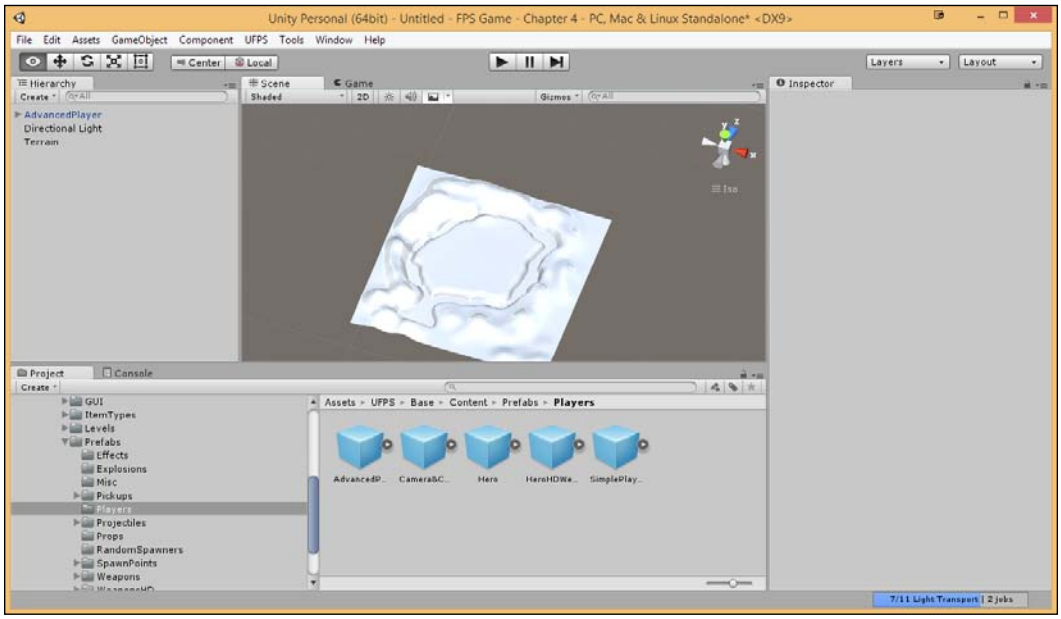
7. Of course, we should see how the Terrain looks in the game with our player first. So, we select the **Main Camera** object in our **Hierarchy** tab and delete it by pressing the *Delete* key.

8. Go to the `UFPS/Base/Content/Prefabs/Players` folder and drag and drop an **Advanced Player** prefab into our world. Place it into the scene and then select it to ensure that the collider is above the Terrain to avoid collision problems, causing the player to fall through the map. Now, if you play the game, you'll get a good view as to how the level currently looks.



9. Once our land is ready, we can create some holes in the ground to fill water in later. This will provide a natural barrier in our world that players will know they cannot pass.

10. To do this, we first need to go back to the **Raise/Lower Terrain** mode. We then create a moat by changing the **Brush Size** value to `50` and then holding down the *Shift* key and clicking around the middle of our texture. In this case, it's okay to use the top view; remember that this will eventually be water to fill in the lakes, rivers, and so on.



To make it easier to see, you can click on the sun-looking light icon in the **Scene** tab to disable the lighting's settings for the time being. Alternatively, if you'd still like to see shadows, you can select the **Directional Light** object in the **Hierarchy** tab and decrease the light's intensity by about 25%.



At this point, in a traditional studio, you'd spend time playtesting the level and iterating on it before an artist or you took the time to make it look great. However, in this case, we want to create a finished project as soon as possible. While designing your own games, be sure to play your level and have others play your level before you polish it.

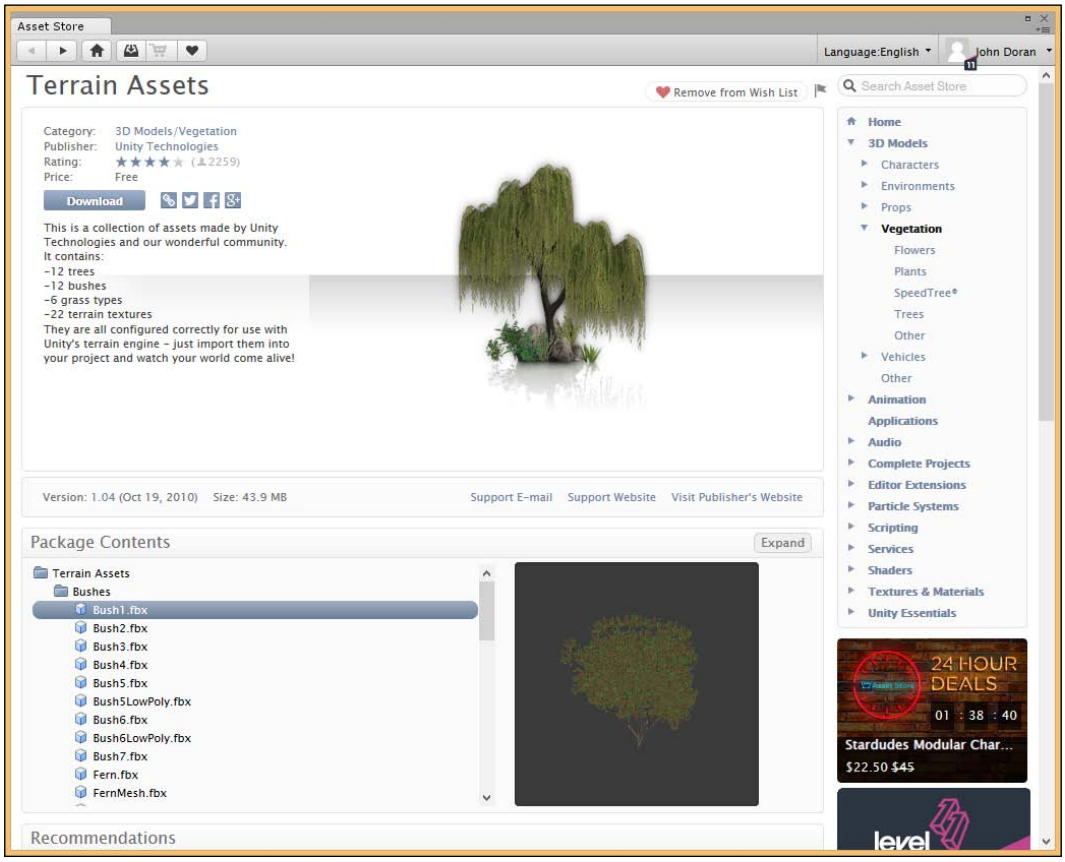
For more information on greyboxing, check out `http://www.worldofleveldesign.com/categories/level_design_tutorials/art_of_blocking_in_your_map.php`.

If you are interested in checking out how a level is built in greybox from start to finish, check out this article from PC Gamer, which has images of each step, at `http://www.pcgamer.com/building-crown-part-one-the-first-look-at-the-next-big-counter-strike-go-competitive-map/`. The whole thing is worth a read.

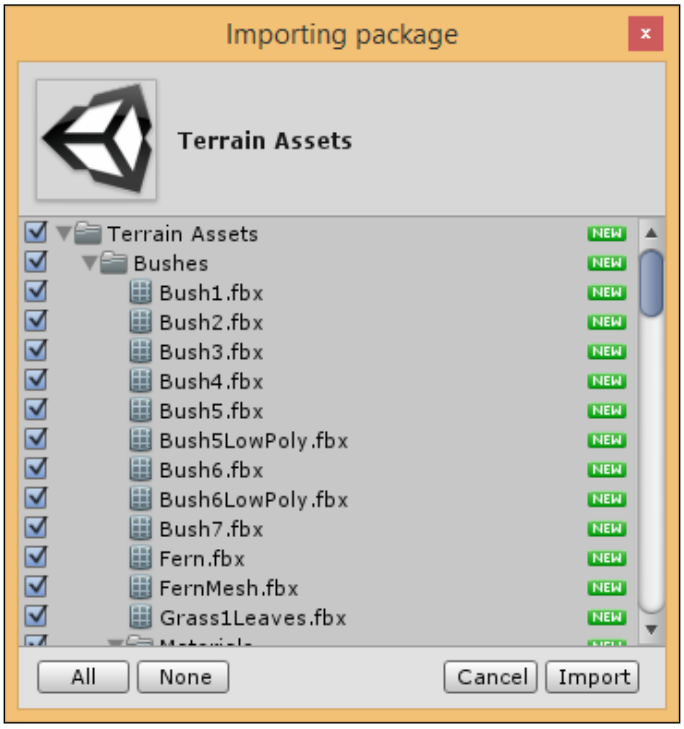
Adding color to our Terrain – textures

This is interesting enough, but it would be quite boring to be in an all-white world. Thankfully, it's very easy to add textures to everything. But first, we need to have some textures to paint onto the world and, for this instance, we will make use of some of the free assets that Unity provides us with. To make use of these assets perform the following steps:

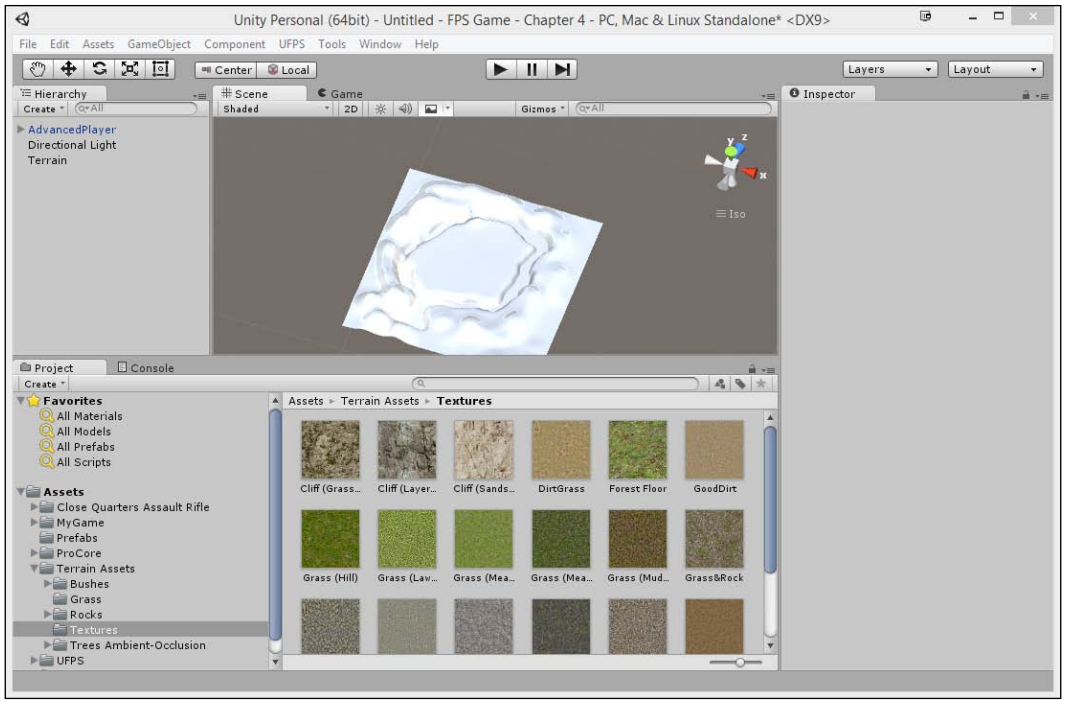
1. To download it, we'll use the **Asset Store** again. So, with this in mind, select **Window** | **Asset Store**.
2. In the top-right corner of the screen, you'll see a search bar where you can type in `Terrain assets` and press *Enter*. Once there, the first asset you'll see is **Terrain Assets** that is released by **Unity Technologies** for free. Click on it and then once in the menu, click on the **Download** button.



3. Once it finishes downloading, you should see the **Importing package** dialogue box popup (if it doesn't pop up, click on the **Import** button, where the **Download** button used to be).

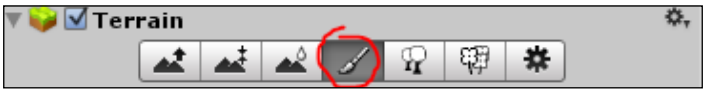


4. Generally, you'll want to select only the assets that you want to use and uncheck the others. But since you're exploring the tools right now, we'll just click on the **Import** button to place them all.
5. Close the **Asset Store** if it's still opened and go back to our **Game** view. You should notice the new `Terrain Assets` folder placed in our `Assets` folder. Double-click on it and then enter the `Textures` folder.



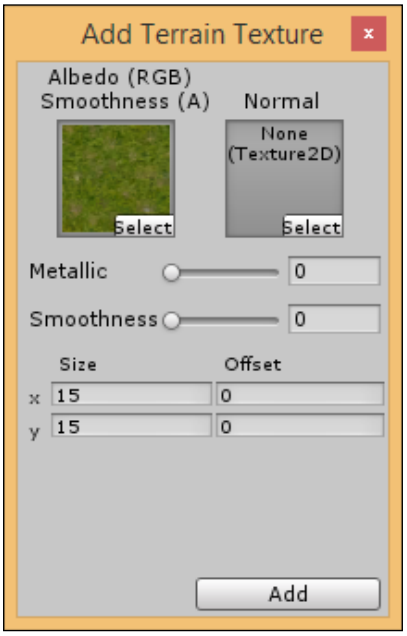
These will be the textures we will be placing in our environment.

6. Select the Terrain object and click on the fourth button from the left (that looks like a paint brush) to select the **Paint Texture** button.



7. Here on, you'll notice that it looks quite similar to the previous sections we've seen. If you look under the **Terrain Assets** section for **Textures**, you will notice that it says **No Terrain textures defined**. So, let's fix this. Click on the **Edit Textures** button and select **Add Texture**.

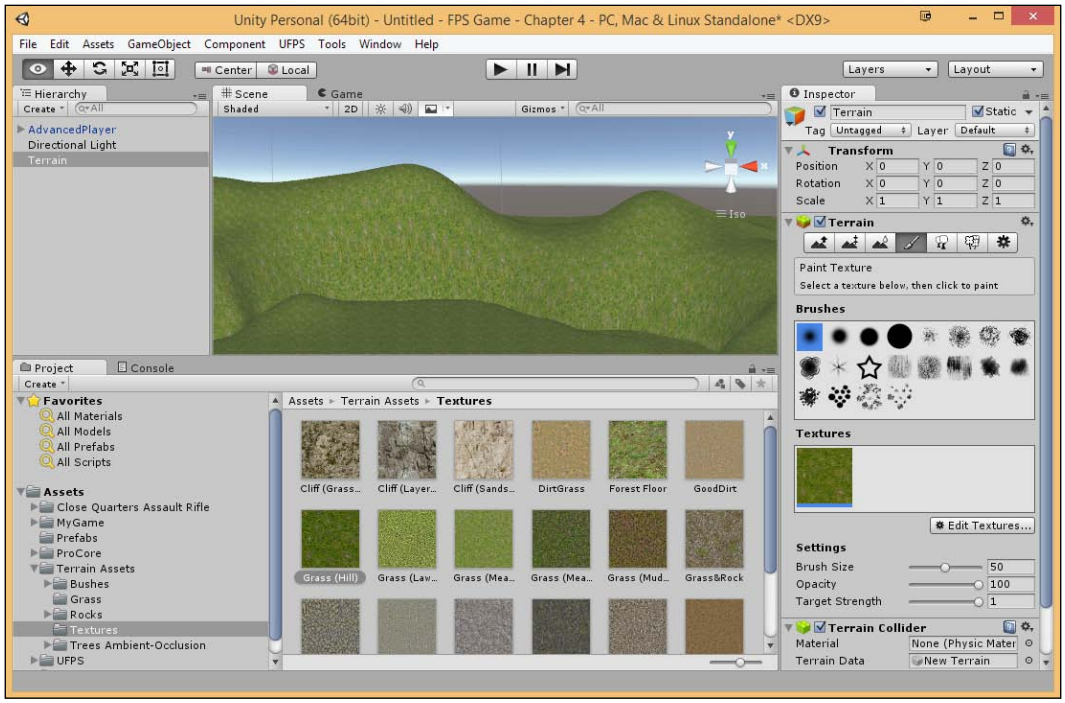
8. You'll see an **Add Terrain Texture** dialogue popup. Under the **Albedo (RGB)** variable, place the **Grass (Hill)** texture (use the search dialog) and then click on the **Add** button.



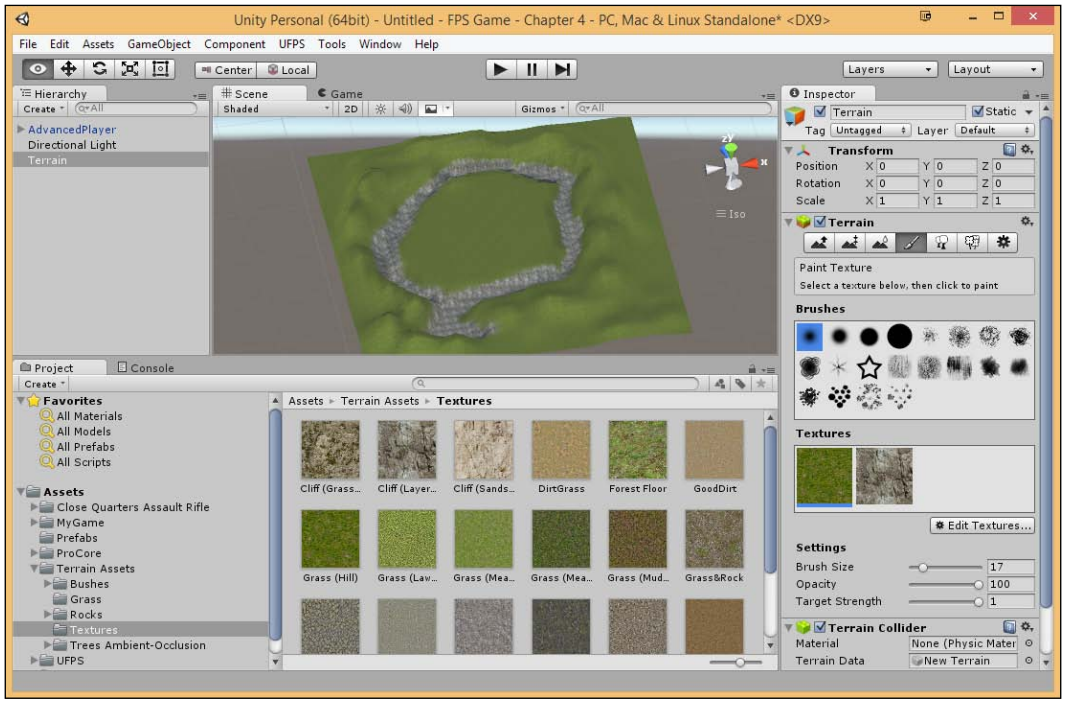
The left texture slot labeled **Albedo (RGB) Smoothness (A)** is just a complex way of saying diffuse/color map. **RGB** stands for the colors red, green, and blue and **A** stands for Alpha. This texture slot has the ability to provide color and opacity to a texture, which is usually saved in a separate channel within the image file. Grass and shrubs in videogames frequently have alphas within their color/diffuse maps to create an illusion that they aren't a bunch of planes with a texture slapped on.

The **Normal** map slot to the right is generally used in tandem with colored maps. It uses brightly colored RGB information to give more realism and depth to a color texture. We will be using these more later on in the book.

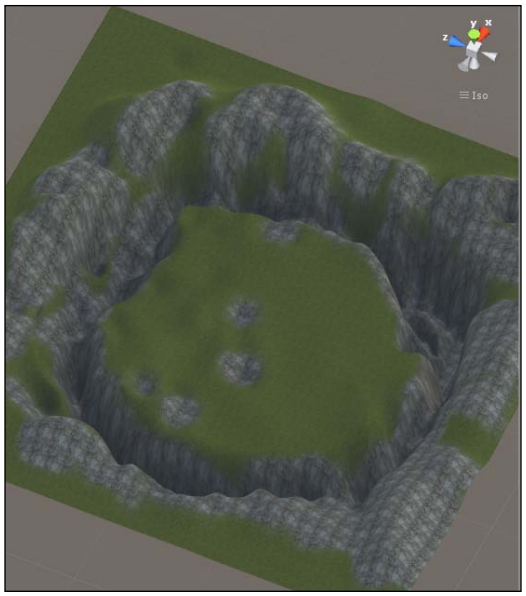
At this point, you should see the entire world change to green if you're far away. If you zoom in, you'll see that now the entire Terrain is using the **Grass (Hill)** texture.



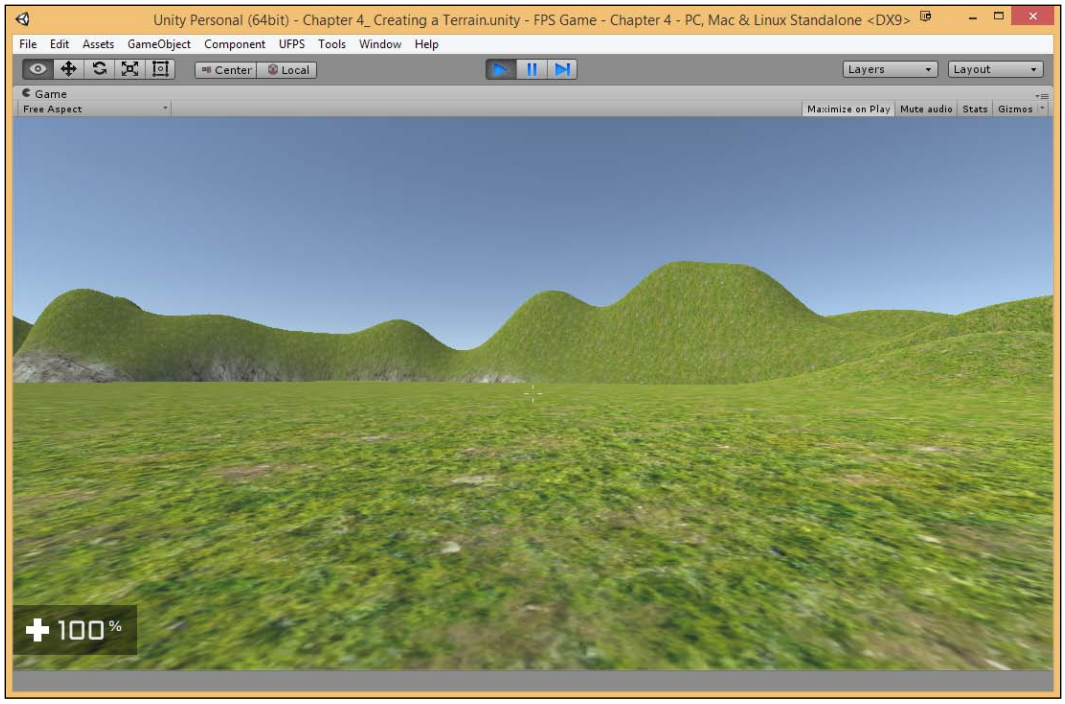
9. Now, we don't just want the entire world to have grass. We will next be adding cliffs around the edges where the water is. To do this, we will add an additional texture by going to **Edit Textures... | Add Texture**. Select **Cliff (Layered Rock)** as **Texture** and click on **Add**. Now, if you select **Terrain** from the **Inspector** tab, you should see two textures. With the **Cliff** texture selected, paint the edges of the moat by clicking and holding the mouse, modifying the **Brush Size** value as needed.



You may also spend time painting the mountains on the outside of the area to create a nicer looking environment such as the following screenshot:

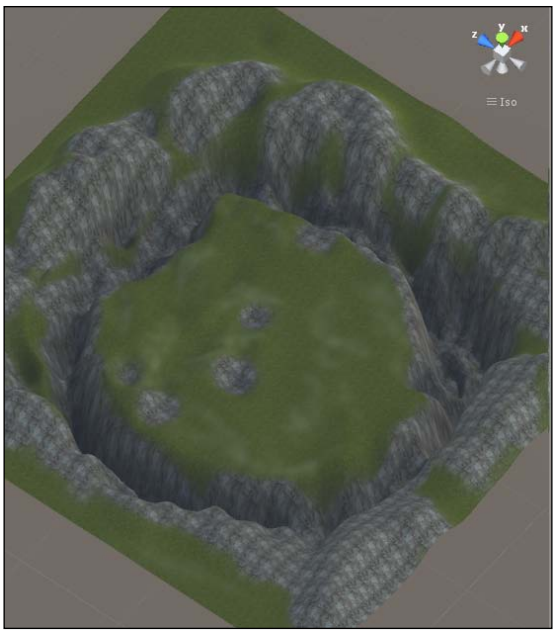


Now, it's starting to look really nice. But if we play the game and look at the ground, at a distance, you will see some repetitiveness in the textures.

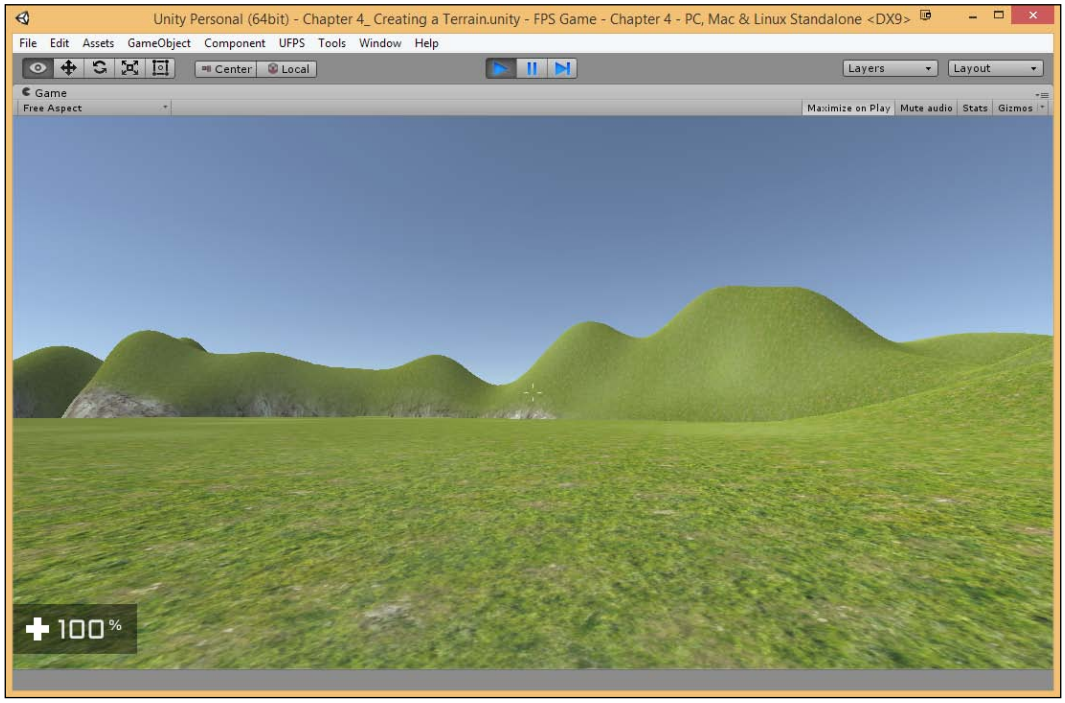


Thankfully, Unity has some things we can do in order to break up this monotony, namely that we can mix the textures.

10. To reduce the appearance of texture duplication, we can introduce new materials with a very soft opacity that we will place in patches in the areas where there is just plain ground. For example, let's create a new texture with the **Grass (Meadow)** texture. Change the **Brush Size** value to `16` and the **Opacity** value to something really low, like `6`, and then start painting in the areas that look too static. Feel free to select the first brush again to have a smoother touch up.



11. Now, suppose we were to zoom into the world as if we were a character there. I can tell that the first grass texture is way too big for the environment, but this can be changed very easily. Double-click on the texture to change the **Size** to `(8,8)`. This will make the texture smaller before it duplicates. It's a good idea to have different textures with different sizes so that the seams of each texture aren't visible to others.

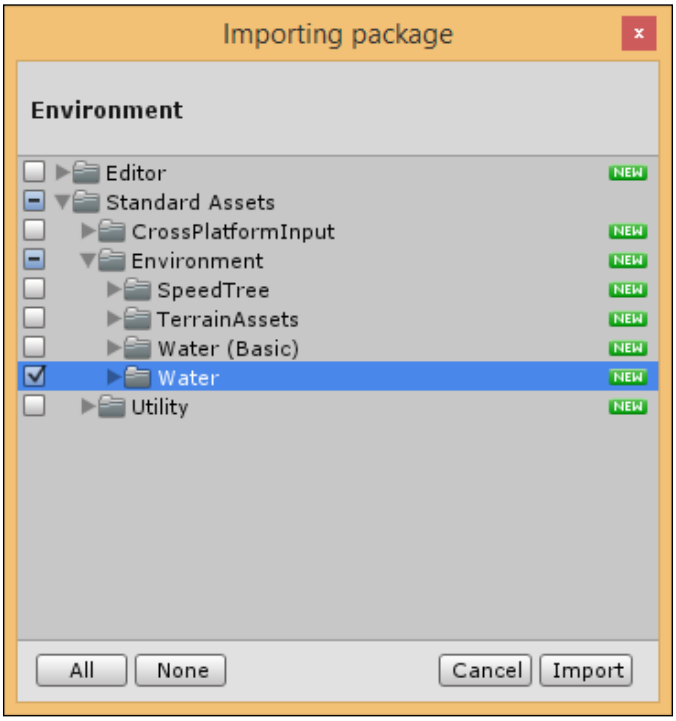


As you can see, it already looks a lot nicer in the scene, but as it stands, it's just a couple of hills without much else to guide the player. To really add to the high quality of this level, we're going to need to add in some additional features to make it appear like a real place.

Adding water

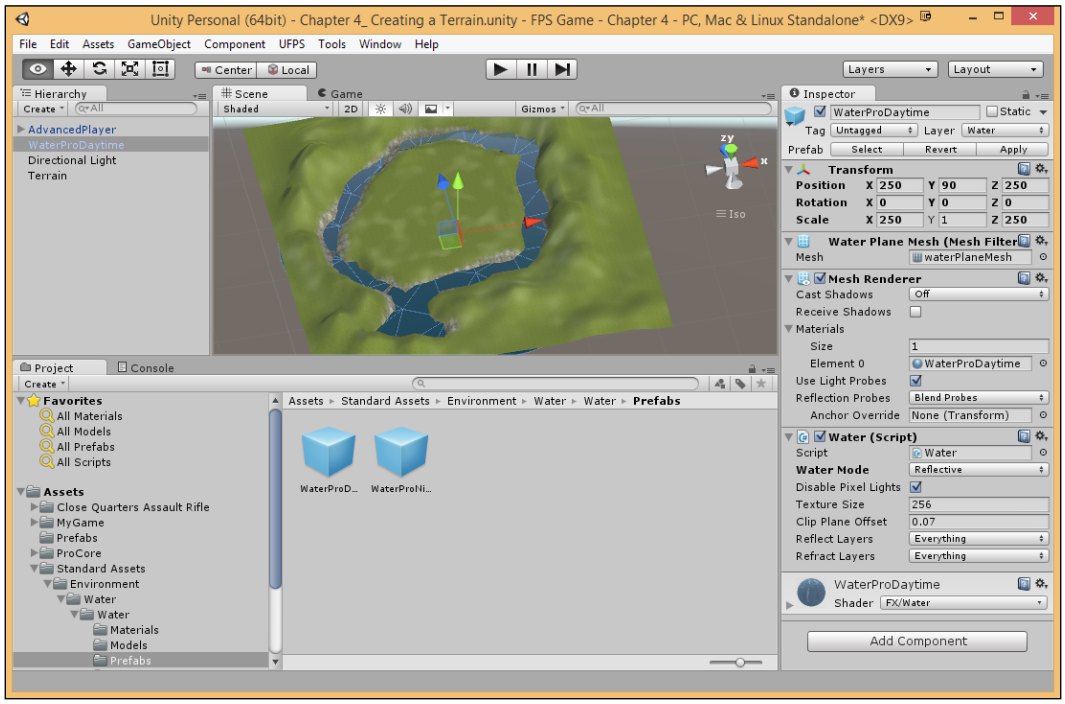
We created the lower section of our gameplay area to be at the water level or where the water would be. This can be a useful tool as a level designer to designate places where players can't go to if they can't swim. Thankfully, it's also quite easy to add to our level and, due to it being included in Unity, we won't need to go to the **Asset Store** to get it.

1. Go to **Assets** | **Import Package** | **Environment** and wait for it to load all of the objects it has. Once the entire package loads, we only want the basic water materials. First click on the **None** button at the bottom, shrink all of the folders, expand `Standard Assets` folder, and then check the `Water` folder. After you've done all of this, click on the **Import** button.

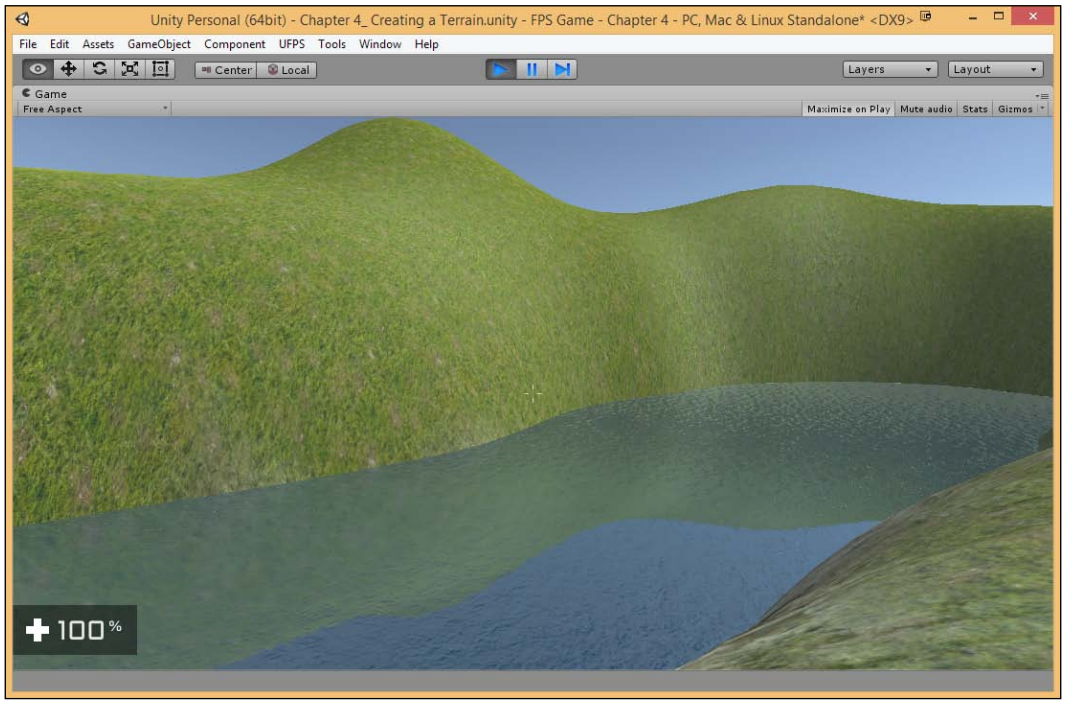


During the import process, keep in mind that sometimes it may seem like Unity is frozen if the game is still running.

2. Next go to the `Standard Assets/Environment/Water/Water/Prefabs` folder to drag and drop the `WaterProDaytime` prefab into the scene.
3. Once in the scene, change the object's **Position** value to `250`, `90`, `250` and give it a **Scale** value of `250`, `1`, `250`, so it covers the entire area. Next, in the **Water (Script)** component from the **Inspector** tab, change **Water Mode** to **Reflective**. So, instead of seeing what's underneath the water, we see the reflection of what's around us.



4. Save the level and play the game to take a look at your newly created water.





For more information on creating water or learning how to create water from scratch, check out `http://docs.unity3d.com/Manual/HOWTO-Water.html`.

Adding trees

Hills typically aren't just grass. Vegetation can be used to block a player's visibility and give a better look to our environment. Perform the following steps to add tress:

1. From the **Project** tab, go to the `Assets/Terrain Assets/Trees Ambient-Occlusion` folder and drag and drop a tree into the world (I'm using **ScotsPineTree**).

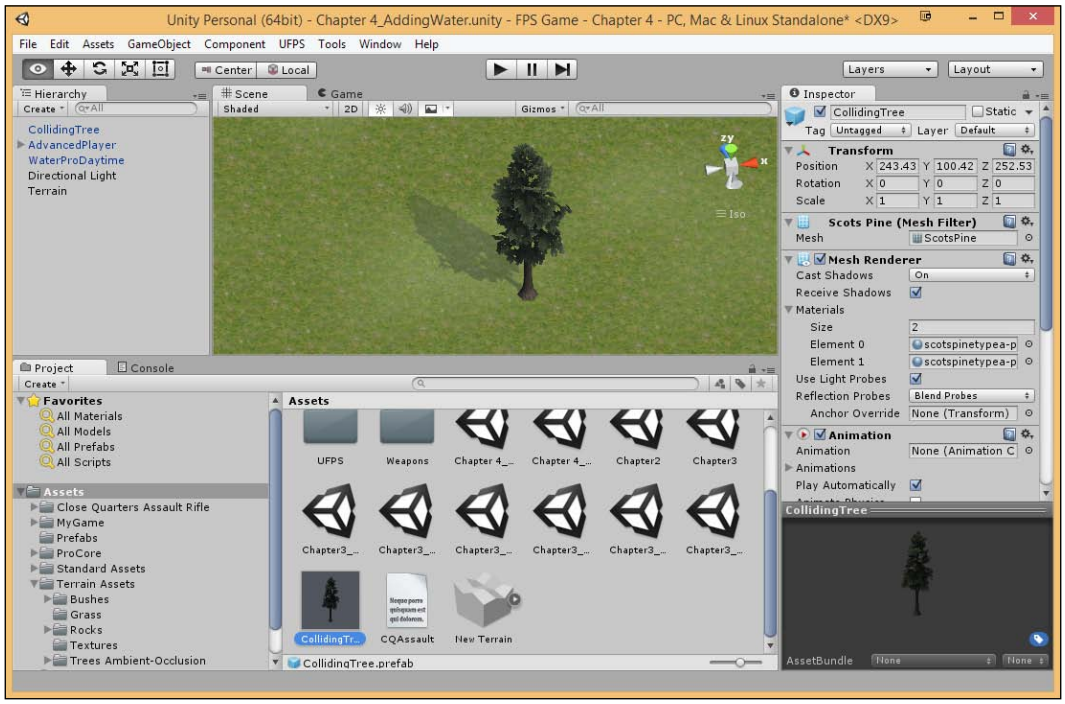
   By default, these trees do not contain collision information, so our player could just walk through it. This is actually great for the areas that the player will not reach, as we can add more trees without having to do meaningless calculations. But we need to stop the player from walking through them, so we're going to add a collider.

2. To do so, select the tree, select **Component** | **Physics** | **Capsule Collider**, and then change the **Radius** value to `1`.



You have to use a **Capsule Collider** for collision to carry over to the Terrain.

3. After this, move our newly created tree into the `Assets\MyGame\Prefabs` folder under the **Project** tab and change its name to `CollidingTree`.



4. Once we verify that the object exists as a prefab, delete the object from the **Hierarchy** tab. With that done, go back to our Terrain object and click on the **Place Trees** mode button (the one that looks like two trees). Just like with **Textures**, there are no trees in it by default, so click on **Edit Trees…** | **Add Tree**, add our **Colliding Tree**, and then select **Add**.

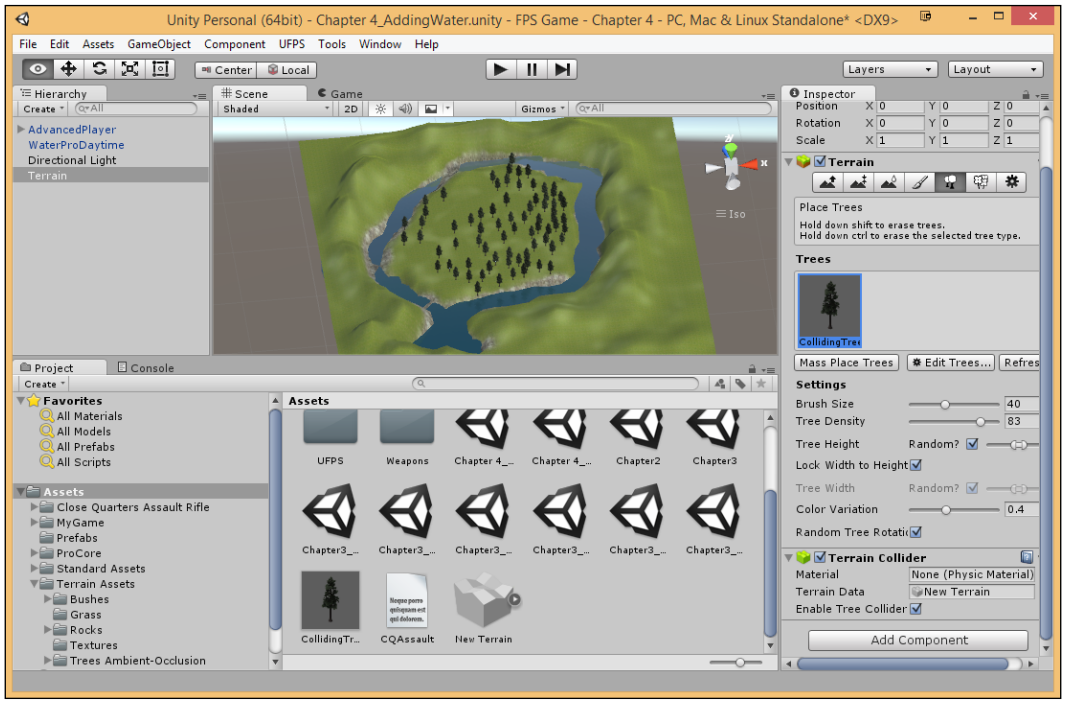


To check the names of each button, you can hold the mouse over the image for a second. It will display what it is.

5. Next, under **Settings**, change **Tree Density** to `15` and, with our new tree selected, paint the areas where you'd like to see trees. However, do not place them on the hills yet, since we know the player will not go there so they do not need to have collision.

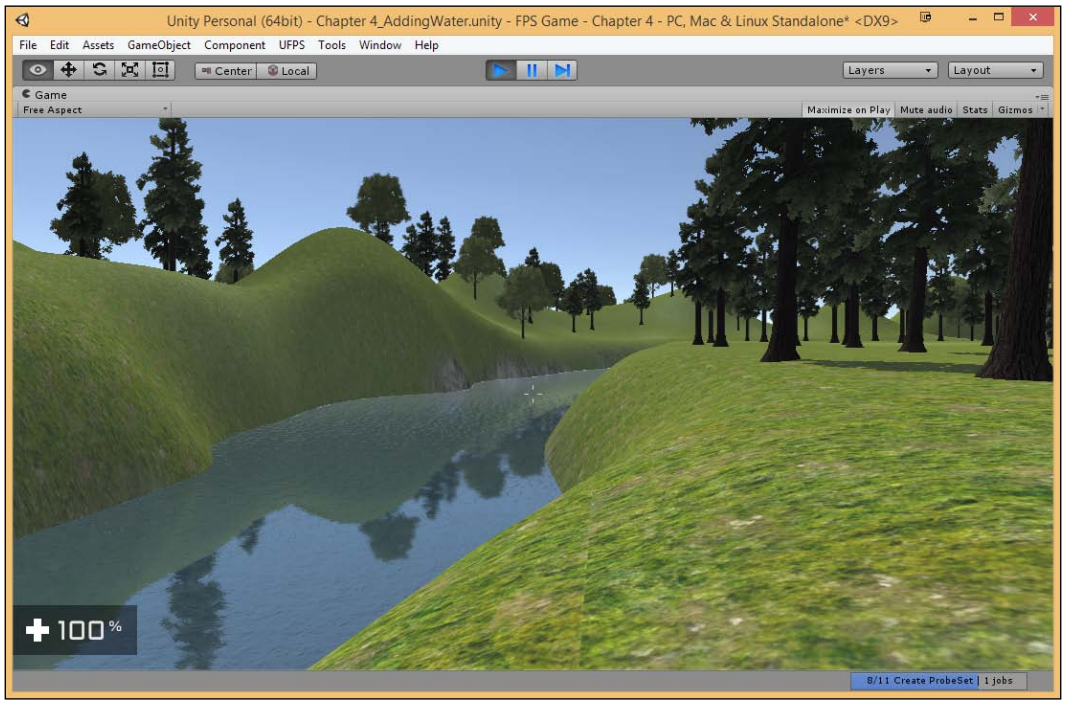
   If you hold down *Shift* and paint, you will remove trees instead of placing them.

6. You should also verify that **Enable Tree Colliders** in the **Terrain Collider** component is enabled.



7. Find the **Colliding Tree** prefab in the **Project** tab, duplicate it by pressing the *Alt* + left mouse button, and drag it to the negative space to create a copy. Select the new tree and rename it to `TreeNoCollision`. Remove the collision component from it by scrolling down to **Capsule Collider**. Click on the **Gear** drop down menu and select **remove component**.

8. Next, go to **Edit Trees** | **Add Tree** and in the **TreeNoCollision** object. We can add these in much higher numbers if needed and place them on the other side of the water where the hills are.
9. Save your level and play the game.



As you can see, the world is already much better to look at.



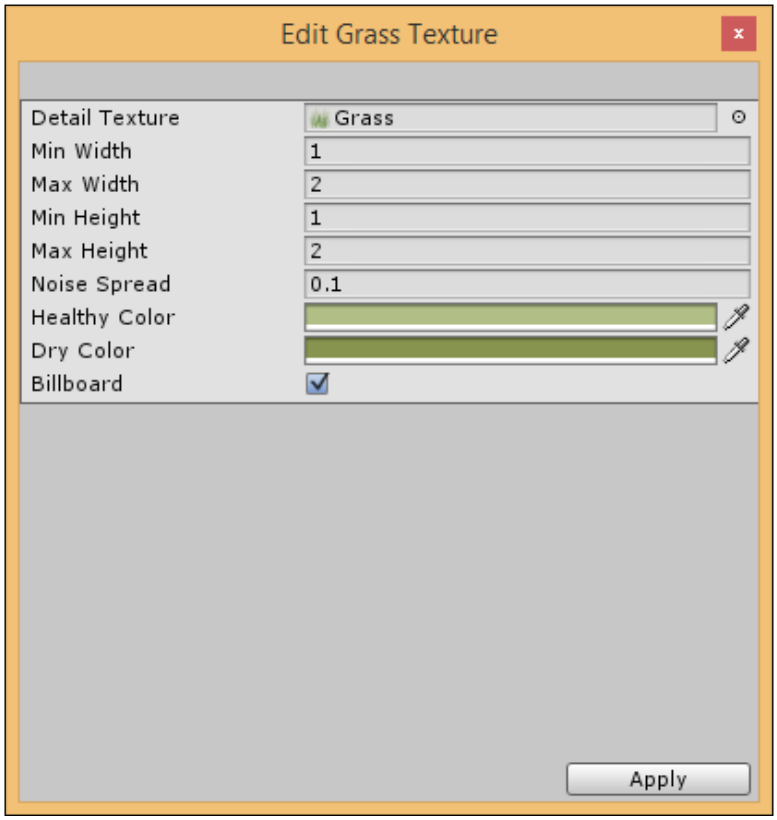
If you want to add more details to your levels, you can add additional trees and/or materials to the area as long as it makes sense for them to be there.

For more information on the Terrain Engine that Unity has, please visit `http://docs.unity3d.com/Manual/script-Terrain.html`.

Adding details – grass

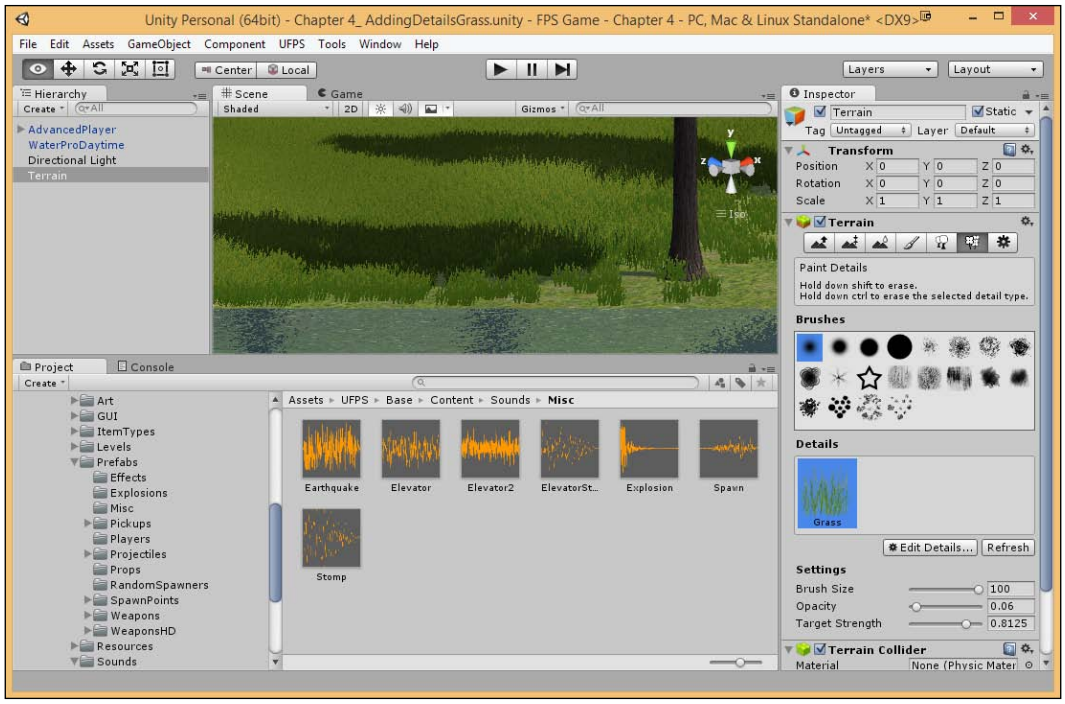
Let's now see how we can use the **Paint Details** tool to add more details to our maps.

1. The mode to the right of the **Plant Trees** mode is the **Paint Details** mode. Click on it, then click on the **Edit Details…** button, and select **Add Grass Texture**. Select the **Grass** texture inside of the `TerrainAssets\Grass` folder for **Detail Texture**. After this, set **Healthy Color** and **Dry Color** by using the eyedropper and choosing a color similar to our textures. Once you're done modifying the settings, click on **Apply**.



The eyedropper can actually pick up colors from anywhere, including the textures on the Terrain, making it very easy for us to pick out similar colors.

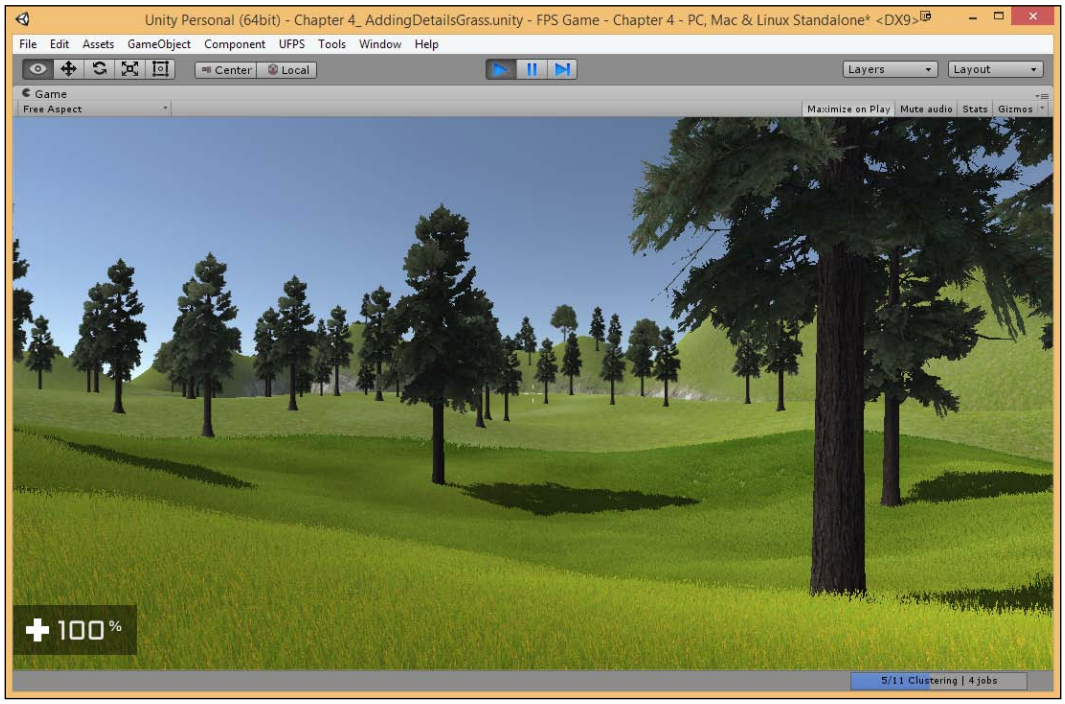
2. Once created, start dragging around to paint the grass in the world in the same way we did the textures. If you can't see it being placed, zoom in as you need to be close to see it. You can adjust **Opacity** and **Target Strength** to a lower value to make the grass more dispersed and not so close to each other.



Depending on the power of your computer, you may want to have less grass and/or details. You can see the grass further away by increasing the **Detail Distance** property if you have a powerful computer.

3. Lastly, our current island is very flat and while that's okay for cities, nature is random. From the **Inspector** tab with the Terrain object selected, go back to the **Height Raise/Lower** mode in the **Terrain** component and gently raise and lower the levels of some areas to give the illusion of depth. Do note that your trees and grass will rise and fall with the changes you make.

4. Once you're finished, save your level and play the game.





This aspect of level creation isn't very difficult, it is just time-consuming. However, it's the time taken to put in these details that really sets a game apart from other titles. Generally, you'll want to playtest and make sure that your level is fun before you perform these actions. But I feel, it's important for you to have an idea of how it is done for your future projects.

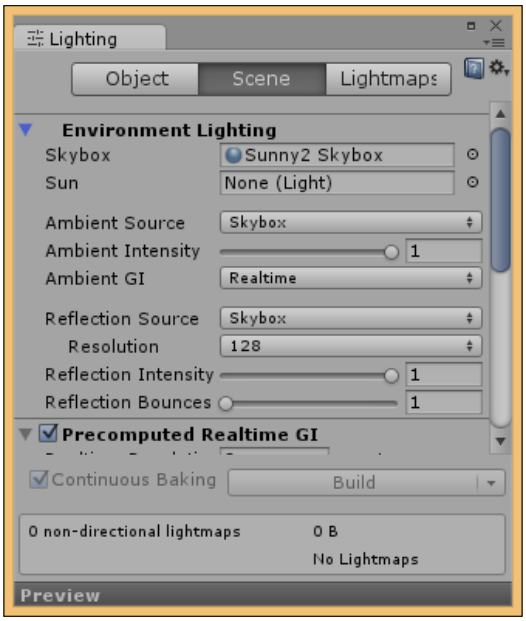
Building the atmosphere – Skyboxes and Fog

Now the base of our world is created. Let's add some effects to make the game more visually appealing.

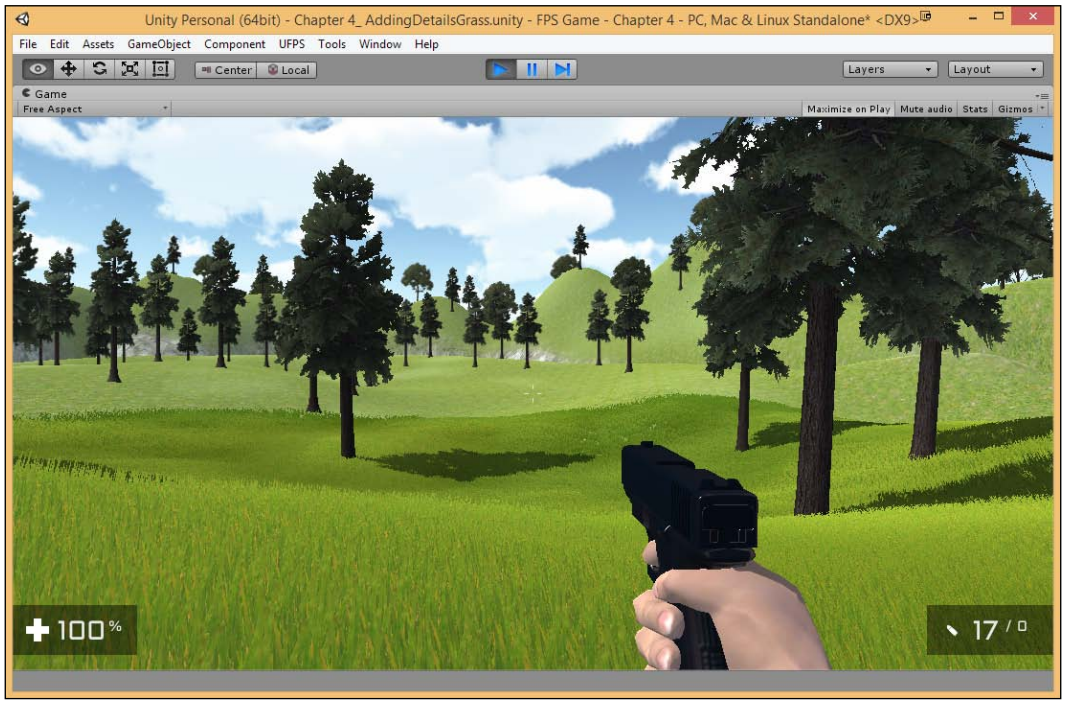
The first part of creating the atmosphere is to add something to the sky. Now, in Unity 5, we are given a Skybox by default, but we will be using one of our own that's provided by UFPS. **Skybox** is a method of creating backgrounds to make the area seem bigger than it really is by putting an image in the areas that are currently being filled with the light blue color, staying still while we move just like the sky doesn't move when we go around because it's so far away.

The reason we say Skybox is because we save six textures (one for each side of a cube). Game engines such as unreal have **skydomes** that do the same thing just with a hemisphere instead of a cube:

1. To modify the skybox, we can go to **Window** | **Lighting**. Once in there, click on the **Scene** button to look at the lighting properties of the current scene.
2. In the **Project** tab, click on the search bar in the top-right corner and type in `Skybox`. You'll see that there are a couple of Skyboxes that we've already imported that are a part of UFPS. Drag and drop the `Sunny2 Skybox` object into the **Skybox** property from the **Environment Lighting** tab.



3. Save your scene and play the game.





For more information on building you own Skybox, check out `http://docs.unity3d.com/Manual/HOWTO-UseSkybox.html`.

This already makes our level look a lot nicer and full of life, but there are also some other things we can do to make it look even better.

Fog is another way to create the atmosphere of a level. Fog obscures far away objects, which both, adds to the atmosphere and saves rendering power. The denser the fog, the more the game will feel like a horror game. The first game of the *Silent Hill* franchise used fog to make the game run at an acceptable frame-rate as it had a large 3D environment on the early PlayStation hardware. Because of how well it spooked players, they continued using it in the later games even though they could render larger areas with the current technology.

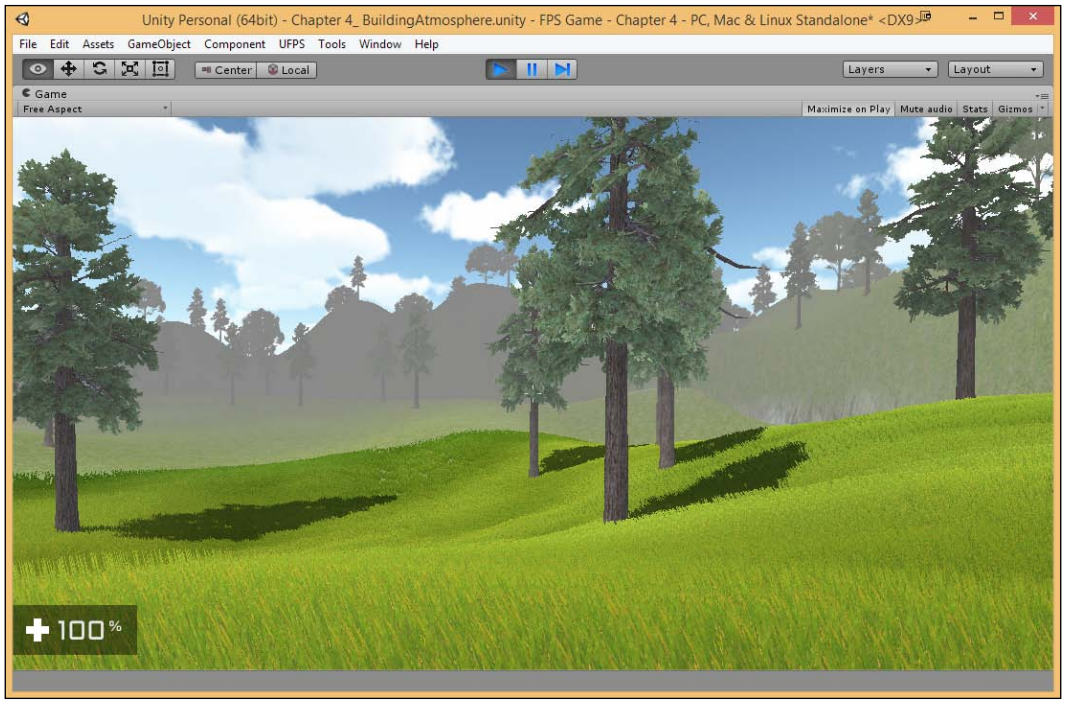
This being said, if you think fog is only used in this way, you'd be mistaken. If you've ever been to a mountain range or seen something further away at a distance, you may notice that things get more tinted in a blue color with an aerial perspective.



You can find more information on aerial perspective at `http://en.wikipedia.org/wiki/Aerial_perspective`.

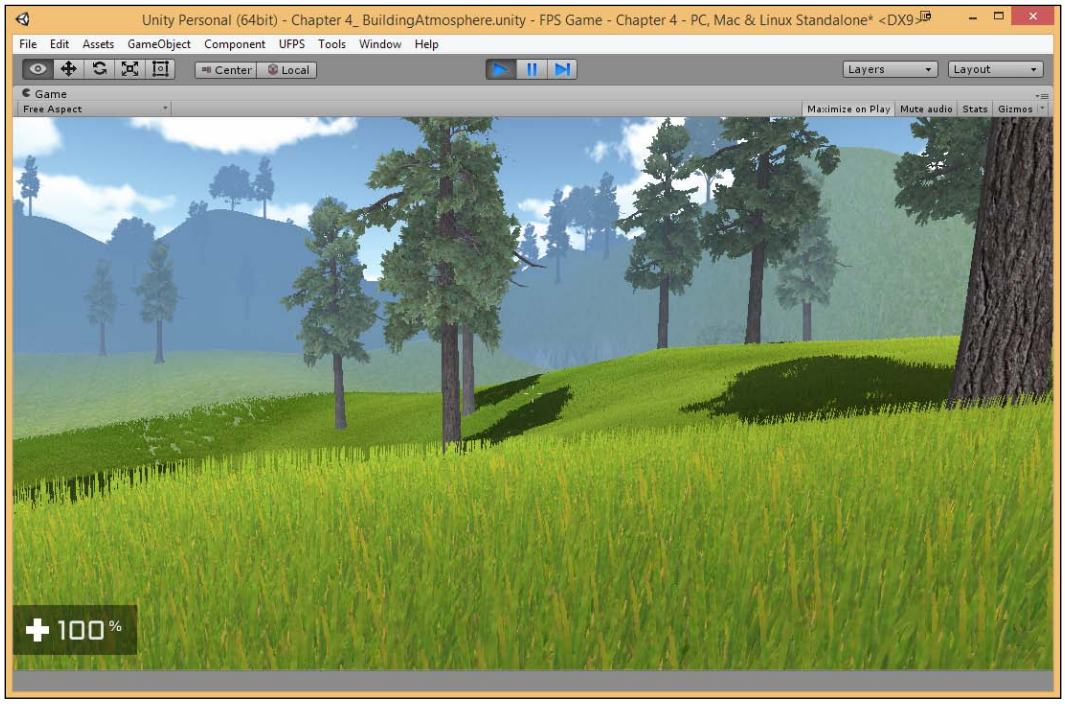
To create the atmosphere using **Fog** perform the following steps:

1. Scroll down the **Lighting** tab till you see the **Fog** section and check it.



If you were to play the game right now, it would look like the previous image. Right now, it looks a bit too grey because of the **Fog Color** property that is set. Change it instead to the same color as the sky.

2. Go to **Fog Color** and change the color to be the same as they sky using the eyedropper tool and then run the game.



Fog is also really great in terms of not letting our players know what's too far ahead of them to create suspense. The scene is already looking better.

Lastly, our current form of lighting is very bland and one-dimensional. Due to the way the lighting is done, if your lighting is done at an angle, all of the areas affected by shadows will have little to no light, making it very hard to see things.

To add to this, we can create another directional light in the opposite direction with less intensity (also known as a **fill light**) to ensure that the areas are lit better and we can see as much details as possible.

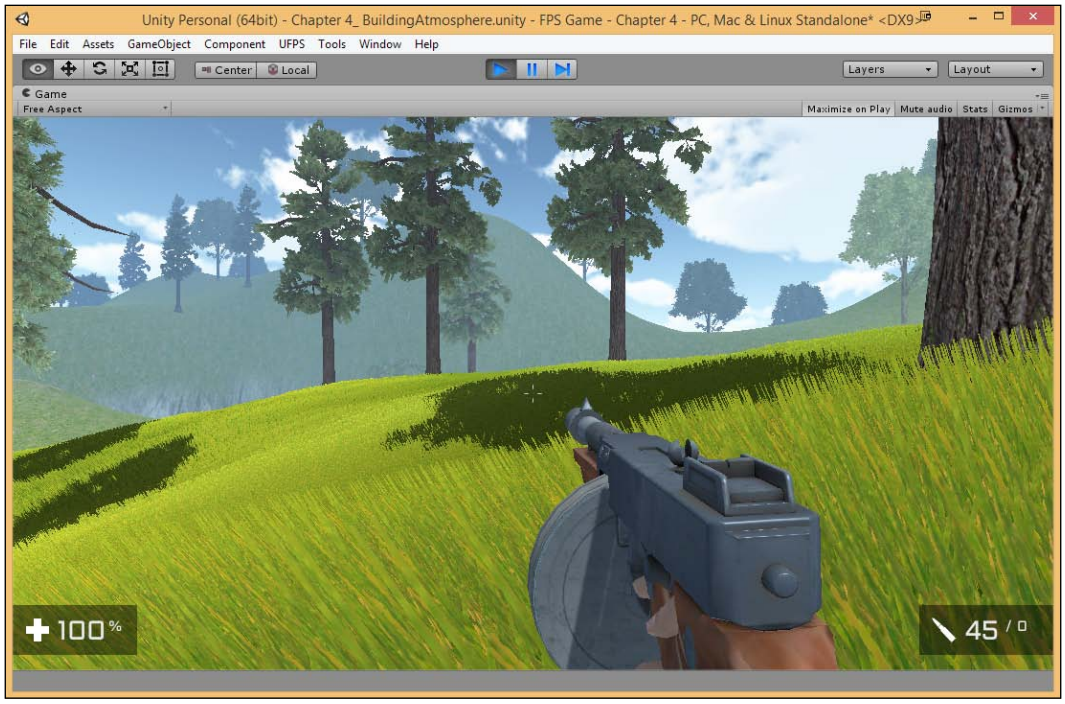
3. Select our **Directional Light** object and press *Ctrl* + *D* to duplicate it.
4. Go to **Edit** | **Snap Settings** and change the **Rotation** snap to `180`. This will ensure that, when we rotate, it will be exactly the opposite (to rotate all the way around an object is 360 degrees. 360/2 is 180).

5. Select the newly created **Directional Light** and then rotate it till it snaps (a **Rotation** of `310`, `150`, `0`).
6. Next, change the color to blue and change the **Intensity** value to `.5`. The reason we are using blue is that it is a complimentary color to yellow, which means that they look good together in general.



For more information on complimentary colors, check out `http://en.wikipedia.org/wiki/Complementary_colors`. To find complimentary colors, you can use a tool such as ColorHexa. For example, here is the page about our light's default color: `http://www.colorhexa.com/fff4d6`.

After this, save the level and play the game.



We now have a completed environment.

Summary

With this, we have a good looking exterior level for our game. In addition, we covered a lot of features that exist in Unity that you can use in your own future projects. Specifically, we covered how to build our Terrain; use textures; add water, trees, and details with grass; how to create atmosphere.

With this in mind, in the next chapter, you will learn how to take your knowledge of environments to build different combat scenarios.

5
Building Encounters

We now have all the knowledge that we need in order to create our gameplay environments, but right now, they're just places that we can walk around in and shoot at. In this chapter, we are going to learn how to create various types of encounters in order to generate effective gameplay scenarios.

Now, as a level designer and/or scripter for a project, you'll often need to implement various encounters that a player will need to go through in the project. There are way too many possibilities for encounters to cover here, but I'll go into some of the most common ones that you'll often see in an FPS.

This project will be split into a number of tasks. It will be a simple step-by-step process from the beginning to the end. Here is an outline of our tasks:

- Adding a simple turret enemy
- Integrating an AI system
- Setting up enemy characters
- Spawning a group of enemies with the help of a trigger
- Cleaning up dead AI
- Placing ammo/healthpacks

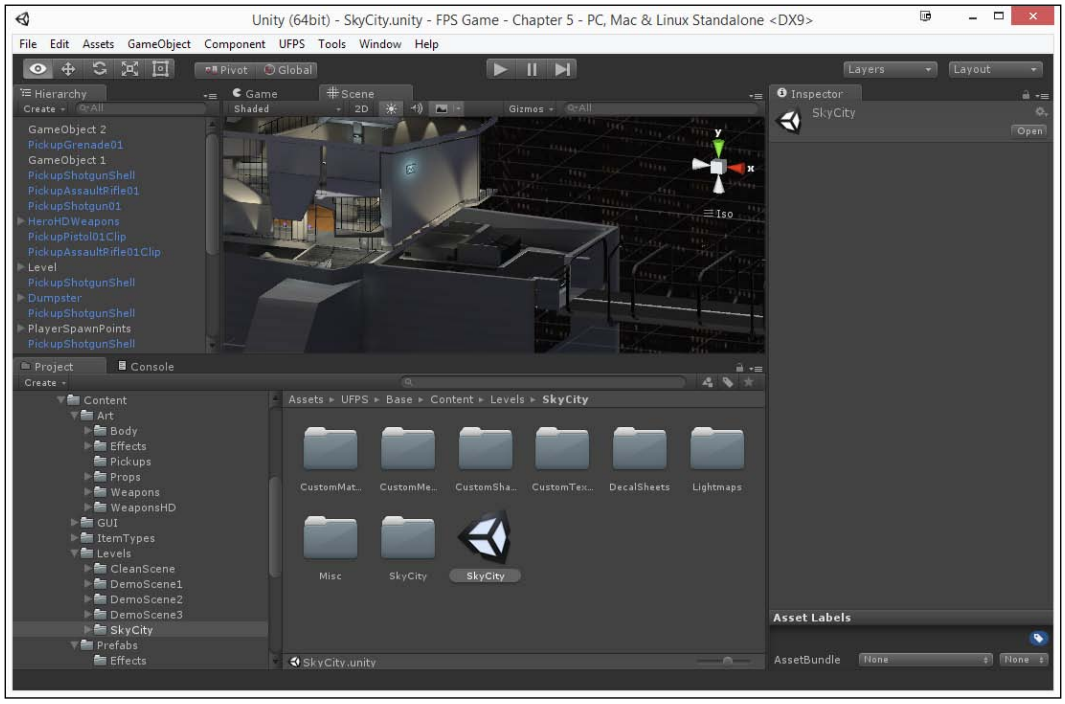
Prerequisites

Before we start, we will need to have a project created that already has UFPS and Prototype installed. If you do not have these installed already, follow the steps described in *Chapter 1*, *Getting Started on an FPS*. In addition to this, I am also assuming that you have a level you want to add encounters to.

Adding a simple turret enemy

To start off with, UFPS already comes built in with an example enemy that we can work with, that is, a turret that will fire at a player if they get too close. More importantly, the turret is an example of how to damage a player, as well as how the player can damage objects. This will be quite useful once we create enemies. Let's take a look at how it's used in an example level:

1. From the Project tab, go to the `UFPS/Base/Content/Levels/SkyCity` folder and then-double click on the `UFPS_SkyCity.unity` file to open it.

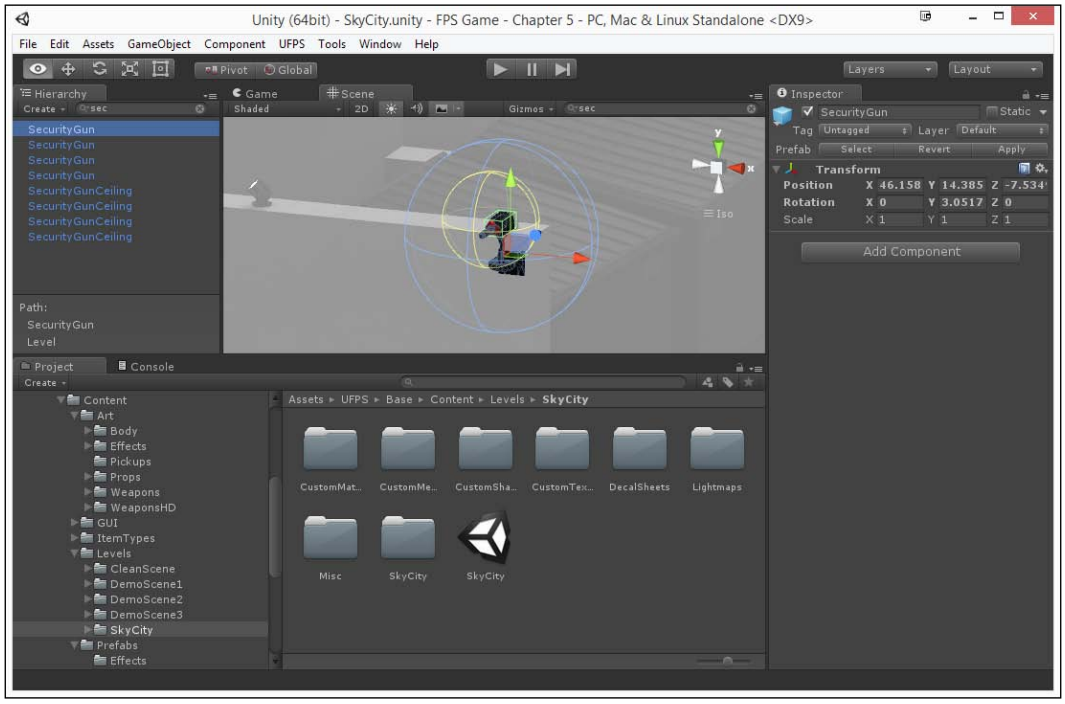


If you haven't gotten a chance to play this demo map yet, give it a try so that you're more familiar with the stuff that we will be talking about.



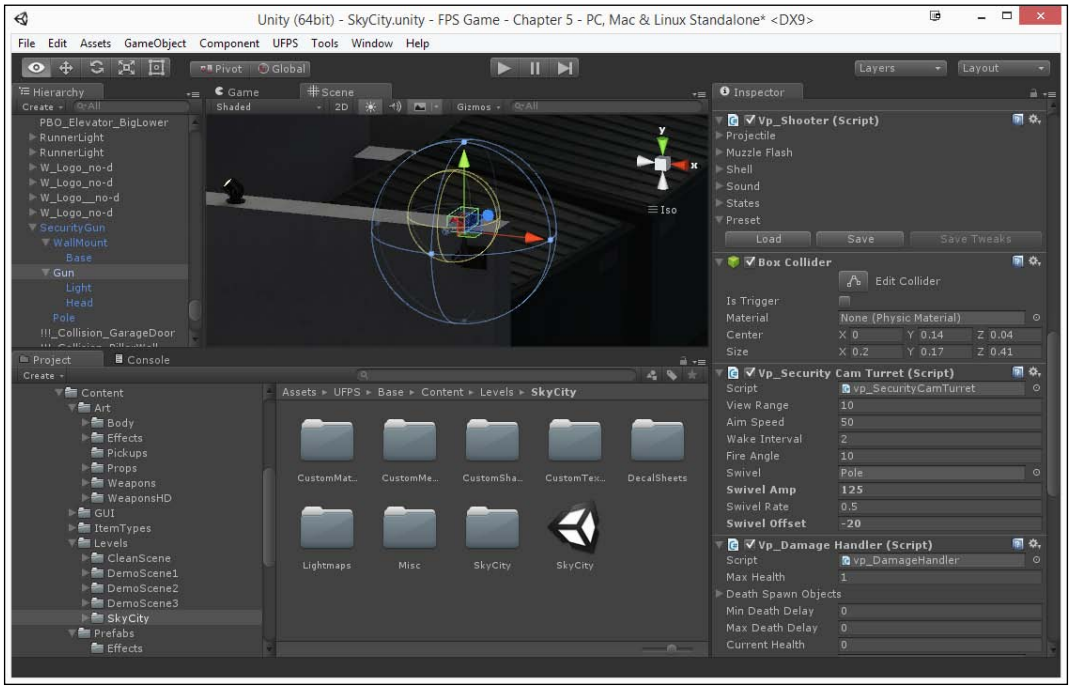
Notice that in certain areas, there will be security cameras on the walls and ceiling that will fire at you if you come close to them in their range. These are the first things that we are going to look into.

2. The turrets that show up in this level are named `SecurityGun`. So, with this information, we can go to the **Hierarchy** tab and then type in the name to be able to find them easily. Double-click on the first name to be taken to the position of it.
3. It is very difficult at this point for us to see the turret due to the light that's placed on top of it. To fix this issue, underneath the **Scene** tab, click on the **Gizmos** dropdown and drag the **3D Gizmos** property all the way to the left-hand side of the window to get rid of it. Then, select the object once again if needed.



4. Now that we can see the `SecurityGun` object, click on the **X** button on the right-hand side of the search bar to once again see everything in **Hierarchy**. Now we can extend the `SecurityGun` object to see the objects that are its children.

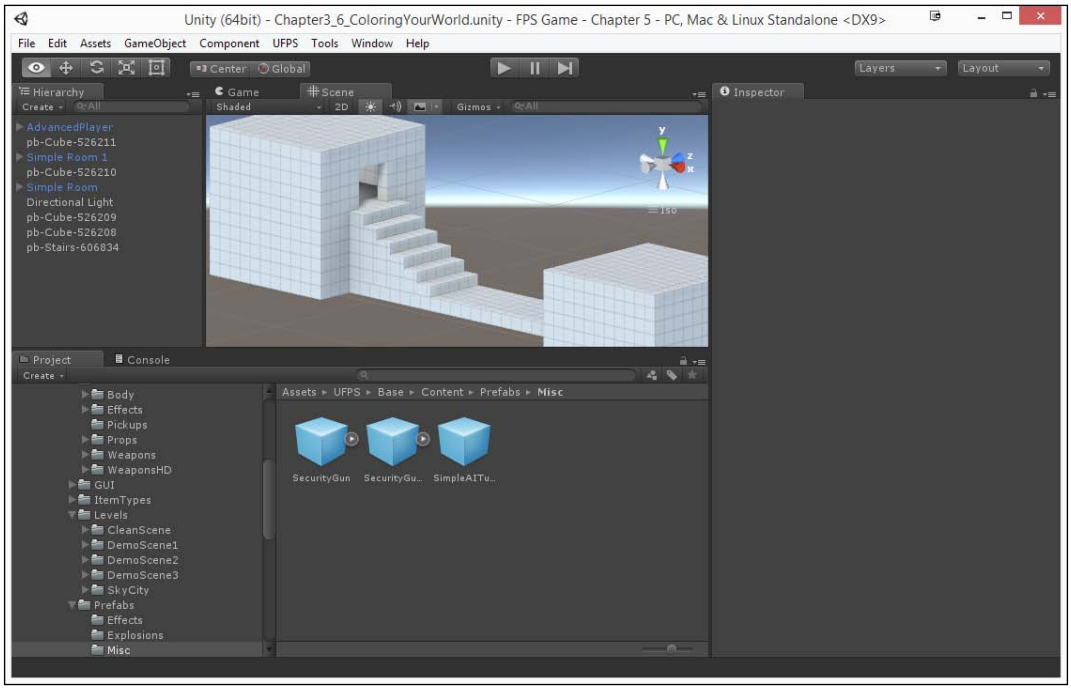
5. `WallMount` and the `Pole` objects are simple meshes and are there for fluff (aside from the security cam that uses the pole to rotate from). The one that we want to take a closer look at is the `Gun` object. Select it, and then you'll see some important classes in the **Inspector** window: `vp_Shooter`, `vp_DamageHandler`, `vp_SecurityCamTurret`, and `vp_Respawner`.



- **vp_Shooter**: This component gives the object that is attached to it the ability to fire projectiles, including but not limited to bullets (it's fired by calling the `TryFire` function). Here, we can set the properties for what it will shoot, very similar to what we did earlier in the custom weapons chapter.
- **vp_DamageHandler**: This component gives an object the ability to take damage, die, and respawn if needed. This will allow us to set the health of the object as well. We damage this object by calling the `SendMessage` function, which we will see later.

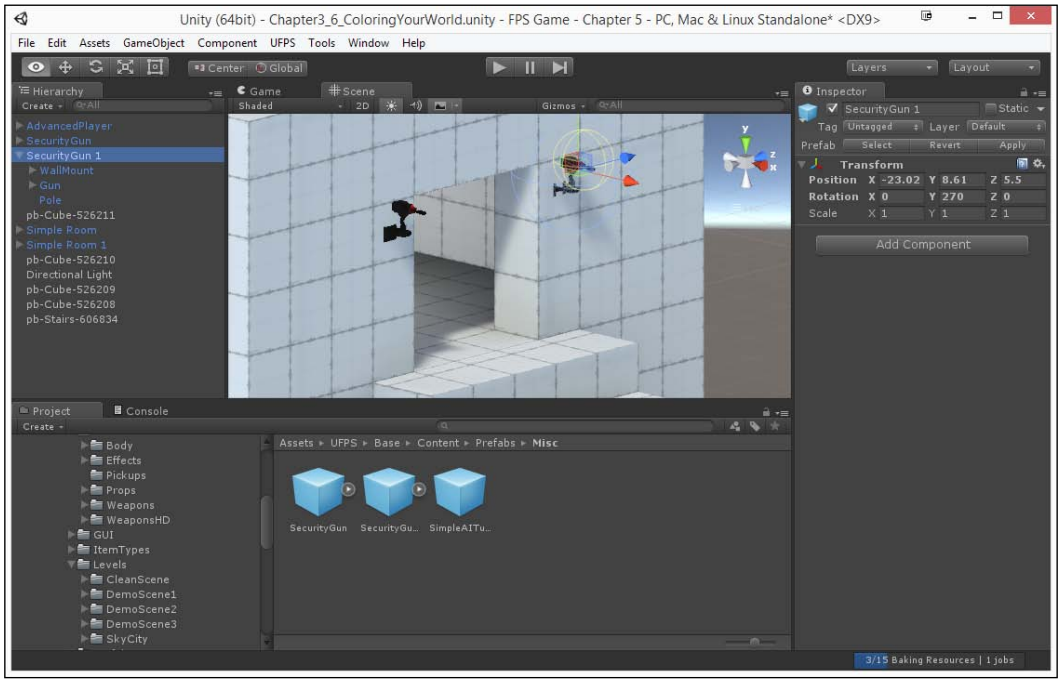
- **vp_SecurityCamTurret**: This component gives the object the back and forth movement that looks for a player to fire at. It's a child of the `vp_SimpleAITurret` class, which looks for a player (`ScanForLocalPlayer`) and attacks them (`AttackTarget`) as needed.
- **vp_Respawner**: This script specifies how an object will respawn. This will determine if the object will respawn at the same position as before or at a random spawn point such as (`vp_SpawnPoints`).

6. Now that we've seen this turret in a professional example, let's put it into our own level. Let's open the level that we created over the course of *Chapter 3, Prototyping Levels with Prototype*.

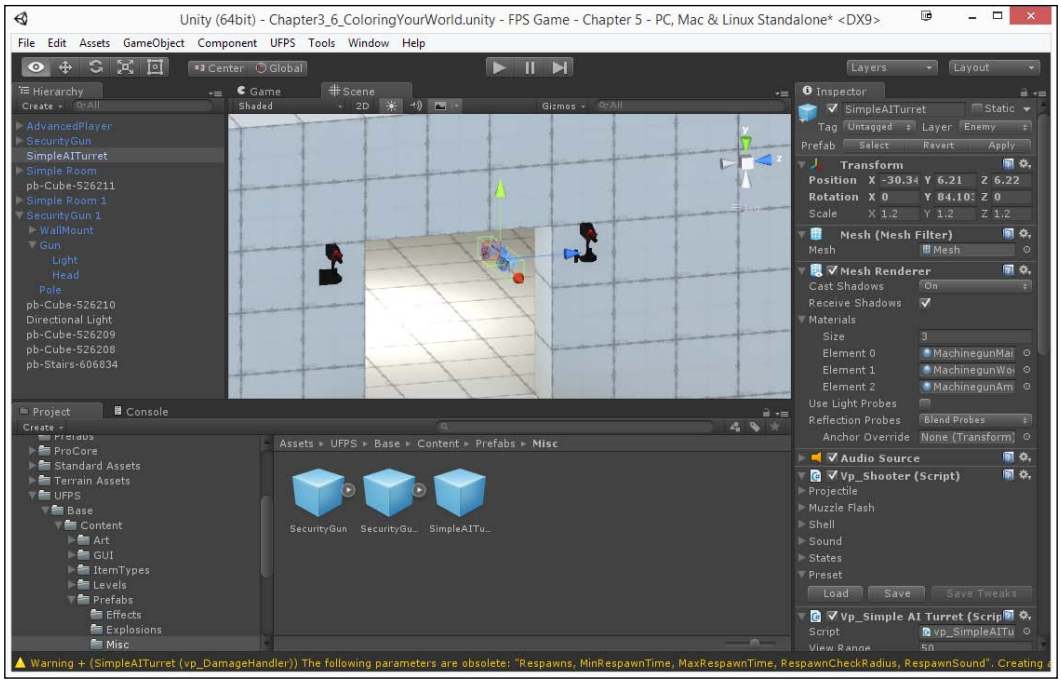


7. `SecurityGun` is already created as a prefab that we can easily place into our levels. With this in mind, go to the **Project** tab, the `UFPS/Base/Content/Prefabs/Misc` folder, and then drag the `SecurityGun` object into the world.

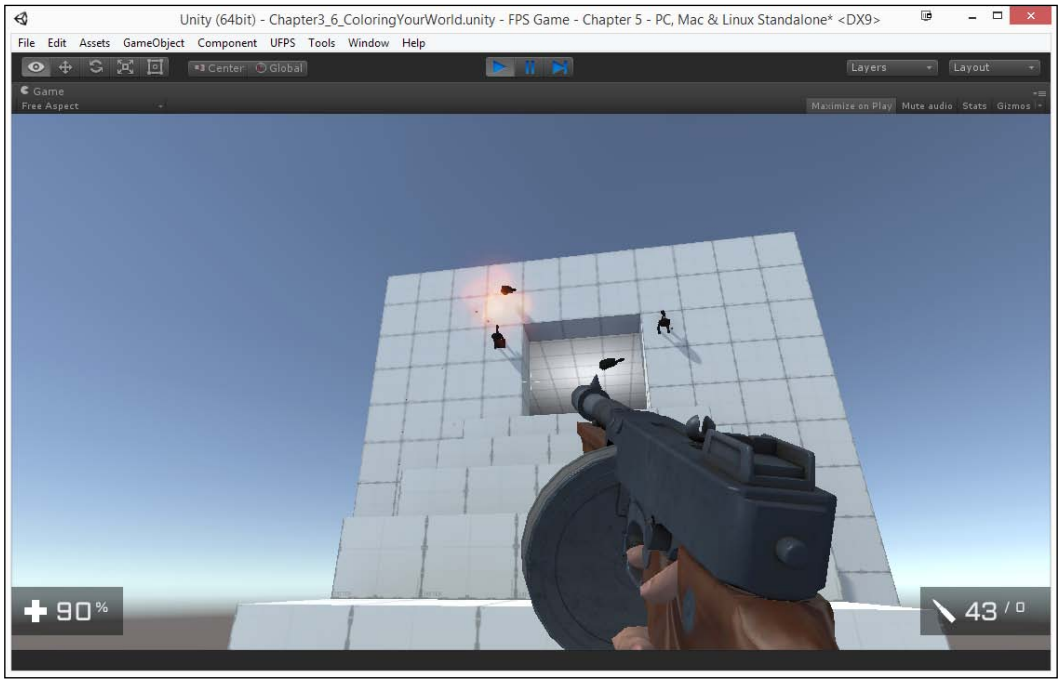
8. I'm going to position the object so that it faces the upward side of the door that we're walking to. Then, from the **Inspector** tab, I'm going to change the **Rotation's Y** axis to `-90`. Once I've positioned the object on one side of the door, I'll duplicate the object by hitting *Ctrl* + *D* and dragging it to the other side.



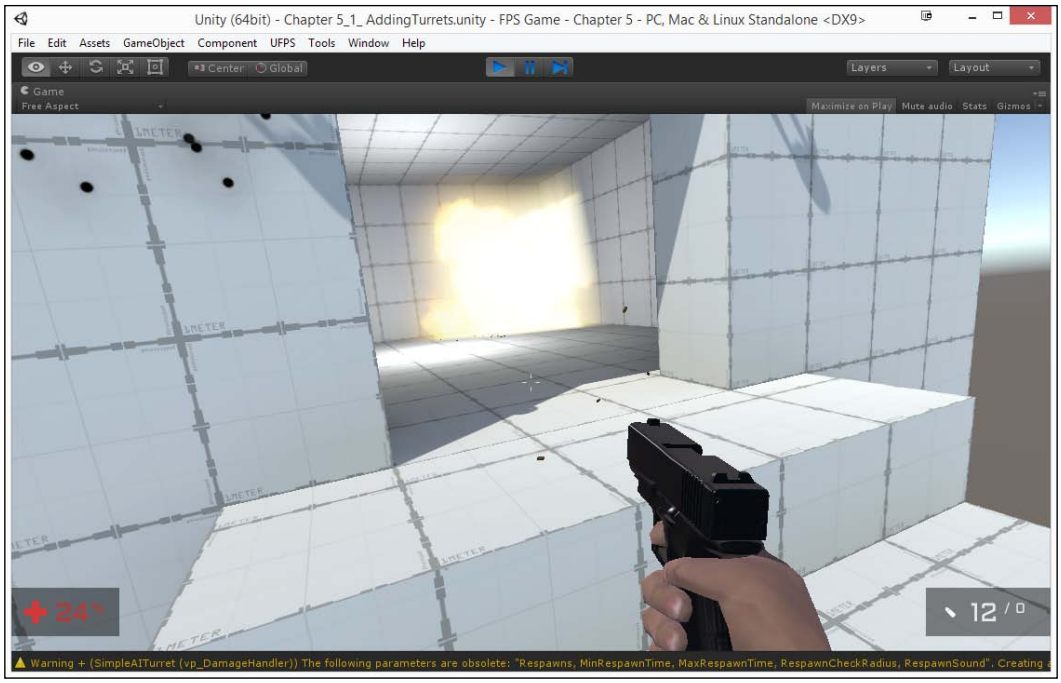
9. In addition to a security camera that moves back and forth, there's also a simple AI Turret, which will only look at you and fire if you get too close. Drag and drop the `SimpleAITurret` object into the room that the turrets are protecting.



10. Now that we have the turrets inside the game, let's save our level and start the game!



With this, we now have two security cameras covering a particular area and if we walk into the room, we come up to another turret that has an explosive reaction when it's defeated!



This is a great start! Now we know how to place stationary enemies into our level!

Integrating an AI system – RAIN

Artificial intelligence (**AI**) is one of the things that can make or break a game. In this section, you will learn how to integrate an AI package for Unity in order to have characters roam around and attack players.

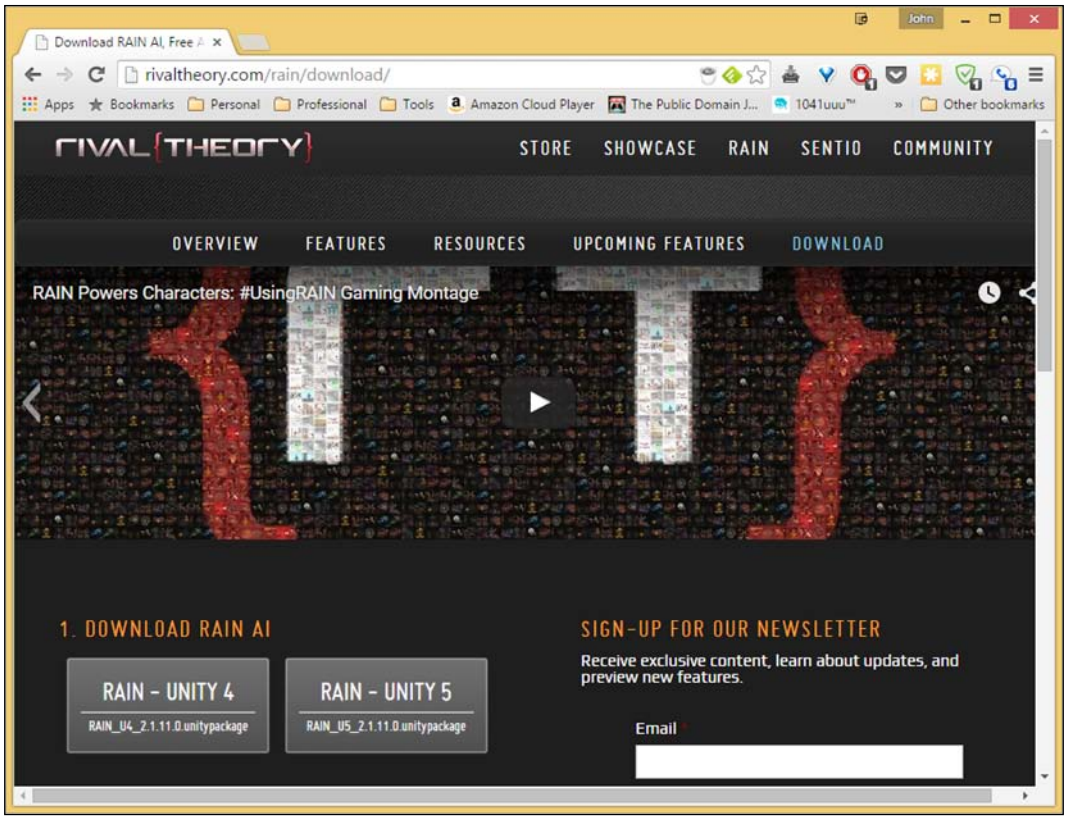
At the time of writing this, people generally tend to either use RAIN by Rival Theory (`http://rivaltheory.com/rain/download/`) or Shooter AI by Gateway Games (`https://www.assetstore.unity3d.com/#/content/11292`) if they are not writing their own package when dealing with AI using UFPS.

In this book, we will discuss both tools using RAIN to create a melee attacking enemy and Shooter AI to create a basic ranged shooting enemy. This is so that you have a better idea of how each works, and you can then weigh their pros and cons for your own title.

Note that I will mention the differences in using both tools in this chapter, but in later chapters, you will be able to choose which of the following you'd like to use for your own levels and encounters.

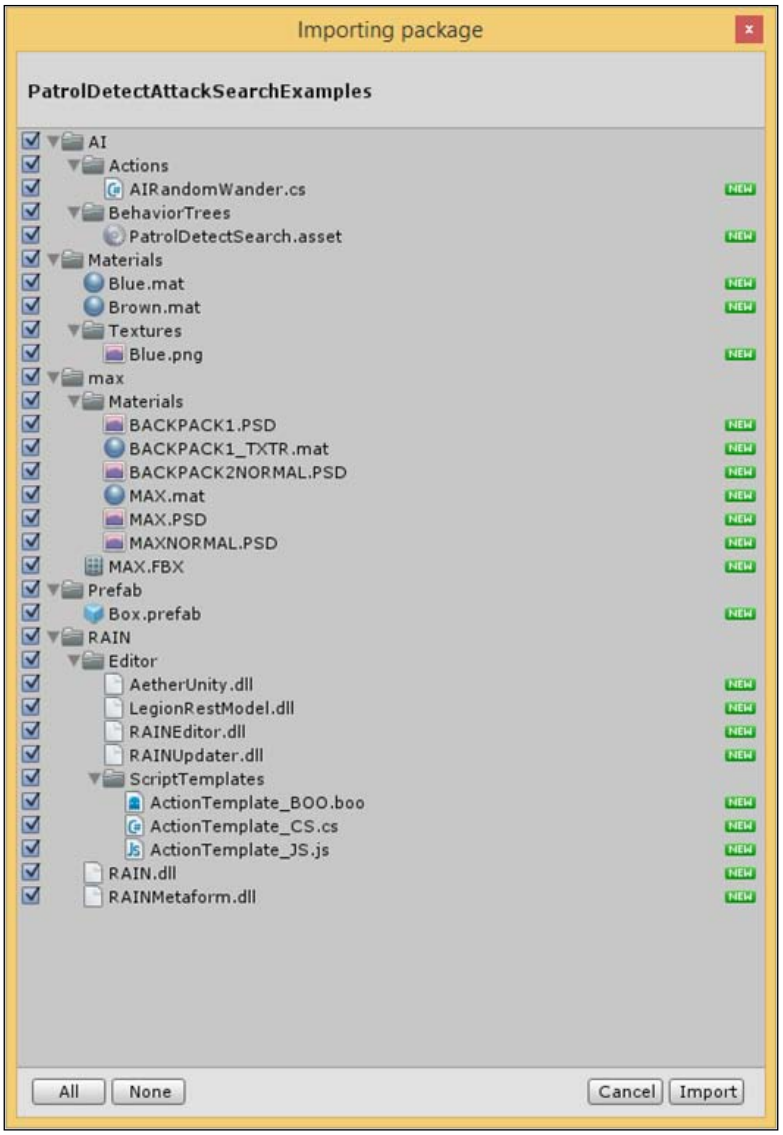
To start off, we will use RAIN in this particular example as it is a free solution that has been out for a while and is pretty mature since it is the most widely used AI engine in digital entertainment. It also has its own active community that can help you should you wish to take this example further.

1. Before starting, create a copy of your current project to save the progress you've made so far. While there shouldn't be any problems when using this project, it's always better to be safe than sorry.
2. To download it, we'll need to visit Rival Theory's website to download RAIN. So, keeping this in mind, open up your web browser and go to `http://rivaltheory.com/rain/download/`:



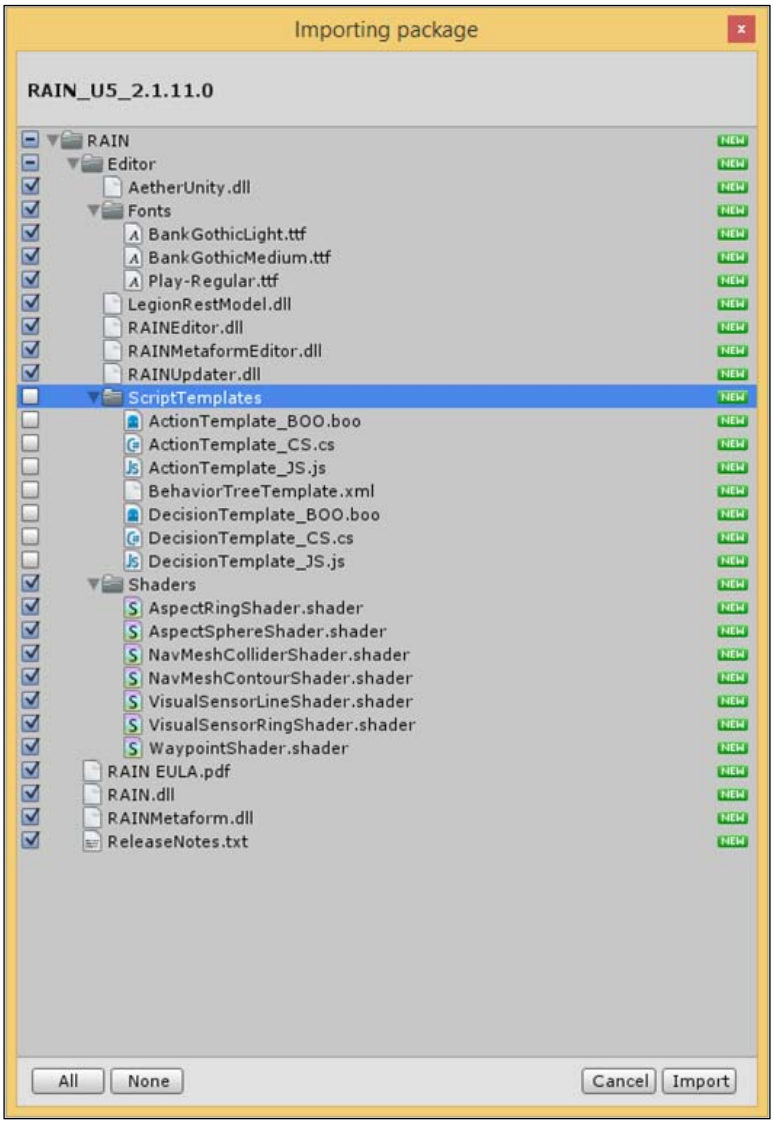
3. From there, click on the **RAIN – UNITY 5** button to download the latest version of RAIN.
4. After this, we will want to download a sample project, which will contain an animated character with sample behavior, that we can plug into the game. CodersExpo has created such a sample project that can be found in Rival Theory's forum under the Sample Projects section at `http://rivaltheory.com/forums/topic/using-waypoint-routes-and-paths/`.
5. From there, click on the `PatrolDetectAttackSearchExamples.unitypackage` file for us to use at `http://rivaltheory.com/?ddownload=10240`.

6. Once the `PatrolDetectAttackSearchExamples.unitypackage` and `RAIN_U5_2.1.11.0.unitypackage` files finish downloading, it's time to import them into our Unity project. Start with the Sample Project, click on the **Import** button, and wait for the project to finish importing:



7. While importing, it may ask you to update your scripts. Since we already made a backup in step 1, click on the **I Made a Backup. Go Ahead!** button.

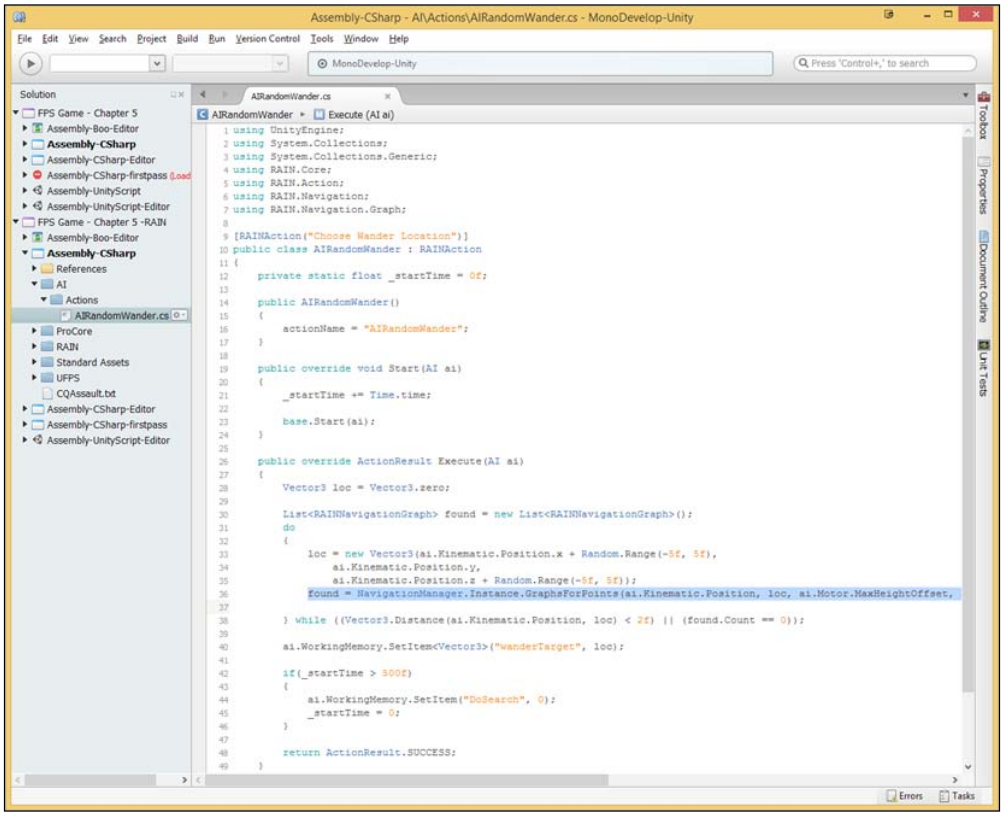
8. You may need to wait for the **Updating RAIN Components** menu. Afterwards, it will ask you if you'd like to automatically search for any updates. In our case, we want to do it manually, so for now, click on **Don't Allow** and then **Ok** in the following dialog box.
9. Next, let's update RAIN to the latest version ourselves by double clicking to open the RAIN unitypackage file and unchecking the files in the `RAIN/Editor/ScriptTemplates` folder as it will attempt to create duplicates of the files that are already causing errors.



10. Once imported, you may see the Console window open up with an error in the `Assets/Actions/AIRandomWander.cs` file. If so, open it in MonoDevelop (you can double-click on the warning/error that's to be taken to the line) and change line 36 with the following code (I've made the changes in bold):

```
found = NavigationManager.Instance
  .GraphsForPoints(ai.Kinematic.Position, loc,
  ai.Motor.MaxHeightOffset, NavigationManager
  .GraphType.Navmesh, ((BasicNavigator)
  ai.Navigator).GraphTags);
```

To help clarify where in the file to look, take a look at the following screenshot:

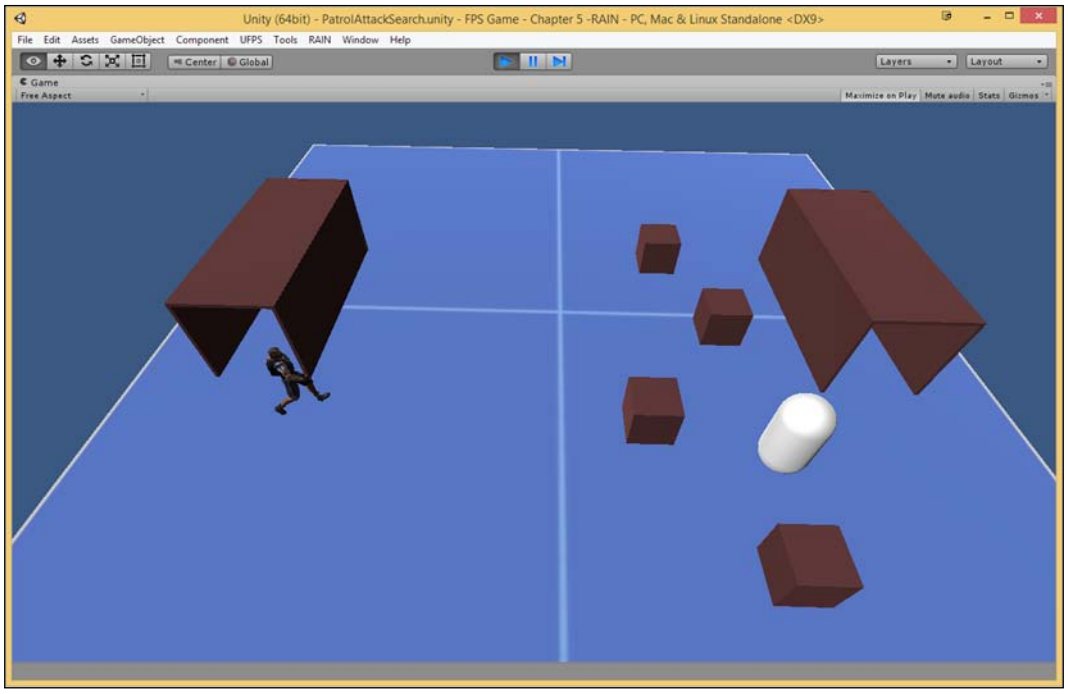




Downloading the example code

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.

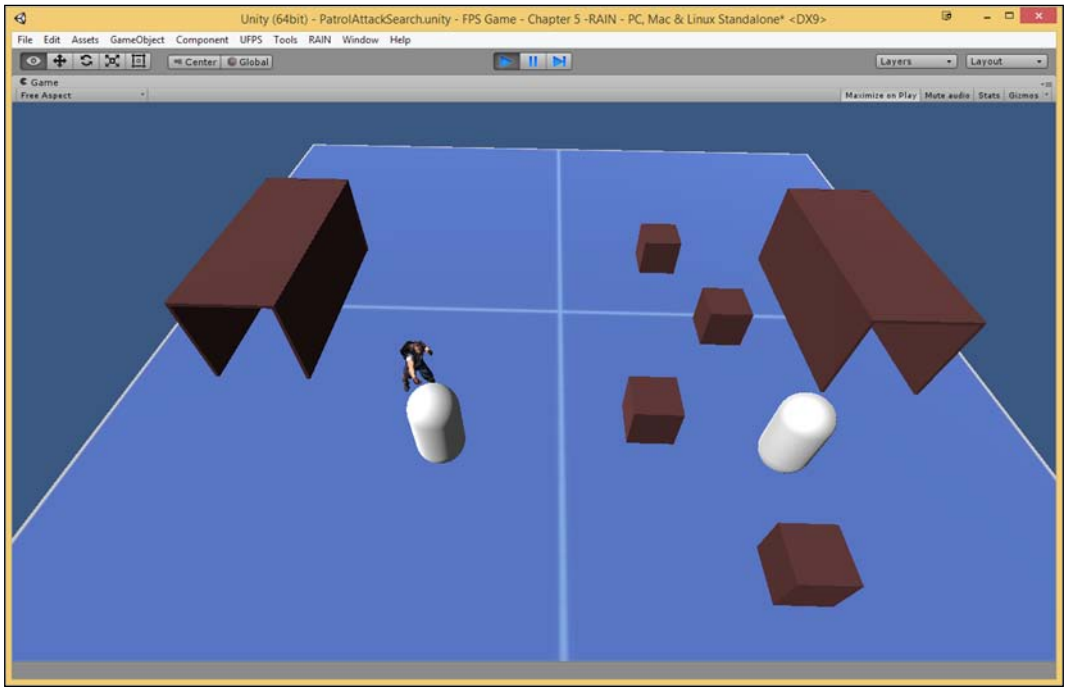
11. Save the file, and then once all of the errors have gone away, open the `PatrolAttackSearch.unity` file from the `Assets` folder in the **Project** tab. To verify whether everything is working properly, if you play the game, you should see a character called Max walking along a path defined in the scene.





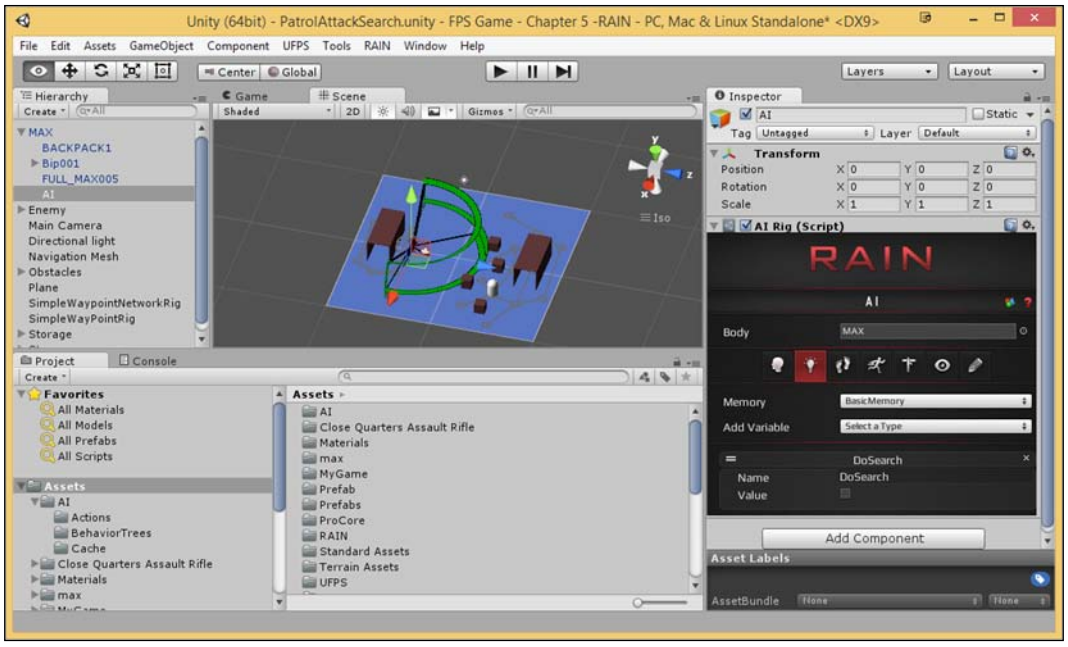
To see an explanation of what this sample project consists of, how it was built, and what each of these things mean, in more detail, check out CodersExpo's own video on starting projects with RAIN at `https://www.youtube.com/watch?v=sQlewYFwCOU`. You can also check out this Quick Start guide by RAIN's creator, Rival Theory, at `https://www.youtube.com/watch?v=YuaBBCL5PSs`.

You may also notice that if you move the white enemies closer to Max, he will run up and attempt to punch them:



Now, RAIN and UFPS are two separate systems that work well in their respective fields, but we can make them work together if we can give RAIN a way to receive damage from UFPS as well as make RAIN damage a UFPS player. Keeping this in mind, we will need to create two scripts in order to carry out these actions:

1. To start off with, we need some way to communicate our health to RAIN. To do so, go to the **Hierarchy** tab, extend the `MAX` object, and select its child, which is named `AI`. From there, you should see the **AI Rig** component in the **Inspector** tab. Next, click on the lightbulb button to open the **Memory** tab.





Information on the AI Rig component and what each tab does can be found at `http://rivaltheory.com/wiki/rainelements/airig`.

2. The **Memory** tab is used to store and share values within different AI elements that exist. We want the AI elements to be able to die at some point, so we will need to add in the ability to check when our health is 0. To do this, from the **Add Variable** dropdown, select **float** and see the new item that's to be added. With regard to this new item, change its **Name** to `currentHealth` and the **Value** to `2`.

   A regular pistol bullet that can be found in the `UFPS/Base/Content/Prefabs/Projectiles` folder, currently has a **Damage** value of `1` from its **Vp_Hitscan Bullet** component. Therefore, it will require the player to shoot our enemy twice in order to defeat it.

3. Create a new C# Script called `RAIN_DamageReciever`. Open it in MonoDevelop and fill it with the following code:

```
using UnityEngine;
using RAIN.Core;        // AIRig
using RAIN.Memory;      // RAINMemory

/// <summary>
```

```
/// Allows RAIN to receive damage messages from UFPS
/// </summary>
public class RAIN_DamageReciever : MonoBehaviour
{
  private AIRig aiRig;
  private RAINMemory memory;

  void Start()
  {
    // Access the AI Rig component which allows us to use
    // any of the properties in them
    aiRig = GetComponent<AIRig>();

    if (aiRig != null)
    {
      memory = aiRig.AI.WorkingMemory;
    }
  }

  void Damage(float damage)
  {
    if (memory != null)
    {
      // Get our current health from RAIN
      float currentHealth =
                  memory.GetItem<float>("currentHealth");

      // Subtract damage from the current health value
      currentHealth -= damage;

      // Update RAIN where our health is at
      memory.SetItem<float>("currentHealth",
        currentHealth);
    }
  }

  // Secondary support for this function when using
  // projectiles
  private void Damage(vp_DamageInfo projectileInfo)
  {
    Damage(projectileInfo.Damage);
  }
}
```

This script allows us to receive the damage events sent by UFPS. Once UFPS finds a bullet that collides with another object (which means it needs to have a collider of some sort), it will attempt to call a function called Damage using either a float or the `vp_DamageInfo` input. We then take this information and modify the `currentHealth` variable we created earlier in the Memory tab. Later on, we will see the enemy die as a result of falling down.

4. Attach the newly created component to the AI child of the MAX object.
5. Next, create a new file called `RAIN_DamageSender` and use the following code:

```
using UnityEngine;

public class RAIN_DamageSender : MonoBehaviour
{
  /// <summary>
  /// When called will send a message to the target as
  /// well as its
  /// parent objects to call the Damage function if it
  /// exists.
  /// Note: UFPS uses a function called Damage to damage
  /// the player
  /// </summary>
  /// <param name="target">The target to inflict
  /// damage.</param>
  /// <param name="damage">The amount of damage to do to
  /// the player.</param>
  public static void SendDamage(Transform target, float
    damage)
  {
    target.SendMessageUpwards("Damage", damage,
                  SendMessageOptions.DontRequireReceiver);
  }
}
```

Since we've made this function static, it allows us to use the function without having to create an instance of the class, and we can simply refer to it as `RAIN_DamageSender.SendDamage`.



For more information on the static modifier, check out: `https://msdn.microsoft.com/en-us/library/98f28cdx.aspx`.

6. We then need to write new behavior for actually having this AI damage the player. To do this, we will create a new script called `ForwardAttackBehaviour`, which will look like this:

```
using UnityEngine;
using System.Collections;

public class ForwardAttackBehaviour : MonoBehaviour {
  /// <summary>
  /// Where the Raycast will start drawing from.
  /// </summary>
  public Transform origin = null;

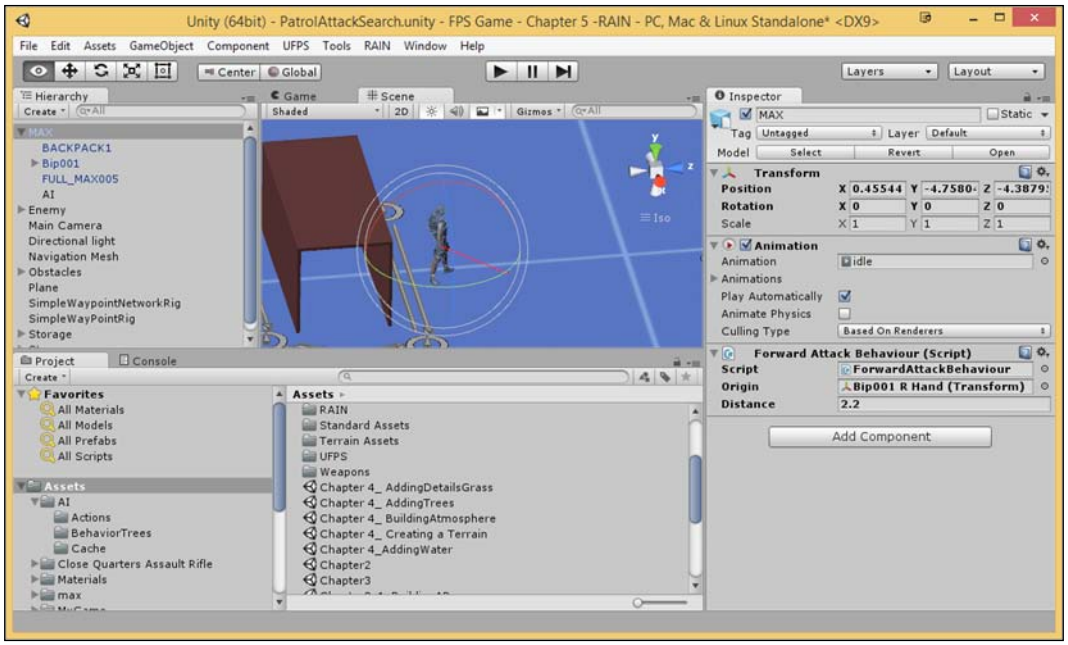
  /// <summary>
  /// How far from the origin will we look for an object to
  /// hit.
  /// </summary>
  public float distance = 2.2f;


  public void AttemptAttack(float damage)
  {
    RaycastHit hit;
    Vector3 direction = Quaternion.Euler(0, 90, 0) *
      origin.forward;

    if(Physics.Raycast(origin.position, direction, out hit,
                       distance))
    {
      RAIN_DamageSender.SendDamage(hit.transform, damage);
    }
  }

  private void OnDrawGizmos()
  {
    if(origin != null)
    {
      Vector3 direction = Quaternion.Euler(0, 90, 0) *
                          origin.forward;
      Debug.DrawLine(origin.position,
              origin.position + (direction * distance),
                       Color.red);
    }
  }
}
```

7. Add the **ForwardAttackBehaviour** component to the `MAX` object, and then from the **Inspector** tab, go to the component and assign **Origin** to the child of Max called `Bip 001 R Hand` (use the **Hierarchy** tab's search bar to find the object easily).

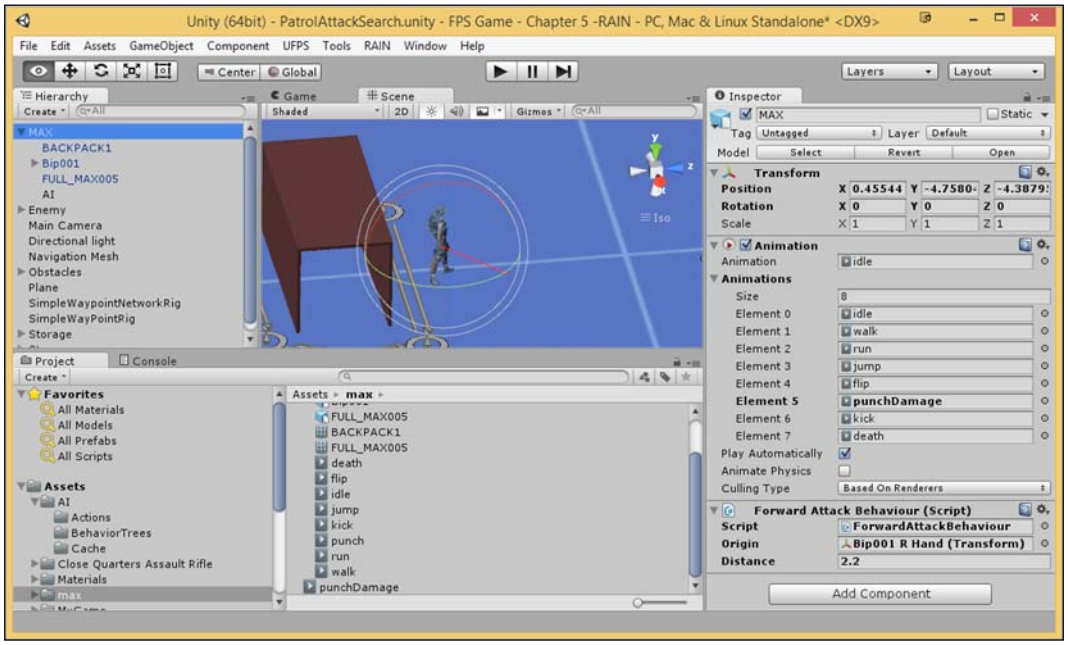


Note the red line being drawn. This is due to the `OnDrawGizmos` function that we added previously, which draws a line in the editor for us to see exactly where we will be checking when `AttemptAttack` is called.

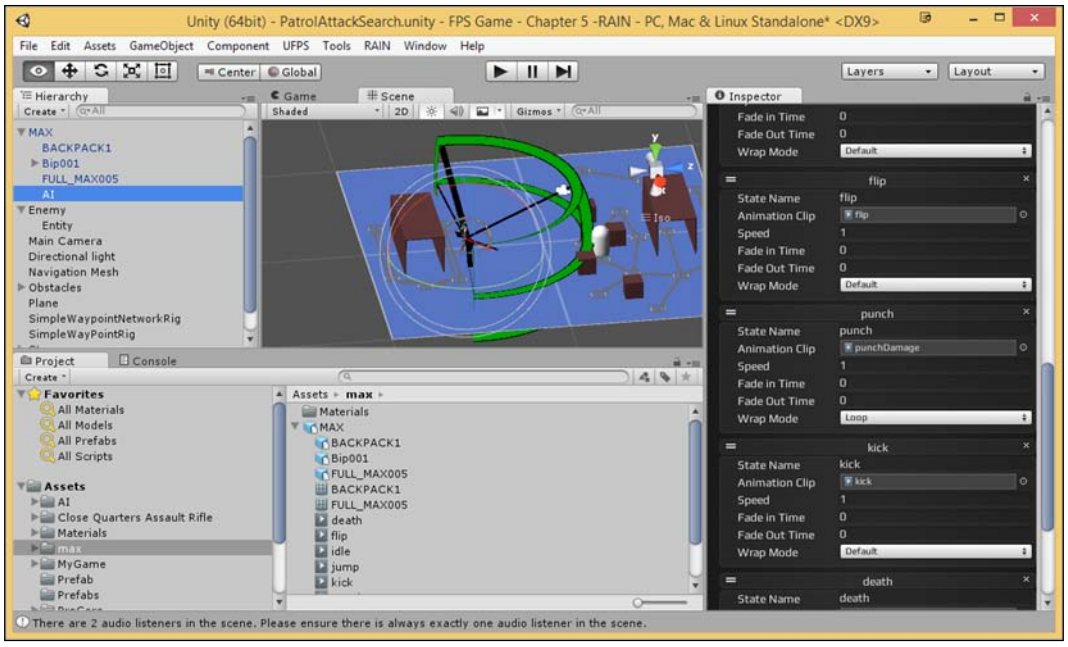
Next, we need to customize our punch animation so that it calls the `SendDamage` function for us when the AI attacks. When the `.fbx` file is imported with animations, it is in a **ReadOnly** mode, so we will need to duplicate it in order to make any modifications to the original version of the animation by performing the following steps:

1. To do this, go to the `Assets/max` folder and select the punch animation. Once selected, hit *Ctrl + D* in order to duplicate `AnimationClip`. Rename the newly created animation as `punchDamage`.

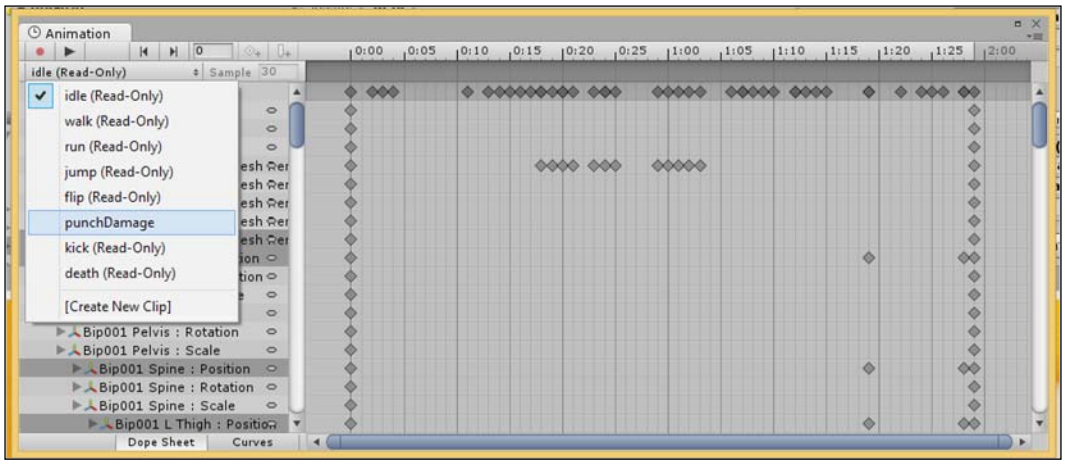
2. Then, we need to tell the character to use this punching animation instead. To do this, go the **Hierarchy** tab, select the **MAX** object, and from the **Animation** component, expand the **Animations** property. From there, replace **Element 5** from `punch` to our `punchDamage` by dragging and dropping the new element into it.



3. We also need to update `AI` so that it plays the correct animation. Select the `AI` object, and then from the **AI Rig** component, select the **Animation** tab, which looks like a running person. From there, scroll down until you get to the punch state, and then change **Animation Clip** to `punchDamage`.



4. Next, we need to actually modify our animation, which we can take a look at by selecting the **MAX** object and then navigating to **Window** | **Animation**.
5. After this, go to the drop-down menu below the record button and select the `punchDamage` option:



Once this is done, you'll see a number of little diamonds that are keys and ways in which an animator can set where each part of this character will be at a certain time.



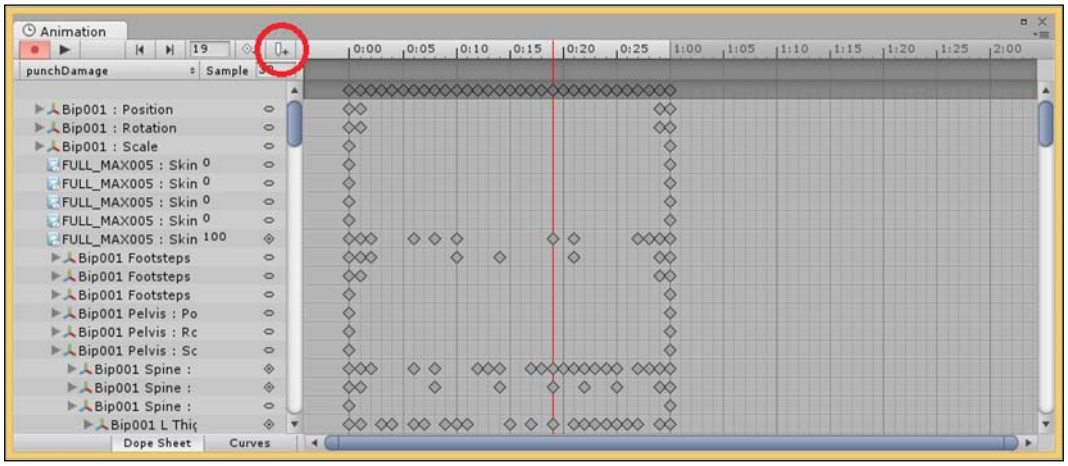
If you have not used the **Animation** tab before, or you want to know more about this, check out `http://docs.unity3d.com/Manual/animeditor-UsingAnimationEditor.html`.

6. We want to attempt an attack when the enemy's fist is all the way extended, which is at frame 19, so from the little box that currently says `0`, type `19` and hit *Enter*.



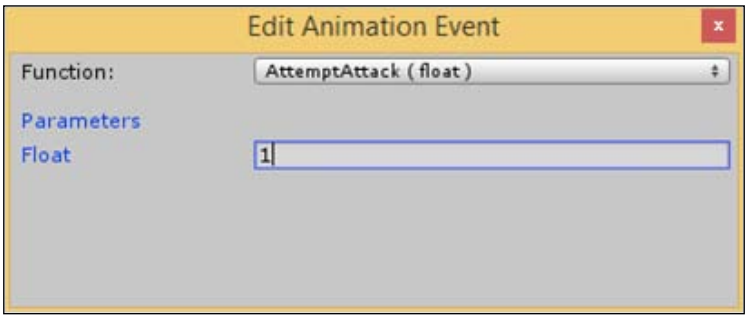
Alternatively, you can also use the **playback head** (the red line) to view the animation and find where you'd like to place your keyframe.

7. From here to the right-hand side of the little box showing the frame we are at, you'll see two buttons to the right-hand side of the first one with a diamond (**Add Key**). Then, you'll see the second one, which looks like a vertical rectangle with a pointed end. This one is called **Add Event**; click on it.



8. In the menu that pops up from the **Function:** menu, select the `AttemptAttack` function, and then for the **Float** value, put in the amount of damage you want to do to the player if he punches you. In my own case, I'll use `1`.

   Currently, in the `AdvancedPlayer` prefab, the **Vp_FP Player Damage Handler** component lists our player's **Max Health** as `10`, so if we decrease the value by 1, it'll be 10% of their health:



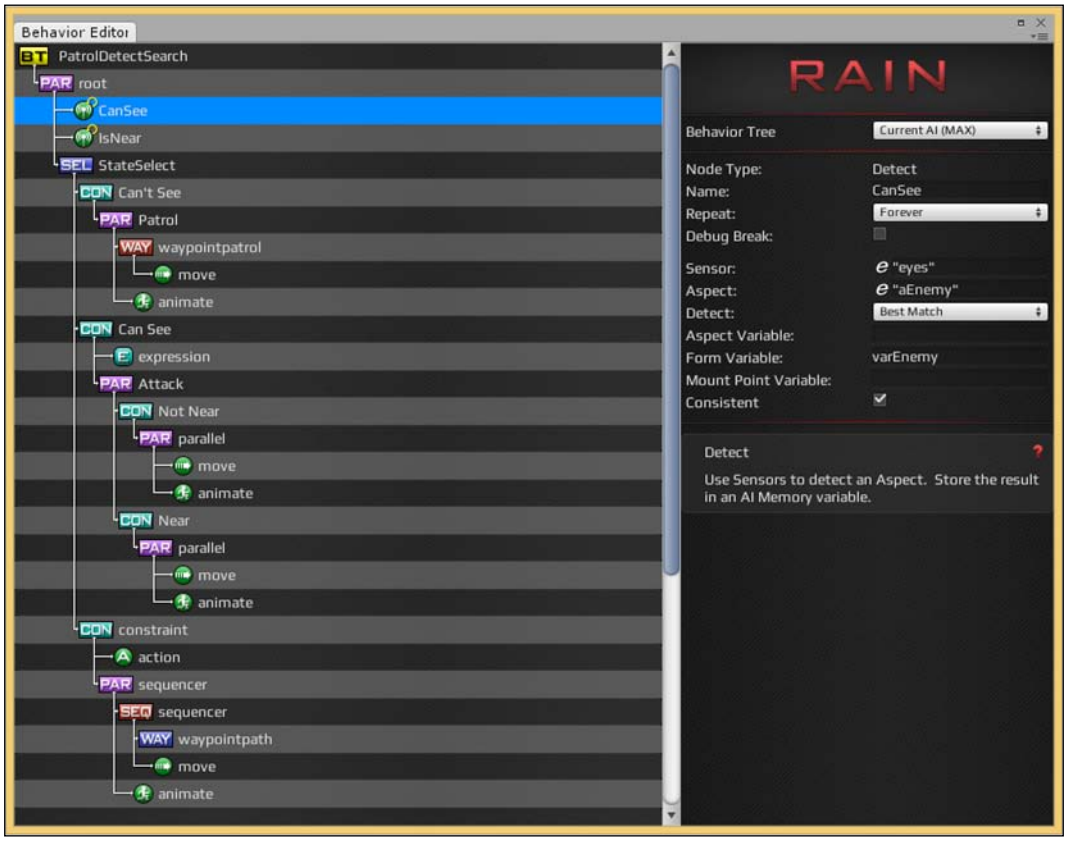
9. Next, let's bring in a player and create a scenario where our RAIN character recognizes this player. Drag and drop an `AdvancedPlayer` prefab from the `Assets/UFPS/Base/Content/Prefabs/Players` folder into the game world. After adding our player, we won't need to have `Main Camera` in the scene anymore. So, select it in the **Hierarchy** tab and then delete it by hitting the *Delete* key.

   If you were to play the game, you'd notice that Max doesn't notice us because we aren't registered as an enemy to him.

10. From the **Hierarchy** tab, select the `Enemy` object and expand it to show its child named `Entity`. Select this object and note that it has an **EntityRig** component.

    RAIN provides the **Entity** component as a way to encapsulate the attributes, properties, or characteristics of the game object it is associated with, which the AI can then reference later on in its behavior. This can be done by defining custom elements and/or aspects. In this instance, we have said that this object has an aspect called `aEnemy`.

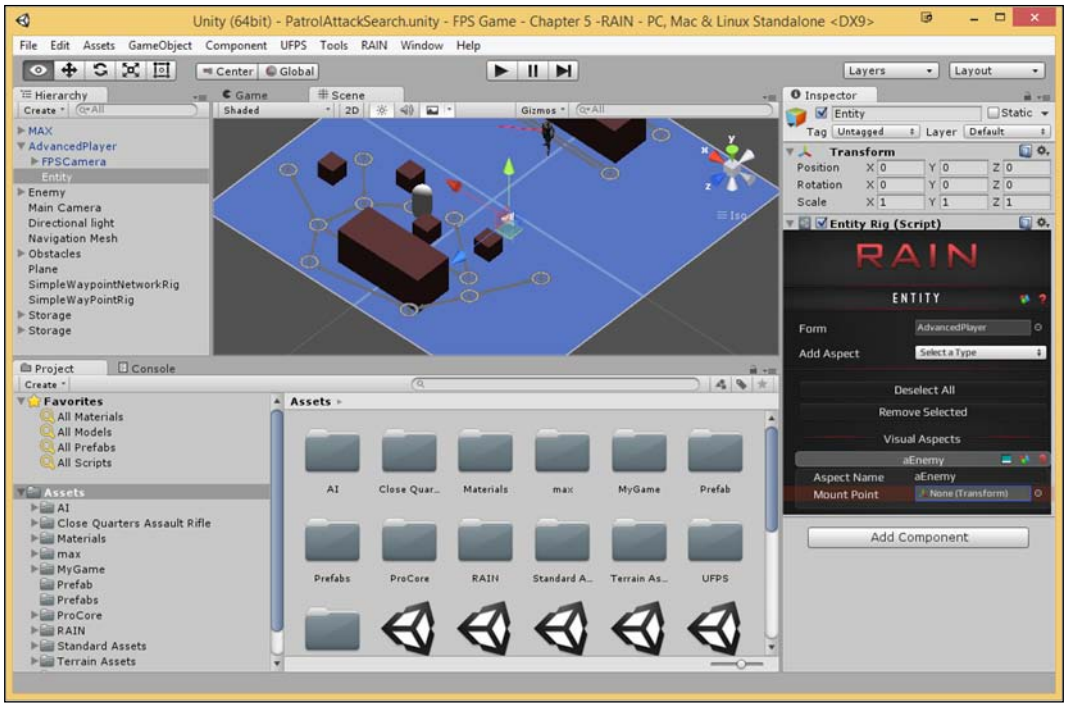
If we open **Behavior Editor** for the `MAX` object by navigating to **RAIN | Behavior Tree Editor**, you'll notice that the `CanSee` sensor is looking for an aspect of `aEnemy`. If it finds it, it will assign the `varEnemy` variable to this object, and when `varEnemy` has a value, the AI will follow this enemy to the best of its ability.





For more information on RAIN's Entity Rig component, check out `http://rivaltheory.com/wiki/gettingstarted/createentity`.

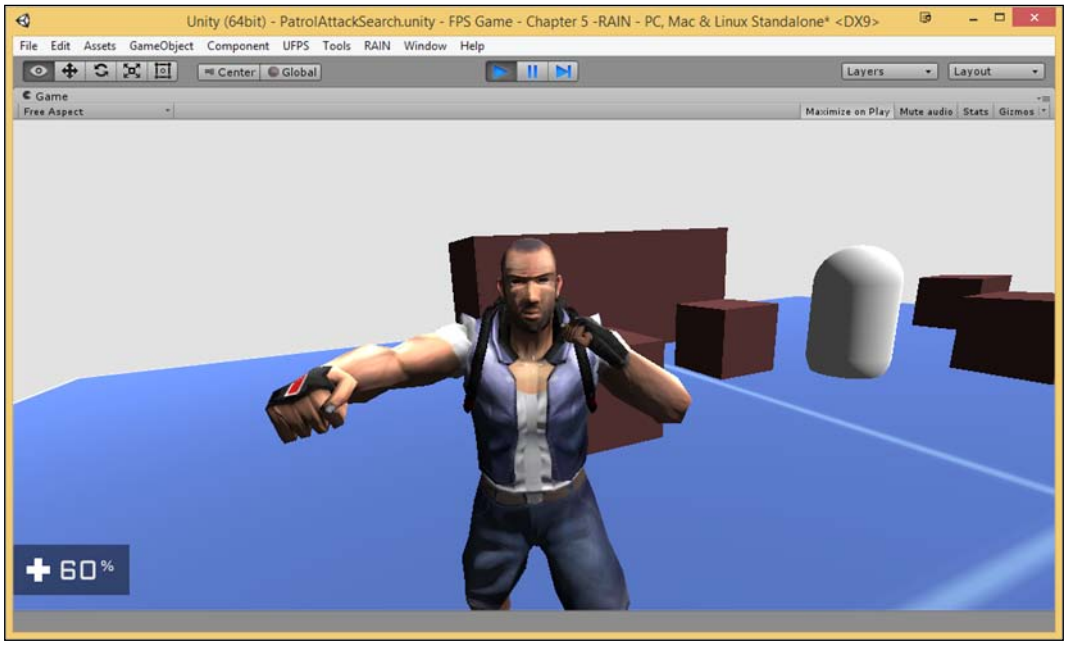
11. Duplicate the Entity object (*Ctrl* + *D*), and then drag it over to be a child of the `AdvancedPlayer` object that we created in the **Hierarchy** tab. Next, change the **Entity Rig** component's **Form** variable to the `AdvancedPlayer` object by dragging and dropping the object into the slot.





It's also possible for you to create an Entity by selecting the **Advanced Player** object from Hierarchy. Then, from the toolbar at the top of the window, navigate to **RAIN** | **Create New** | **Entity**. After it's created, move over to the Inspector tab and go down to our newly created component. From there, select **Visual Aspect** from the **Add Aspect** dropdown, and then from the newly created aspect, change **Aspect Name** to `aEnemy`. Remove the value for **Mount Point** by selecting it and hitting the *Delete* key.

12. Save our project and play the game to verify that everything is working correctly.



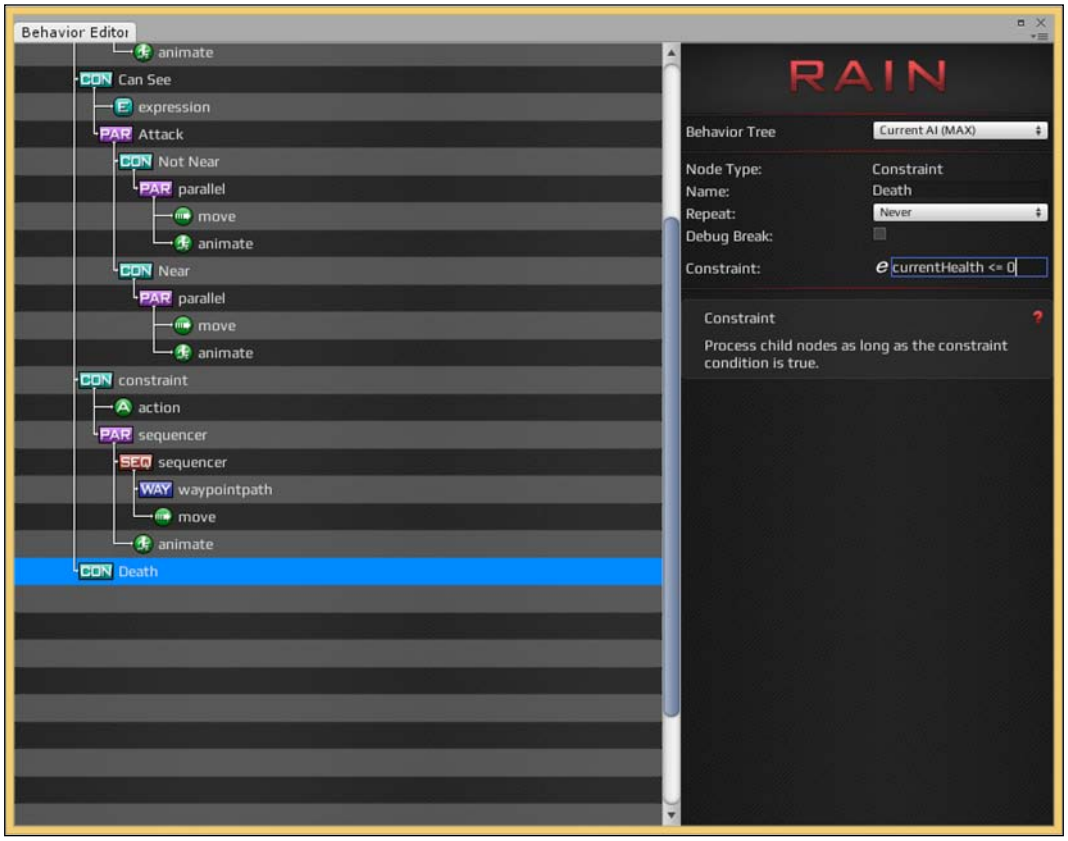
And as you can see, we can now be attacked by the character and UFPS updates its health correctly! Now, we just need to get the other way around working.

13. The first thing we need to do is create a collider of some sort for UFPS to recognize that the player has been hit as the current AI has no collision. Select the AI object once again and add a **Capsule Collider** component. Once done, change the **Y** value of **Center** to `1`, **Radius** to `.1`, and **Height** to `2`. This is done so that UFPS can communicate with the **RAIN_Damage Reciever** function.

    If we wanted the character's collision box to be very accurate, we can always add the `Box Collider` and `DamageReciever` components around each of the parts of the character rig. However, the more you add, the more computationally expensive the calculations will be. The advantage of doing this is that your bullet marks will look much better.

14. After this, we need to have the AI react to taking damage and dying. Select `MAX`, go to the toolbar at the top of the window, and navigate to **RAIN | Behavior Tree Editor**.

15. Once there, right-click on the `StateSelect` option by navigating to **Create** | **Decisions** | **Constraint**. Then, change **Name** of the newly created constraint to `Death`, and under **Constraint**, add `currentHealth <= 0`.





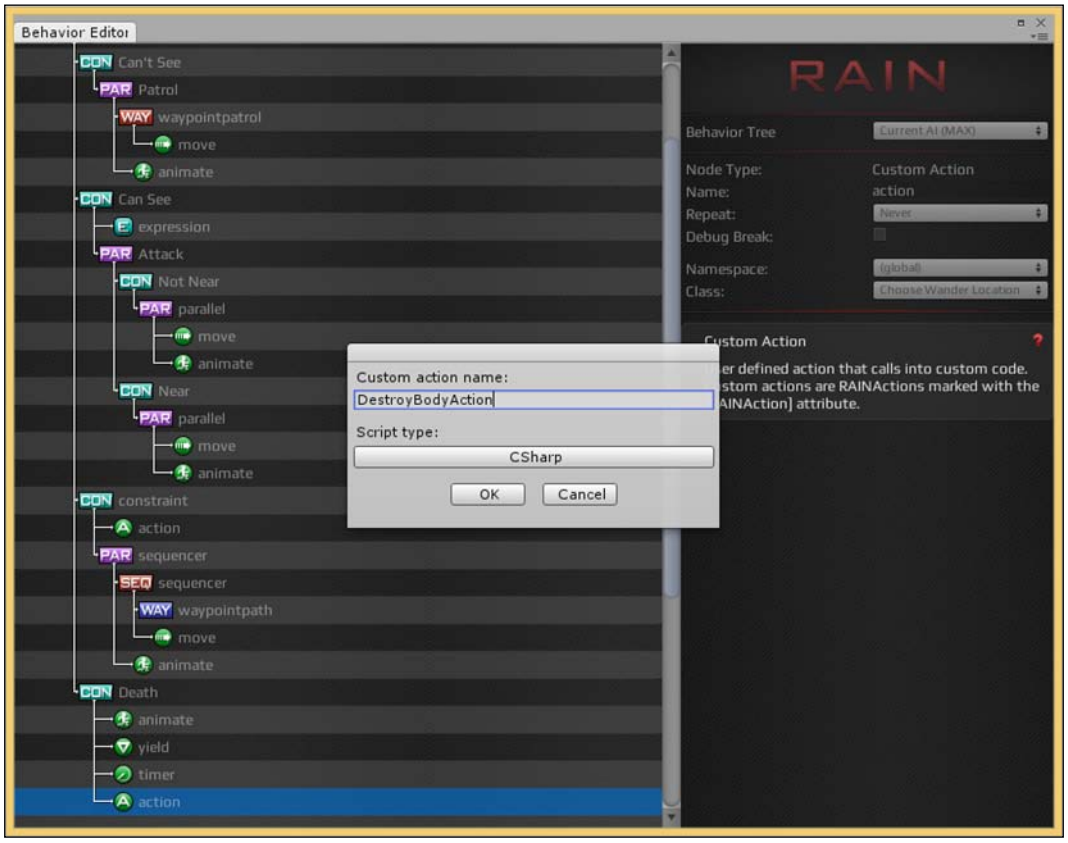
For more information on the Behaviour Tree Editor and to see how you can debug it while the game is on, check out `http://rivaltheory.com/wiki/behaviortrees/behaviortreeeditor`.

16. Next, we need it to play the proper animation, so right-click on the Death constraint, and then navigate to **Create** | **Actions** | **Animate**. In **Animation State:**, put in `death`. This is done so that it plays the death animation.
17. After this, we don't want the character to start moving again as we want it to get destroyed after a while. So, let's create a new yield action first by right-clicking on and navigating to **Create** | **Actions** | **Yield**.

18. Afterwards, create a timer by navigating to **Create** | **Actions** | **Timer** and plug in `5` for **Seconds**.

    This is done so that when we die, we can play the animation before removing the character from the scene.

19. Next, we need to add in some custom code to actually destroy our character, so right-click on `Death` once again, and this time, navigate to **Create** | **Actions** | **Custom Action**. From the **Class:** property, select `Create New Custom Action`. Under **Custom action name:**, add `DestroyBodyAction` and make sure that **Script type:** is still at `CSharp`.



20. Once you've confirmed the values, click on the **OK** button and open up the script MonoDevelop (it's located in your `Assets/AI/Actions` folder) and put in the following bold lines:

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

```
using RAIN.Core;
using RAIN.Action;

[RAINAction]
public class DestroyBodyAction : RAINAction
{
    public DestroyBodyAction()
    {
        actionName = "DestroyBodyAction";
    }

    public override void Start(AI ai)
    {
        base.Start(ai);
    }

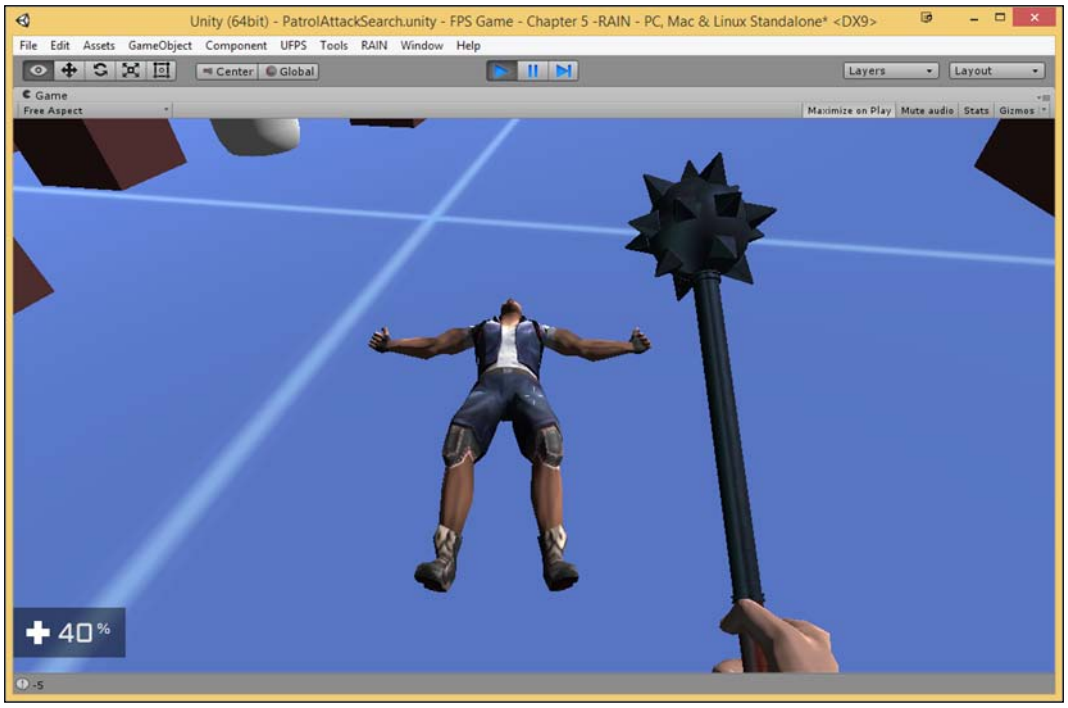
    public override ActionResult Execute(AI ai)
    {
      MonoBehaviour.Destroy(ai.Body);
        return ActionResult.SUCCESS;
    }

    public override void Stop(AI ai)
    {
        base.Stop(ai);
    }
}
```

In addition to this, you can also remove the lines that I've commented out since the script doesn't actually use them.

21. In the behavior tree, we also need to make sure that we can exit out of the constraints in order to go to `Death`. Under **Can See** for **Constraint**, put in: `(varEnemy != null) && (currentHealth > 0)`. For **Can't See**, use `varEnemy == null && DoSearch == 0 && currentHealth > 0`. Then, under the first constraint, put in `varEnemy==null && DoSearch==1 && currentHealth > 0`. With this in place, we should be finished with the behavior, so go ahead and exit the menu!
22. Finally, let's rename `MAX` to `RAIN_Enemy` and drag and drop it to the `MyGame\Prefabs` folder so that it can be used later on.
23. In addition to this, select the **Advanced Player** object, and click on the Prefab Apply button so that the other advanced players contains the changes we've made as well.

24. Save your script and now try out the project!





For those of you interested in using RAIN, check out another useful tutorial that is specific to RAIN itself at `http://cocoateam.com/post/105258624839/creating-a-fully-functional-enemy-behavior-using`.

At this point, we now have an enemy that can both damage us as well as be damaged, and we've also touched on how to remove enemies from a scene after a period of time!



Rival Theory have a premium version of AI characters made in RAIN called Squad Command or the Advanced Warfighter AI, which can be found at Unity Asset Store. This can be useful for those of you who wish to get more complex AI up and running fairly quickly, but it does cost money. At the time of writing this, you'll also still need to add the changes we made in this tutorial to get it to work with UFPS characters. If you're interested in checking the assets out, take a look at `https://www.assetstore.unity3d.com/en/#!/content/23526`.

Integrating an AI system – Shooter AI

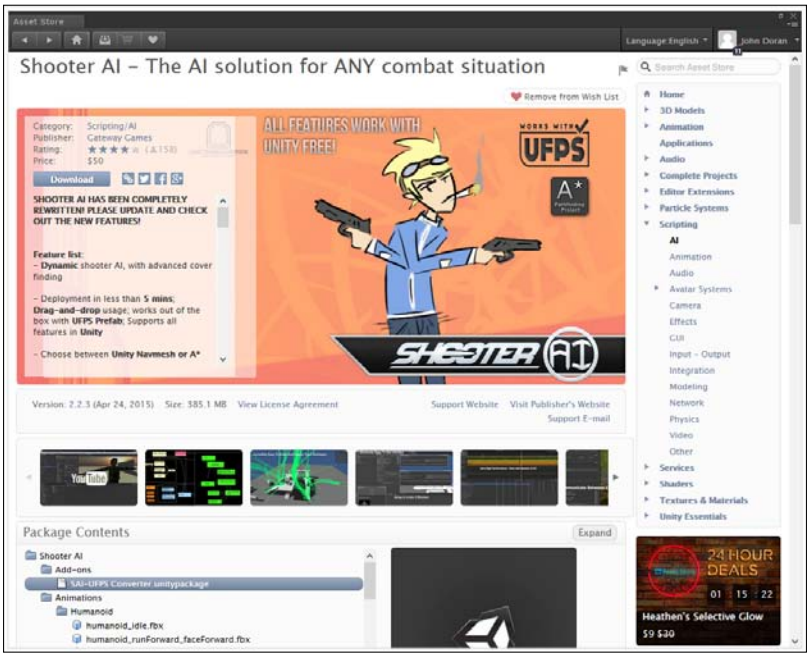
As mentioned in the previous section, there is another tool called Shooter AI, which is currently available at the Asset Store. However, unlike the base RAIN program, it is not free and will require a purchase. However, it is the simpler of the two to set up and also uses Unity's own NavMesh system instead of RAIN's proprietary version. Keeping this in mind, let's create a ranged enemy to shoot at us.



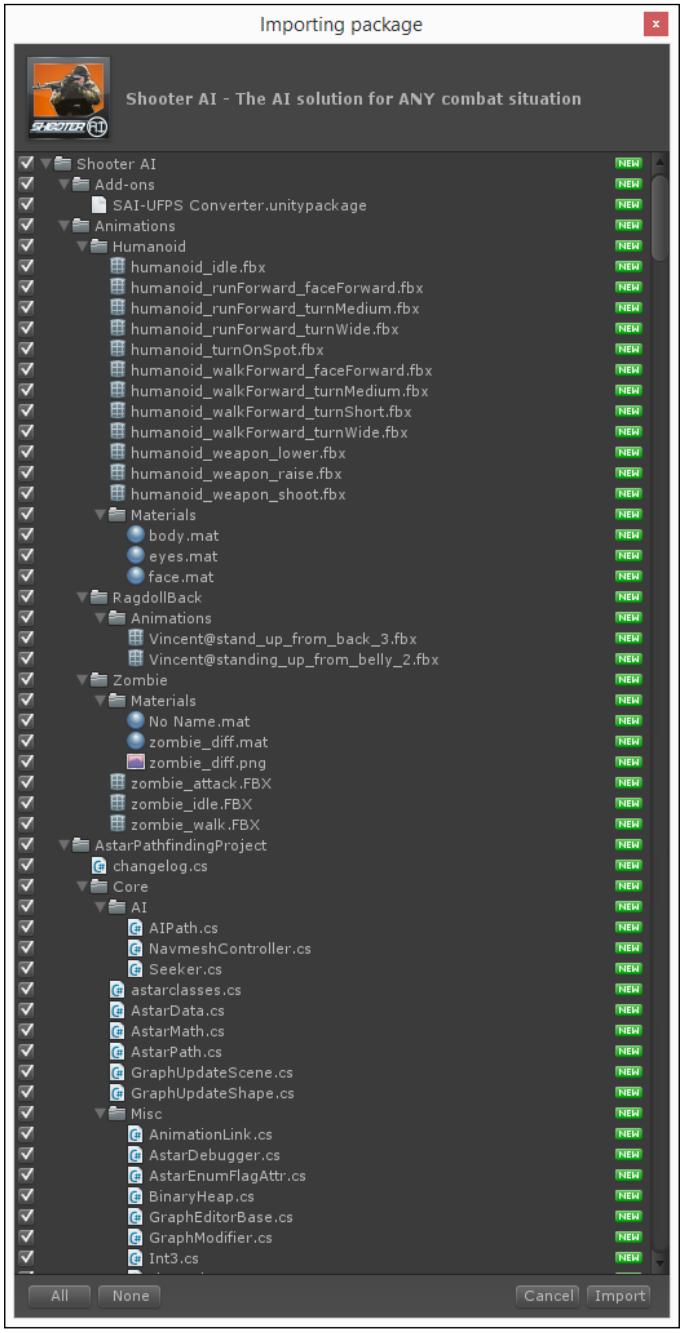
As of this writing, there is talk of AI Shooter being replaced by another AI System called Paragon Shooter AI.

Most of the content regarding the AI system should work similarly to what's described here. For more information on that check out: `http://forum.unity3d.com/threads/paragon-ai-first-third-person-shooter-ai.305971/`.

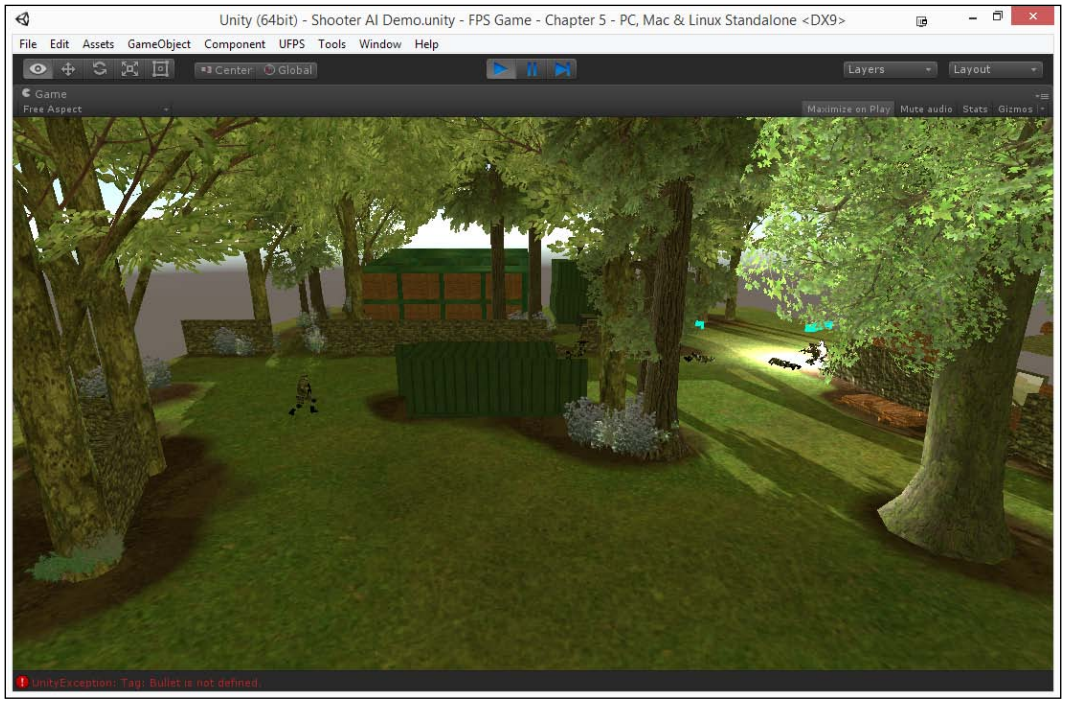
1. To download Shooter AI, we'll use the Asset Store again. In order to do this, navigate to **Window | Asset Store**.
2. In the top-right corner, you'll see a search bar. Type in `Shooter AI` and press *Enter*. Once you do this, the first asset you'll see is **Shooter AI – the AI solution for ANY combat situation**. Left-click on it, and then once you've purchased the asset, click on the **Download** button and accept the license agreement.



3. Once it finishes downloading, you should see the Importing Package dialogue box pop up. (If it doesn't, click on the **Import** button where the Download button used to be):

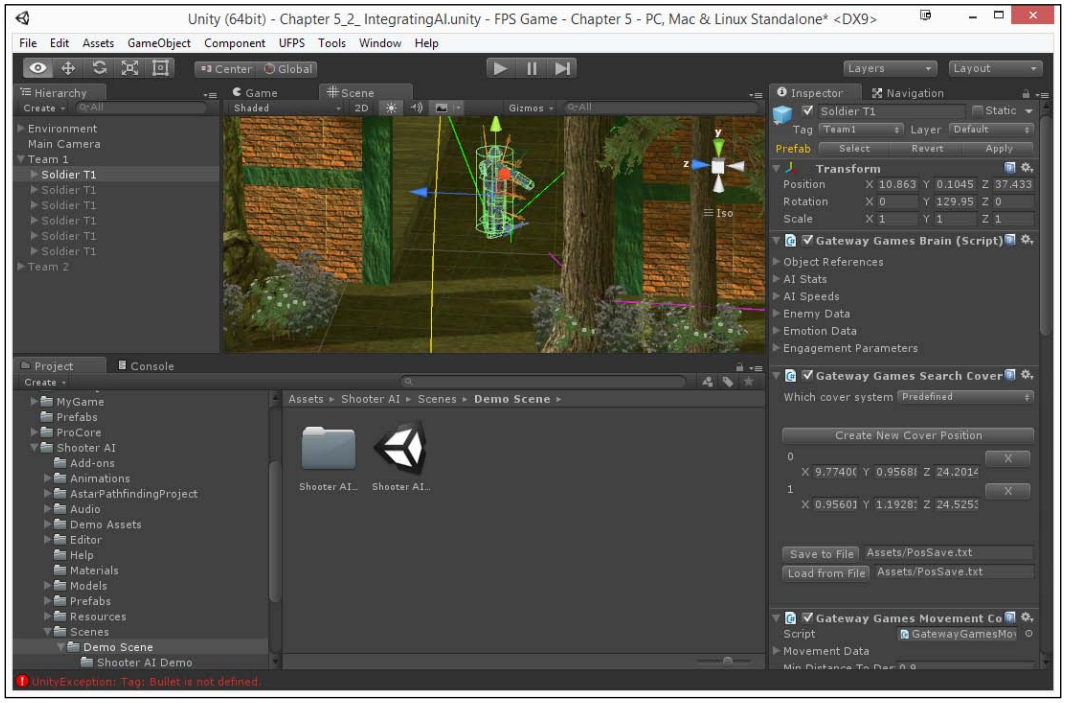


4. Click on the **Import** button to place all the packages into our project and wait for it to finish importing into the project.
5. Close the Asset Store if it's still open, go back into our game view, and you will notice the new **Shooter AI** folder placed in the **Assets** folder. Double-click on it and then enter the **Prefabs** folder.
6. Now, Shooter AI contains a lot of new things for us to work with, the first of which is a new demo map and prefabs for enemies. To take a look at the new demo level, go to the **Project** tab. Then, go to the `ShooterAI/Scenes/Demo Scene` folder and click on `Shooter AI Demo.unity`.
7. Upon opening the map, you should be able to click on **Play** and watch a team versus team combat.

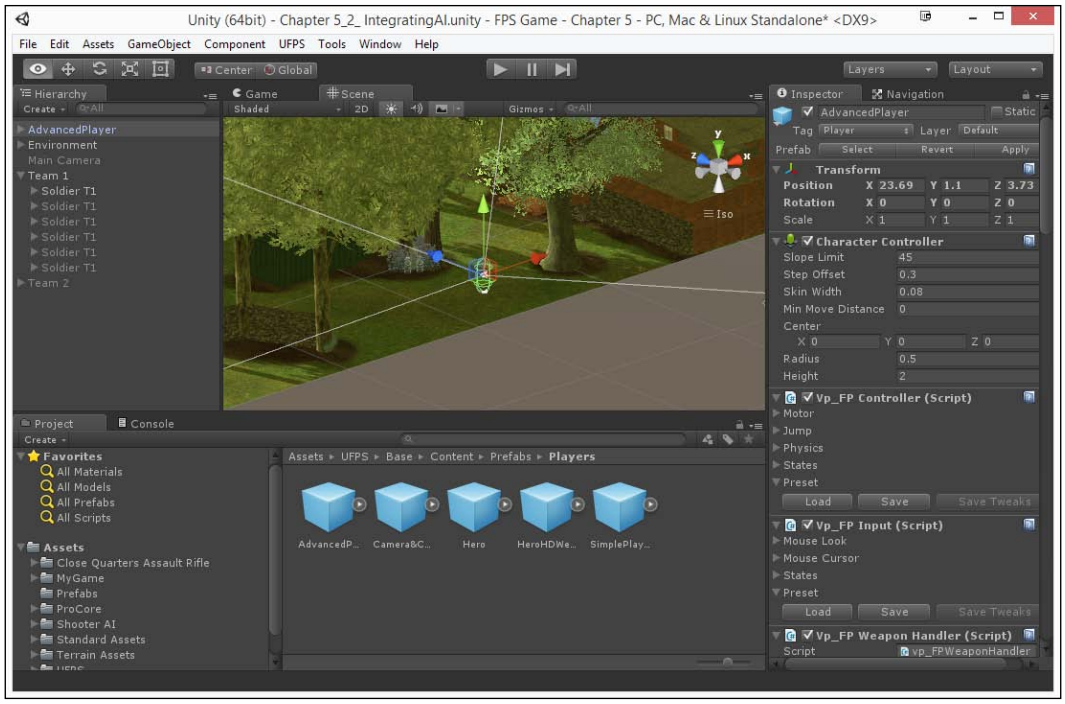


Pretty cool, huh? Once this is done, we'll know that Shooter AI is installed properly.

8. To get started working with Shooter AI, we're going to place our player into this level and get the soldier to work as an enemy for us. This level contains two objects in **Hierarchy**: `Team 1` and `Team 2`. Disable `Team 2` completely by checking off the checkmark beside the object's name. Also, deactivate all but one of the soldiers from `Team 1`. Double-click on the enabled object to zoom into him.



9. Now, we need to customize the solder to actually work with UFPS. To do so, select the `Solder T1` object, select the **Gateway Games Brain** component, and open the **Enemy Data** variable. Under **Tag of Enemy:**, select **Player**.
10. After this, we will want to add our player into this map. Then, go to `UFPS/Base/Content/Prefabs/Players` and drag and drop the `AdvancedPlayer` prefab into the level away from the enemy.

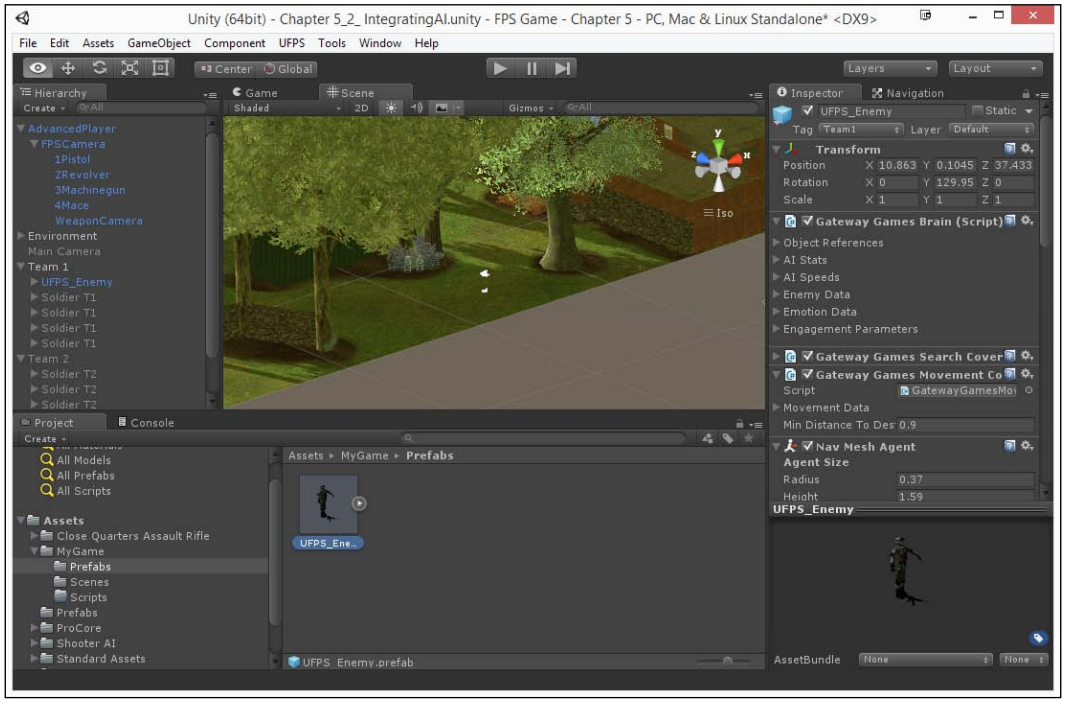


These are a great first few steps. If we play the game now, the player can walk around and can damage the enemy… but when the enemy attacks back, it does not damage us! To fix this we will need to modify how damage is done by the Shooter AI.

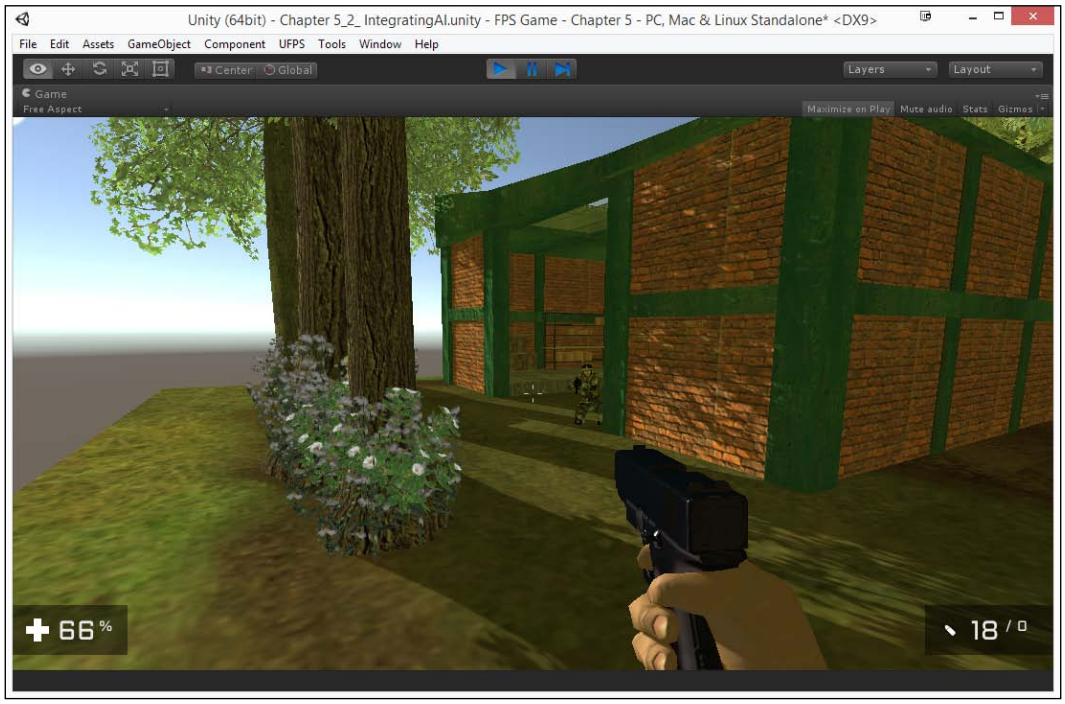
11. Open the `Soldier T1` children and select the `MainWeapon(Clone)` object, and then in **Inspector** under the **Gateway Games Weapon** component, scroll down and change the **Damage Model** property to **Global**.

    The reason we have chosen **Global** is that UFPS does damage to enemies no matter where you hit the object; Shooter AI by default will do more damage if you hit the head instead of the body, but UFPS doesn't support this by default.

12. Now that we have enemies ready to be used in our project, let's create a prefab of our own that we can use in our own levels. Select our Soldier object and navigate to **GameObject | Break Prefab Instance**. After this, rename it to `UFPS_Enemy` so we know that it's to be used with our system. Then, open up the `MyGame/Prefabs` folder and drag and drop the `UFPS_Enemy` object there.

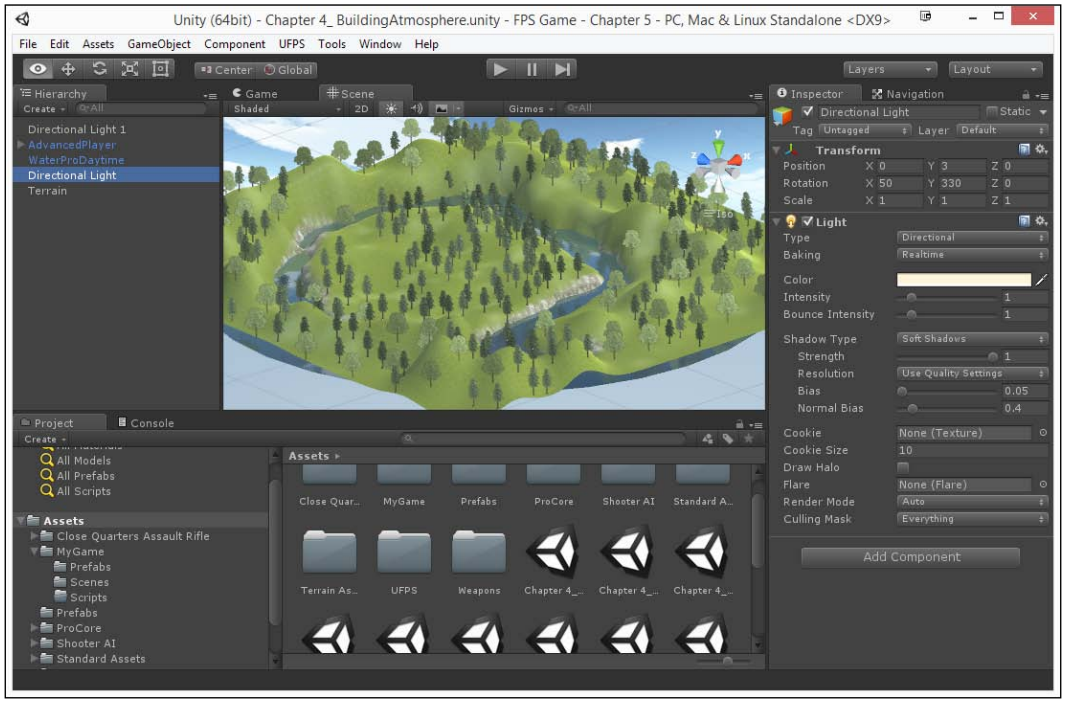


13. Lastly, save your level and play the project!



This is now starting to look really nice! We have an enemy that is ready to be used inside our actual game!

14. Now to continue with this train of thought, let's go ahead and spawn the AI inside one of the levels we've created. To do this, let's open the map that we created in *Chapter 4, Creating Exterior Environments*.

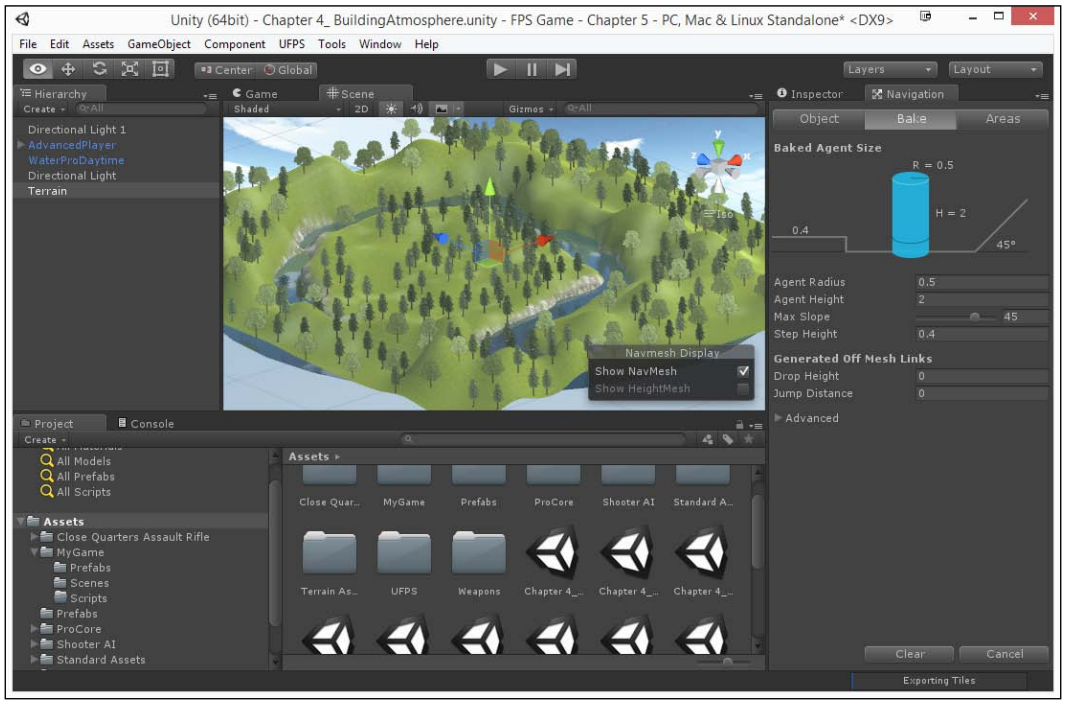


We next need to add Navigation information to the map in the form of a navigation mesh, **NavMesh** for short, which is what gives AIs information on where they can and can't go.

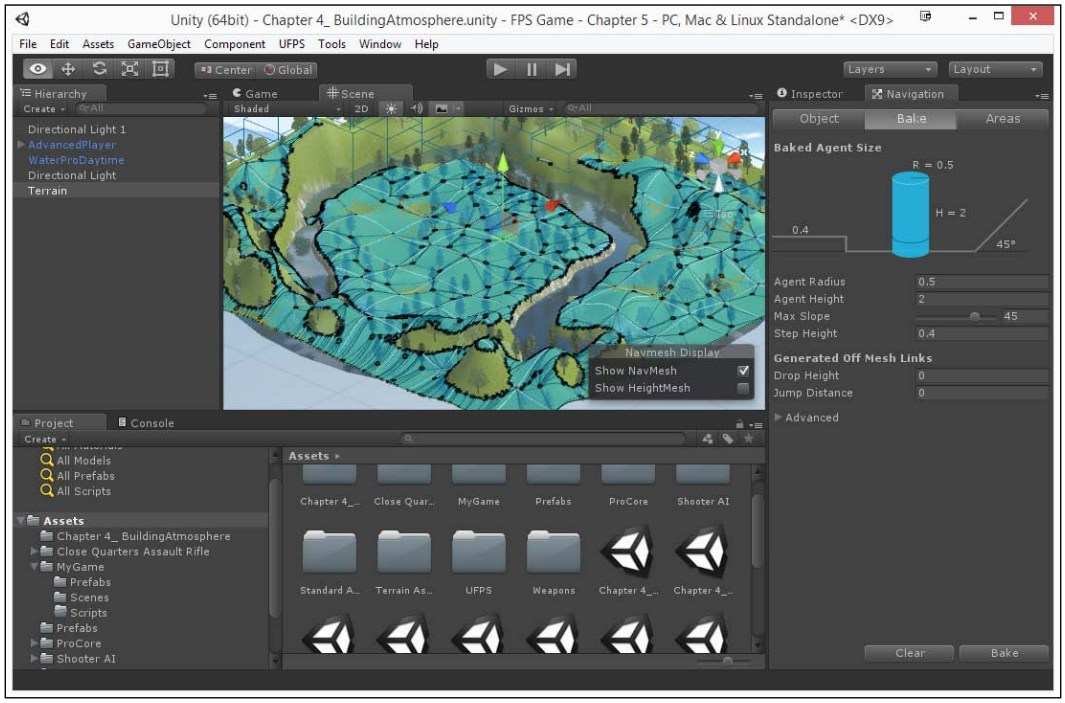


Note that RAIN does not use this method of navigation for its characters. We will talk about how RAIN's form of NavMeshes work in the *Spawning enemies with the help of a trigger* section.

15. Before we can make a NavMesh, we need to make our environment static, which is to say that our characters cannot move. Select the `Terrain` object and confirm that the **Static** checkbox from the top right-hand side menu is checked.
16. To create the NavMesh, we need to navigate to **Window** | **Navigation**. From here, we will then select the **Bake** tab. Leave the default parameters as they are now, and then click on the **Bake** button.

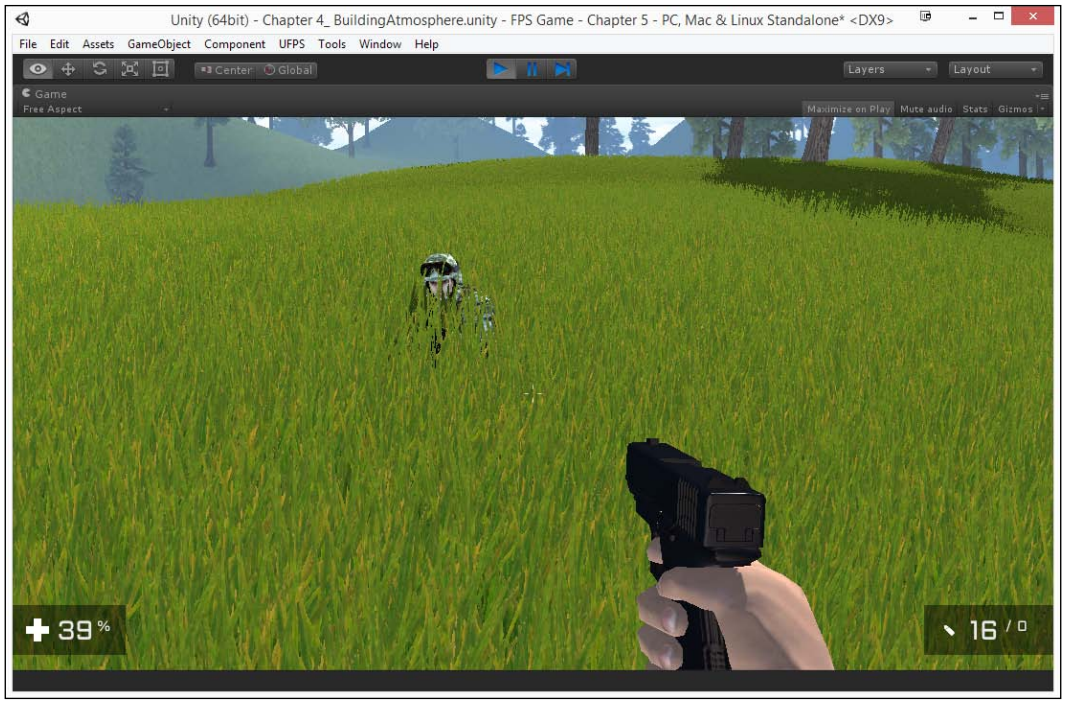


Next, we will need to wait for it to finish, which may take a while based on how powerful your computer is. Once it's finished, we should be able to see a ton of blue on the screen:



This blue area represents places that the AI can actually travel to.

17. Now, drag and drop `UFPS_Enemy` into the game world where the player was spawned. He will now wander the area using the waypoint specified in the **Gateway Games Patrol Manager** component and will engage with you when he sees you.



With all this in place, we've seen how easy it is to integrate an AI system into our project!



If you're interested in checking out the documentation for Shooter AI and learning more about the specific variables that are used for it, check out `https://www.dropbox.com/s/u7wi7wb8azmmlaf/Manual.pdf?dl=0`.

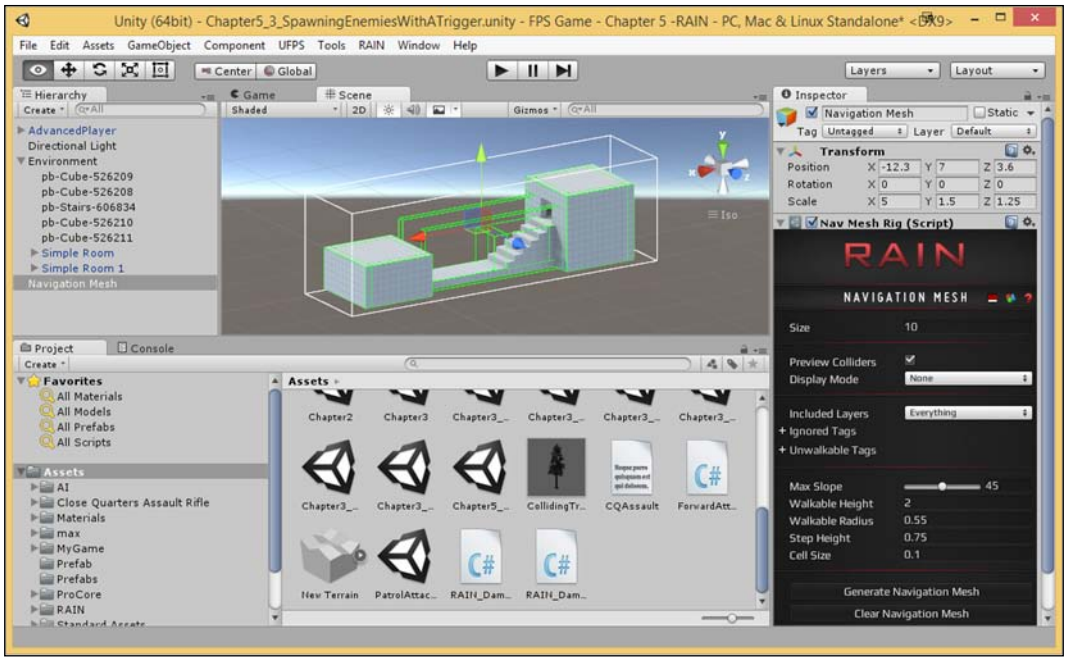
Spawning enemies with the help of a trigger

Enemies are computationally expensive to have spawned throughout the entire game project. A little trick that we use as game developers is spawning enemies right before a player enters an area; this way, they don't need to be created until they're ready to be seen! We can to this through the use of a property called a **trigger**.

1. Let's first open the level that we created in *Chapter 3*, *Prototyping Levels with Prototype* that we used earlier in this chapter. We will need to add in navigation data to make things easier to work with. Create an empty game object by navigating to **GameObject | Create Empty** and set **Position** as `0,0,0`. Then, call the game `Environment`.
2. Once we have the object created, drag and drop all our cubes and rooms into it as children. After this, click on the **Static** button to ensure that the objects don't move by selecting **Yes, change children**. This is an optional step as RAIN can still make a NavMesh without this, but it's a way for us to show that we will not, make these objects move. If you are using Shooter AI, however, you must perform this step.

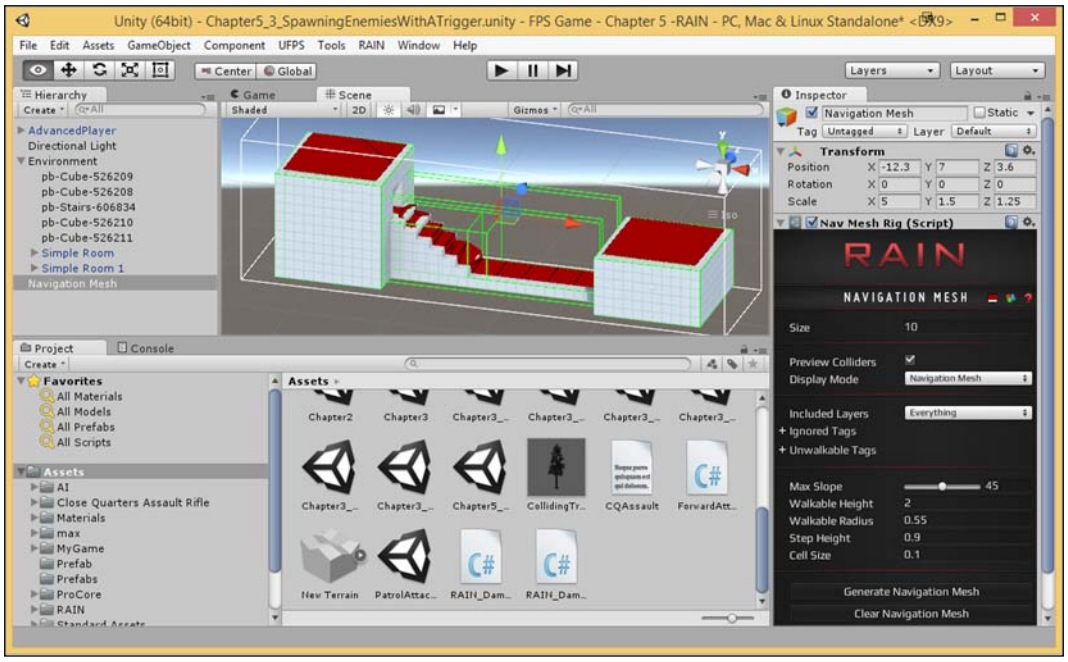
   For RAIN, follow these steps:

   1. Now, in order for the RAIN characters to recognize where to travel, we need to use RAIN's own Navigation Meshes. To do this, we will need to go to the toolbar at the top of the window and navigate to **RAIN | Create New | Navigation Mesh**. This will create a white box within which we will want to have our entire area fit in by scaling the object.



2. Once you've finished resizing, click on the **Generate Navigation Mesh** button and wait for it to finish its calculations.
3. After this, switch to the navigation tab and under the **Bake** section, click on the **Bake** button. You'll see it complete, but the steps will not show up correctly.

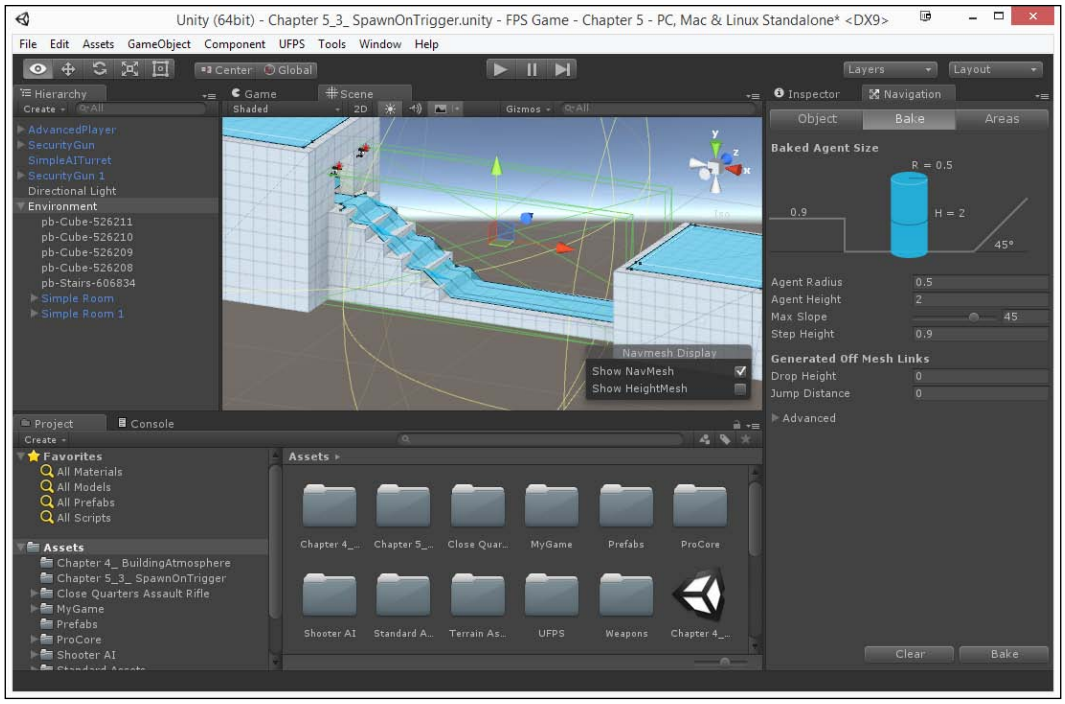
4. Set **Step Height** to `.9` and run it again.



5. Now, jump to step 3.

For Shooter AI, follow these steps:

1. Switch to the navigation tab and under the **Bake** section, click on the **Bake** button. You'll see it complete, but the steps will not show up correctly.
2. Set **Step Height** to `.9` and run it again.



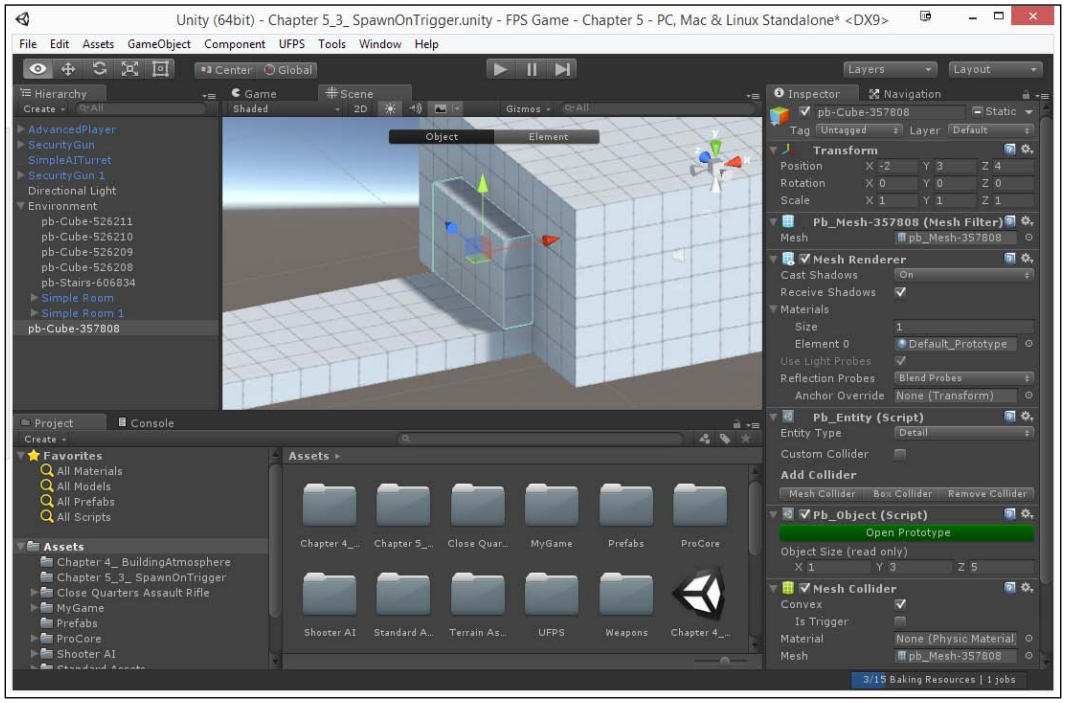


Another way to use navigation is through a system called **A***, which finds paths at runtime and is included in Shooter AI. For more information on how to set it up, check out `https://www.youtube.com/watch?v=4p3ZehTNL6s`.

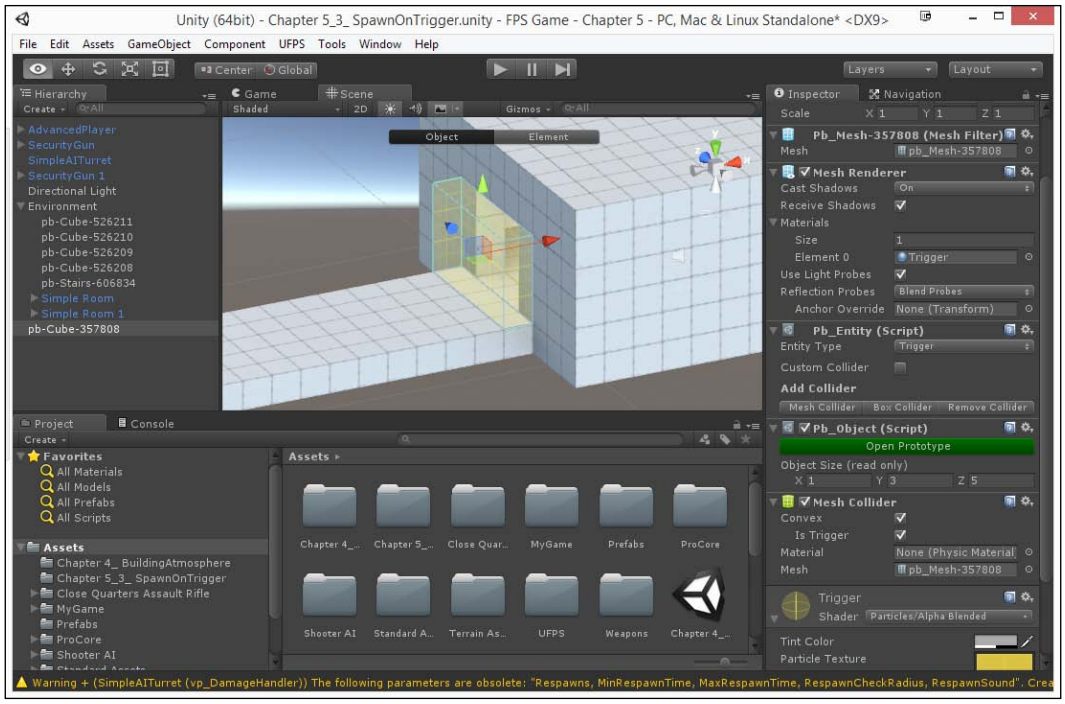
3. Now, jump to the next step.

3. Now that we have our navigation mesh (NavMesh for short) completed, it's time to spawn our enemies. My plan is that when we exit the first door, an enemy will be spawned in the other room and run out to see the player.

4. Open up Prototype again, which we learned before in *Chapter 3*, *Prototyping Levels with Prototype*, and create a cube (*Ctrl* + *K*) that covers the door to the first room.



5. Once you have the room created, from the Prototype window, switch to Object mode if you haven't already, and then click on the **Set Trigger** button. You'll see that it has changed color to a yellow semi-transparent box, as shown in the following screenshot:



6. You'll also notice that on the **Mesh Collider** component, the **Is Trigger** property has also been toggled. This means that Unity will let us know when a collision has occurred, but it will not stop the player from moving.

   Now we need to create a new component for the trigger to spawn our enemy. In order to do this, we will need to dive into some code.

7. Go to the `MyGame/Scripts` folder, navigate to **Create** | **New C# Script**, and call it `SpawnEnemyOnTrigger`. Once finished, double-click on the created file to open up MonoDevelop, the built-in development environment for Unity.
8. Once inside, change the code to this:

   If you are using RAIN:

```
using UnityEngine;
using System.Collections;

public class SpawnEnemyOnTrigger : MonoBehaviour {

  // Enemy to spawn
  public GameObject enemy;

  // Where to be spawned at
```

```
  public Transform spawnPoint;

  // Has this happened already?
  private bool hasTriggered = false;

  void OnTriggerEnter(Collider other)
  {
    //If the player touches the trigger, and if
      //it hasn't been triggered before
    if(other.tag == "Player" && hasTriggered == false)
    {
      // Spawn a new enemy using the properties from the
        // spawnPoint object
      GameObject newEnemy = Instantiate(enemy,
                                spawnPoint.position,
                                spawnPoint.rotation)
                                as GameObject;

      // We only want this to happen once.
      hasTriggered = true;
    }
  }
}
```

If you are using Shooter AI, add the following code to have the AI go to the player:

```
using UnityEngine;
using System.Collections;

public class SpawnEnemyOnTrigger : MonoBehaviour {

  // Enemy to spawn
  public GameObject enemy;

  // Where to be spawned at
  public Transform spawnPoint;

  // Has this happened already?
  private bool hasTriggered = false;

  void OnTriggerEnter(Collider other)
  {
    //If the player touches the trigger, and
```

```
        //if it hasn't been triggered before
      if(other.tag == "Player" && hasTriggered == false)
      {
        // Spawn a new enemy using the properties from the
           // spawnPoint object
        GameObject newEnemy = Instantiate(enemy,
                              spawnPoint.position,
                               spawnPoint.rotation)
                               as GameObject;

        //Tell enemy to go to the player
        newEnemy.GetComponent<NavMeshAgent>()
          .SetDestination(other.transform.position);

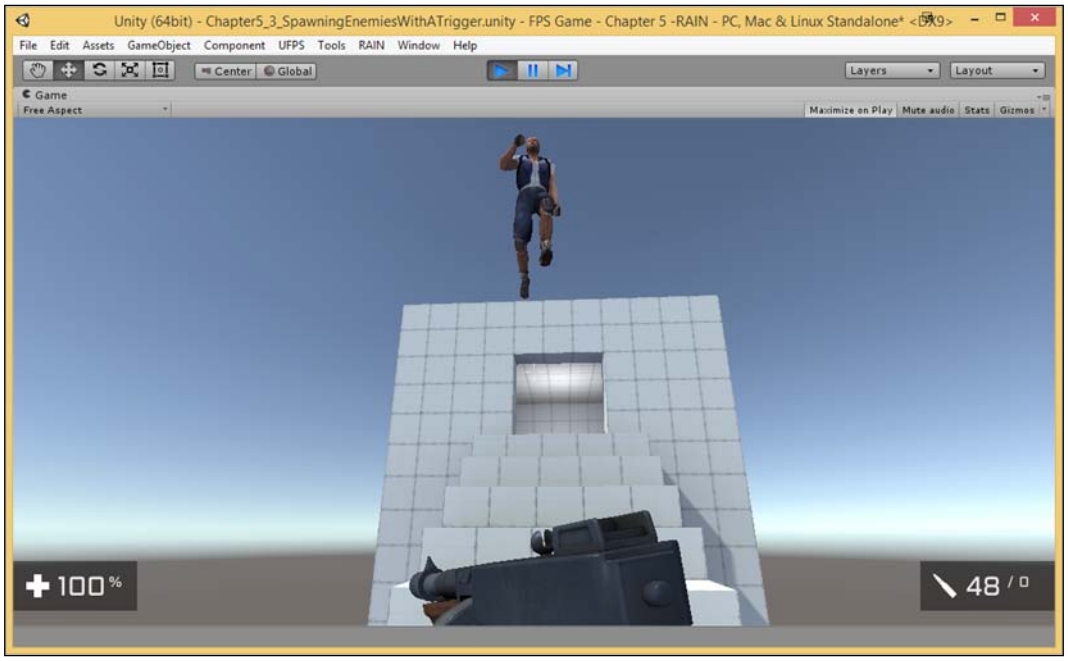
        // We only want this to happen once.
        hasTriggered = true;
      }
    }
}
```

This code does a number of things for us. It uses three variables and a single function. Variables are holders for data, and functions are ways for us to act upon this data. The two public properties we have here are for the object that we wish to spawn (the enemy) and where we want to spawn it (in the room). We also have a private property, which is only available to the class. Since we only want to trigger this once, we need to know whether we've executed this before or not.

The `OnTriggerEnter` function is a built-in method in Unity that is called for us whenever an object of any kind has touched the trigger or when the object has a collider and the **Is Trigger** option is toggled. We first check if it's the player and if it is, we then spawn an enemy. We can later extend this to make it fit a more interesting encounter, but this is a good starting point.

9. Next, we need to attach this component to our trigger. Select it and then drag and drop the appropriate `SpawnEnemyonTrigger` file on top of it. If you are using RAIN, follow along; if you are using Shooter AI, skip ahead to just after step 16.
10. Once the component is attached, drag and drop our `RAIN_Enemy` prefab under the Enemy property and the `SimpleAITurret` object as the spawn point. This way, we will use the turret's position and rotation when we spawn our enemy, but we could also create an empty object and use this as well.

Now, if we were to run the game now and run up to the enemy, he would indeed start chasing us… but there's a bit of a problem:



By default, the **motor** that the object is using is called a `BasicMotor`, which is great and efficient but doesn't make use of gravity. Thankfully, there is another motor class included called a `CharacterControllerMotor` that we can add, which will work well for us.

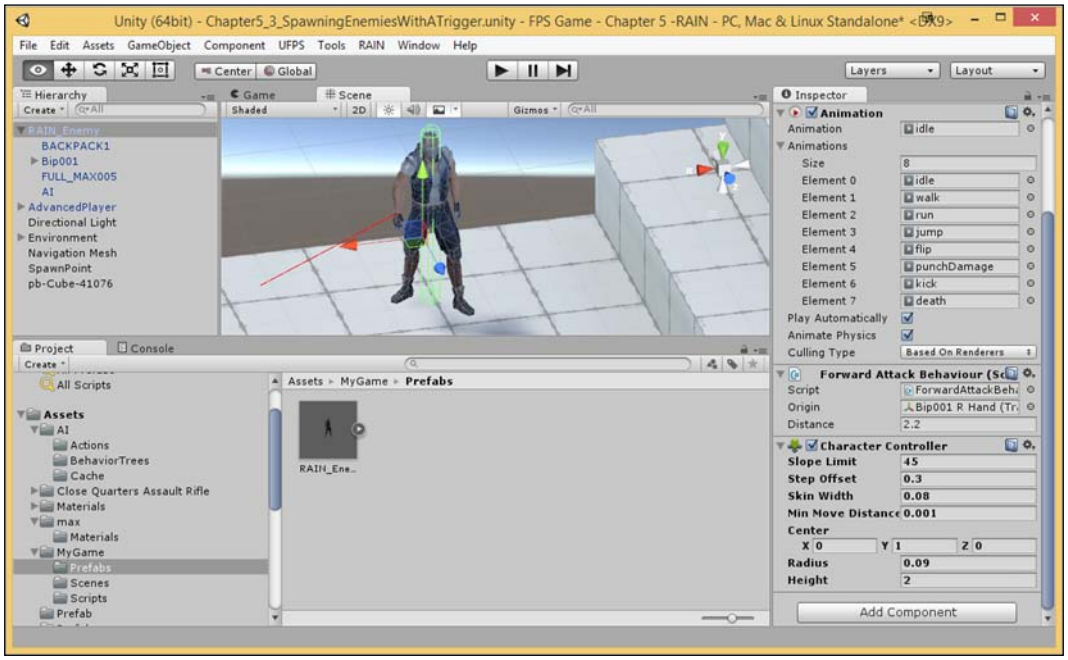


For more information on the different kinds of motors as well as how to make your own, check out `http://rivaltheory.com/wiki/rainelements/charactercontrollermotor`.

11. First, in order to make it easier to modify the prefab, let's bring one into the scene. So with this in mind, from the **Project** tab go to the `MyGame/Prefabs` folder, and then drag and drop a `RAIN_Enemy` prefab into the scene.

12. From the **Hierarchy** tab, extend the newly created object's children and select the AI object. From here, click on the feet icon to open the **Motion** tab. Then, change the **Motor** property to `CharacterControllerMotor` from the dropdown that has popped up.
13. This will require our `RAIN_Enemy` object to have a **CharacterController** component, so let's add one real quick. Select the parent object from the **Hierarchy** tab, and then from the top bar, navigate to **Component** | **Physics** | **CharacterController**.

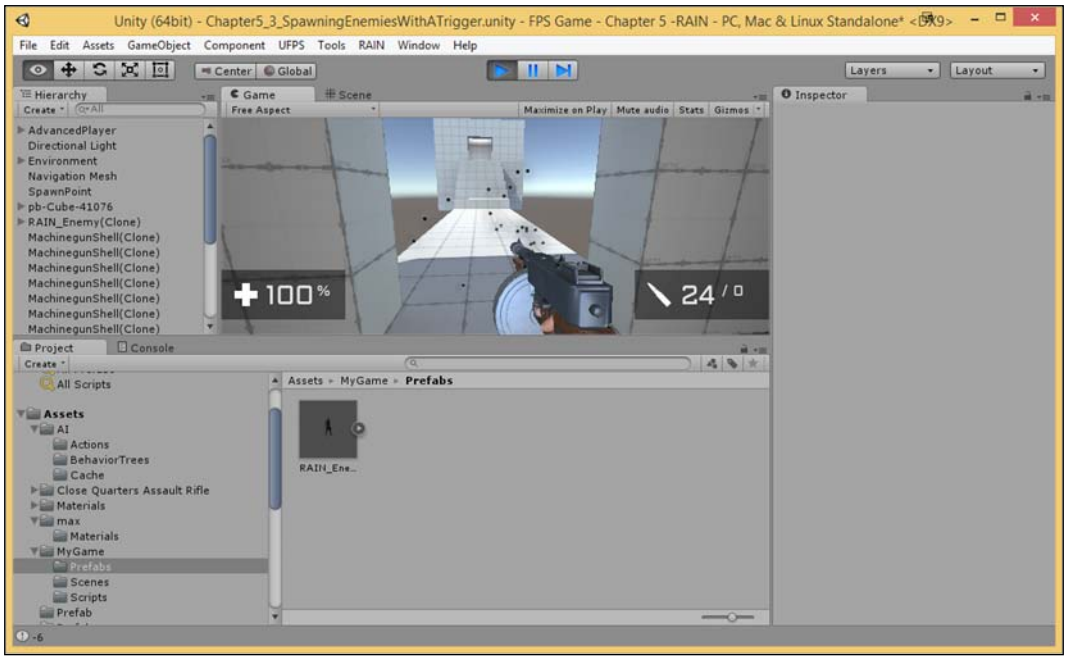
    You'll notice that there is now a new capsule that's been added. This is what RAIN will use to base its movement on.
14. From the **Inspector** tab, scroll down to the **Character Controller** component and change the **Center Y** property to `1`, **Radius** to `.9` (slightly smaller than the collider we created to receive hit events), and leave **Height** at `2`.



15. After making all these changes, let's update our original prefab by scrolling up to the top of the **Inspector** tab, and then from the top **Prefab** section, click on the **Apply** button. This is so that our prefab object now has all the changes we made to the previous version.

16. After this, you can delete the object from our **Hierarchy** as we won't need it anymore.

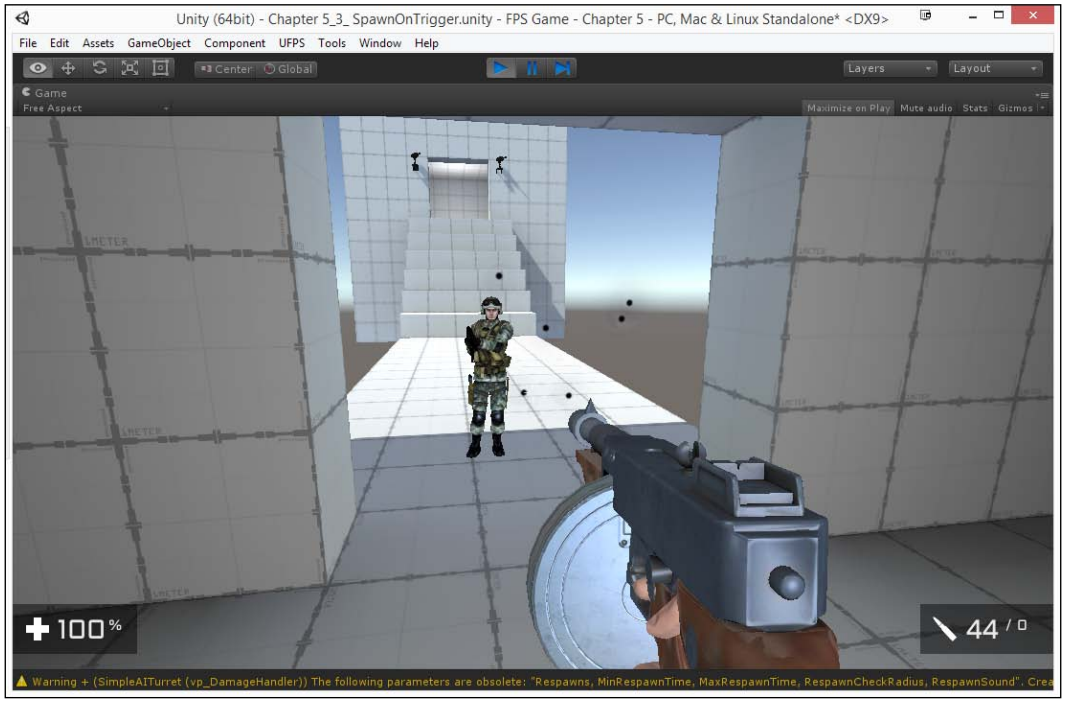
    At this point our enemy will now go up and down stairs correctly and move towards us. However, if we happen to go back to our first room to hide from the enemy...



Our bullets aren't going through the trigger. This is because when we shoot in UFPS it draws an invisible line and sees if anything hits it. Right now, the trigger we created counts as something so it uses that as what it hits. (by default, things will be hit by bullets even if their colliders are triggers). Thankfully, this is a pretty simple fix.

17. Select our trigger object, and from the **Inspector** tab, select **Layer** and **IgnoreBullets**.

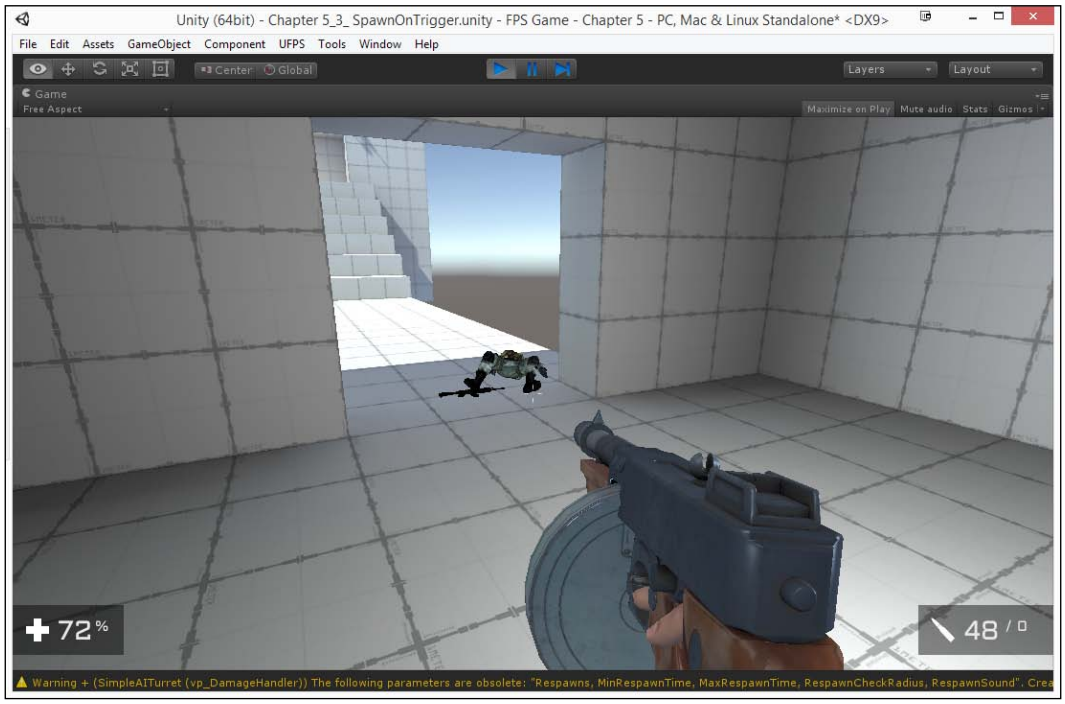
    The enemy can't follow us and our bullets aren't going through the trigger. This is because the bullet is drawing a line and seeing that there is something there due to the layer that it has (by default, things will be hit by bullets even if their colliders are triggers). Thankfully, this is a pretty simple fix.



If you're using Shooter AI, you should now do the following:

1. Select our trigger object and from the **Inspector** tab, select **Layer** and **Trigger**.

2. Now, save your scene and run again!

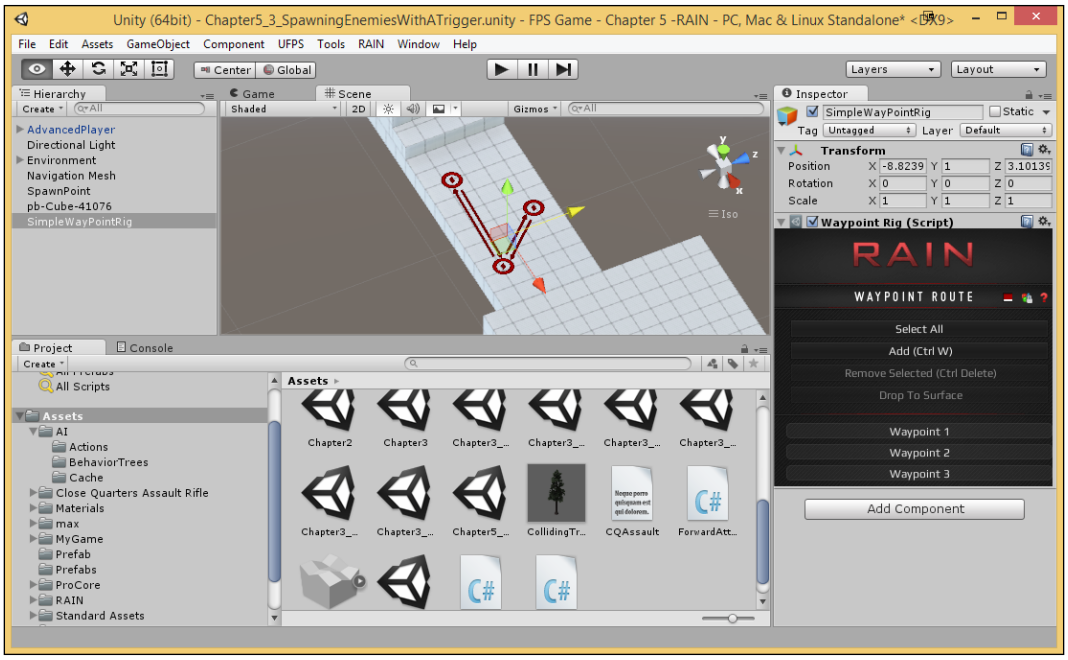


With all this place, we now have our trigger working perfectly, and we can shoot through it as well! Now, go ahead and skip the rest of this section and continue on to the *Clean Up Date AI* section.

We may also want to have the AI move rather than just stay in place for the beginning of the combat. Our current behavior tree states that if we don't have an enemy nearby, we can search for a Waypoint Route component from an object called `SimpleWayPointRig` (which you can check out for yourself in the Behaviour Editor from the `waypointpatrol` decision). If you're using RAIN, follow these steps:

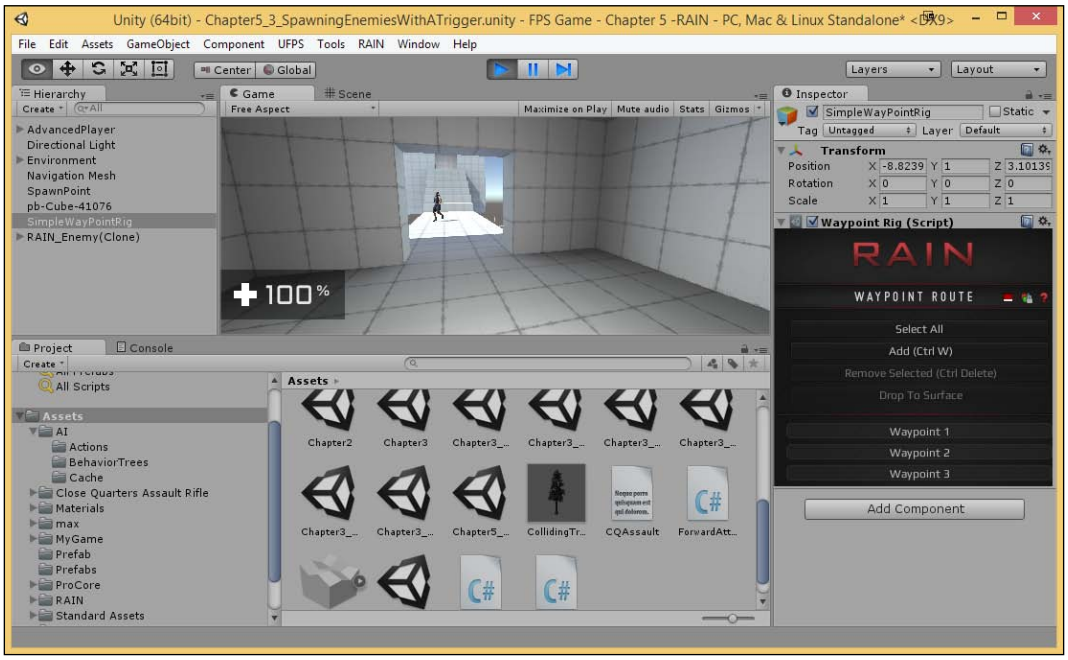
1. From the toolbar at the top of the window, navigate to **RAIN** | **Create New** | **Waypoint Route**. From **Inspector**, rename this object to `SimpleWayPointRig`. Next, click on the **Add** button to add a waypoint and you'll see a single circle there, which will be the first place that the enemy will go to.

2. After creating this first key, press *Ctrl* + *W* to create a new key, which will show up nearby and you can move this via translation tools. You can create as many parts of this key as you like to create a patrol route for the AI to take.

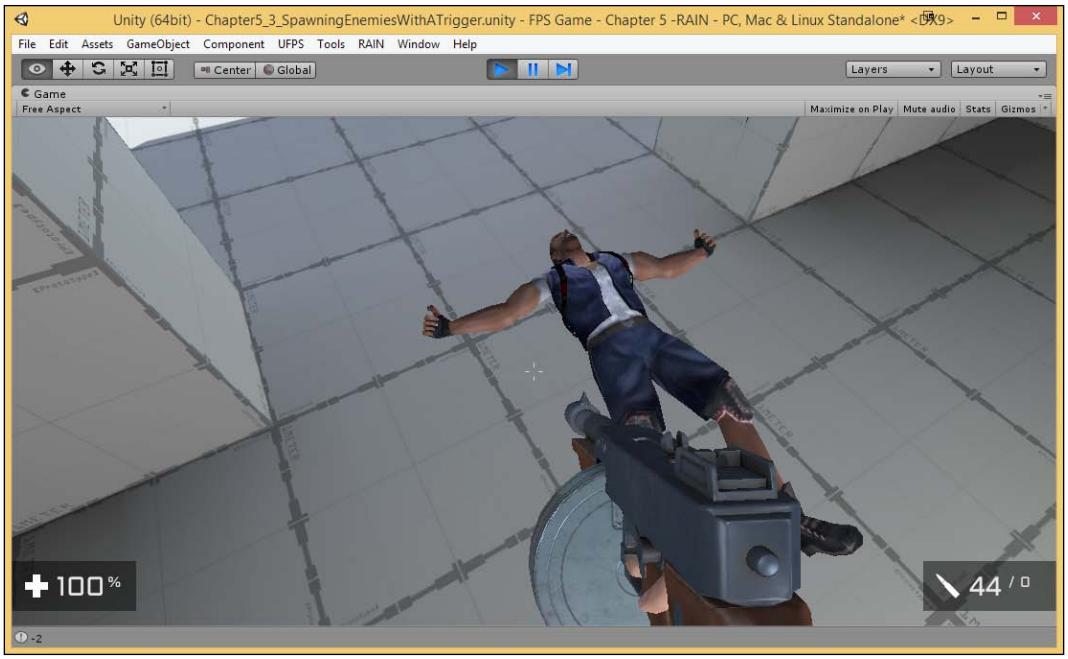


Currently, when you finish the route, you will go back to the previous waypoints, so there is no need to try to make them all match.

3. Now, save your scene and run it again!



As you can see, our enemy will now spawn and follow the route we've created, and if we have him get to the trigger, bullets will fire as well!



Now, our enemy's functioning as we want it to!

Spawning multiple enemies at once

Taking this a little further, we can create another script that can be used to spawn multiple enemies at once.

1. Go to the `MyGame/Scripts` folder, navigate to **Create** | **New C# Script**, and call it `SpawnEnemiesOnTrigger`. Once you've finished, double-click on the created file to open MonoDevelop.
2. Once the file is opened, put in the following code if you're using RAIN:

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

public class SpawnEnemiesOnTrigger : MonoBehaviour
{
  // Enemy to spawn
  public GameObject enemy;

  // Where to be spawned at
```

```
  public List<Transform> spawnPoints;

  // Has this happened already?
  private bool hasTriggered = false;

  void OnTriggerEnter(Collider other)
  {
    //If the player touches the trigger, and if it hasn't
      // been triggered before
    if(other.tag == "Player" && hasTriggered == false)
    {
      foreach(var spawnPoint in spawnPoints)
      {
        // Spawn a new enemy using the properties from the
          // spawnPoint object
        GameObject newEnemy = Instantiate(enemy,
                        spawnPoint.position,
                        spawnPoint.rotation)
                        as GameObject;
      }

      // We only want this to happen once.
      hasTriggered = true;
    }
  }
}
```

If you're using Shooter AI, add in the following code:

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

public class SpawnEnemiesOnTrigger : MonoBehaviour
{
  // Enemy to spawn
  public GameObject enemy;

  // Where to be spawned at
  public List<Transform> spawnPoints;

  // Has this happened already?
  private bool hasTriggered = false;

  void OnTriggerEnter(Collider other)
```

```
  {
    //If the player touches the trigger, and if it hasn't
      // been triggered before
    if(other.tag == "Player" && hasTriggered == false)
    {
      foreach(var spawnPoint in spawnPoints)
      {
        // Spawn a new enemy using the properties from the
        // spawnPoint object
        GameObject newEnemy = Instantiate(enemy,
                              spawnPoint.position,
                              spawnPoint.rotation)
                              as GameObject;
        //Tell enemy to go to the player
        newEnemy.GetComponent<NavMeshAgent>()
          .SetDestination(other.transform.position);
      }

      // We only want this to happen once.
      hasTriggered = true;
    }
  }
}
```

This code is different from our previously created one in a few ways. For instance, now, `spawnPoint` is `spawnPoints` and its type is `List<Transform>` or a list of transforms. This is basically a group of multiple objects with the `Transform` component attached, which we can then walk through (known in programmer lingo as **iteration**, which is what `foreach` is for) and do something about for each point, in this case, spawning an enemy.

3. Save the file and go back into the editor. Now, remove our previously created component by right-clicking on the component in the **Inspector** tab and selecting **Remove Component**. After this, drag the `SpawnEnemiesOnTrigger` script that we've just written onto the object to add it.

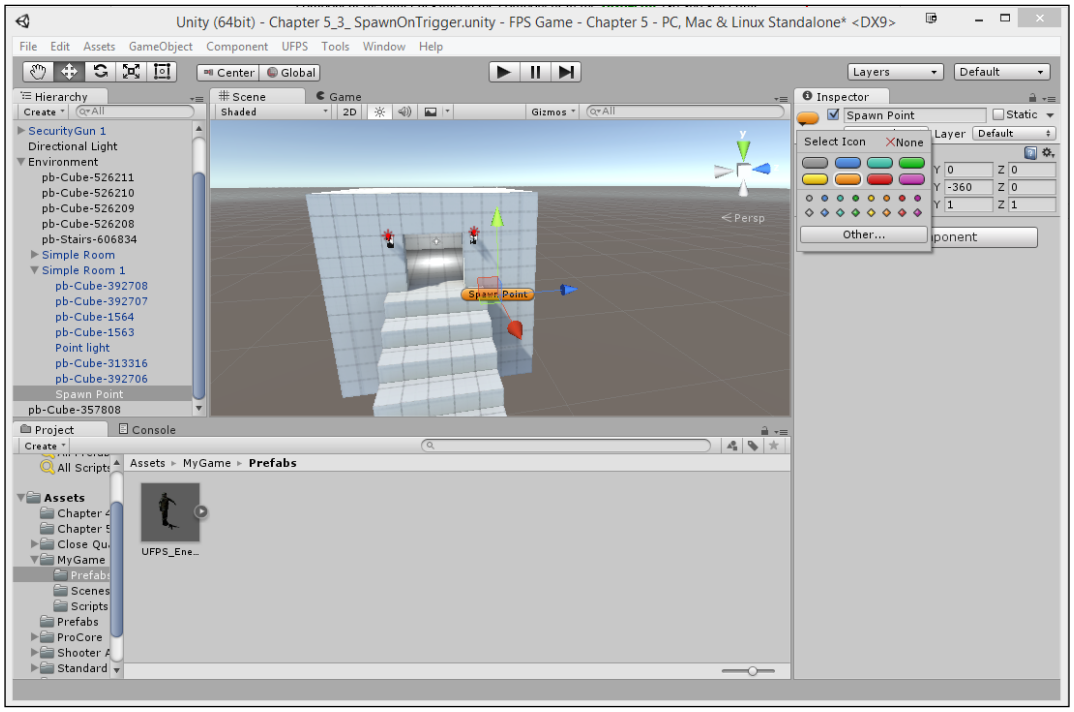
   At this point, you may notice that we can do everything that we did with `SpawnEnemyOnTrigger` inside the `SpawnEnemiesOnTrigger` class, so you can delete it if you'd like.

4. Next, in the **Inspector** tab, go down to the newly created component and drag and drop our Enemy prefab into the **Enemy** slot just like we did before. Then, expand the **Spawn Points** property. Next, change **Size** to `2` and you'll see two slots open up.

5. We need to create two objects to be our spawn points. A little trick to this is that when we select an object in the **Hierarchy** tab and create a new Game Object, it will be created in the center of this object. So, select the Simple Room that our enemies are in, and then navigate to **Game Object | Create Empty Child**. With the new object created, change its name to `Spawn Point`.

   By default, the object is invisible unless we have it selected, in which case, we can see the transform gizmo, so we may want to add some visuals to it to make it easier to see and position it where we want it to be placed.

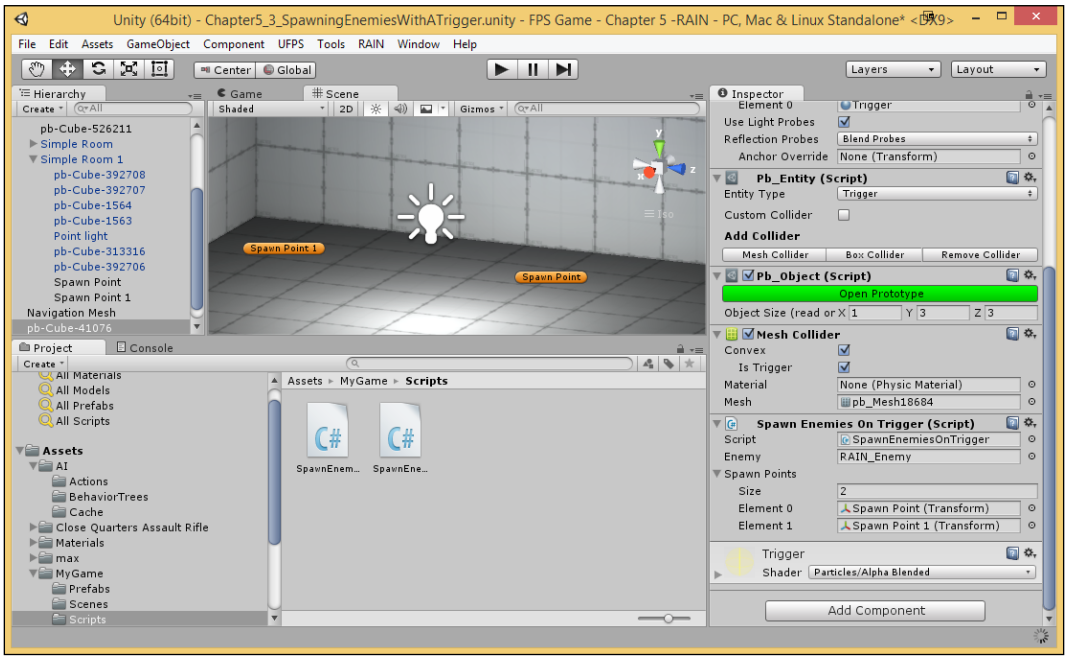
6. Click on the icon to the left-hand side of the name and you'll see the **Select Icon** window pop up. Select one of the top icons and you'll see that the object will have the object's name appear where it is currently located.



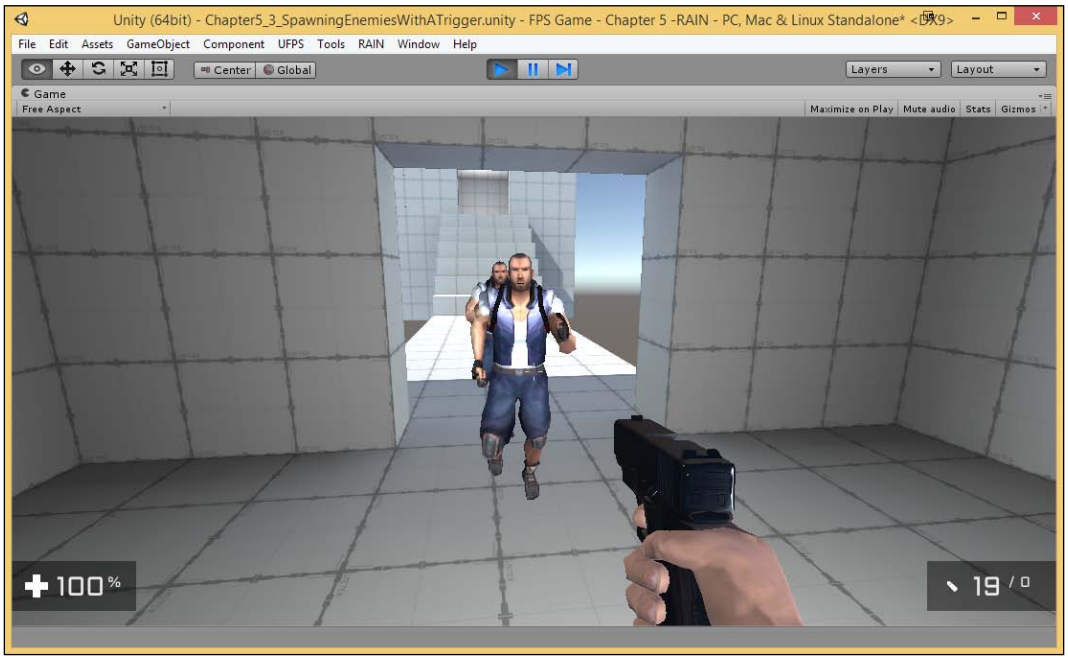


For information on the Icon Display controls, check out `http://docs.unity3d.com/420/Documentation/Manual/GizmoandIconVisibility.html`.

7. Now, we can move this object where we want it, duplicate it, and move it to another position so that we now have two spawn points to spawn something from.
8. Lastly, select our trigger object once more, go back to the **Inspector** tab, and assign the spawn points into each element of the spawn points.



9. Save your scene and play the game!



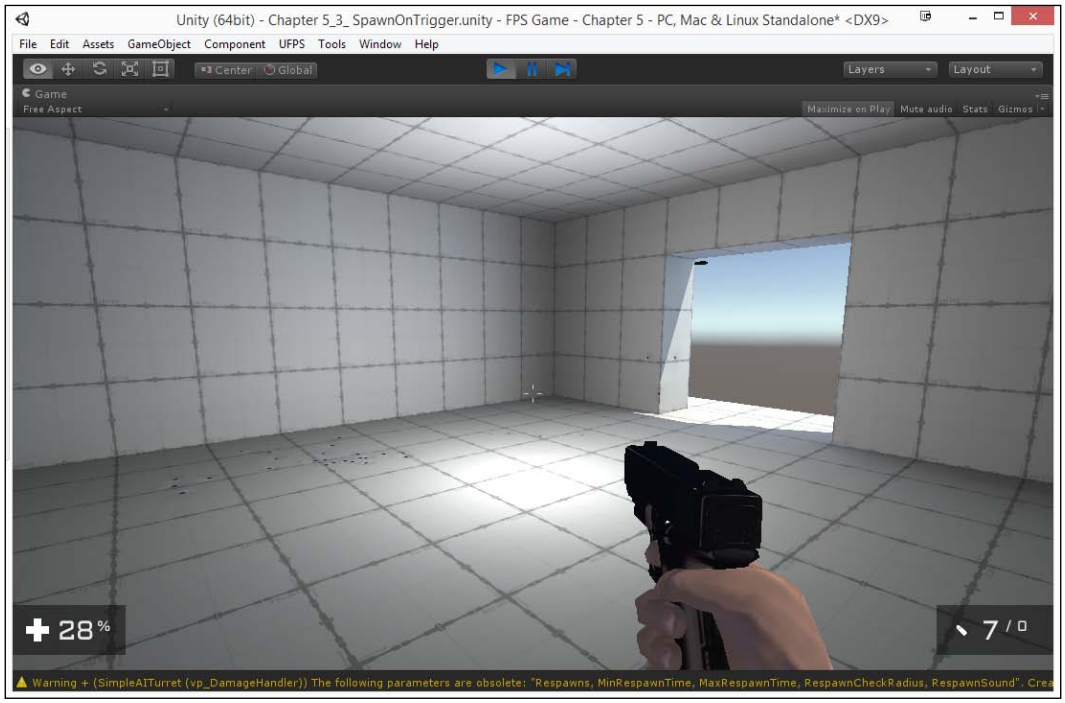
We can now spawn as many enemies as we want and have them come in from multiple directions! We're now in a position to create many different kinds of combat scenarios!

Cleaning up dead AI

Another thing that you will need to do when dealing with AIs is that if you're using Shooter AI, when they die, they don't disappear, at least by default. This adds realism to our world, but it's still very computationally expensive to have them around. Thankfully, we can add another component to make it quite easy for us to clean up enemies of our choice.

1. From the **Project** tab, open the `MyGame/Prefabs` folder and select the `UFPS_Enemy` that we created earlier.

2. From the Inspector tab, scroll all the way down and press the **Add Component** button. From here, navigate to **Shooter AI | Clean Up After Death**.
3. By default, it has the **Cleanup Time** property set to `10`, which means that after 10 seconds it will disappear. Since we want to see it quickly, change the value to `2`.
4. Save your project and start the game again.

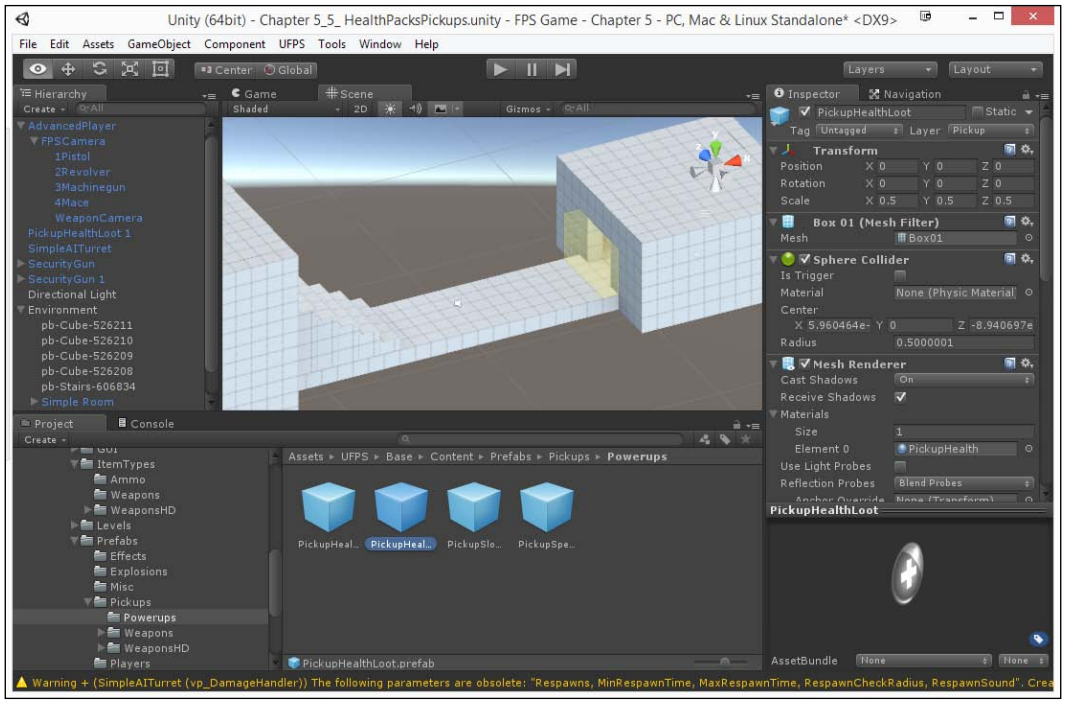


After we kill our enemy, it will disappear! This will really help with things, such as performance and keep our game running smoothly!

Placing healthpacks/ammo

Other tools in a level designer's toolbox are ammo/health pack pickups. These are often used as rewards for a player who's exploring a layout as well as providing an incentive to travel to certain places, sometimes at risk.

1. From the **Assets** tab, go to the `UFPS/Base/Content/Prefabs/Pickups/Powerups` folder. Once here, you'll see a number of powerups that can be placed in the world as follows:



   - **PickupHealth**: This will increase a player's health and will also respawn
   - **PickupHealthLoot**: This will increase a player's health by a little bit and it will not respawn
   - **PickupSlomo**: This will slow down enemies with a bullet-time effect for time-based gameplay.
   - **PickupSpeed**: With this, players will be able to travel much faster than they normally can when there is a need for speed.



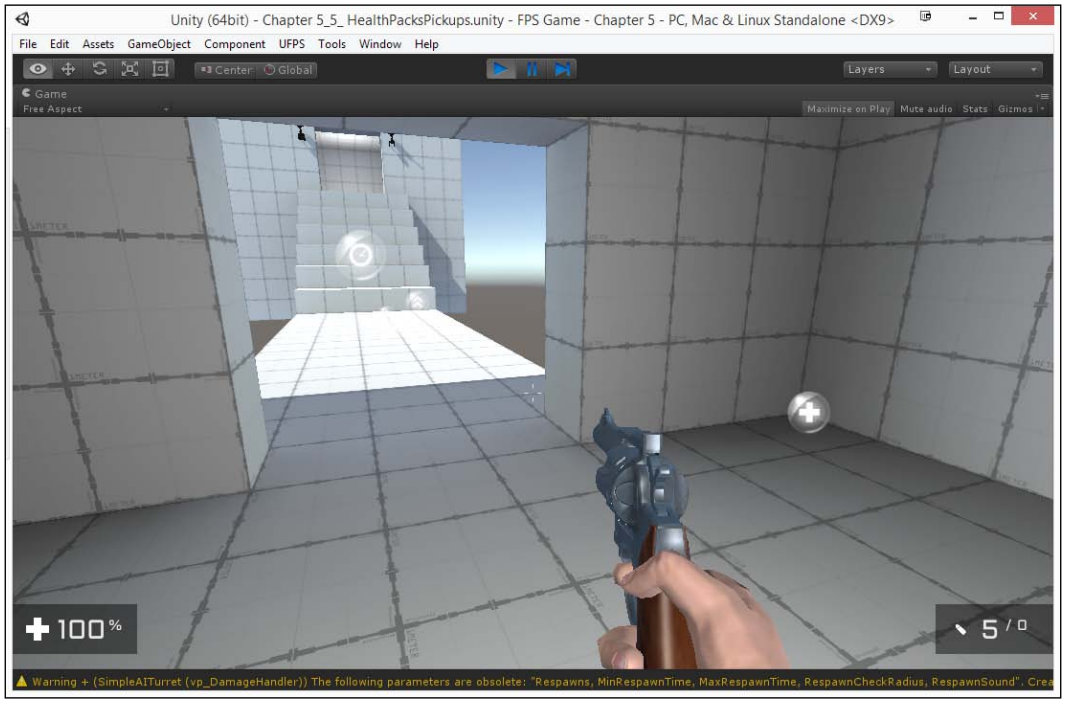
A tutorial to create your own custom health and ammo pickups can be found at `https://www.youtube.com/watch?v=BEHimn5UeF0`.

2. Drag and drop some pickups along the way to the other room to lead the player toward this area. Also, place some ammo for weapons that you want the player to use (in the `UFPS/Base/Content/Prefabs/Pickups/Weapons/Ammo` folder).



A nice write-up on item placement in FPS maps can be found at `https://dfspspirit.wordpress.com/2015/03/26/designing-great-1vs1-fps-maps-part-1/`.

3. Once you're finished, save your level and play the game!



Now, our world is a little more interesting and our players have the incentive to go out and face danger!



For more information on Shooter AI and to resolve any questions that you may have, visit its official thread on the Unity forums at `http://forum.unity3d.com/threads/shooter-ai-the-ultimate-artificial-intelligence-solution.204220/`.

Summary

We now have some interesting encounters for our game! In addition to these, we covered a lot of features that exist in Unity for you to be able to use in your own future projects. We specifically covered placing turrets, integrating RAIN in our project, integrating Shooter AI in our project, spawning enemies using triggers, placing pickups and healthpacks, and so on. Keeping all this in mind, in the next chapter, we'll learn how to take our knowledge of environments and build different combat scenarios!

6
Breathing Life into Levels

Of course, it's great to have an environment and enemies. But if these were all there was in a game, you'd just be walking from one encounter to the next with nothing more interesting to look forward to. In this chapter, we will be exploring some of the ways we can breathe life into our levels with moving objects and more things players can interact with.

This project will be split into a number of tasks. It will be a simple step-by-step process from the beginning to the end. Here is the outline of our tasks:

- Building an explosive barrel prefab
- Triggers for gameplay (doors)
- Creating an elevator with interaction

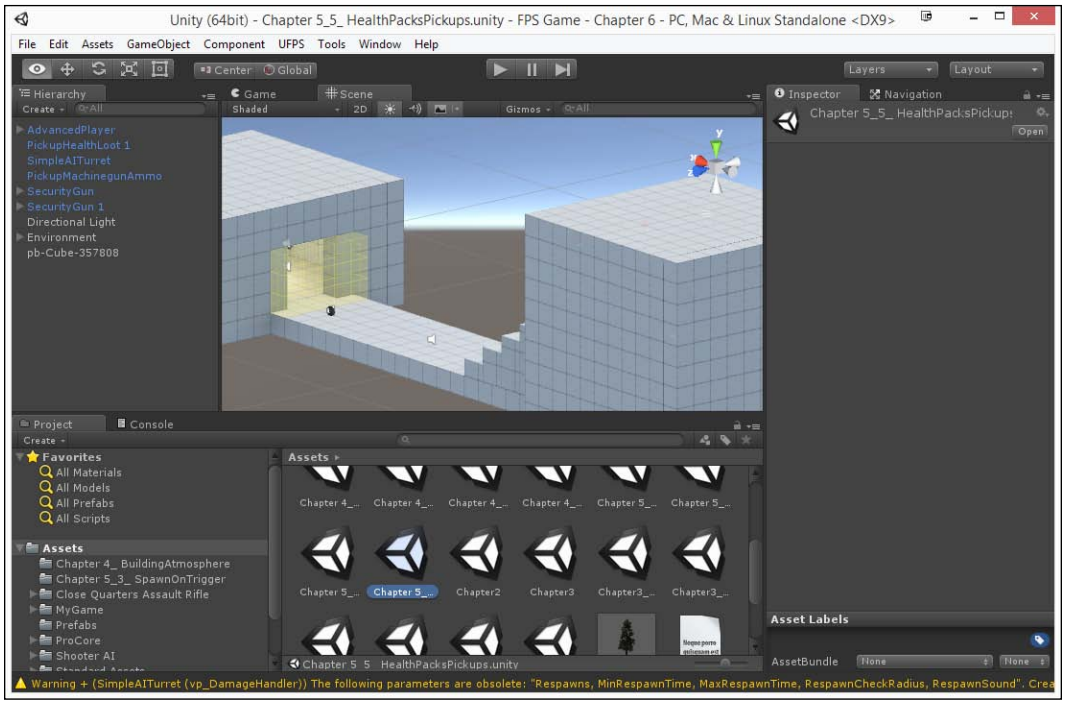
Prerequisites

Before we start, we need to have a project created that already has UFPS and Prototype installed. If you do not have these, follow the steps described in *Chapter 1*, *Getting Started on an FPS*. In addition, I am also assuming you have a level that you want to add encounters to.

Building an explosive barrel

One of the first things we are going to look at is the common FPS staple: explosive barrels. These will allow the player to get a "leg up" on their enemies by causing explosions when damaged enough. They're also fairly simple to create.

1. To start off, let's open up the level we created in the *Chapter 5, Building Encounters* (`Chapter 5_5_ HealthPacksPickups.unity` in the example files you can get from the Packt Publishing website).

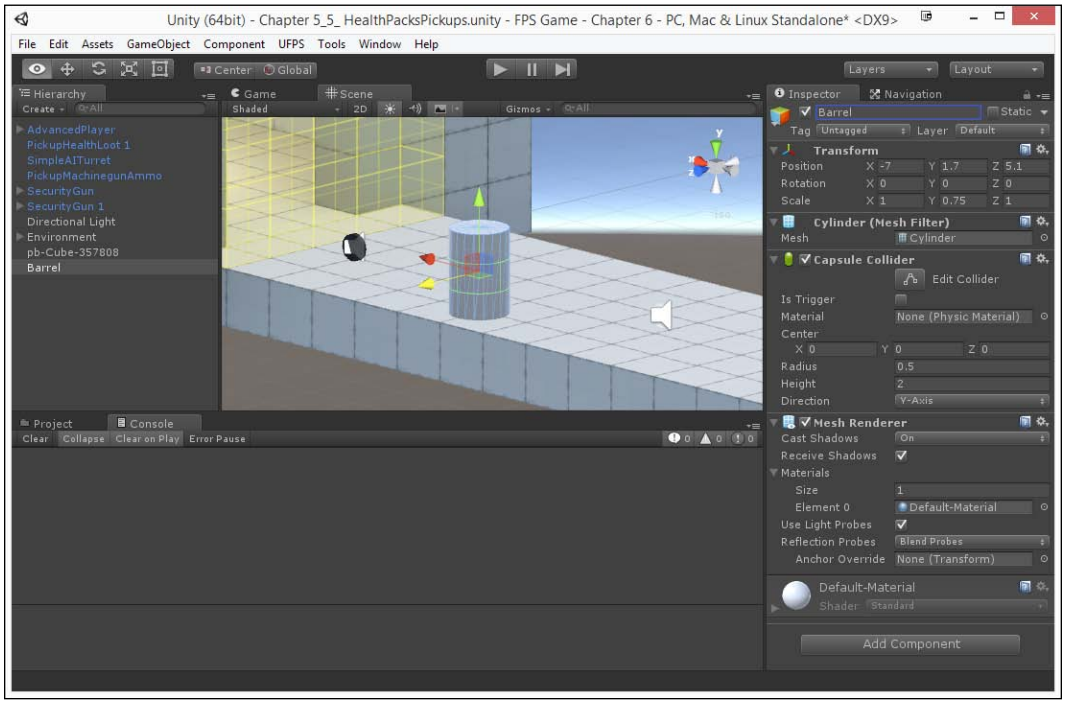


Let's get the implementation of the functionality to work correctly, then we can make it look nice.

2. Create an object to represent our barrel. To do this, go to **GameObject** | **3D Object** | **Cylinder**.

   This will create a 3D object that will look like a cylinder and give it a collider so that it blocks our player if he/she walks into it. It can also be hit by bullets.

3. Change the object's name to `Barrel`. The default cylinder is a bit too large to be a barrel, so we will change the **Y** value of the **Scale** property to `.75`. Then, we will position it so that it fits the ground (`-7, 1.7, 5.1` in this case).



4. Next, we want it to react to being damaged. So, we will need to add a **VP_Damage Handler** component to it. To do this, from the **Inspector** tab, click on the **Add Component** button, type in `damage`, and select it from the dropdown menu.

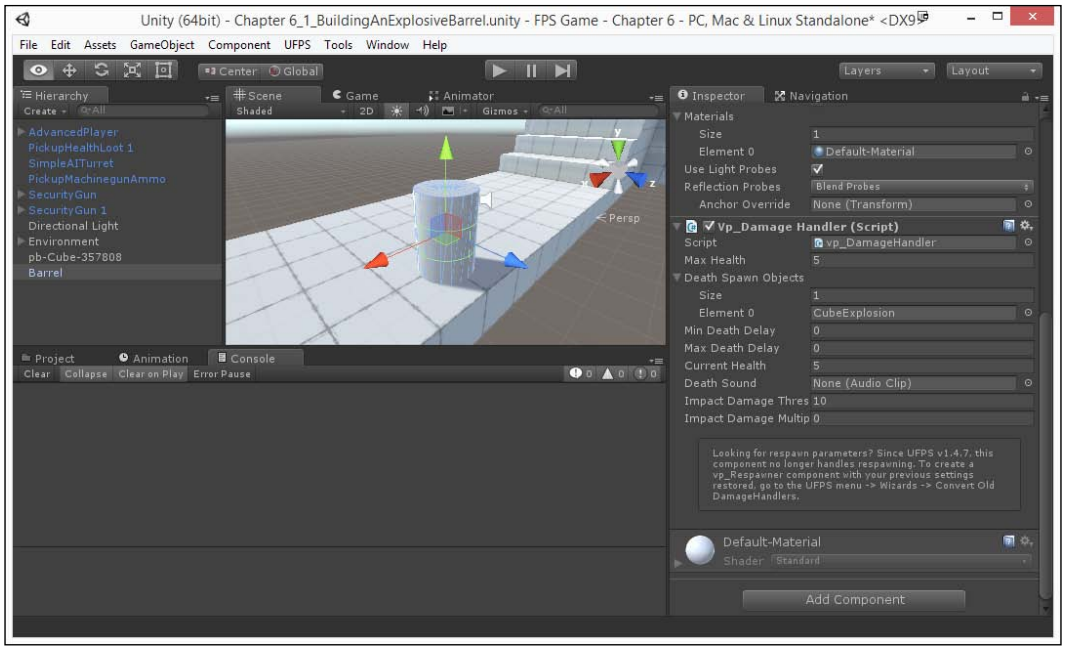


The **VP_DamageHandler** component gives objects the ability to take damage and die. It's important to note that this will not respawn objects (use the **VP_Respawner** component for that). As we don't want the barrels to respawn, it is fine to use this component. In this case, if we play the game and shoot the barrel, it will disappear.



For more information on the **Damage Handler**, check out `http://www.visionpunk.com/hub/assets/ufps/manual/damage`.

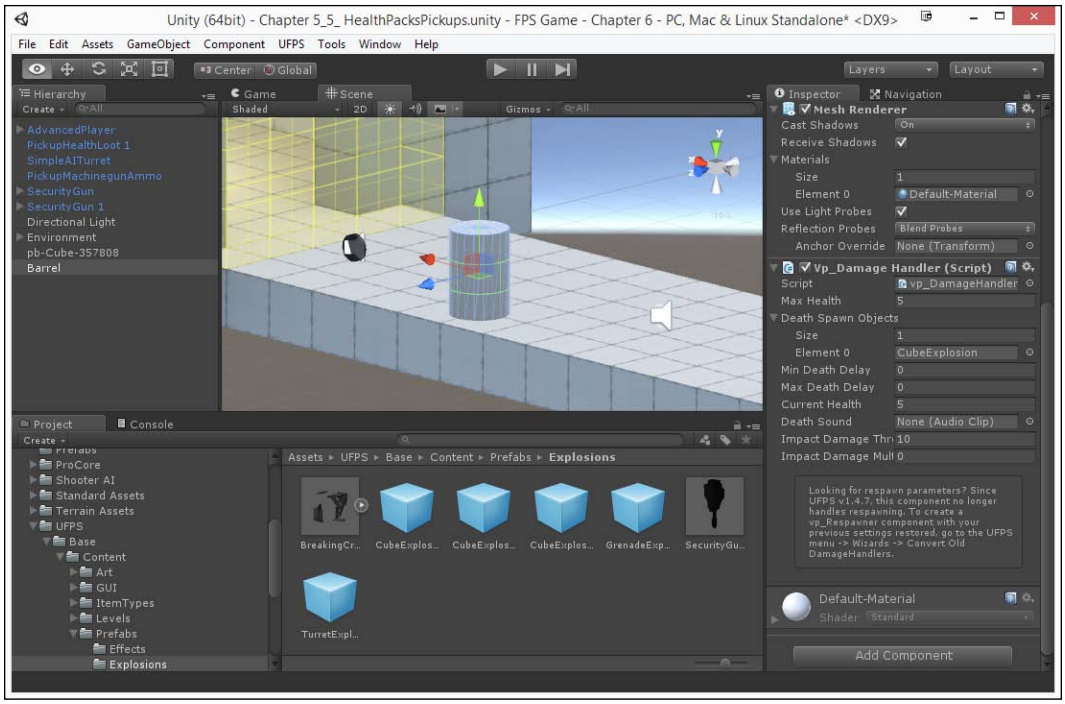
5. We don't want a single small bullet to destroy the barrel, so in the **Vp_Damage Handler** component, we will set the **Max Health** and **Current Health** variables to `5`. The higher the number used here, the more is the damage needed to be done to the barrel before it is destroyed.



Currently, when we kill the barrel, the object just disappears. Instead, let's have an explosion occur that will damage everything nearby.

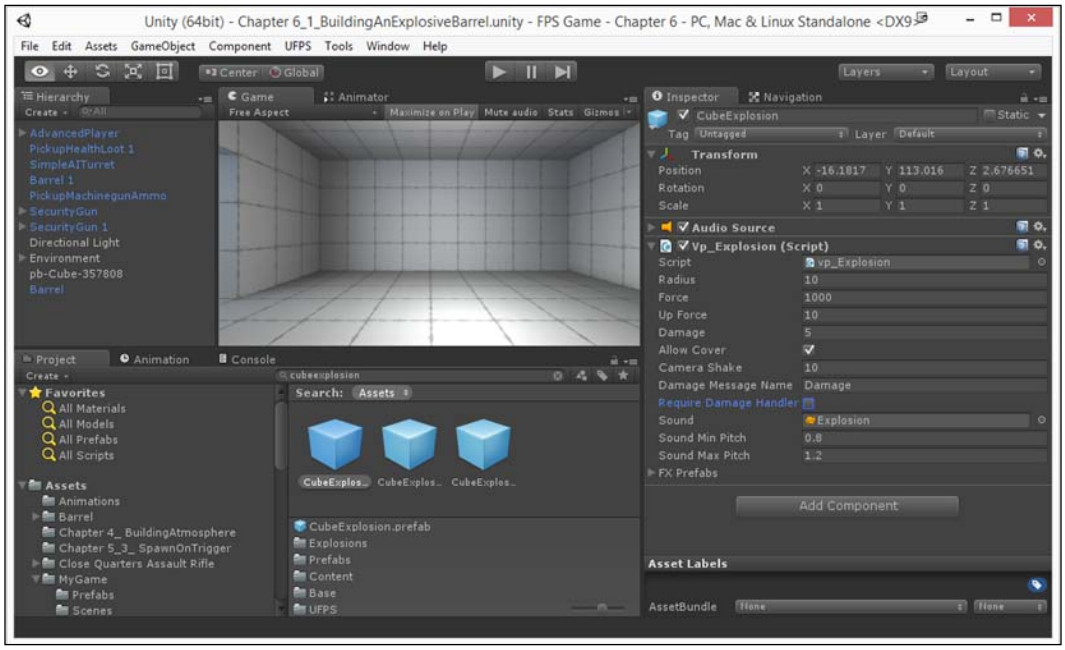
6. Under the **Vp_Damage Handler** component, expand the **Death Spawn Objects** variable and change the **Size** value to `1`.

7. From the **Project** tab, go to the `Assets/UFPS/Base/Content/Prefabs/Explosions` folder and move the `CubeExplosion` prefab to the **Death Spawn Object** slot by holding down the mouse button over it and dragging it into the slot.



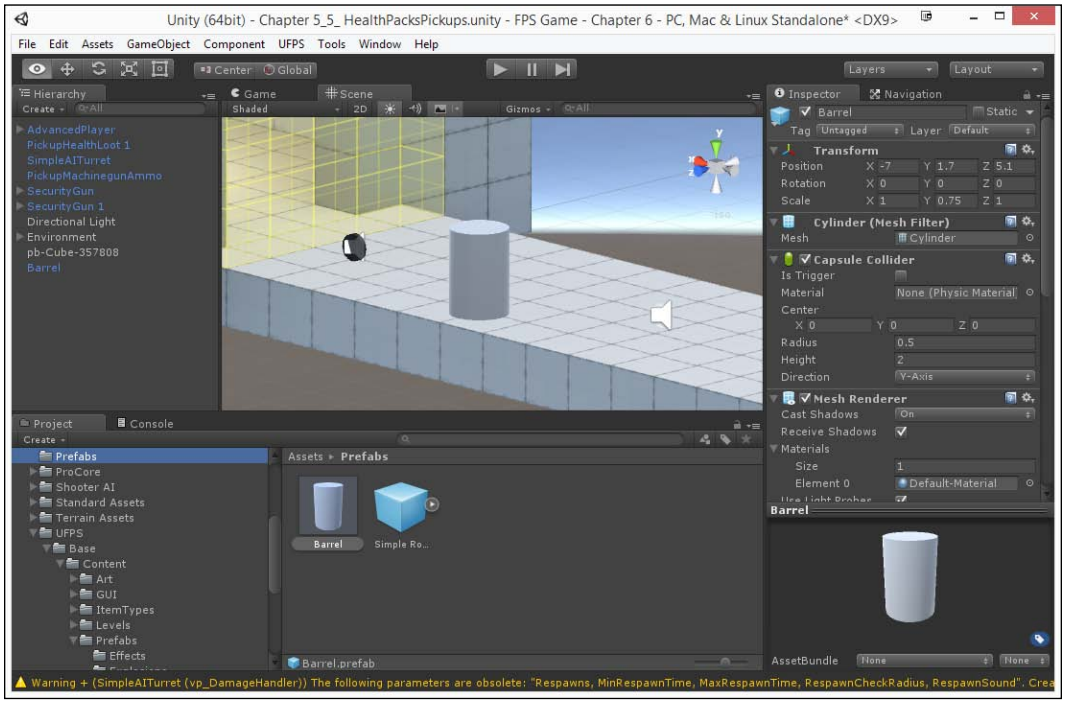
Now, if we were to walk up to the barrel and attack it, we would get knocked back. However, enemy AIs don't seem to respond to it. This is because the Shooter AI characters do not come with a **Damage Handler** component. However, we can do a quick fix for this to work.

8. Back in the **Project** tab, select the `CubeExplosion` prefab. From the **Vp_Explosion** component, uncheck the **Require Damage Handler** property.



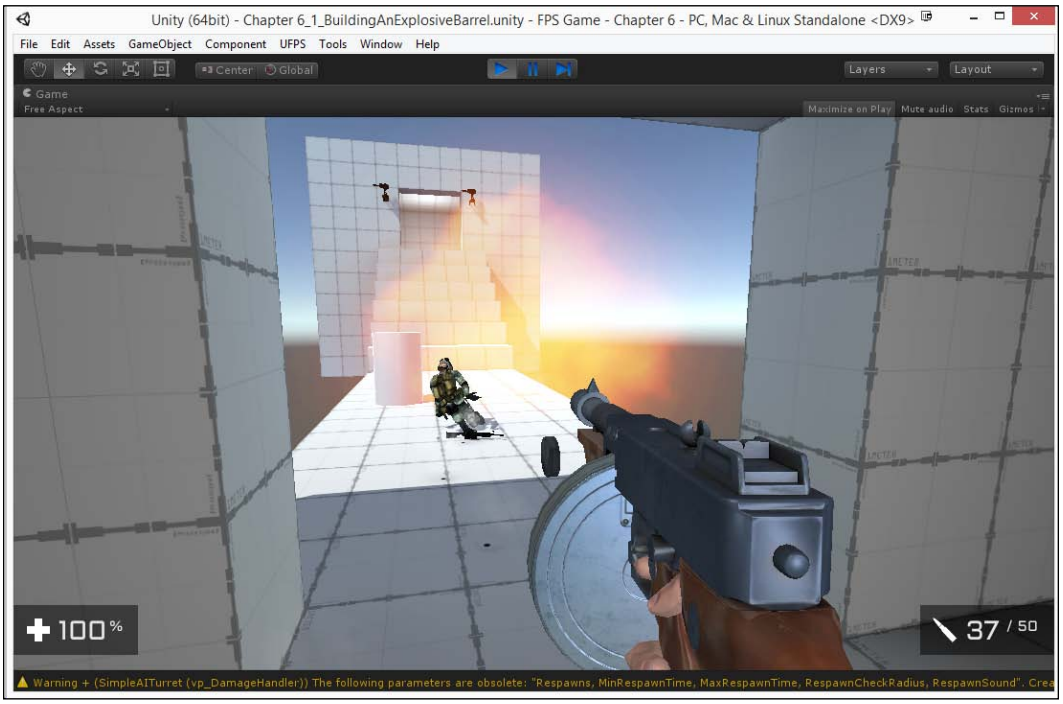
This tells the explosions that any object within its radius will have the `Damage` function called, which our enemies have an implementation for, even if it doesn't have a **Vp_Damage Handler** component.

9. Now, let's create a prefab of our barrel by going to the **Project** tab, opening the `Assets/Prefab` folder, and dragging the `Barrel` game object from the **Hierarchy** tab into the folder.



10. Now, to confirm whether everything is working correctly, create a copy of the barrel by dragging it from the `Prefabs` folder into the game world. You'll notice that it will automatically try to place itself on the floor for us.

11. Save the level and run the game.



There we go! When we destroy the barrel, our enemies will be damaged by being caught in the explosion.

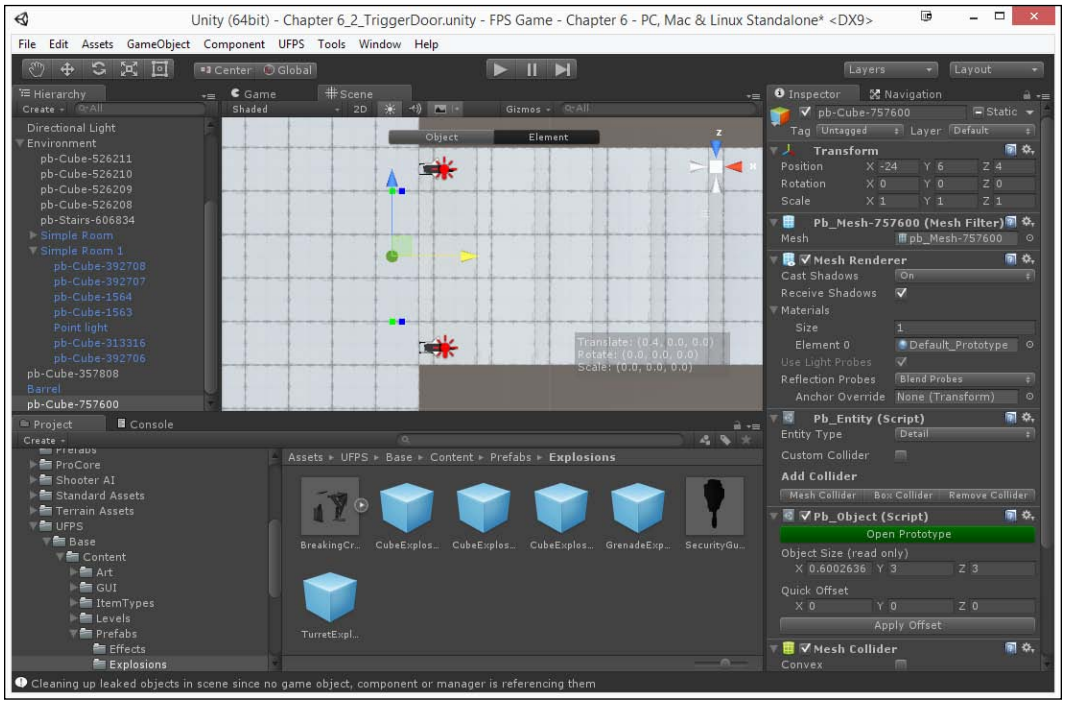
Using triggers for doors

Earlier, you learned how to use triggers to do something simple, such as spawning enemies to be attacked. In this section, you're going to learn how to use the same principles for doors.

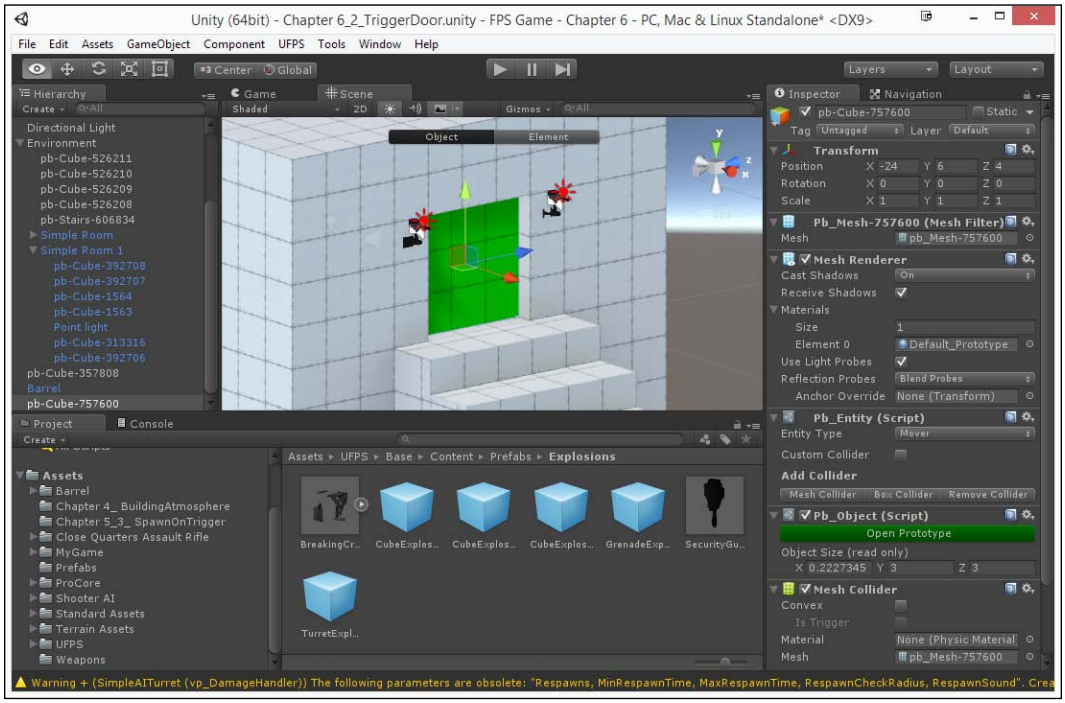
1. Since we're using the first doorway we created to spawn enemies, let's create our new door in the second one. To do this, open up **Prototype** and select **Create** to add a new **Cube** into the world, placing it in such a way that it fits the front of the doorway we created earlier.



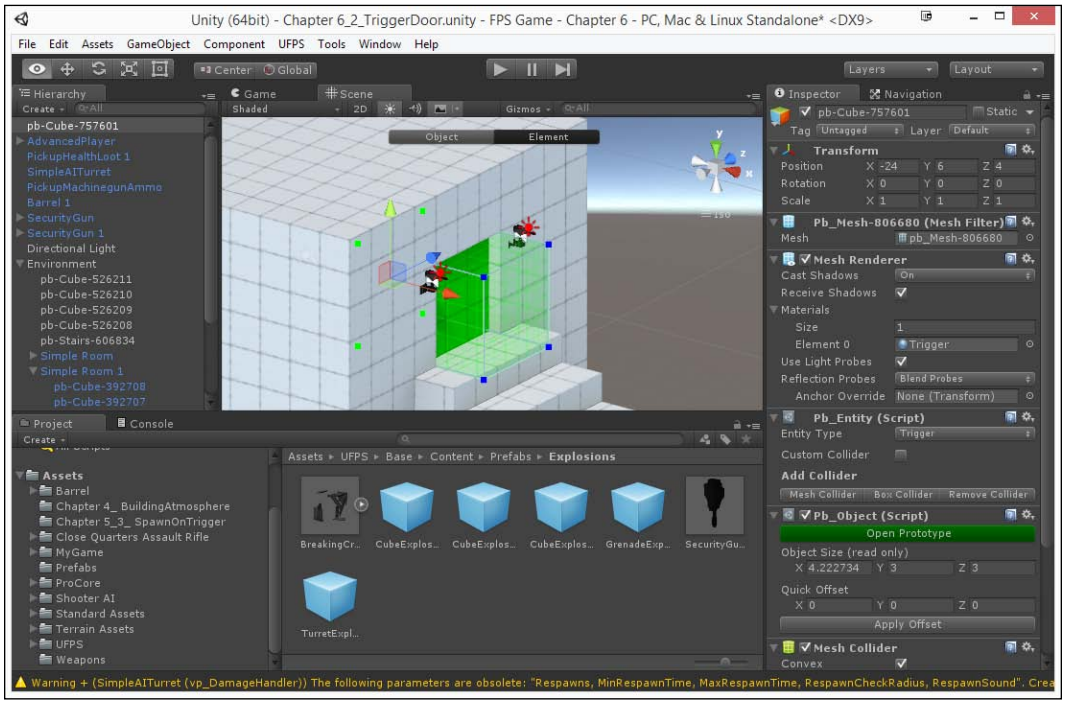
For a refresher on how Prototype works, refer to *Chapter 3, Prototyping Levels with Prototype.*



2. Now, switch to the **Object** mode and press the *M* key to specify this object as a **Mover.** To make it easier to tell the difference between the normal stuff and it, select the **Vertex Colors** option and make the object green.

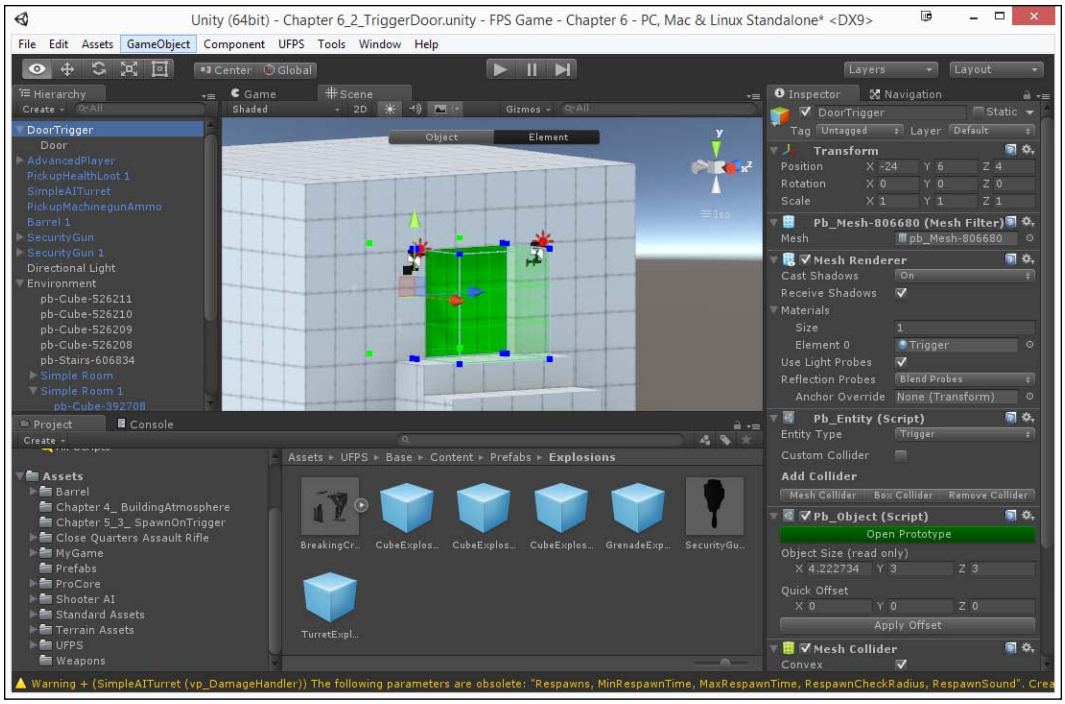


3. Next, duplicate the object, click on the **Set Trigger** button, and drag it out toward both the sides of the door.



4. To make it easier to look at, rename the trigger to `DoorTrigger` and the door to `Door`. Then, make the `Door` object a child of `Door Trigger`.

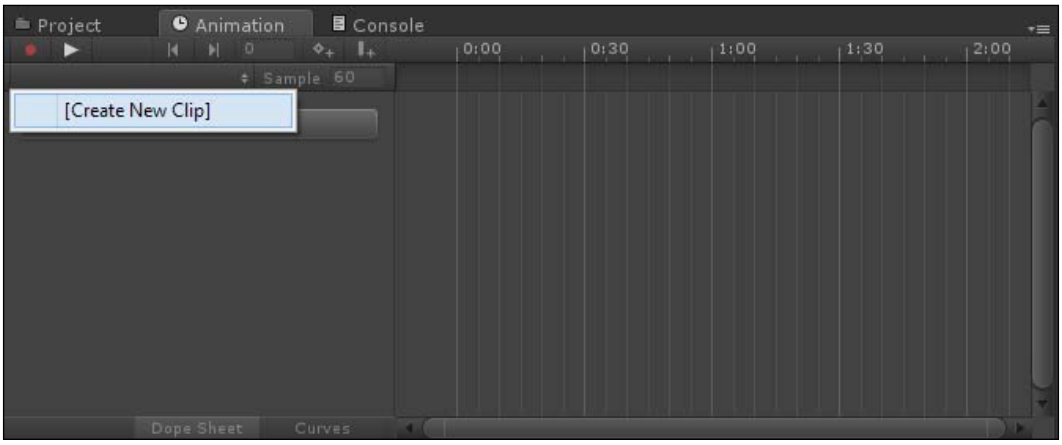
   Now, why are we making the trigger the parent? Well, later on, when we are creating animations, the door will be moving. If the door was the parent, then the trigger would move as well.



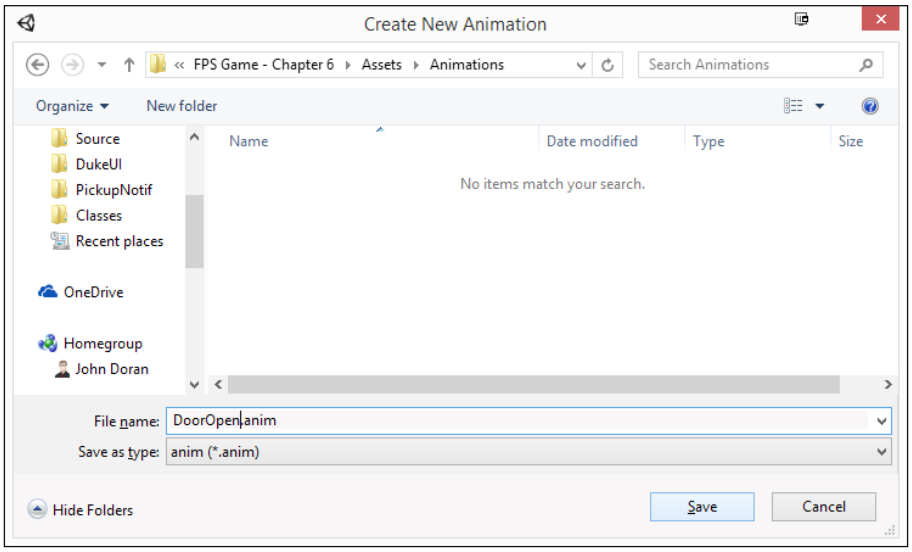
5. We also need to set the **Layer** value of `DoorTrigger` to `Trigger` so that it cannot be hit by bullets. A dialogue box will open up, asking if you want to apply the layer to the child; in this case, we will say *no*, because the bullet should actually hit the door, but not the trigger.

   Now, we need to make it such that when the player hits the trigger, the door will open and when they leave, it will close. To do this, we will need to write some script; but before we do that, we'll need to create an animation for the door to move.

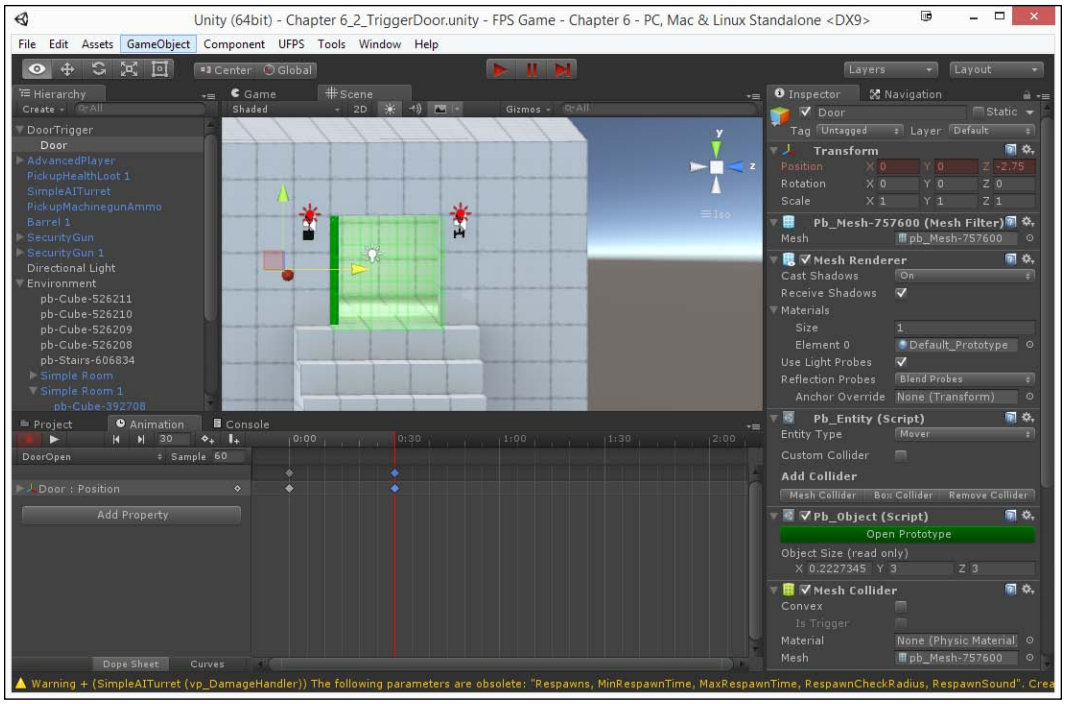
6. From the **Hierarchy** tab, select the `Door` object and go to **Window | Animation**. This will open up the **Animation** window, which is where we create animations inside Unity. Drag and drop the window to where the **Project** tab is to work with it with ease. Then, click on the **Create** button on the left-hand side of the **Animation** tab (clicking the up and down arrow keys for the dropdown menu) and select **[Create New Clip]**, as shown in the following screenshot:



7. Unity will ask you where you want to save the animation file. Go to the `Assets` folder and create a new folder called `Animations`. Type in `DoorOpen.anim` and press **Save**.



8. Exit Prototype if it is open and switch to the **Translate** tool (*W*). Click on the record button and drag the red bar to the `0:30` spot by dragging the mouse on the bar with the image's measurements. Then, with the `Door` object selected in the **Hierarchy** tab, drag the door over to the side.



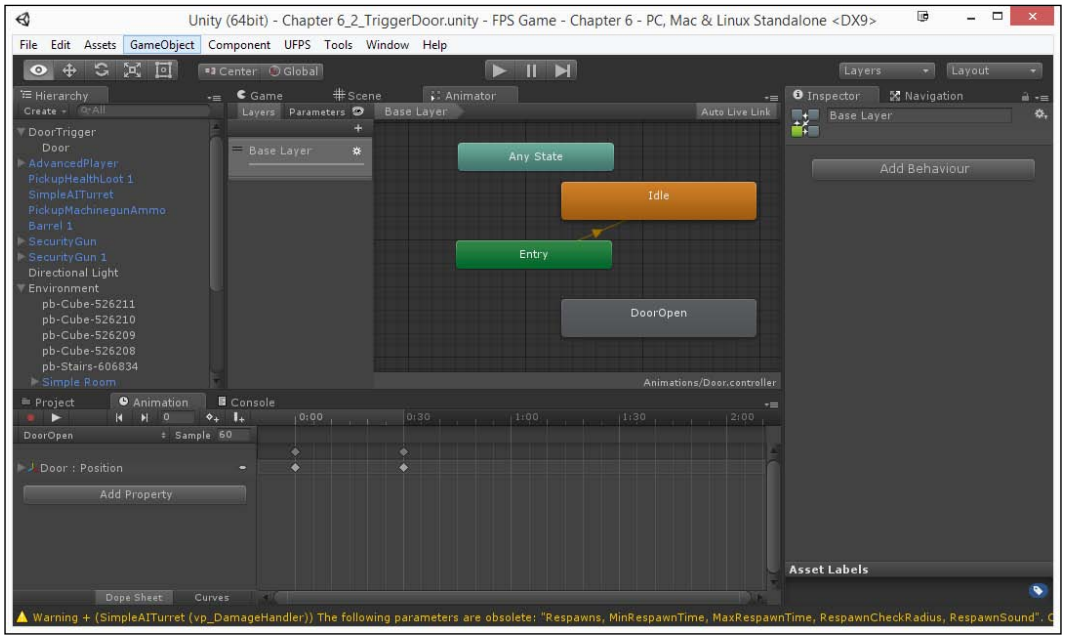
You should notice that the animation has automatically created two key frames, one at the start `0:00` and one where we moved the red bar to.



For more details on the **Animation** tool and the many ways you can use this tool to modify any value, check out `https://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/animate-anything`.

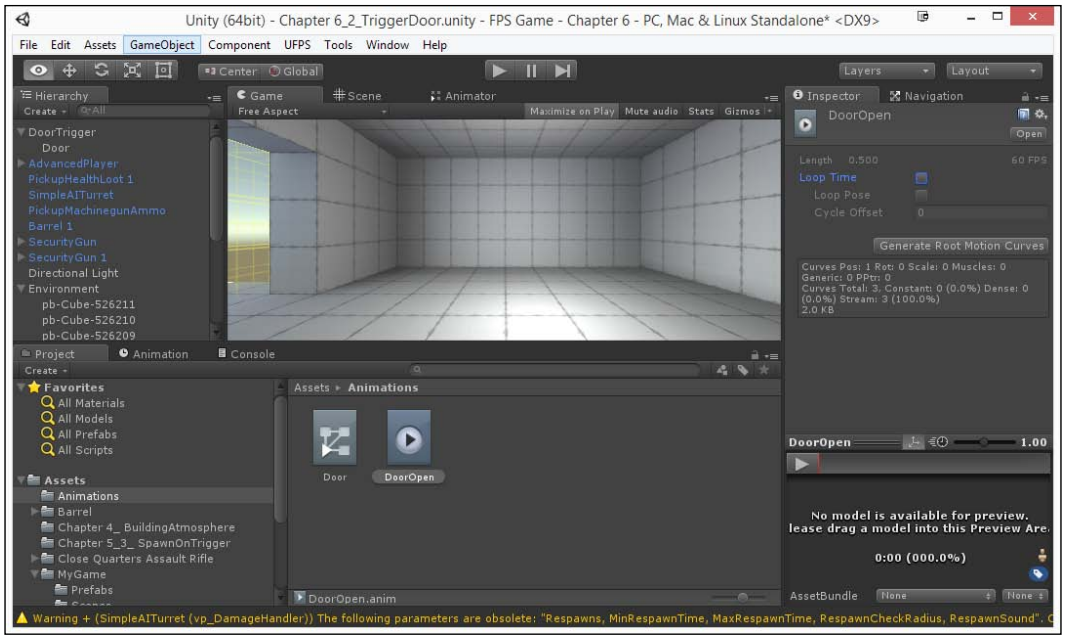
9. Click on the record button again to stop recording. Now, by default, this newly created animation will be played over and over again. To fix this, go to the `Door` object and, from the newly added **Animator** component, double-click on the **Controller** variable (called `Door`) to open up the **Animator** window.

10. Right-click, select **Create State | Empty**, and name it `Idle`. Then, right-click on the newly created `Idle` object again and select **Set As Layer Default State**.



Now, our animation will only play when we ask it to and not by default. However, once we play it, it will play over and over again.

11. To fix this, from the **Project** tab, go to the `Assets/Animations` folder and click on the `Door Open` object. Then, uncheck the **Loop Time** property so that it will not loop.



12. After this, create a `DoorClose` animation doing the reverse of what we did before.

    That being said, now that we have the animation, we need to call it within the code or it'll never happen.

13. Go to the `MyGame/Scripts` folder, select **Create** | **New C# Script**, and name it `DoorBehaviour`. Once finished, double-click on the created file to open `MonoDevelop`.

14. Once inside, change the code to the following:

```
using UnityEngine;
using System.Collections;

public class DoorBehaviour : MonoBehaviour {

  /// <summary>
  /// The Animator component of our door object
  /// </summary>
  public Animator doorAnim;

  // Called whenever an object with a collider enters our trigger
  void OnTriggerEnter(Collider other)
  {
    // Check if the player hit the trigger
    if(other.name.Contains("Player"))
    {
      Debug.Log ("Open Door");

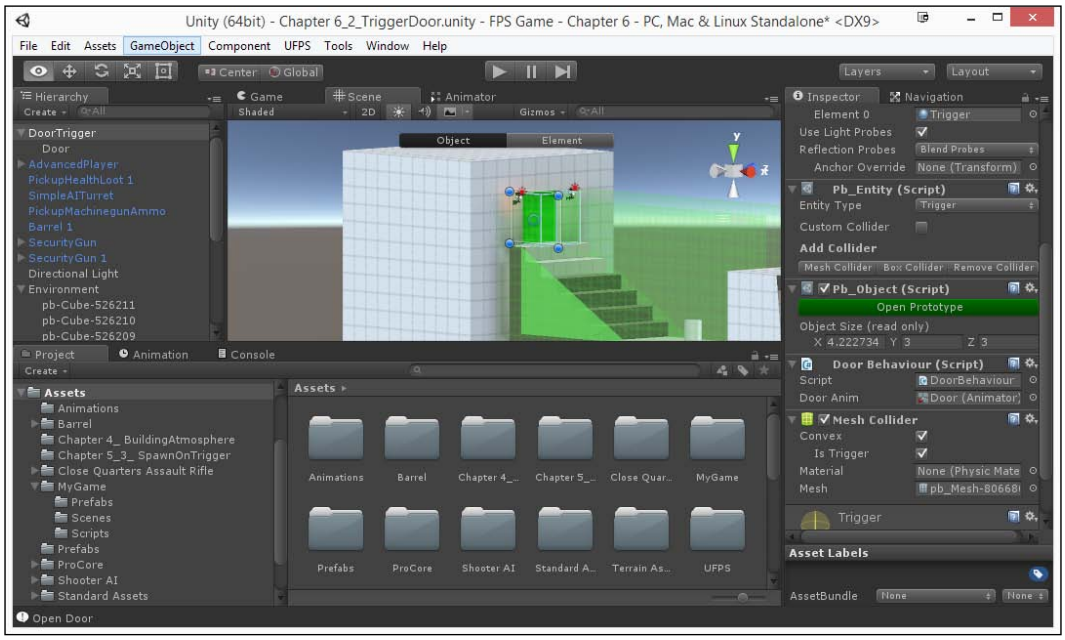
      // Play our door animation like normal (speed at 1)
      doorAnim.StopPlayback ();
      doorAnim.Play ("DoorOpen", -1, 0.0f);
    }
  }

  // Called whenever an object with a collider exits our trigger
  void OnTriggerExit(Collider other)
  {
    // Check if the player hit the trigger
    if(other.name.Contains("Player"))
    {
      Debug.Log ("Close Door");
```

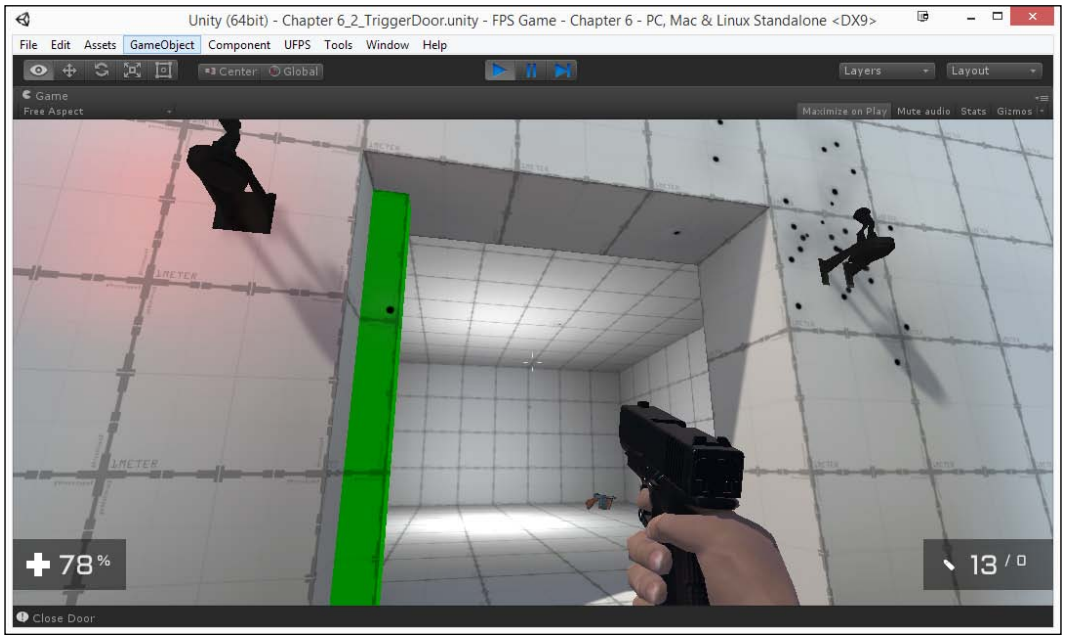
```
        // Play our door animation reversed (speed at -1)
        doorAnim.StopPlayback ();
        doorAnim.Play ("DoorClose");
    }
  }
}
```

This code uses two built-in functions inside Unity: `OnTriggerEnter` and `OnTriggerExit`. When our player hits the trigger, we will play our door animation. On exiting, we will play the same animation, but reversed. This way, the door will close for us.

15. Next, we need to attach this component to our trigger. Select it and drag and drop the `DoorBehaviour` file on top of it.
16. Once the component is attached, drag and drop our `Door` object under the `doorAnim` property.



17. Now, save your scene and run again.

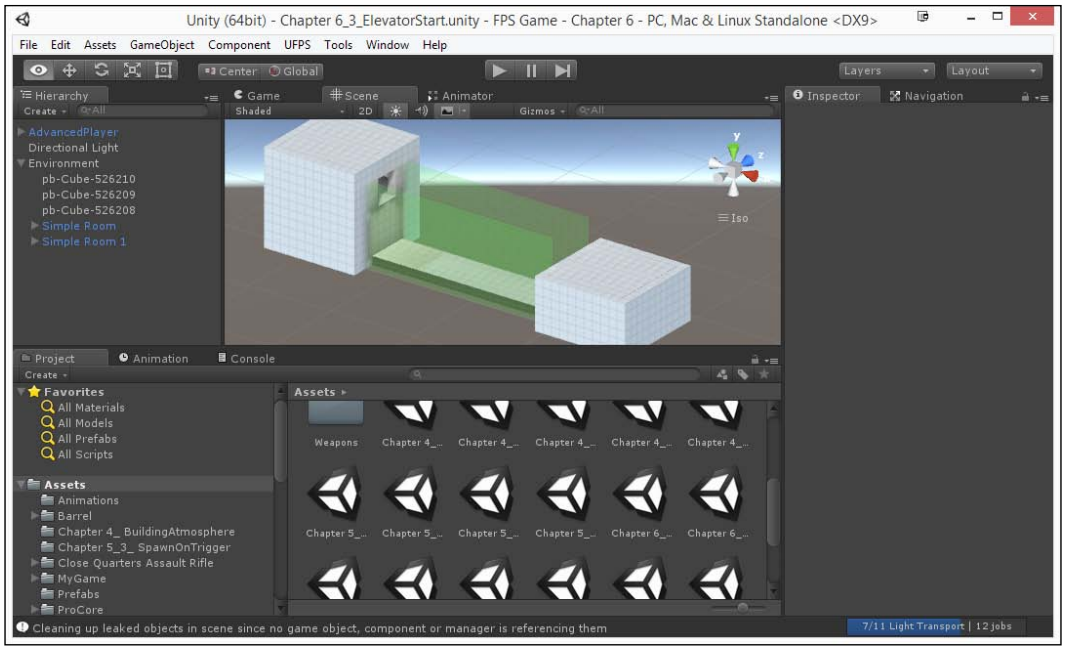


With this, we now have doors that will open and close when we approach them.

Creating an elevator

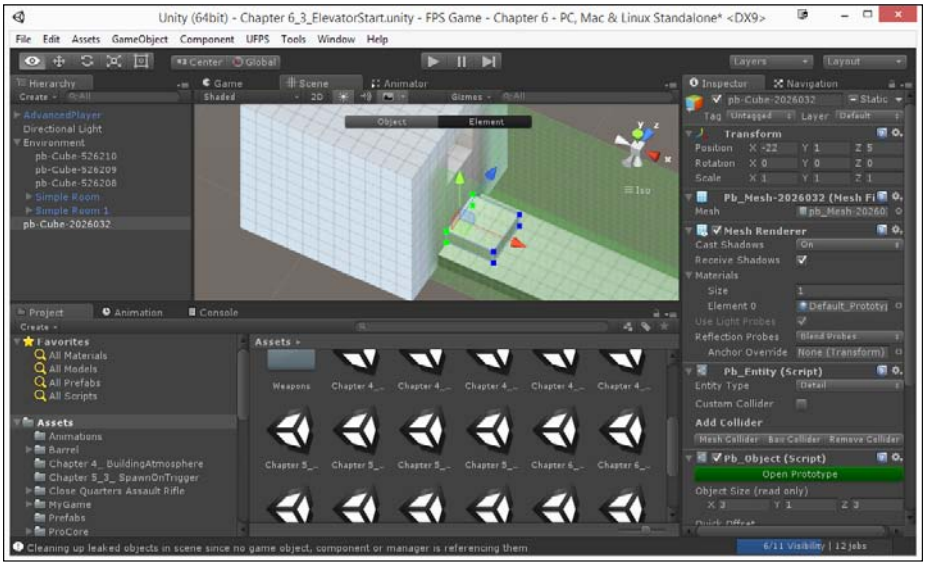
Continuing with what you learned from the previous section, we can take animations to another level by making moving platforms. However, UFPS comes with its own built-in way of doing this.

1. From the **Project** tab, open the `Chapter 6_3_ElevatorStart` level. This will give us a simple starting point to create an elevator.

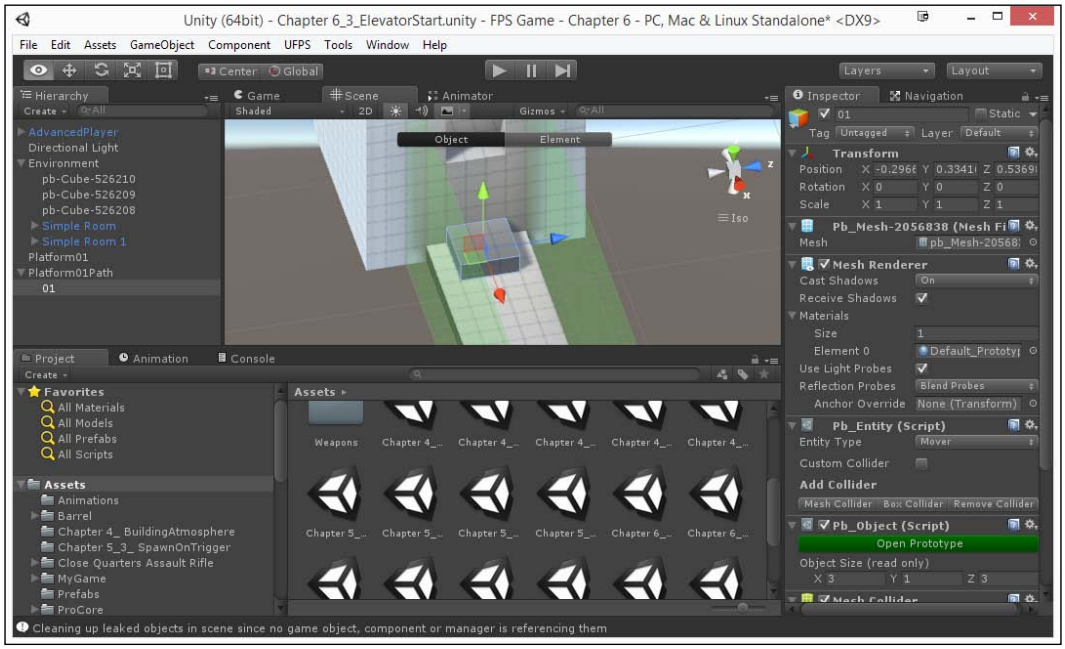


Alternatively, use a level that we created previously and add an elevator to it.

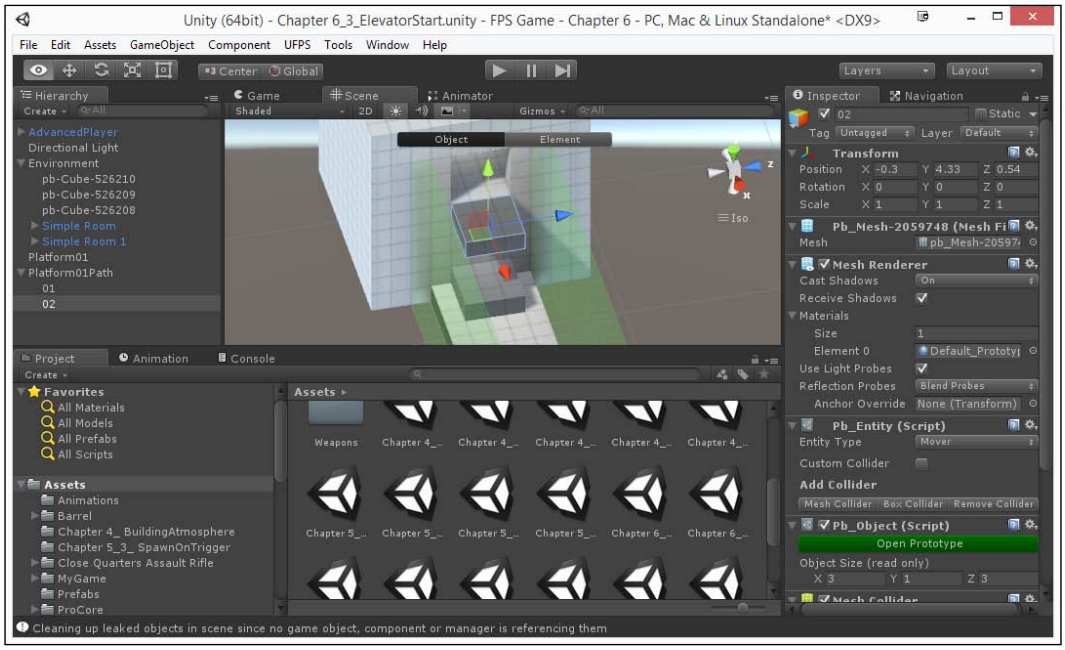
2. From Prototype, create a cube by pressing *Ctrl* + *K*. Move it to ground level in a 3 x 3 x 1 size cube.



3. Go to the **Object** editing mode and set **Vertex Color** to **Grab** to make it easy for us to see.
4. Rename the object to `Platform01`. Finally, from the **Prototype** tab, click on the **Set Mover** button to make the object a mover.
5. Create a new game object by going to **GameObject** | **Create Empty**. Name this new object `Platform01Path`.
6. Create a duplicate of our `Platform01` object by selecting it from the **Hierarchy** tab and pressing *Ctrl* + *D*. Rename this object to `01` and make it a child of `Platform01Path`.

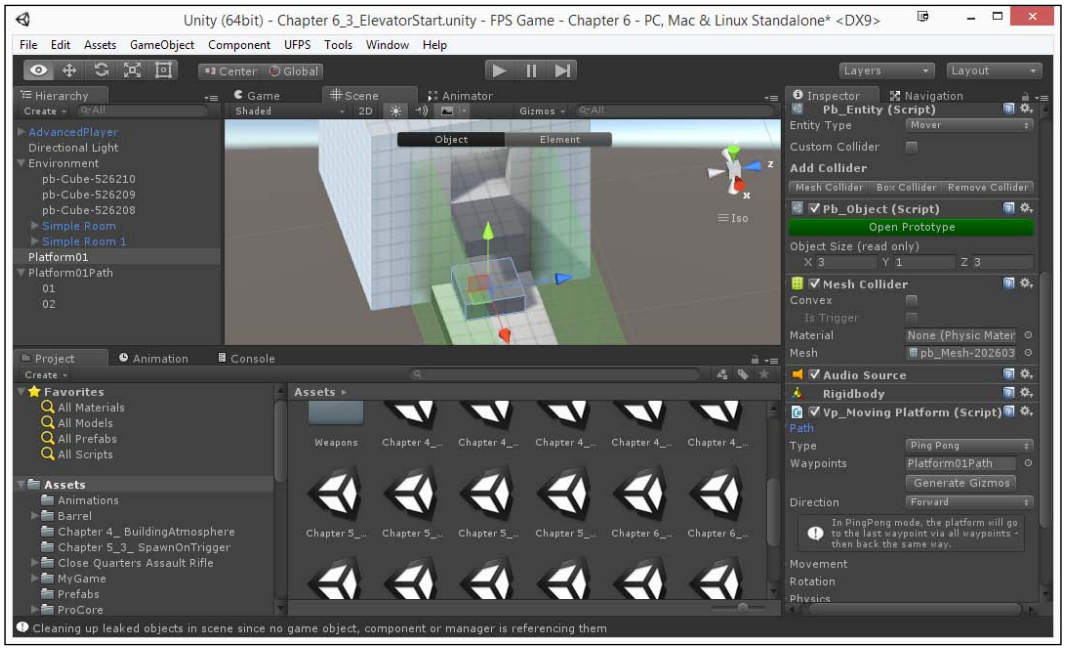


7. Next, duplicate the `01` object and move it up till it faces the point where we want the elevator to go.

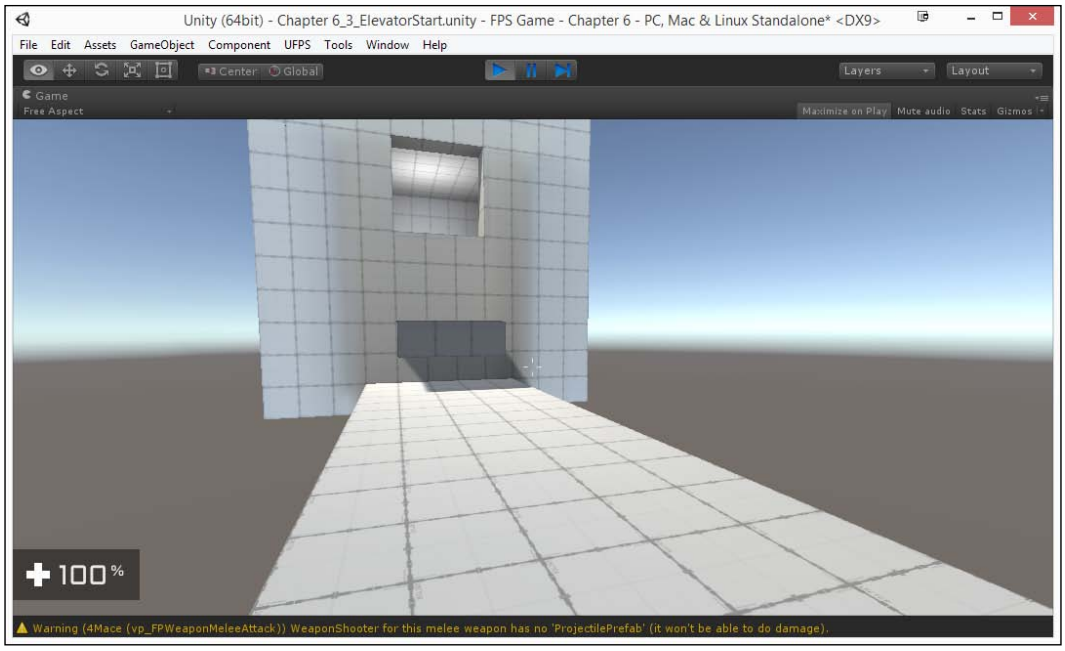


8. Now, select the original `Platform01` object and add a **Vp_MovingPlatform** component to it.

9. Once added, from the **Inspector**, expand the **Path** property and assign the `Platform01Path` object to the **Waypoints** property.



10. Finally, we need to have the path waypoints to not be visible when the game is playing, so select both objects in the path (01 and 02) and then from the **Prototype** tab click on **Set Trigger**. We don't need the **Mesh Colliders** though, so go ahead and remove them by right clicking on them and selecting **Remove Component**.

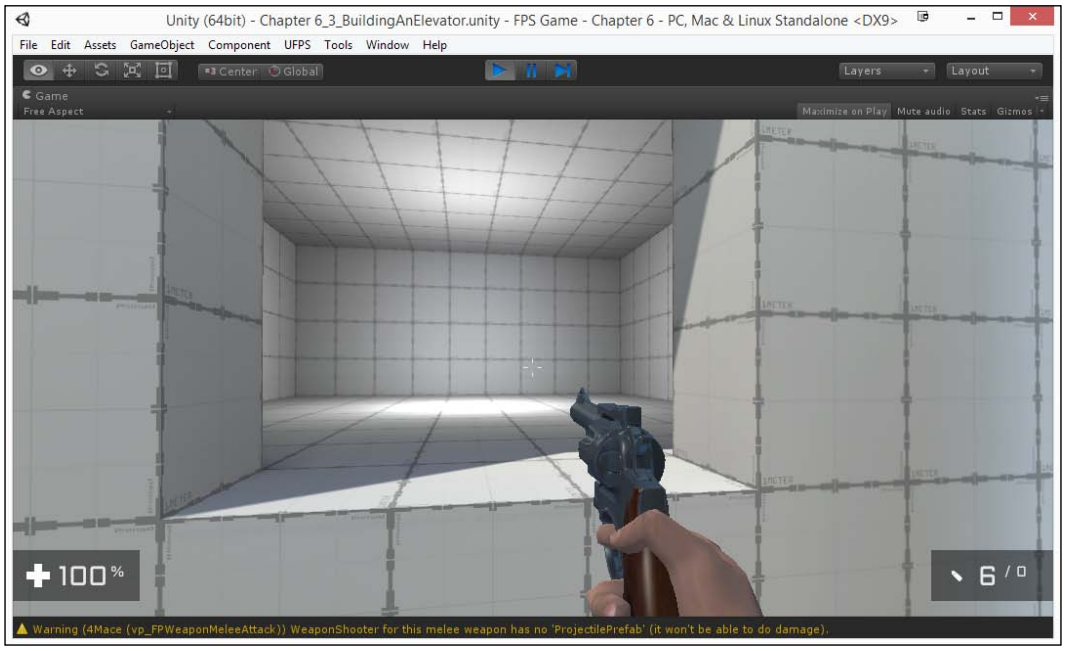


If we play the game now, the elevator should move up and down continuously. This is great, but there are some additional properties we can set to have the elevator react to our player, such as the paths we could have the object go.

11. From the **Vp_Moving Platform** component, change **Type** to `Target`. This mode will have the platform start off still and then move when the player stands on it. We also have some additional parameters we can change. Set **Return Delay** and **Cooldown** to `1`.

    Return Delay is the duration the platform will wait at the destination before it goes back. The **Cooldown** property is how long the platform will stay at the end of its animation before going back.

12. Save your level and start the game.



Now, our new moving elevator platform will work in a much more player-friendly way. We have exactly what we're looking for!



For more information on moving platforms and their paths, check out the UFPS' documentation at `http://www.visionpunk.com/hub/assets/ufps/manual/platforms`.

Summary

At this point, we have a firm foundation for you to create more interesting encounters and functionality in your levels. Specifically, we covered how to build an explosive barrel prefab and triggers for gameplay (doors) and create an elevator with interaction.

With this in mind, in the next chapter, you will learn how to polish our levels with effects while you finalize your game.

7
Adding Polish with ProBuilder

At this point, we have levels that we can work with and have play-tested and refined them to the point where we're ready to start polishing them to get ready for release.

Previously, we have been using Prototype to do our indoor level creation. This has been great, but when we want to start building professional-quality levels, we need to start adding more details such as textures to the level.

Different companies and game engines deal with this in different ways. Some will take their prototyped levels and then export them to a file to bring them into a 3D modeling tool, such as 3dsMax or Maya, to create their environment, often converting the original brushes into colliders. Other companies will instead refine their prototypes, add in more details, and then add materials to the brushes that were created; this is the approach that we are going to use.

Now, Prototype doesn't come with this feature, but its ProBuilder tool does. In this chapter we are going to explore how to use this tool to create the best levels that we can.

Here is an outline of our tasks:

- Upgrading from Prototype to ProBuilder
- Creating materials
- Working with ProBuilder—placing materials and UVs
- Meshing your levels

Prerequisites

Before we start, we will need to have a project created that already has UFPS and Prototype installed. If you do not have that already, follow the steps described in *Chapter 1, Getting Started on an FPS*. In addition, I am also assuming you have a level that you want to add encounters to.

Upgrading from Prototype to ProBuilder

Upgrading our projects so we can use them in ProBuilder isn't as simple as deleting Prototype and adding ProBuilder because of the way it is programmed; they both use different types of components. Thankfully, there is a tool that can act as a intermediate stage and that will save all of the data needed so we won't have to redo everything. So let's learn how to go about doing it.

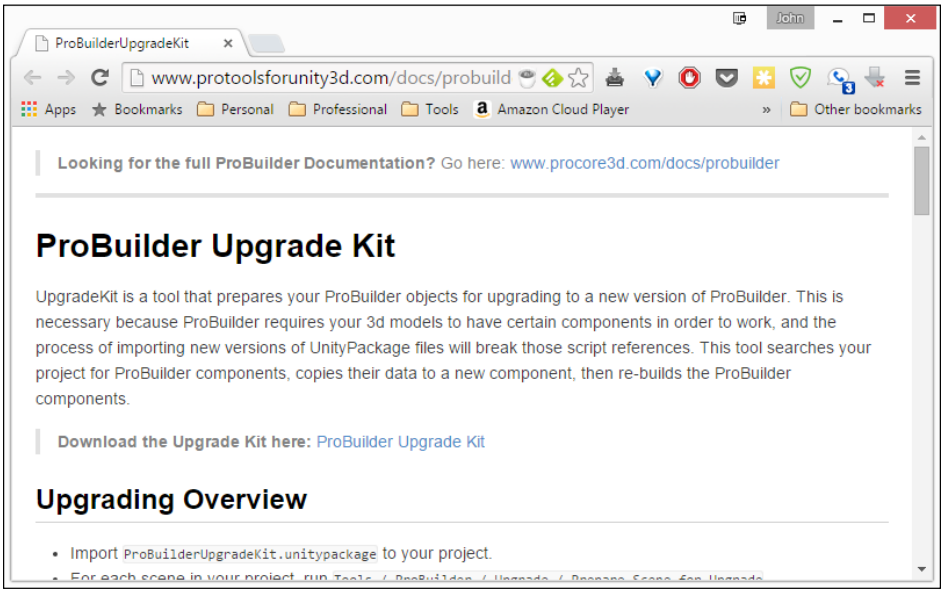


Before continuing, I do want to mention that the upgrade process is *not* reversible, and if an error occurs, you will almost certainly lose all your work. Make a backup of your entire project before attempting to do this!

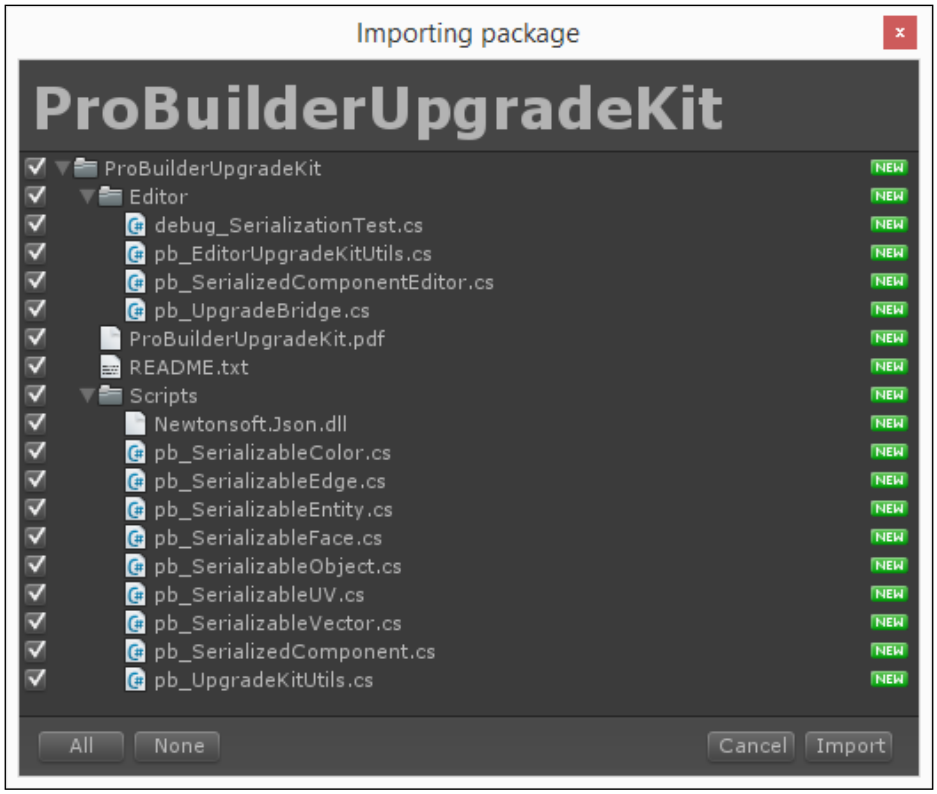
It's also important to note if you are using ProBuilder Basic and updating to Advanced and are using 2.4.7 or higher, all you need to do is import your new package.

To be completely sure of the steps needed and what to do for whatever version of the tools you're using check out the following page: `http://www.protoolsforunity3d.com/docs/probuilder/#installingAndUpgrading`

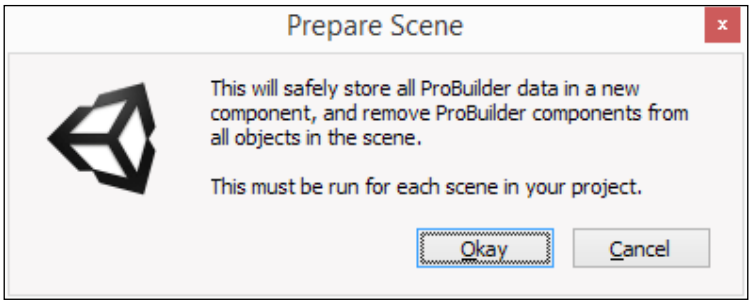
1. To start off we will need to go to the ProBuilder site to download a new Unity package called the **ProBuilder Upgrade Kit**. To do that open up your web browser and visit: `http://www.protoolsforunity3d.com/docs/probuilder/ProBuilderUpgradeKit.html`.
2. From there click on the **ProBuilder Upgrade Kit** link to download the package, clicking on **Download** once you get to the Dropbox file.



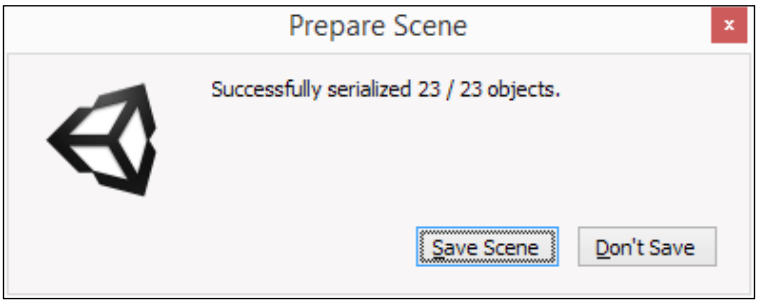
3. Once it's finished downloading, open up your Unity project (making sure to back up your previous one first), then go to the top bar, and then select **Assets** | **Import Package** | **Custom Package...** Then browse to your `Downloads` folder to select the `ProBuilderUpgradeKit.unitypackage` file.
4. Once it comes up, verify that everything is checked and then click on the **Import** button.



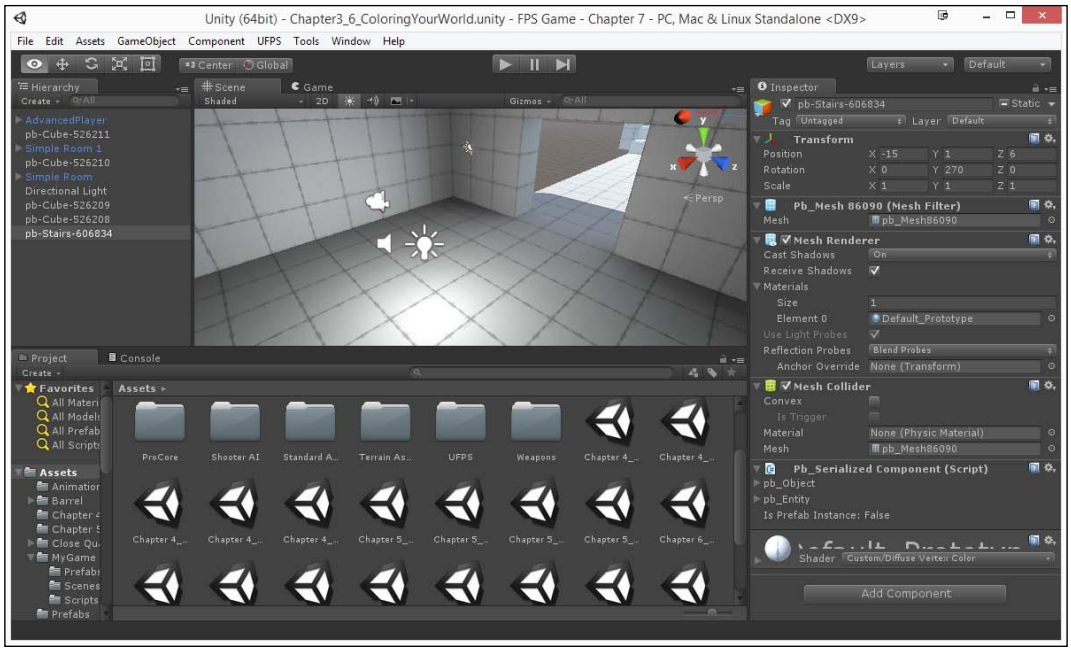
5. Next, open up each scene in your project that uses Prototype and then go to the top bar and select **Tools** | **Prototype** | **Upgrade** | **Prepare Scene for Upgrade**. It will bring up a popup saying that it will store the data and remove the old content. Click on the **Okay** button:



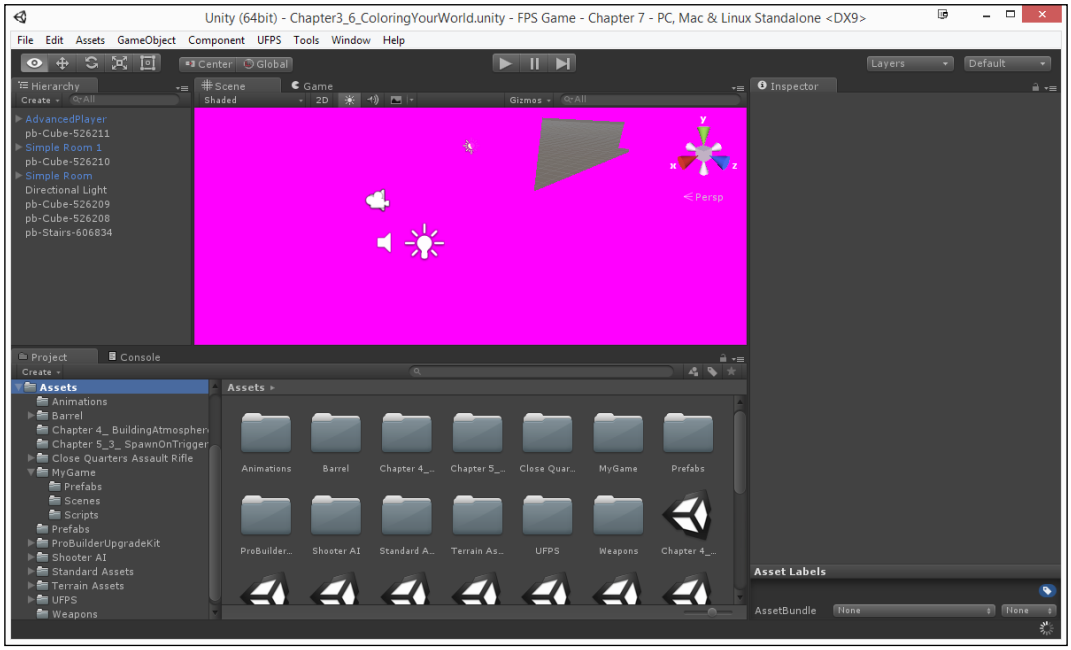
6. Once it finishes, another popup will appear saying that it successfully serialized the objects. Click on the **Save Scene** button and then do this again for every other level.



If you look at the scene, you'll notice that the Prototype components have been replaced with a new `Pb_Serialized Component (Script)` component. This is the data needed by ProBuilder to recreate your scene with the new functionality built in and ensures we can remove Prototype safely.

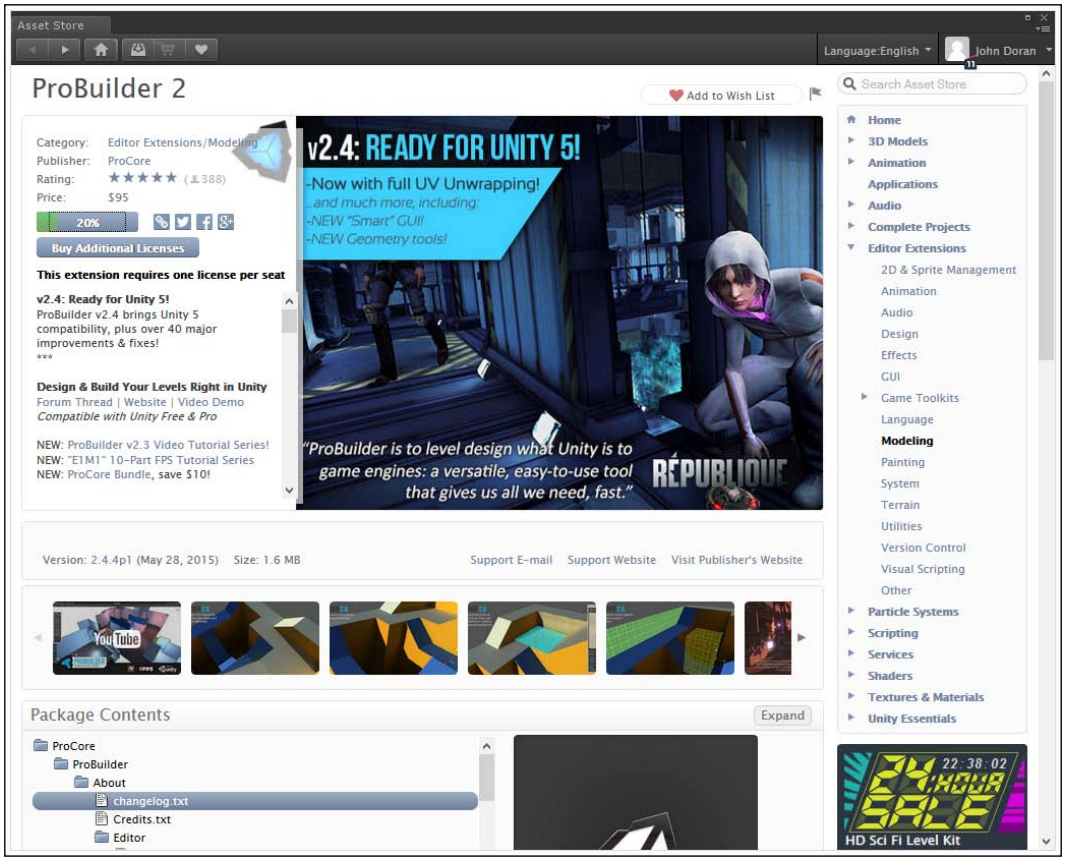


7. Now that all of the objects in all scenes are fixed, we can delete Prototype from our project. From the **Project** tab, select the `ProCore` folder and then delete it. At this point your screen may look something like this:



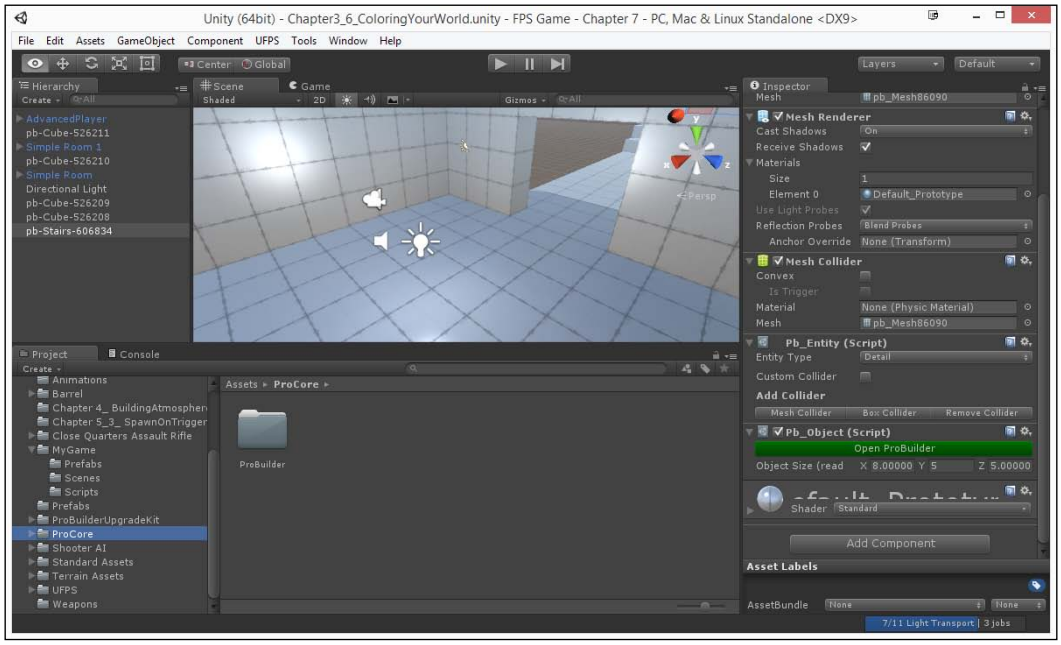
You may also see some errors on the console. This is perfectly normal and expected because some of the data is no longer there.

8. Now we need to get ProBuilder, so let's go to the **Asset Store** by going to the top bar and then selecting **Window | Asset Store**. Log in as discussed before, search for `ProBuilder`, and select **ProBuilder 2**. Purchase the asset if you haven't already, and once finished download the asset.



9. Now, when the popup starts up click on the **Import** button to bring it in.

10. Now that we have ProBuilder completed, we need to go through all of our scenes; from the top bar select **Tools | ProBuilder | Upgrade | Re-attach ProBuilder Scripts**. You should notice everything going back to normal and if you select an object you'll know everything has gone correctly if you see there is a `Pb_Object` component with **Open ProBuilder** as an option.

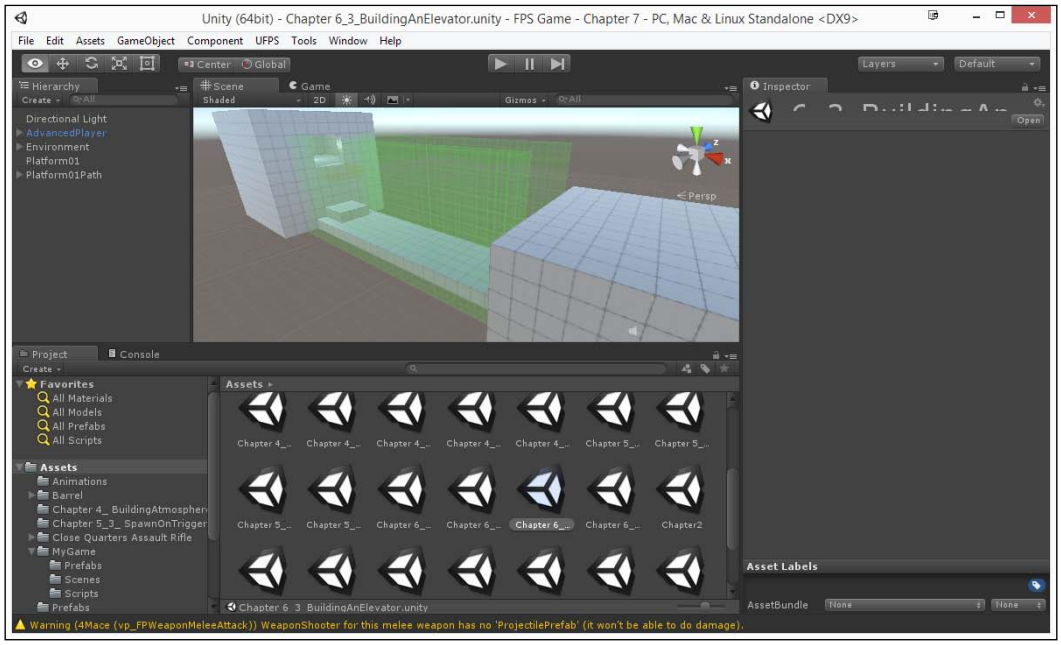


11. In levels where you have special properties such as Triggers, Colliders, and Movers, everything will work correctly when you play the game, but in the Editor mode the materials may be the normal colliders instead of the special ones from Prototype. To fix this, after conversion you may need to select the object again; from **Object** mode, select the **Set Trigger** (or whatever it is) option once more and save the level.

12. Once you are finished with all of the scenes, you can then optionally delete the `ProBuilderUpgradeKit` folder.



If you do delete the folder and there is a scene that hasn't been converted then you'll lose all that data, so be very careful!



And with that, we have ProBuilder completely installed!

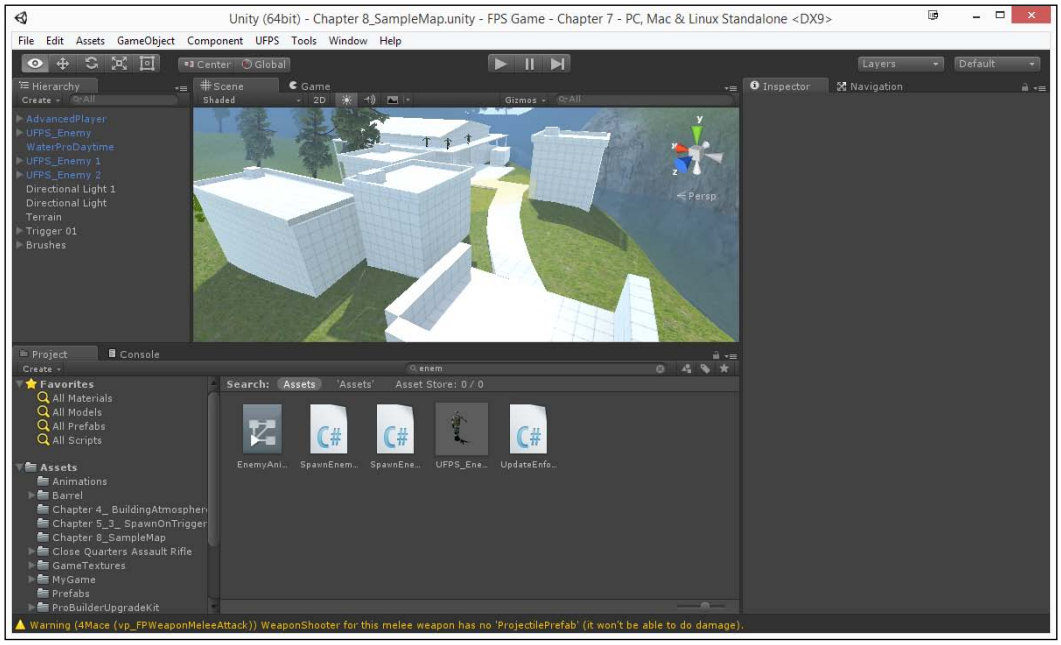


For another video example of the upgrading process, check out: `https://www.youtube.com/watch?v=O-Dz0Q3KgCs&feature=youtu.be`.

Creating material

Of course, we wouldn't have spent all this time getting ProBuilder ready if we weren't about to use it to make our projects nicer, so let's start this process by adding more details to our walls, making use of the materials. But, of course, we will actually need to have materials to do that, so let's get started with that!

1. To start off, we will need to open up a level to work with. Either open up your own level created previously or open `Chapter8_SampleMap` provided in the `Example Code` project.



This level uses the concepts that we learned about in the previous chapters, building a level in Prototype on top of the terrain that we created previously. Notice that, even though the objects are a tad more complicated than what we created before, I simply looked at real-life buildings for inspiration and fleshed them out with more details.

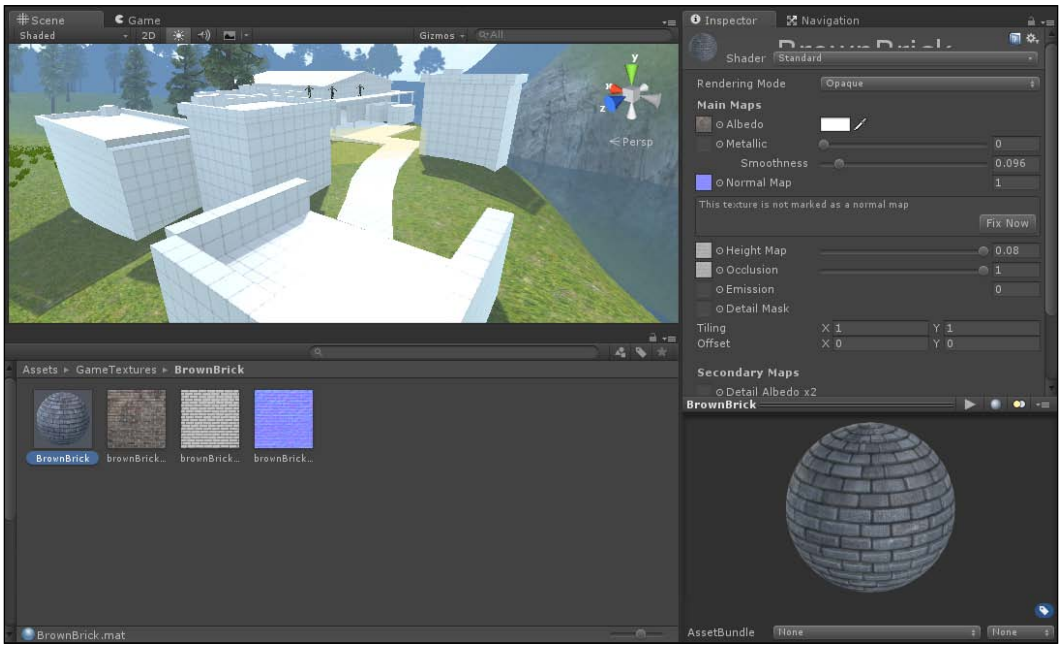
Now that we have a level here for us to work with, we will also need to have something to actually put on the walls. In Unity, we can control the visual appearance of game objects making use of materials. Materials control the visual aspect of an object. This will include the shader used on the object as well as its color, shinyness, and how bumpy it is in addition to many other properties. To create Materials, we usually apply textures, or image files, into various properties to adjust the settings. Let's see how that works.

2. In the `Example Code` folder for this chapter, you'll see a folder called `GameTextures`. Drag-and-drop this folder into your project and wait for it to import.

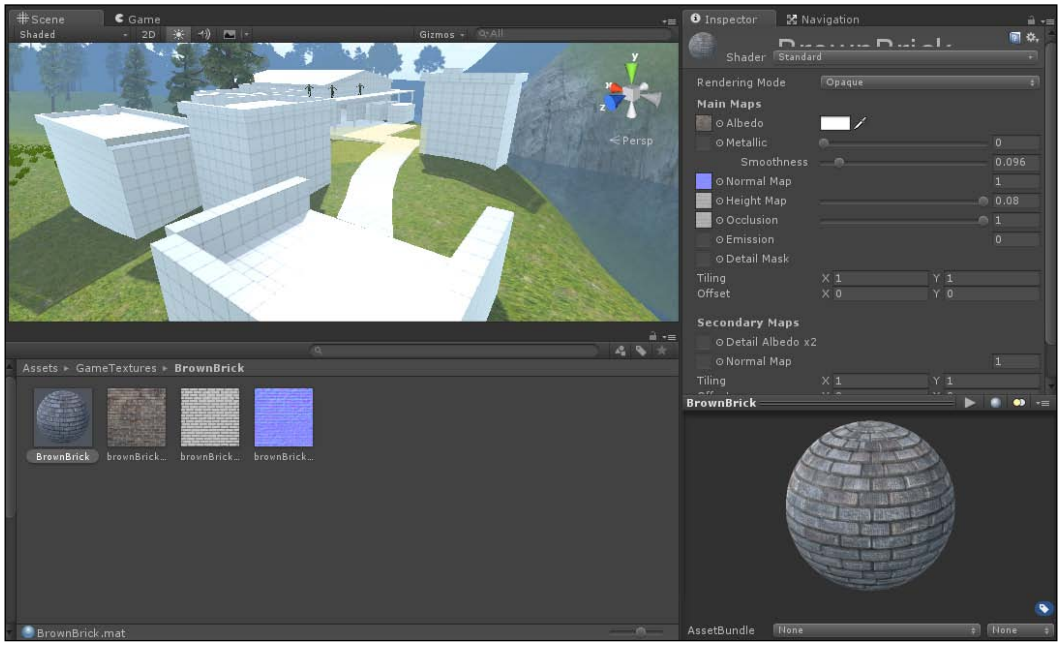


These textures were provided by the great folks at *GameTextures.com*, which has a huge array of ready-to-use materials for use in games with little to no effort. It's been a huge time-saver for someone like myself who doesn't have the time to build materials from scratch and is used by thousands of other indie titles as well as several huge-name AAA studios. You can find them at: `http://gametextures.com/`.

3. Each of these folders will contain the files that we will need in order to create a material. To get started, let's open up the `BrownBrick` folder from the **Project** tab and then click on **Create** | **New Material**. Once it's created give it the name `BrownBrick`.
4. Select the object and you should see a large number of properties in the **Inspector** tab. Drag-and-drop the `brownBrick_unity_albedo` file onto the box to the left of the **Albedo** property, the `brownBrick_unity_height` file onto the **Height Map** and **Occlusion** properties, and the `brownBrick_unity_normal` file onto the **Normal Map** property. Under the **Metallic** property, set **Metallic** to `0` from the slider and **Smoothness** to `.1`. Under the **Height Map**, change the slider to `.08`.



5. Once you do that you'll see text appear below the **Normal Map** saying that the texture isn't marked as a normal map. Click the **Fix Now** button and you should notice the material has become much nicer to look at.



Newly introduced in Unity 5, the **Standard Shader** is now the default way to create all materials for non-mobile platforms. It's a powerful tool, but may be a bit complex for newcomers. Here is a quick description of each of the properties we used.

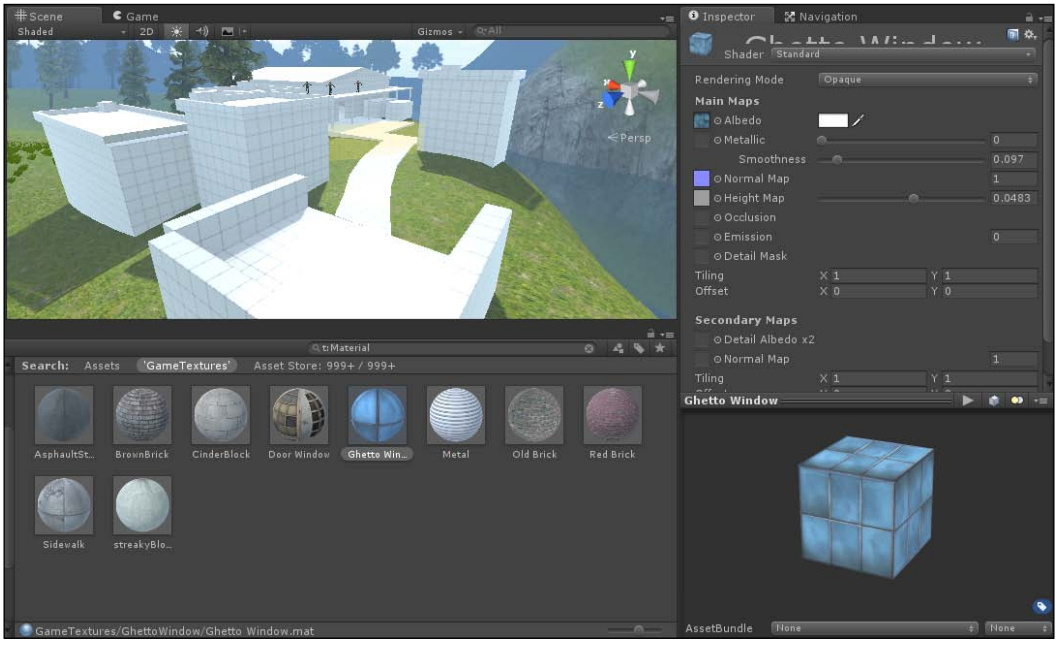
- The **Albedo** property stands for the base color to be used on the surface of the object to be placed.
- The **Normal Map** property is what's referred to as a bump map, which is a special kind of texture that will add bumps and other aspects to the material to artificially add depth to the object, as you can see in the changes done to the material.
- The **Height Map** property is another kind of bump map but, instead of showing the difference in direction the surface appears to face, it shows how high the surface should appear to be raised (black not at all, white all the way). The **Height Map** property has a slider that will allow you to scale how high or low the bricks appear in your image.

- Finally, the **Occlusion** map provides information on what parts of the material should receive indirect lighting from **Directional Light** and the like.
- The **Metallic** parameter sets the reflectivity and light response of the object. It contains two sliders, **Metallic** and **Smoothness**. **Metallic** sets how "metal-like" the surface is. `1.0` is pure metal, `0` is not metal at all. Smoothness is how smooth the surface appears to be. For instance, if **Smoothness** and **Metallic** are both at `1` you'll get a mirror.



For more information on the **Standard** parameters of **Shader** check out: `http://docs.unity3d.com/Manual/StandardShaderMaterialParameters.html`.

6. Go through the other materials and do the same thing, making sure to use the sliders to get what looks right for you specifically while learning how each property modifies the material. While doing so you may want to change the **Ghetto Window** on the bottom so that instead of showing a sphere you show a box instead. You can do this by clicking on the sphere icon to the right of the play button to the right of the **Ghetto Window** text.



And at that point you should have a new group of materials that you can use with ProBuilder!

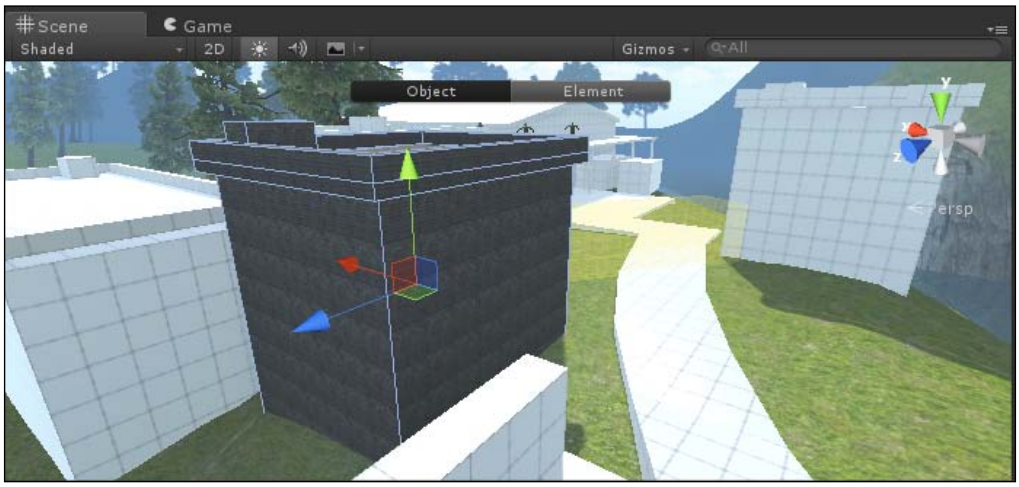


Alternatively, you can also use the `GameTexturesCompleted.unitypackage` file from the `Example Code` folder for this chapter to use what I am using.

Working with ProBuilder – placing materials

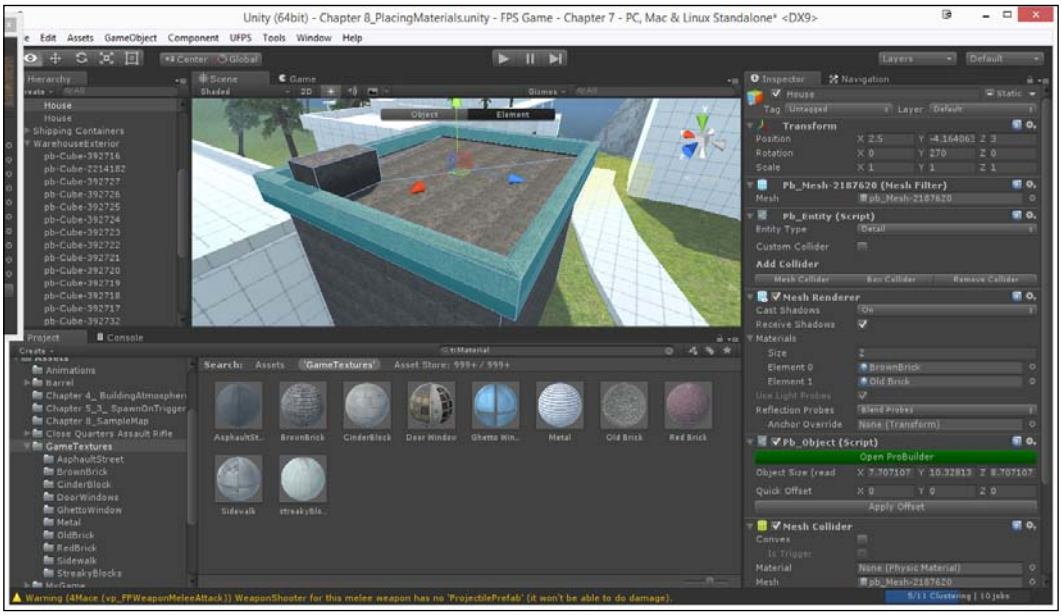
Now that we have the materials to work with, let's learn how we can place them into the scene by making use of ProBuilder's Material tools.

1. There are many ways to apply material to objects and faces with ProBuilder. For the easiest, simply select an object in your scene and then drag-and-drop one of the materials on top of it. For instance, in the next screenshot I've taken the `BrownBrick` material and applied it onto one of the houses in our level.

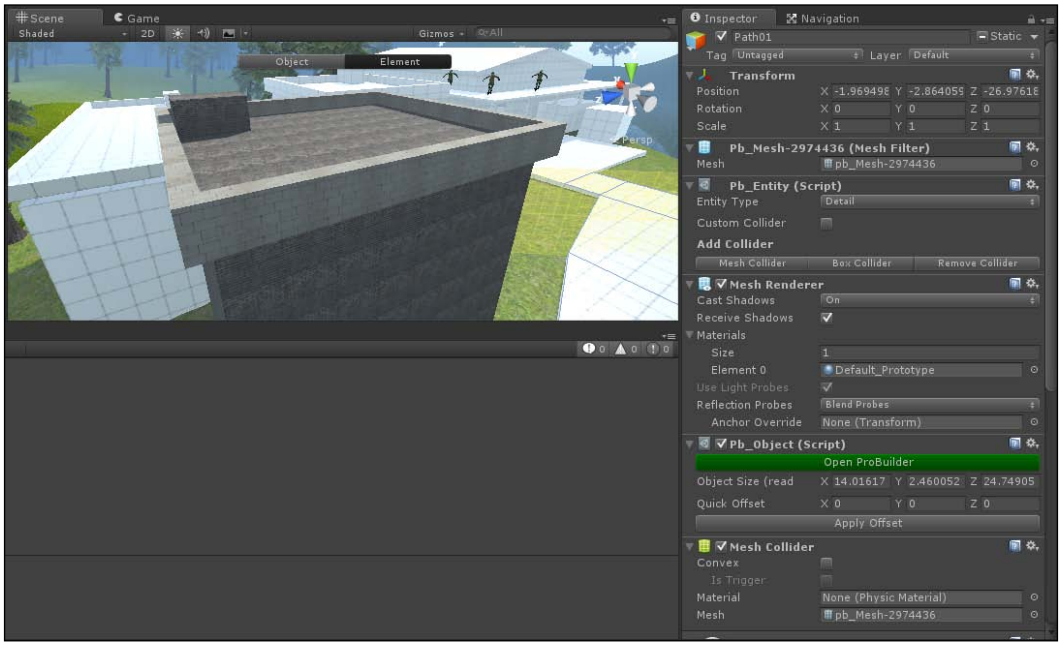


As you can see, all of the faces of the object now are using the `brick` material instead of the default. However, we may not want to use the material on every single face of the building. In addition, we may also want the bricks to be bigger, because right now they look incredibly tiny.

2. Open up the **ProBuilder** window and change to **Face** mode. From there, select the faces on the top of the house that we will want to replace with a different texture; for now, some trim around the roof will do nicely.



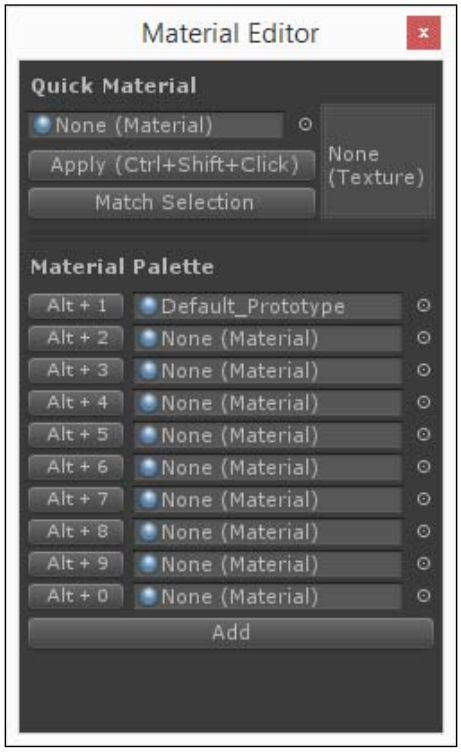
3. After that you can simply drag-and-drop a material onto your selection and it should apply the material to what you currently have selected (and nothing else).





Don't worry if things look a bit freaky for a period of time, Unity needs time to rebuild lighting after the Materials on objects have changed. Alternatively, if you want things to go quicker, go to the **Lighting** tab by going to **Window** | **Lighting** and then from the **Object** section uncheck the **Continuous Baking** button, but be sure to turn it back on before finishing the project!

4. Alternatively, we may want to paint the faces individually. This is also easy enough to do. Select your object once again. From the **ProBuilder** tab, click on the **Material** button to bring up the **Material Editor** dialog box.

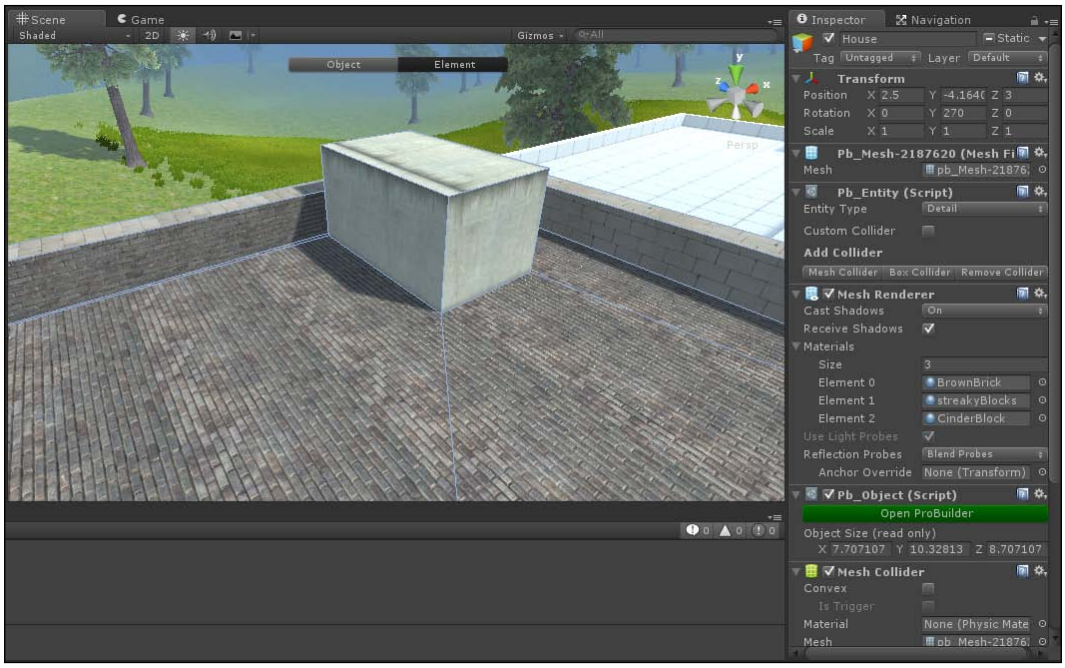


The **Material Palette** allows users to set up often-used materials and then just press certain hotkeys to apply them to whatever is selected. In addition there is the **Quick Material** parameter, which is what we're going to use now.

5. Drag-and-drop the `streakyBlocks` material into the **Quick Material** parameter; once that's completed, hold down the *Ctrl* + *Shift* keys and then start clicking on faces, such as the heater on the top of the house. This will allow you to quickly start applying faces.



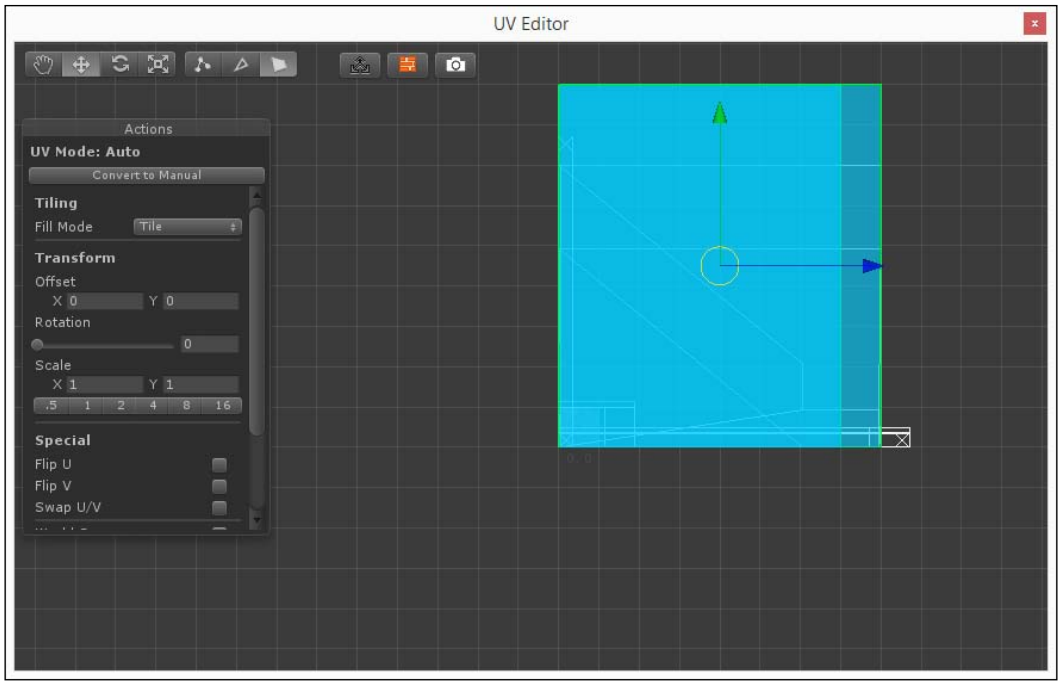
For more information on *Applying Materials* check out: `http://www.protoolsforunity3d.com/docs/probuilder/#texturesAndUVs`.





Alternatively, you can select a number of faces and then click on the **Apply** button to apply them all at once. You can also select any face in your level and click **Match Selection** to automatically select whatever material you used there—nice for when you're trying to find something specific for your level.

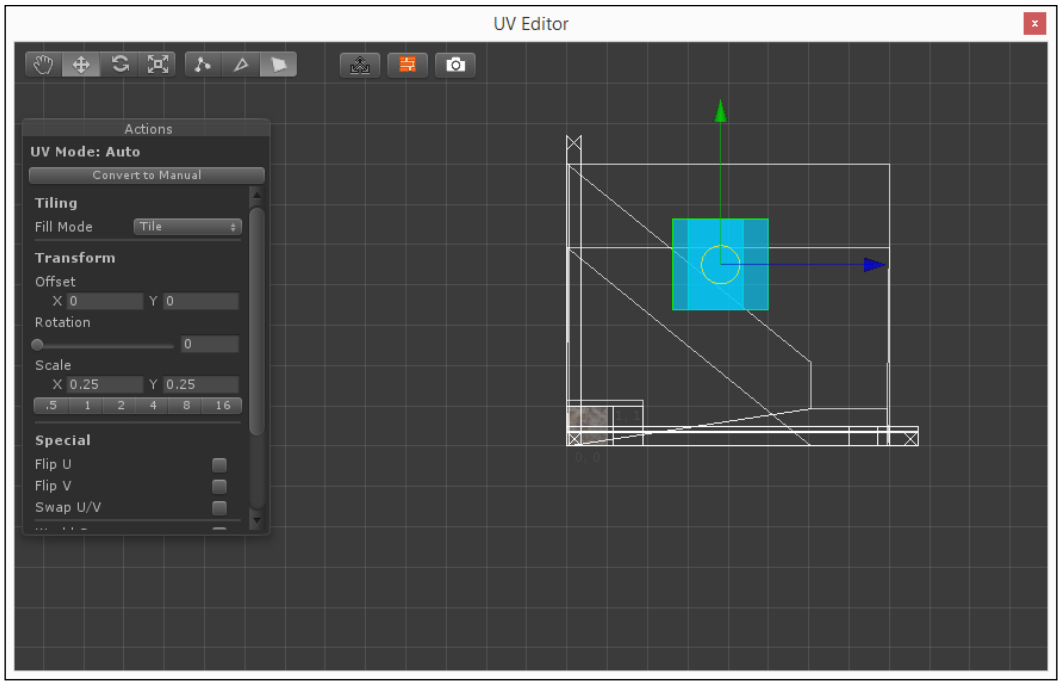
6. Now that we have some materials placed, let's fix the whole size and placement issue. Let's first fix the brick texture. Select the bricks around the house and then open up the **UV Editor** window by going to the **ProBuilder** tab and then selecting **UV Editor**.



When you zoom out the camera, you'll notice a lot of lines and some blue boxes. Well, the blue boxes are what you currently have selected in the **Scene** tab, and the white lines represent the UVs of the object: what will be painted onto the surface of the polygon. Rather than use an *X* and *Y* position, to avoid confusion they call the coordinates *U* and *V*.

Movement in the **UV Editor** is very similar to normal movement in Unity; you can hold the middle mouse button and move the mouse in order to pan the camera and use the mouse wheel to zoom in and out.

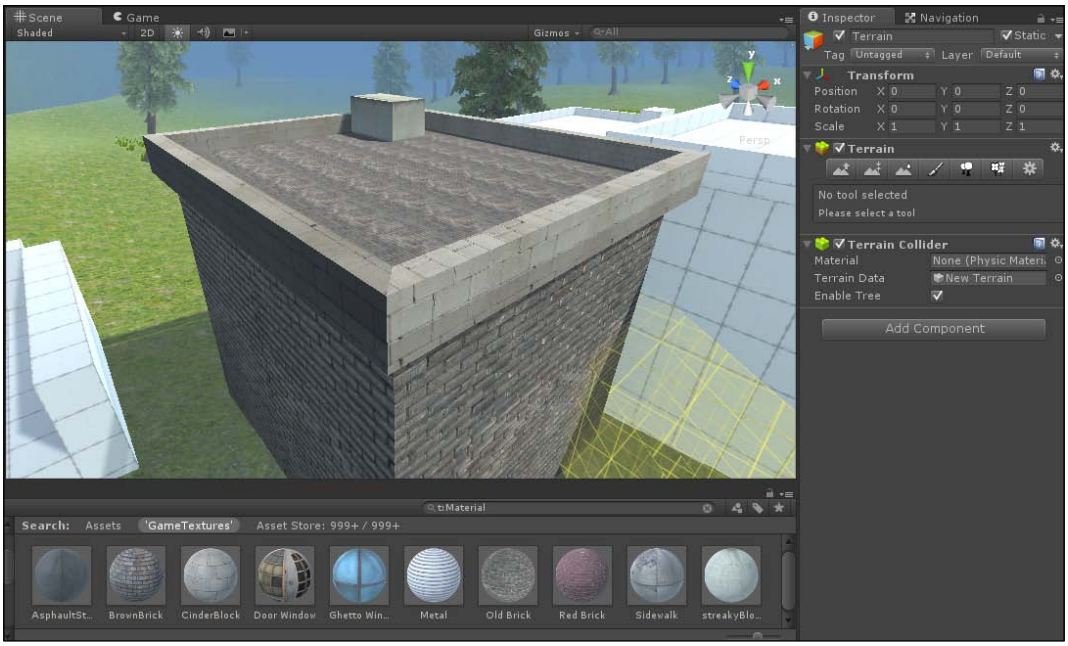
7. On the left-hand side, you'll see a number of properties in the **Actions** tab. What we want to look at is the **Scale** property. From there click on the **4** button to make our object have a scale of `.25`, which makes the plane four times smaller; this will make the texture applied to it four times larger.



This is what it looks from the UV side; from the unreal side, you'll notice that the texture got larger.

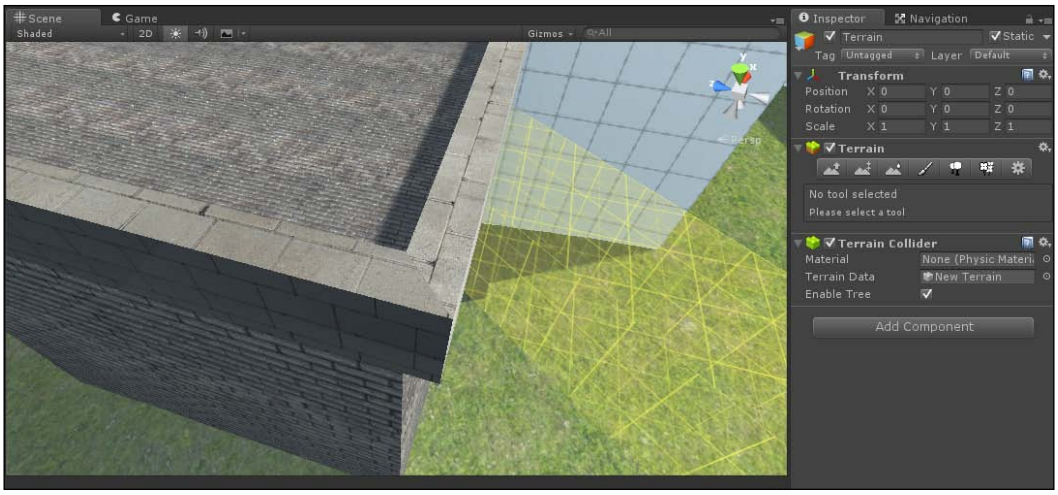


8. Next we'll select the top section and scale that up as well to `2` instead:



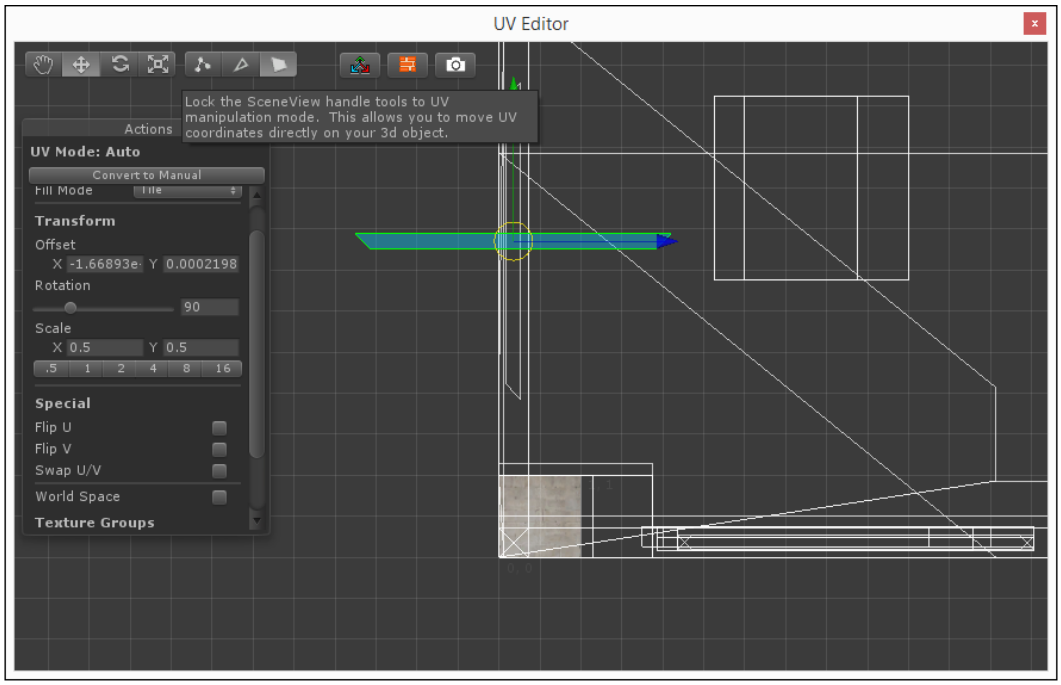
However, at this point there are some additional issues. The top row is facing a different way from the sides, and the bricks are different sizes in different places. Thankfully, we can fix this in a flash.

9. Select one of the top pieces that is rotated in the wrong direction. We can fix this by moving the **Rotation** property in the **UV Editor** with that face selected to `90`.

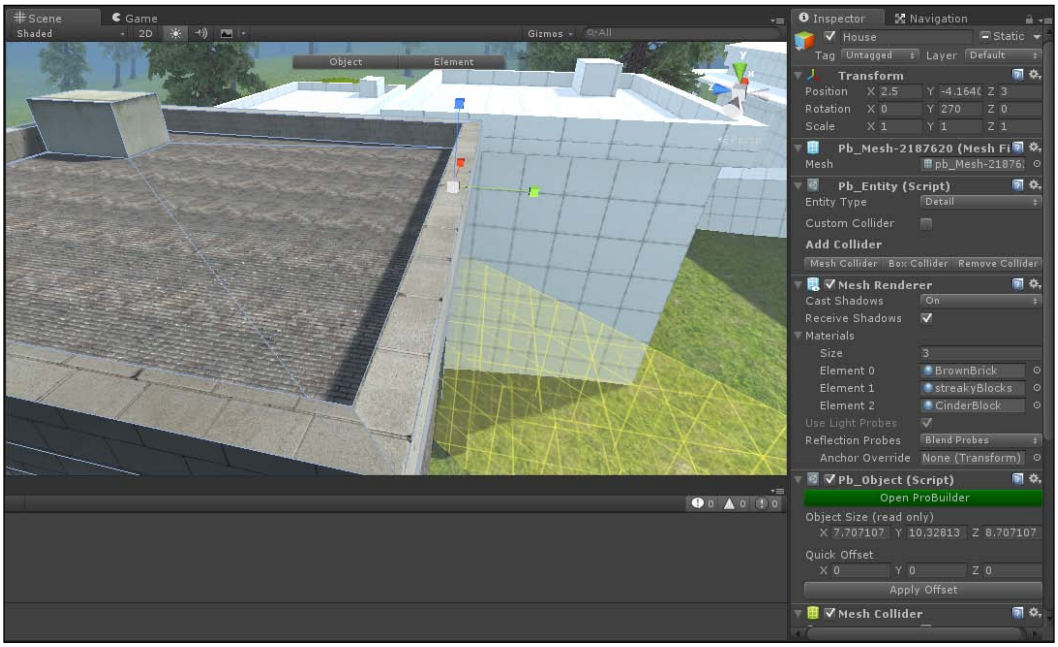


It's starting to get there, but what we really want is for the brick to fit the brush we created. It'd be a lot easier to just tweak it in the **Scene** view till it looks correct, and that's what we're going to do next.

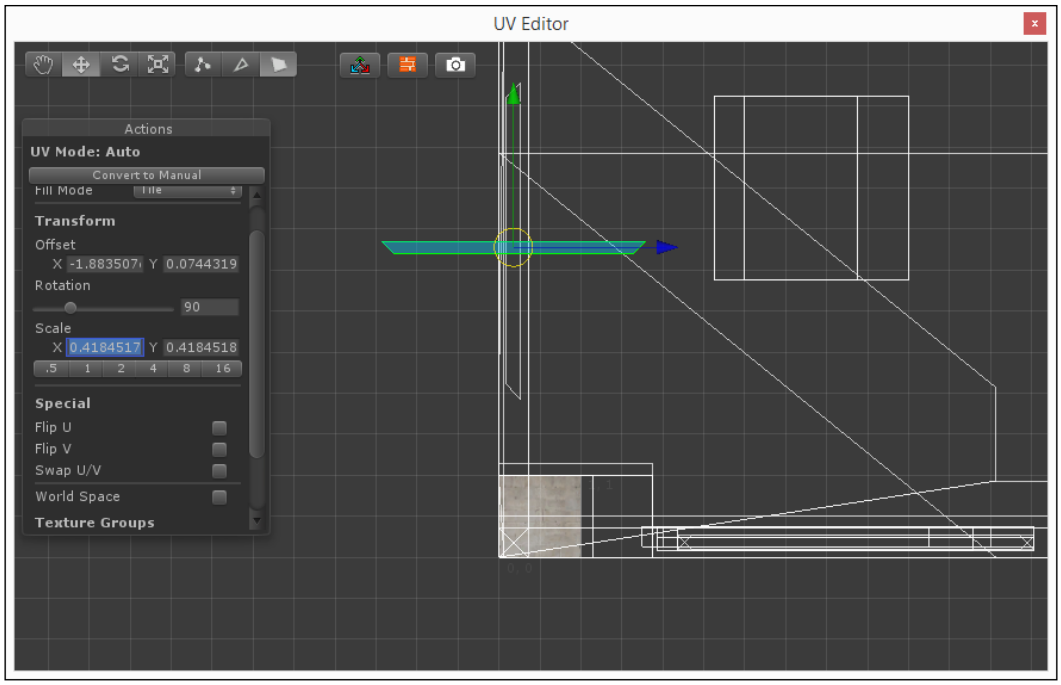
10. With our face selected, at the top (on the left side of the right part of the toolbar), you'll see some arrows pointing out in directions. Click on this to lock the **SceneView** map handles to the UVs instead of the object.



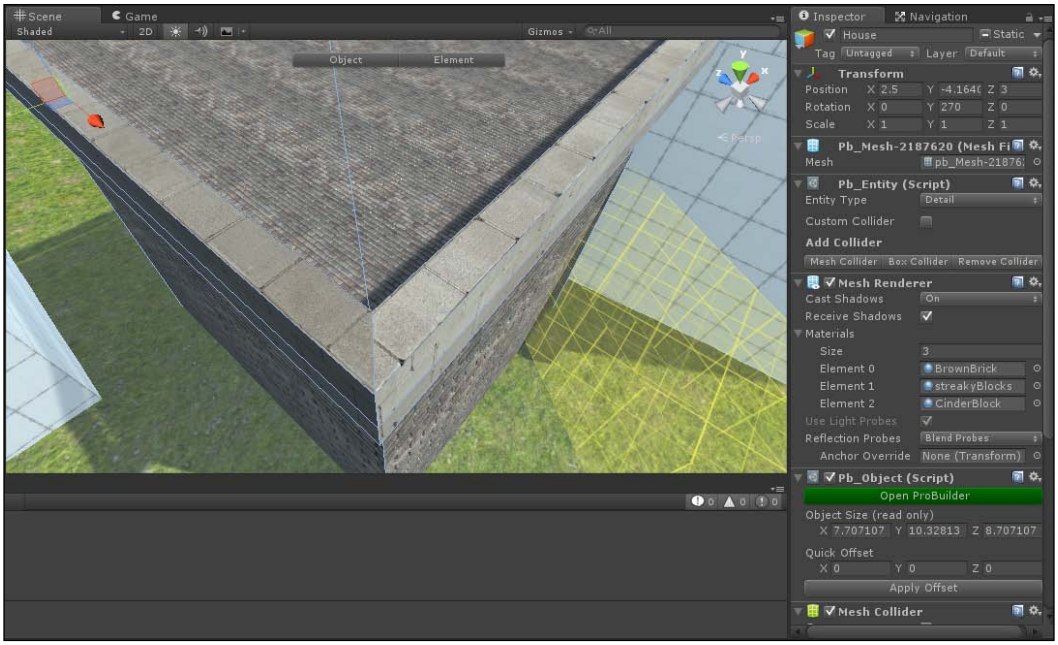
11. Now, back in the **Scene** view in the Unity editor, you can use the same tools we've been using in the environment to modify the UVs.



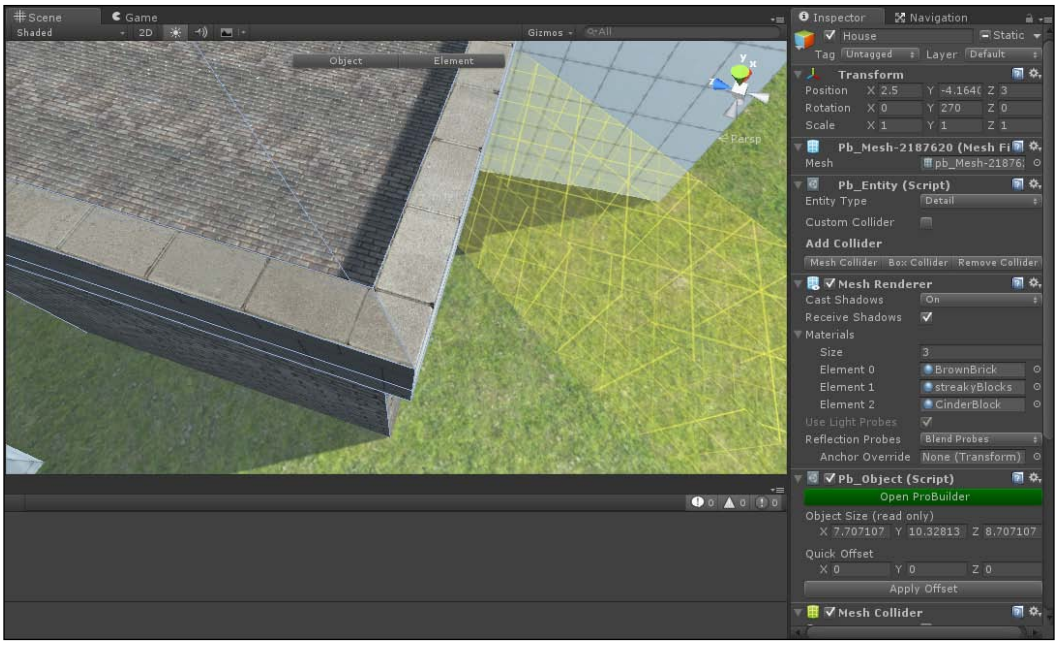
12. When you go back into the **UV Editor** you'll notice the properties have been set to whatever you did in the **Scene** view. Copy the **Scale** value, as we will use it for the other parts.



13. Select the face to the right, paste the **Scale** values in, and then translate them until they fit within the brush we created as well.



It's looking better, but the edges aren't meeting up correctly. Here we have two choices. We could either place a mesh here to hide the issue, or we can translate the UVs to make a small block there. In this case, I will make a small block, as follows:

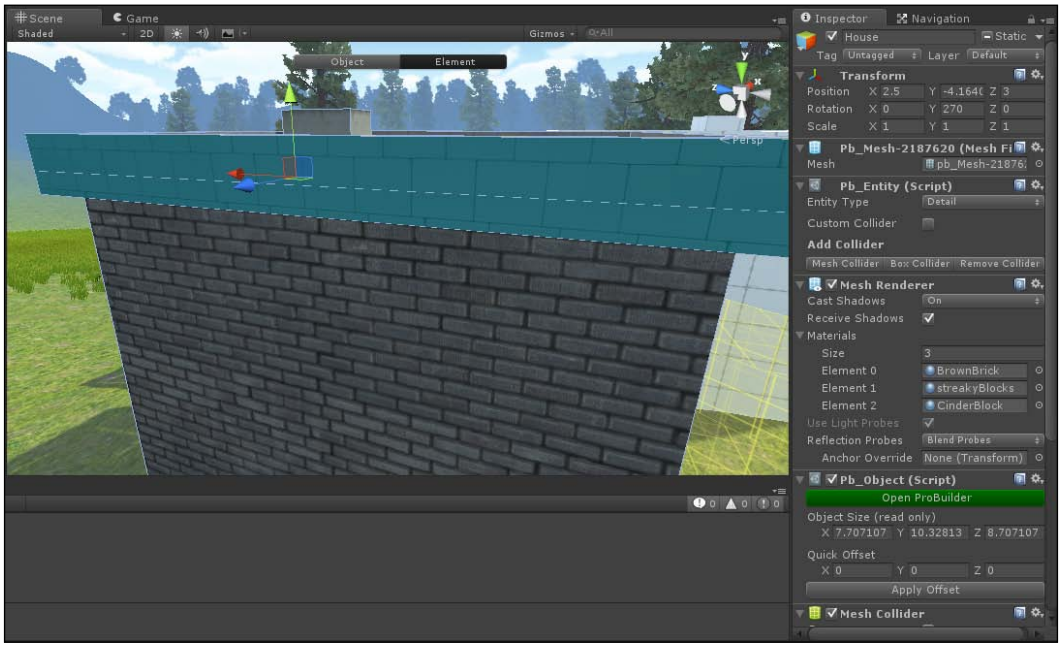


14. Next we will want to do the insides of the blocks.

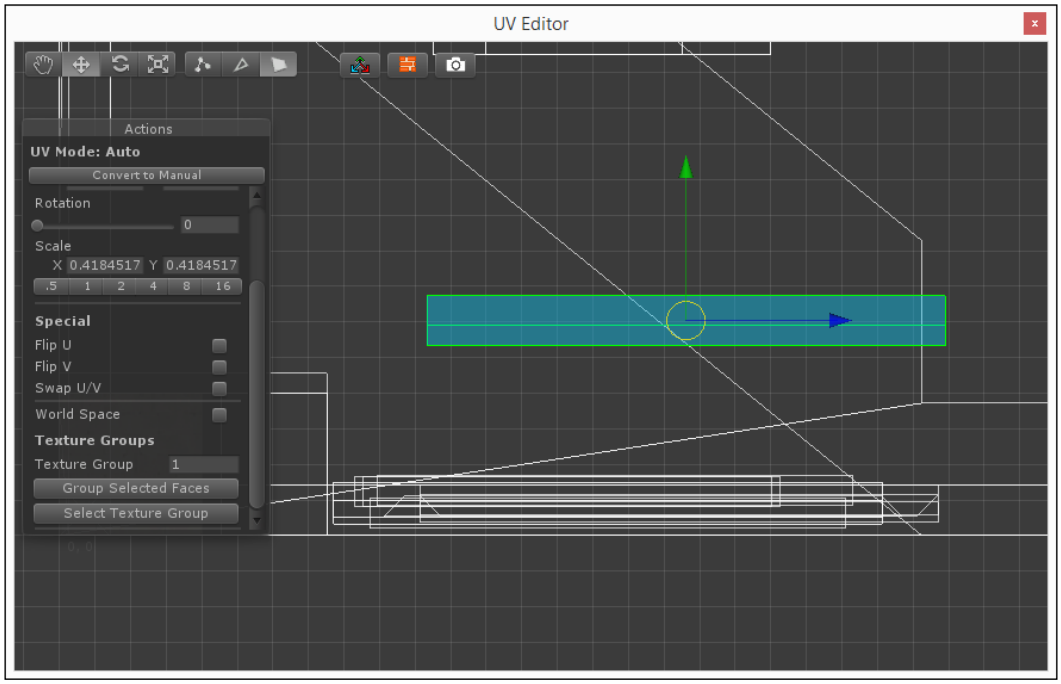


As you can see, the blocks are looking even better!

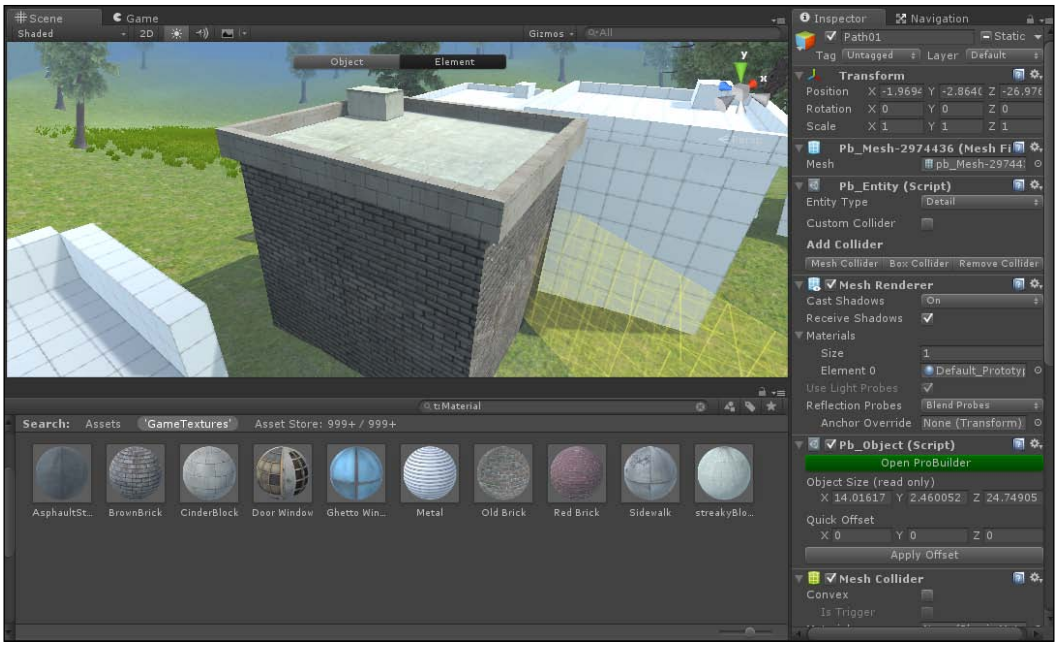
15. You may notice that the blocks on the outside look much larger than the others; that's because the top section here is broken up into two pieces. To have them show up correctly, we can use the **Group** tool. Select both outer faces by clicking on one and then holding down *Shift* and selecting the other.



16. Then back in the **UV Editor** tab click on the **Group Selected Faces** option. Once you've done that you'll see that the two separate UVs are now connected to each other and function as one.

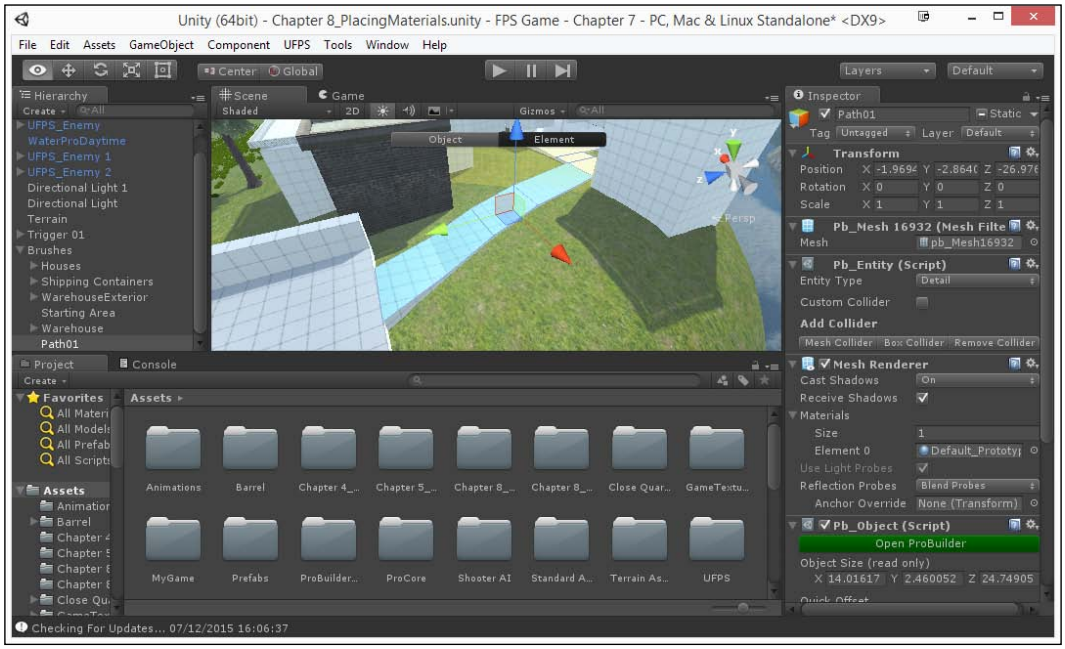


17. You should then be able to move the object and scale it correctly to fit each part of the outside. After that, I scaled up the top of the small addition on the top of the house to get rid of the grime and then applied that material on the top, grouping everything so that it fits the top.

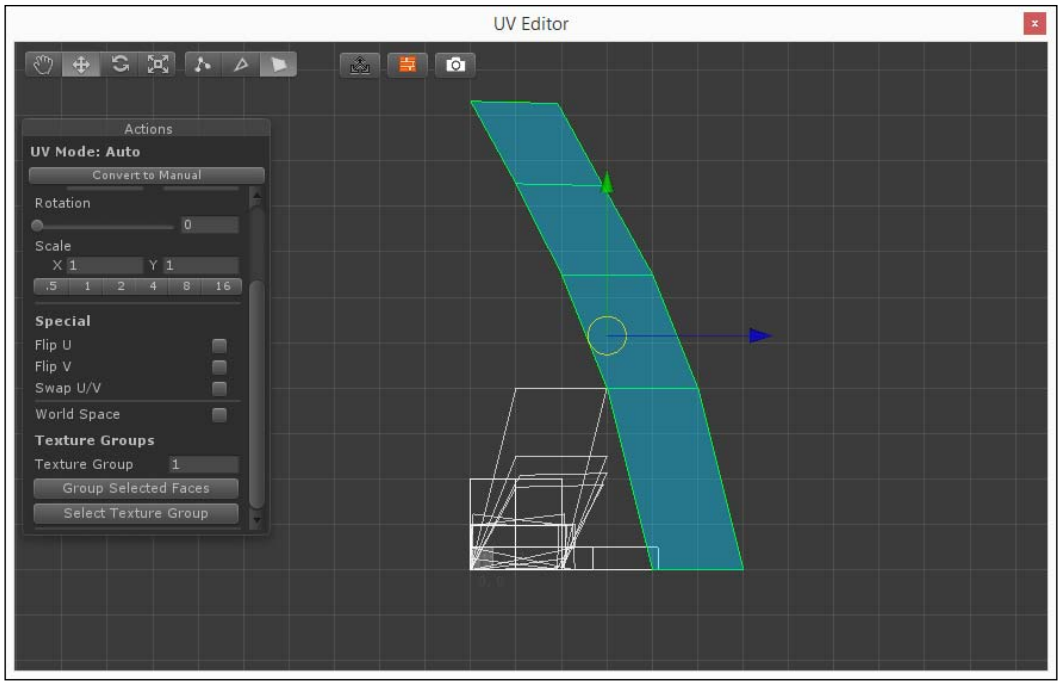


And with that we have one single house completed! This gives us a pretty good picture of how to build UVs for different things, but I wanted to show one more example that may help you deal with more complex things such as curves.

18. Continuing from the player's starting area, you'll see a curved road. Select those faces and then assign the `Sidewalk` material to the object. Once that's done, open up the **UV Editor** once again.

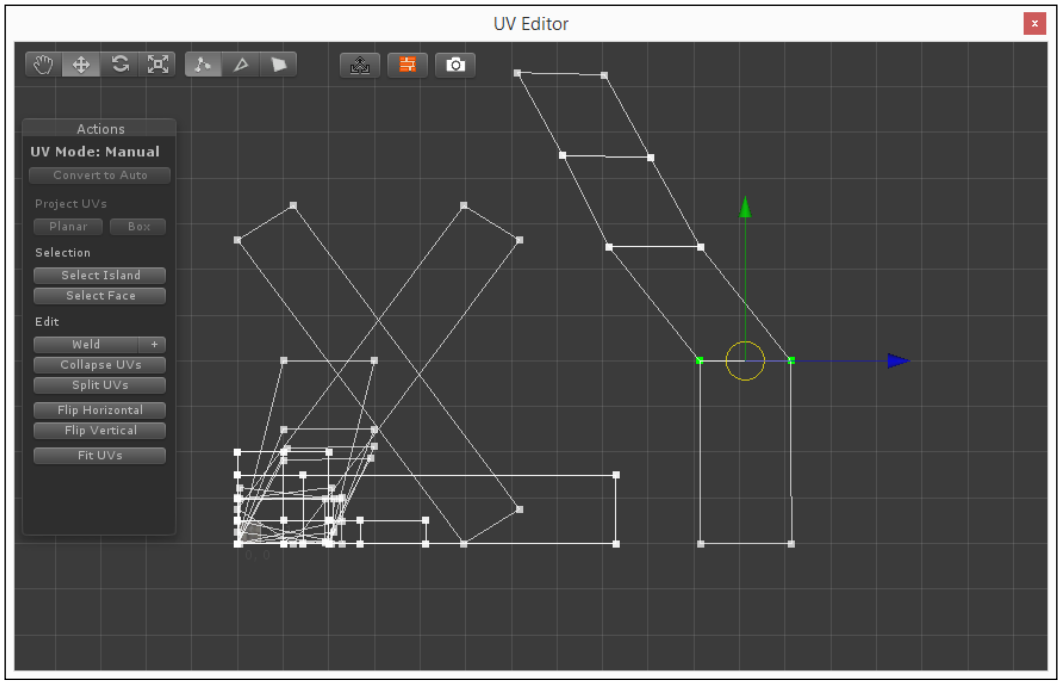


19. Currently things are too small and they're all going in a certain direction and not flowing correctly. To start fixing this, in the **UV Editor** select **Group Selected Faces**.

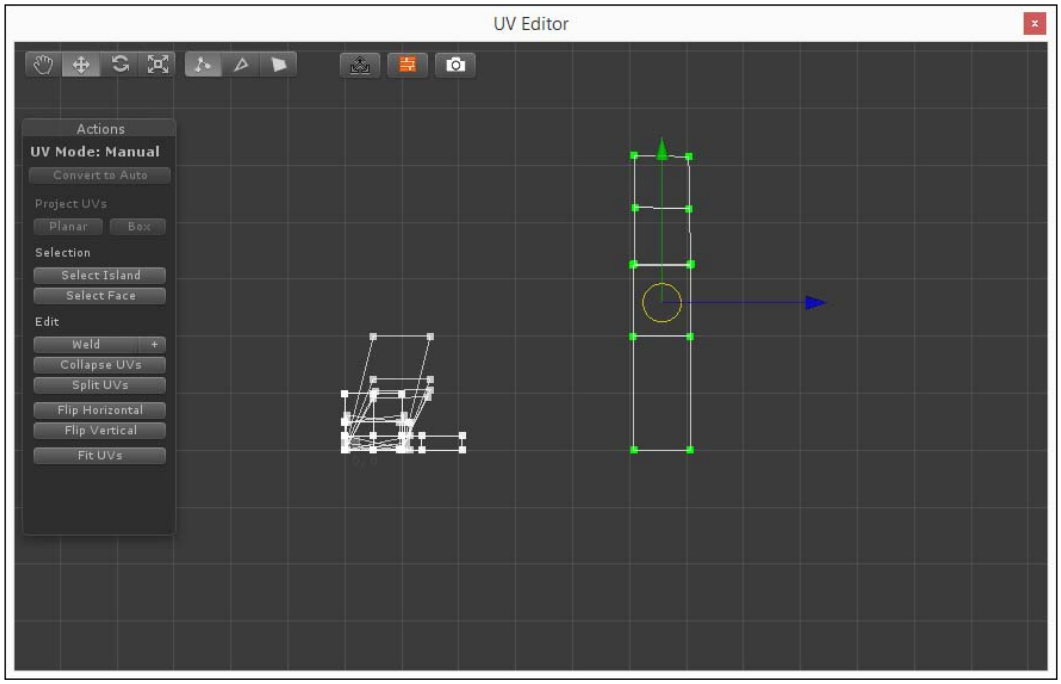


You'll see that the objects are now connected to each other, but still look incorrect in the **Scene** view. What we want to do is have the UVs curve with our brushes.

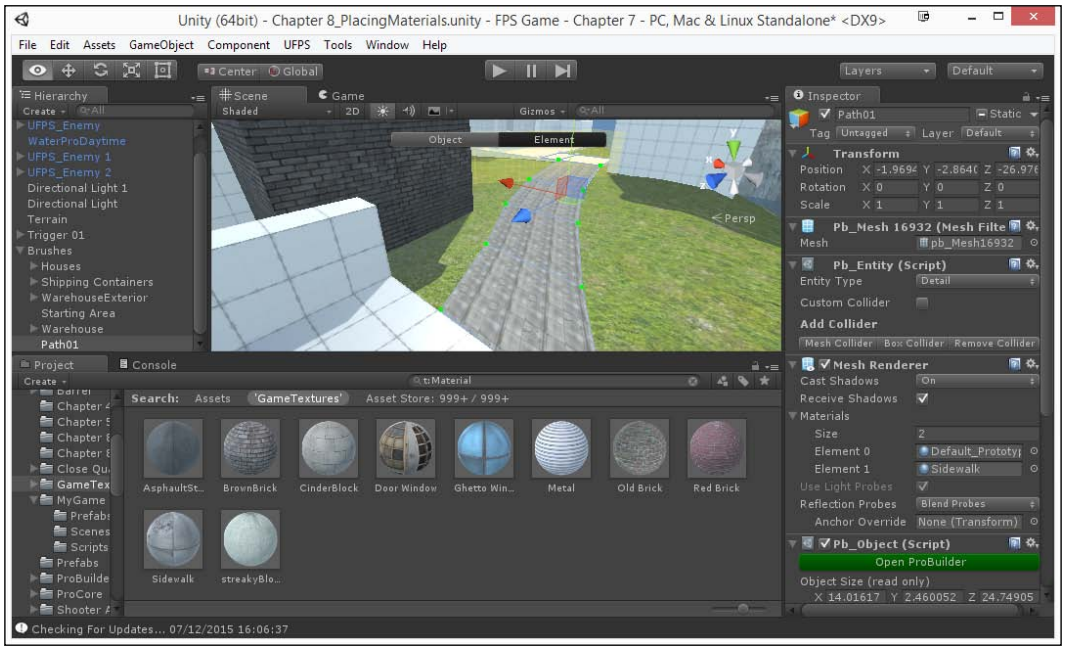
20. Still in the **UV Editor**, click on the three dots button to switch to **Vertex** mode and using the **Translation** tool move the vertices to the right away from the other pieces so we don't make any mistakes. Then select the two vertices above the bottom one and move them so that they are parallel to the ones below it.



21. Then continue to the next edge and move to the side as well. As you do this, the top two pieces will break apart. Just select them again and move them up as well. At the end it should look like this:

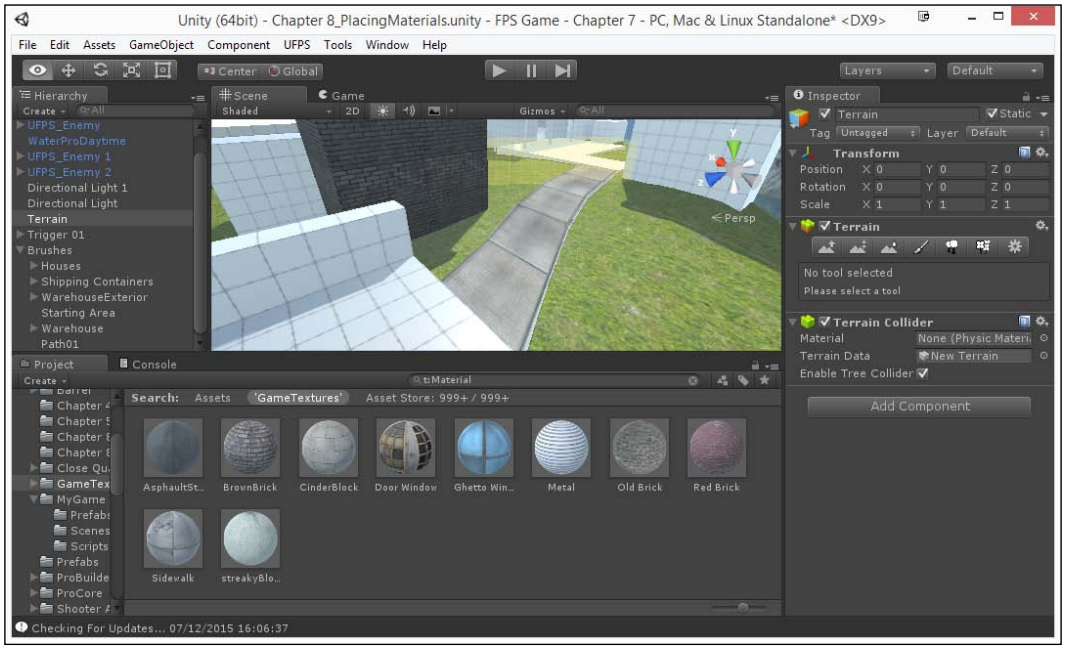


And inside the editor it should look like this:

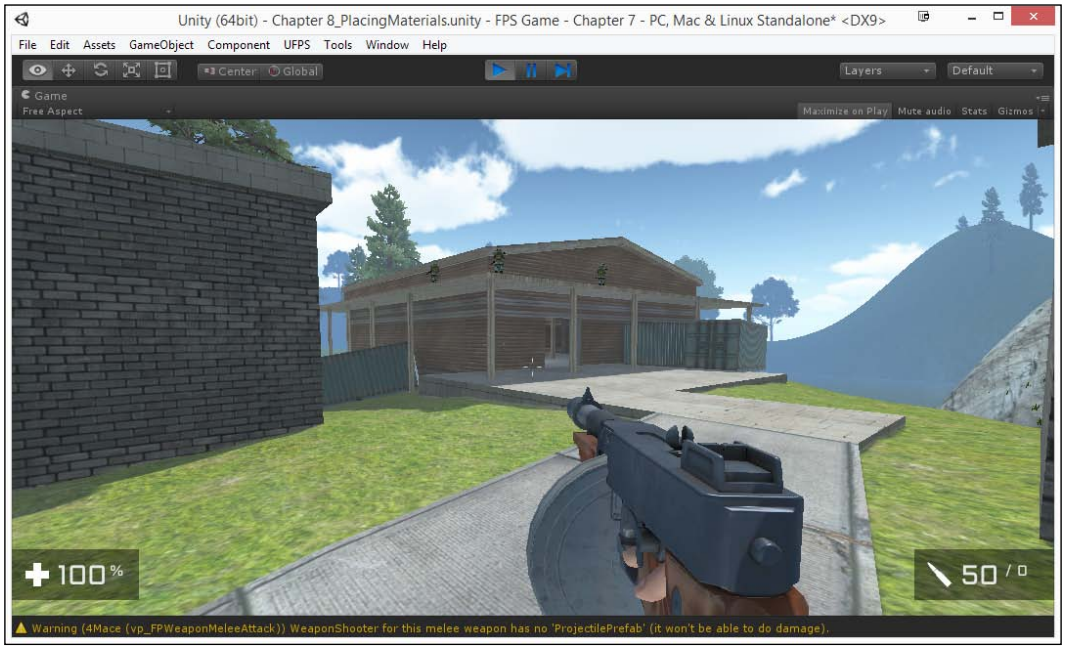


As you can see, the path is going in the correct direction! Now, the material this is based on is four different blocks in two parts, one half normal and the other half with some destruction. For now, we will scale the material up so that it will only show the clean part.

22. Use the **UV Editor** and scale the objects down to fit only one part of the sidewalk; translate them to fit snugly. When you're done, it should look like this:



Now that we have that experience, we can go ahead and texture the remaining parts of the level. When you're finished and you turn **Continuous Baking** back on, your level should look something like this:



We're now well on our way to having a full and polished level!

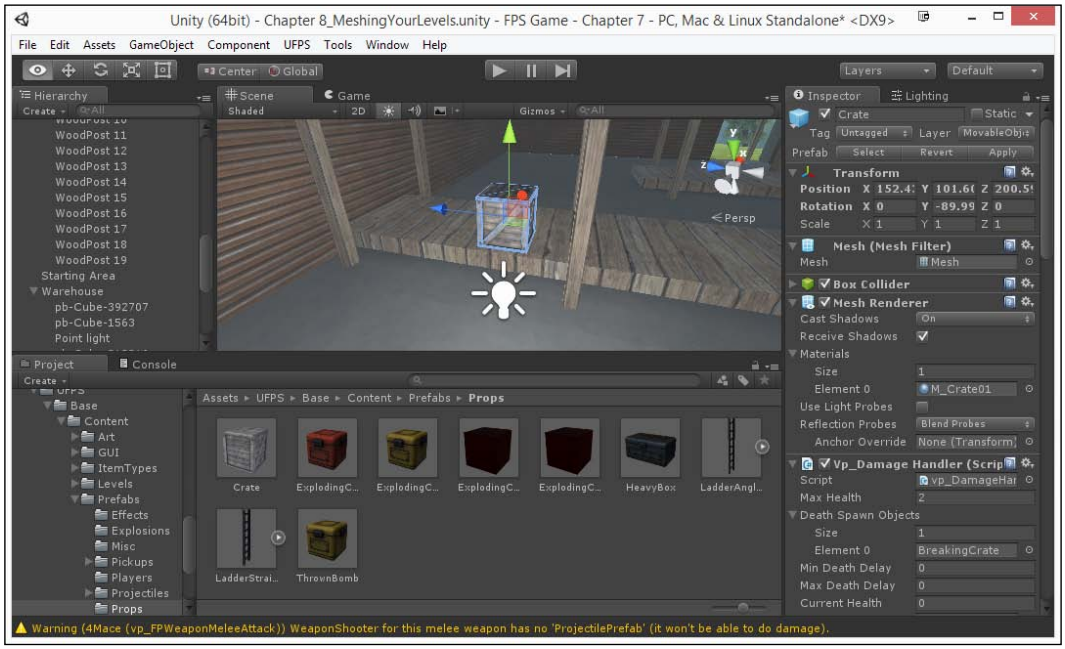


For a more in-depth video showing all aspects of the UV Editing and Unwrapping tools, check out: `https://www.youtube.com/watch?v=U_5f8RlciWQ`.

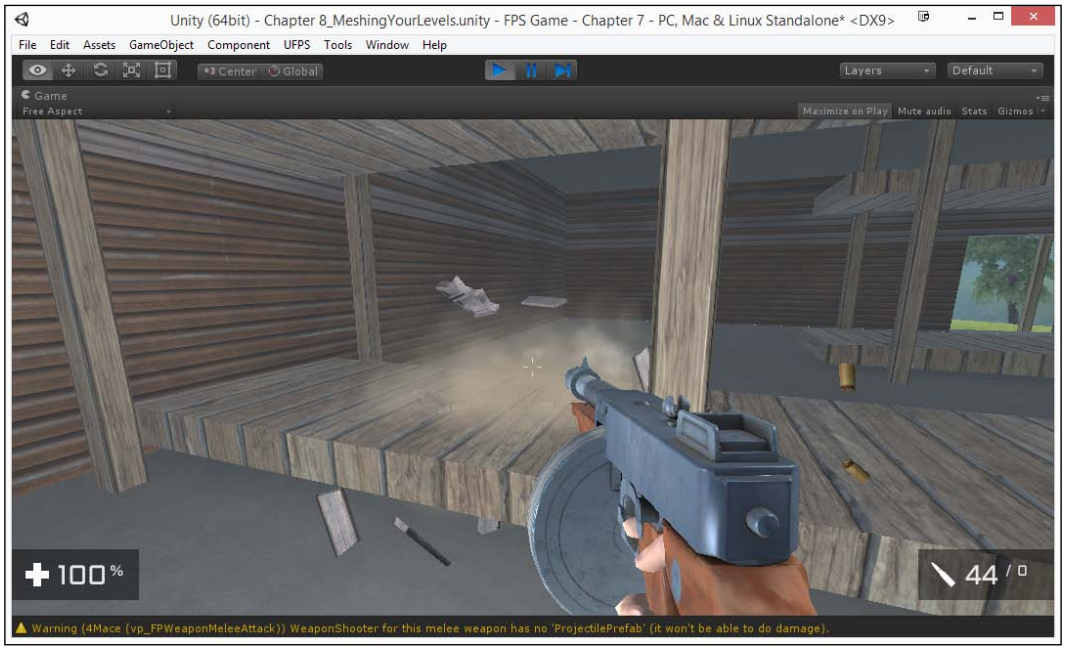
Meshing your levels

In the game industry, there is often a role known as a **mesher**; this is a person whose role is to fill levels with meshes, or additional details, to make the level even nicer. Generally, these additions do not affect the gameplay, or are not meant to. At times, designers and meshers will argue about whether certain things should be added to the scene due to realism or gameplay. However, since you're doing it all, there's no issue this time!

1. From the **Project** tab, open up the `UFPS/Base/Content/Prefabs/Props` folder. This folder contains a large number of objects that we can play with in our level.
2. Move your camera into the warehouse of the scene and drag-and-drop a `Crate` prefab object on top of one of the shelves.



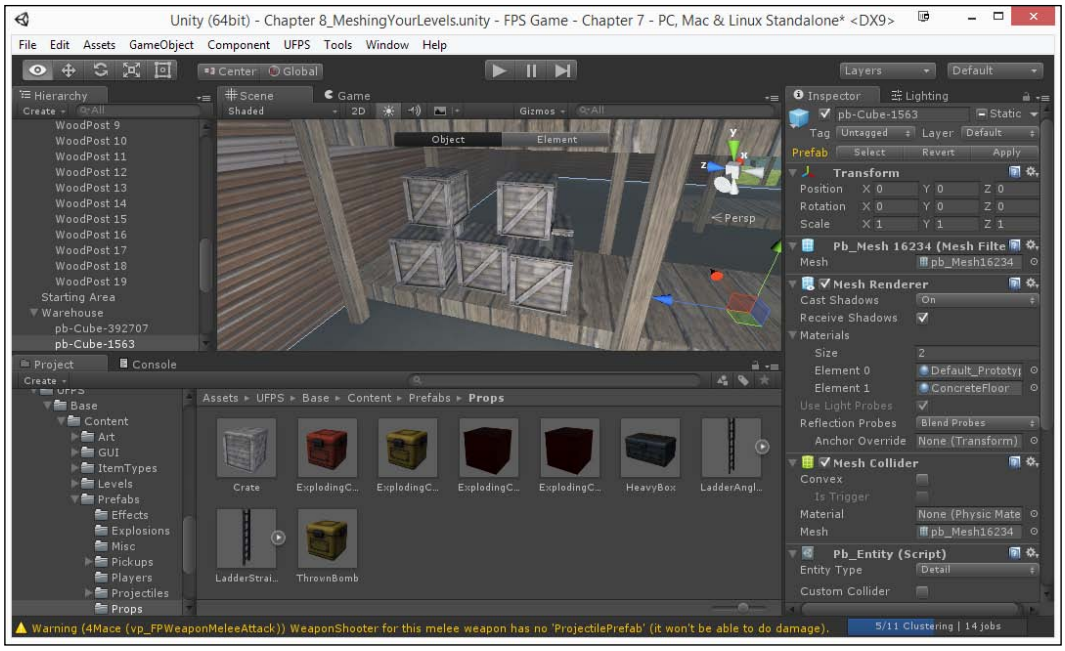
If you play the game now and go to the crate, you'll find that it's already created in such a way that, if you shoot it, the crate will be destroyed!



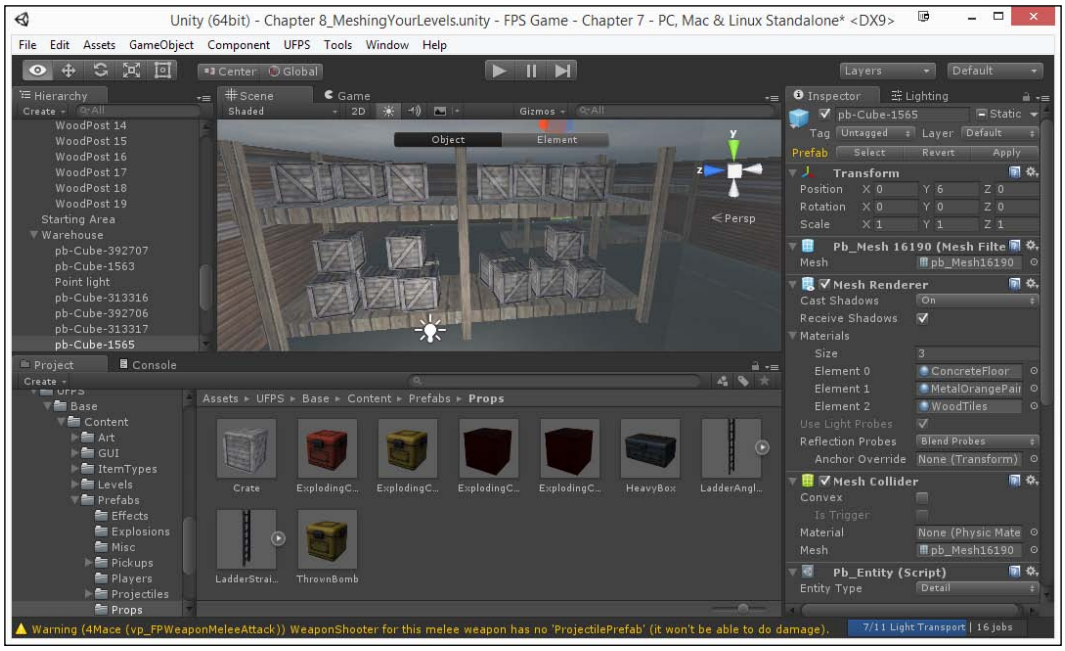


The object can also be picked up and moved—perfect for use in simple puzzles.

3. Now duplicate this box, doing small, repetitive changes in rotation every time to break up the similarities between the objects.

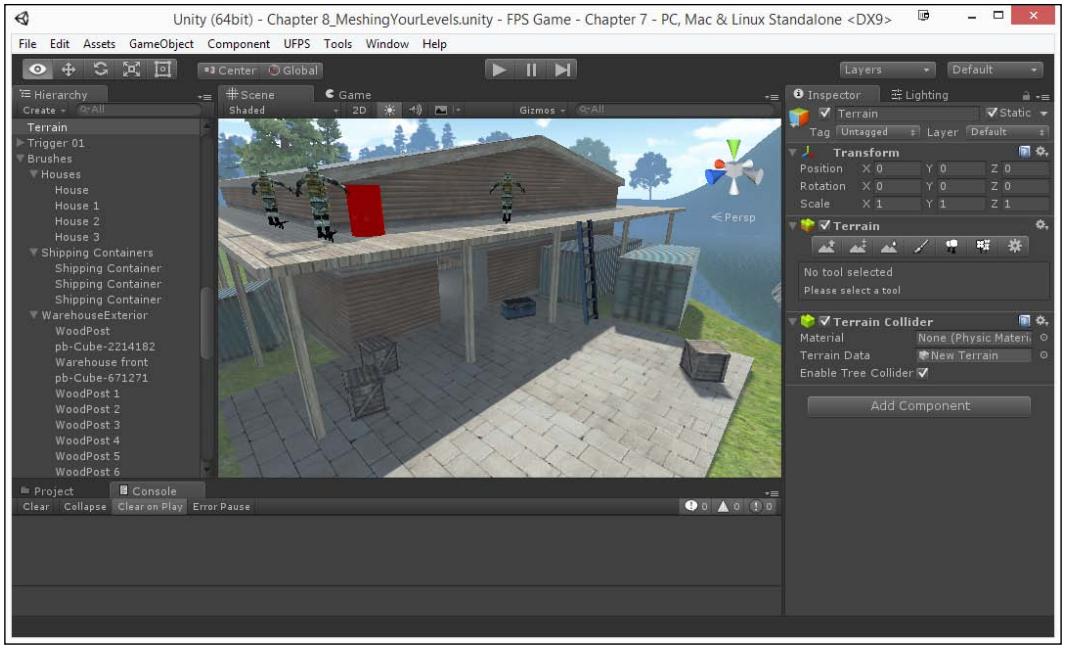


Once you've created a small group, you can select all of those objects and then duplicate them, continuing to make small changes!



As you can see, it already looks much more like a lived-in environment.

4. We can also add in the `Ladder` prefab, to allow our players to move up to the top of the warehouse, and the `Barrel` prefab we made from the last level as well in order to add more cool features to the level!



5. Save your level and start up the game!



As you can see, our level is looking great! Take the time to play around with other models, through the project, from the Asset Store, or of your own creation if you're inclined to make your level unique!

Summary

At this point, we now have our levels in place and ready to publish! Specifically, we learned about upgrading from Prototype to ProBuilder, creating materials, working with ProBuilder to place materials and UVs, and meshing your levels

With that in mind, in the next chapter we will learn how to customize the GUI of our game and add a main menu for us to start and exit our project!

8
Creating a Custom GUI

Now that we've completed all of our levels, it's time for us to customize the **Graphical User Interface** (**GUI** for short).

A Graphical User Interface or GUI is the way players interact with your games. You've actually been using a GUI in the previous chapter, and also when you interact with your operating system of choice. Without a GUI of some sort, you wouldn't be able to make your electronics do anything (apart from by using a command prompt, as in DOS and UNIX).

When working on GUIs we want them to be as intuitive as possible and only contain the information that is pertinent to the player at any given time. There are people whose entire job is programming and/or designing user interfaces and colleges provide degrees on the subject. Thus, while we won't talk about everything having to do with working on GUIs, I do want to touch on aspects that should be quite helpful when working on your own projects in the future.

Here is the outline of our tasks:

- Creating a main menu: part 1 – adding text
- Creating a main menu: part 2 – adding buttons
- Creating a main menu: part 3 – button functionality
- Replacing the default UFPS HUD

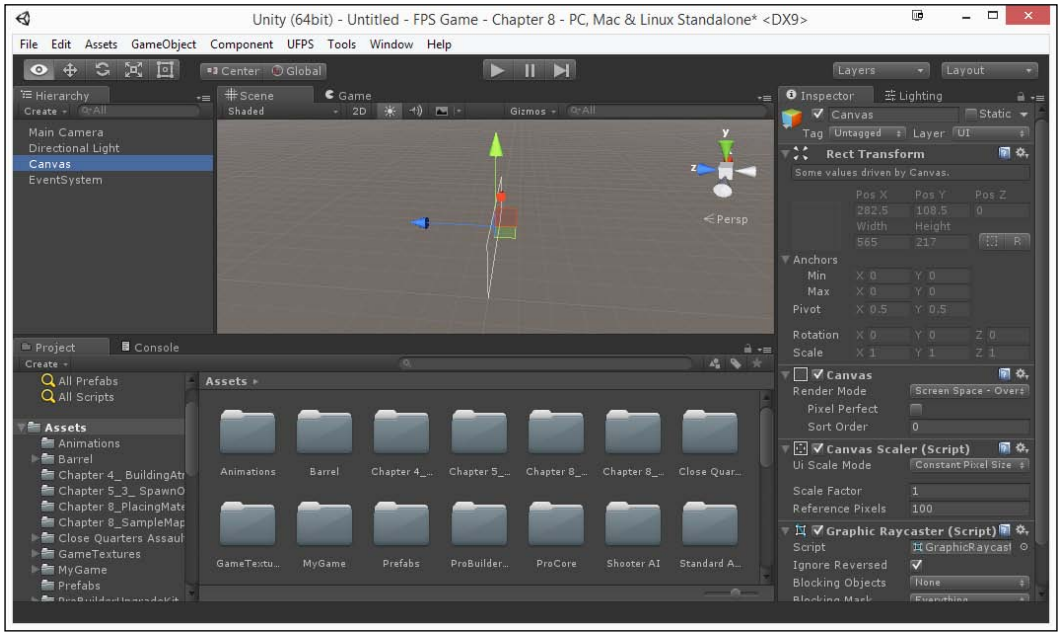
Prerequisites

Before we start, we will need to have created a project that already has *UFPS* and *Prototype* installed with all of the levels that you want to have inside your game project. If you do not have that already, follow the steps described in all of the previous chapters.

Creating a main menu: part 1 – adding text

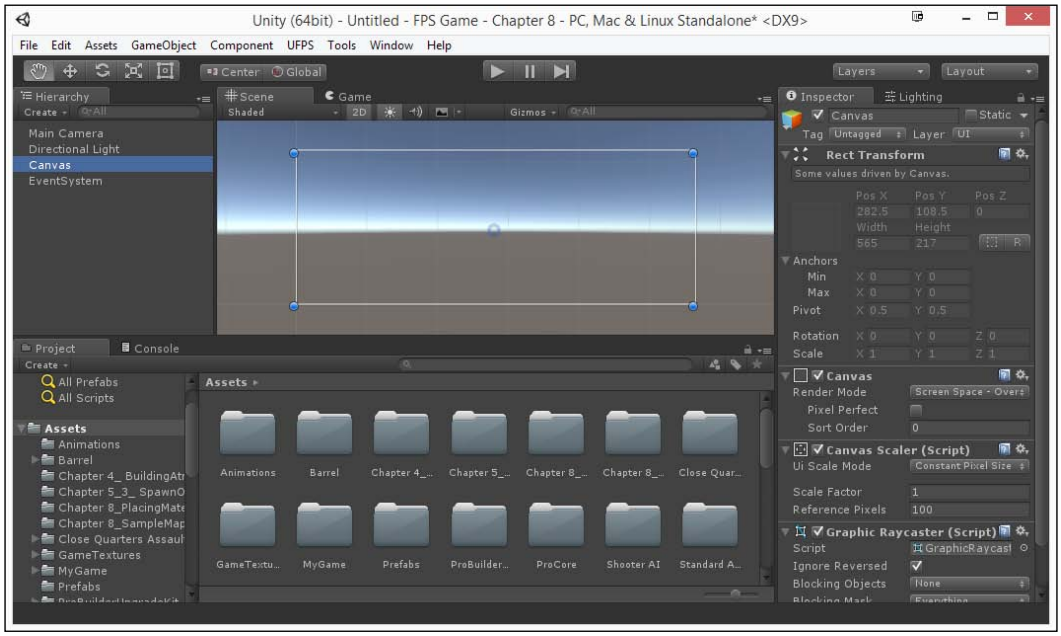
Games don't just launch straight into the gameplay. Generally they start off with some kind of Title Screen with the game's name and the option for you to play or quit. This will be a good way to get some familiarity with how the new Unity GUI system (as of Unity 4.6) works with some simple UI elements. To start off, let's add some descriptive text that will display our game's name.

1. To start off we will need to create a new level for our main menu to exist in, so to do that go to **File** | **New Scene**.
2. Now we could add objects into this level, but for our purposes let's make it blank. From the **Hierarchy** tab, select the `Main Camera` object, under **Clear Flags** select `Solid Color`, and then under **Background** change the color to black.
3. In order to create a GUI we need to have something called a Canvas in our scene, so to do this, we need to go to **GameObject** | **UI** | **Canvas**.



4. It may be hard to see the object at this point, so let's go into 2D mode. We can do that by clicking on the **2D** button below the **Scene** tab.

You'll notice that now when we double-click it becomes a lot easier to see what's there because we're viewing the scene in an orthographic way, or straight-on like a 2D game would be. Of course, 3D interfaces are also popular, but in that case you'll be moving around just like a normal object in 3D space.



The **Canvas** acts as a space in which we can place different UI elements in the scene. It can either be drawn on top of the game like we will be doing here, or it can actually be placed inside the game world, leading to cool effects.



For more information on the canvas, please visit `http://docs.unity3d.com/Manual/UICanvas.html`

You'll also notice that another object was created, **EventSystem**, which will allow us to interact with the UI we'll learn about later.

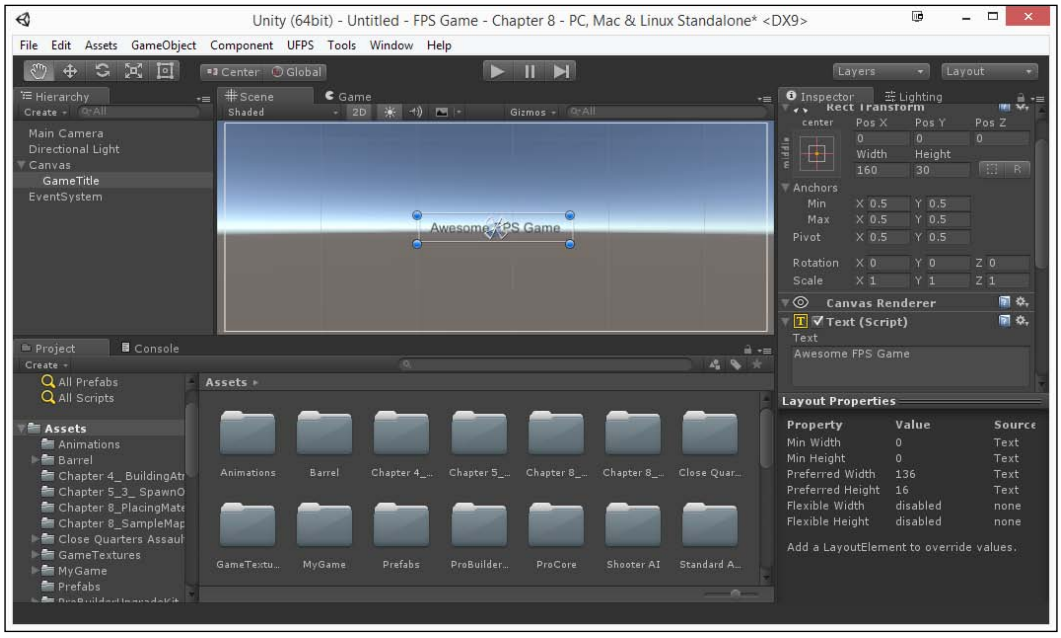
It's important to note that UI elements will only show up in the **Scene** if they are a child of a **Canvas** object.

5. Next let's add some text for our game. With our Canvas object selected from the **Hierarchy** tab, from the top toolbar click on **GameObject** | **UI** | **Text**.

6. From the **Hierarchy** tab, select the newly created `Text` object and then, from the **Inspector** tab, change its name to `GameTitle`. Under the **Rect Transform** component change the **Pos X** and **Pos Y** values to `0` by either typing them in or right-clicking on the **Rect Transform** component and then selecting **Reset Component**.

   You may have noticed that this object doesn't have a Transform component in the Inspector. In its place we have **Rect Transform**. This transform has a number of new properties designed for working with UIs, such as pivots to dictate which directions the UI can scale from as well as Anchors to hold the object at a certain place within the scene. This is very useful in terms of creating UIs because we want to be able to support many different resolutions and have it work correctly. We'll discuss these properties in more detail as they come up in this chapter.

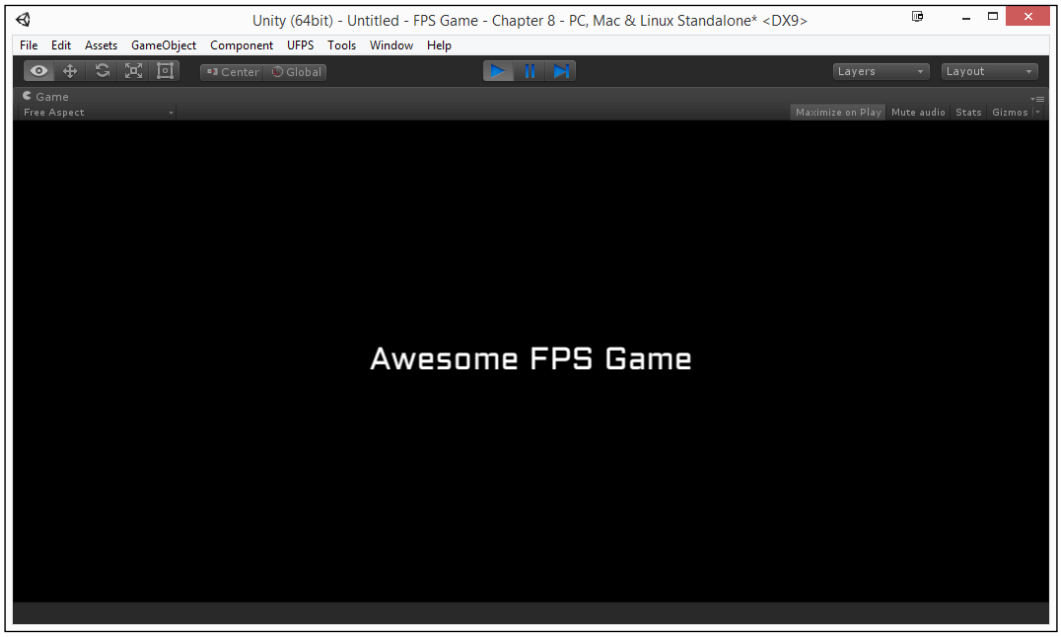
7. Under the **Text** component, change the **Text** property to your game's name (I used `Awesome FPS Game`) and in the **Paragraph** section change the Alignment to **Center Align** and **Middle Align** (the center of the choices).



8. We're making progress, but now let's make the text larger. Change the **Font Size** to `32` and you'll notice it disappears; that's because our width and height are too small for this element. Back in the **Rect Transform** component, change the **Width** to `350` and the **Height** to `60`.

9. Our background is currently black so let's change the text's color as well. Back in the **Text** component change the **Color** property to white.
10. We can also change the font to whatever we'd like as well. UFPS comes with a font, so let's just use it. Click on the circle icon to the right of the **Font** property and select `Aldrich-Regular` from the list that pops up.

    You can also drag-and-drop any font file you have on your computer into the Unity project and use it as well!



Okay, we're starting to make headway. Let's move it up as well.

11. Back in the **Rect Transform** component, change the **Anchors** section's **Min** and **Max Y** values to `.75`.

    Notice that the **Pos Y** value goes from `0` to a negative number. This is saying that the text is a certain number of pixels away from being 75% up to the top of the screen.

    Anchors are convenient because, when working with larger or smaller resolutions, objects will stay in the same relative positions in space, even if we change the aspect ratio. Depending on what the **Anchors** variables are set to, **Rect Transform** will show different variables to allow you to position your object.

If you set **Anchors** to a single point, without stretching, you'll see the **Pos X**, **Pos Y**, **Width** and **Height** Properties, like we have currently.

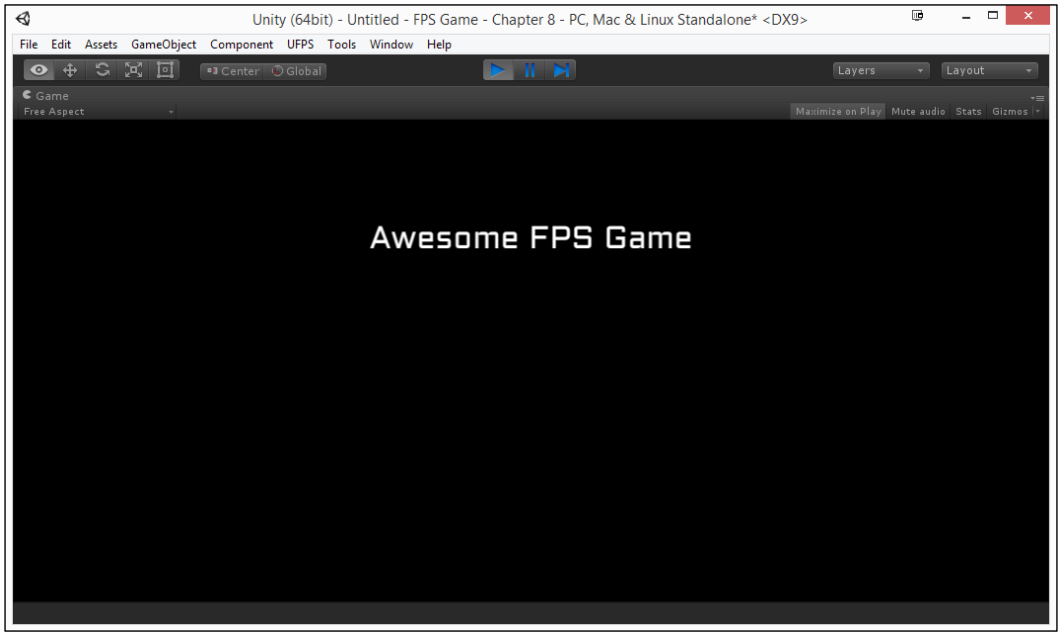


In code, the **Pos X** and **Pos Y** values are stored in a variable known as `anchoredPosition`, which may be a more appropriate name for them.

However, if you set Anchors in a way that stretches your UI Element, you'll get **Left** and **Right** and/or **Top** and **Bottom**. To set your anchors easily, there are a number of presets which you can access from the button in the top left of the **Rect Transform** component. In addition to that, if you click on one of the buttons while holding *Alt* it will automatically move the object selected as well.

Do note that, when you change the value, it doesn't actually move the object as it adjusts the position to put it in the same place; with that in mind, you'll still need to set the position.

12. Change the **Pos Y** value to `0` and you'll see that now the value is higher!



And with that we now have a clean logo for our project!



For more information about the Text component, please visit `http://docs.unity3d.com/Manual/script-Text.html`

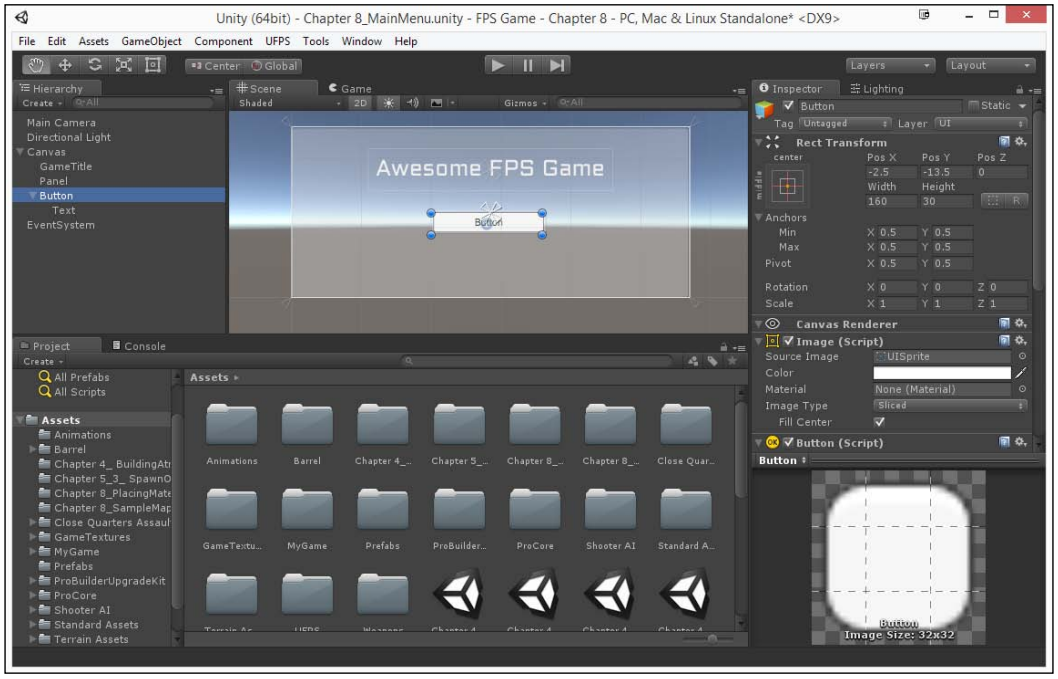
Creating a main menu: part 2 – adding buttons

We have now created the simplest type of UI there is. Let's start adding in some interactive objects with buttons that will allow us to start or close the game while also learning about Layout Groups.

1. To start off, we will need to create a holder for the buttons we want to create to make it easy for us to add additional options to our HUD with minimal fuss. To do that we will select our `Canvas` object from the **Hierarchy** tab and then create a **Panel** by going to the top toolbar and selecting **GameObject | UI | Panel**.

   The panel object is simply an object with **Rect Transform** and an Image component attached that is set to take up the entire screen.

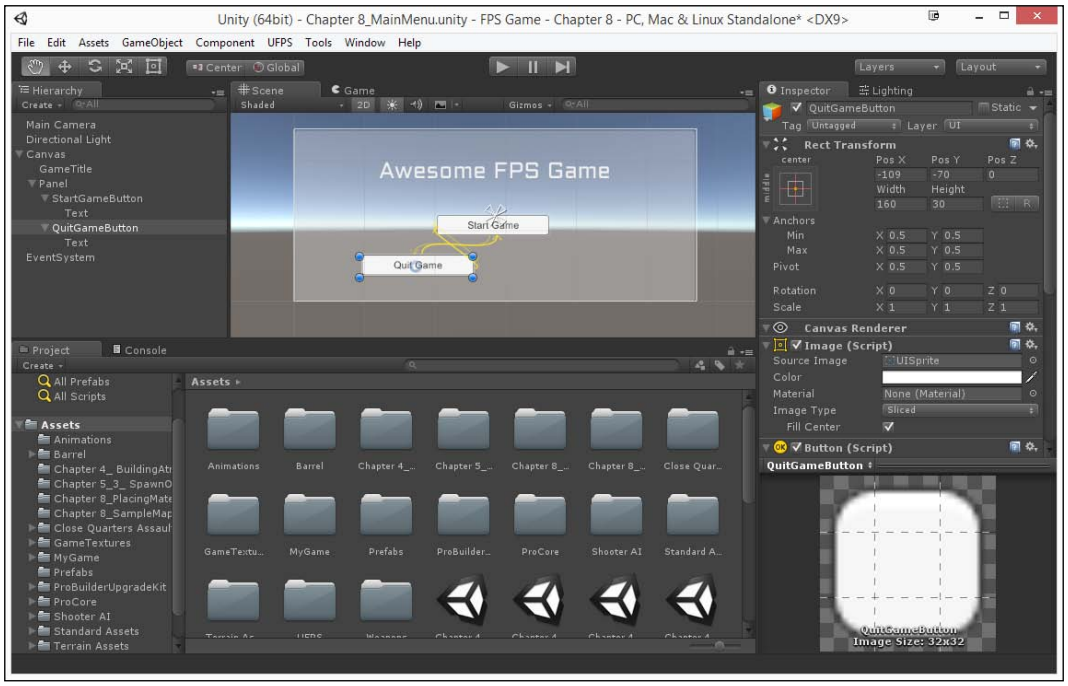
2. Next let's create a button object. From our canvas object, select **GameObject | UI | Button**.



The button that we just created actually consists of two objects: the button itself, which has the background image, and a button script that dictates how it animates and interacts. It also has a child text object that has the text that's being displayed.

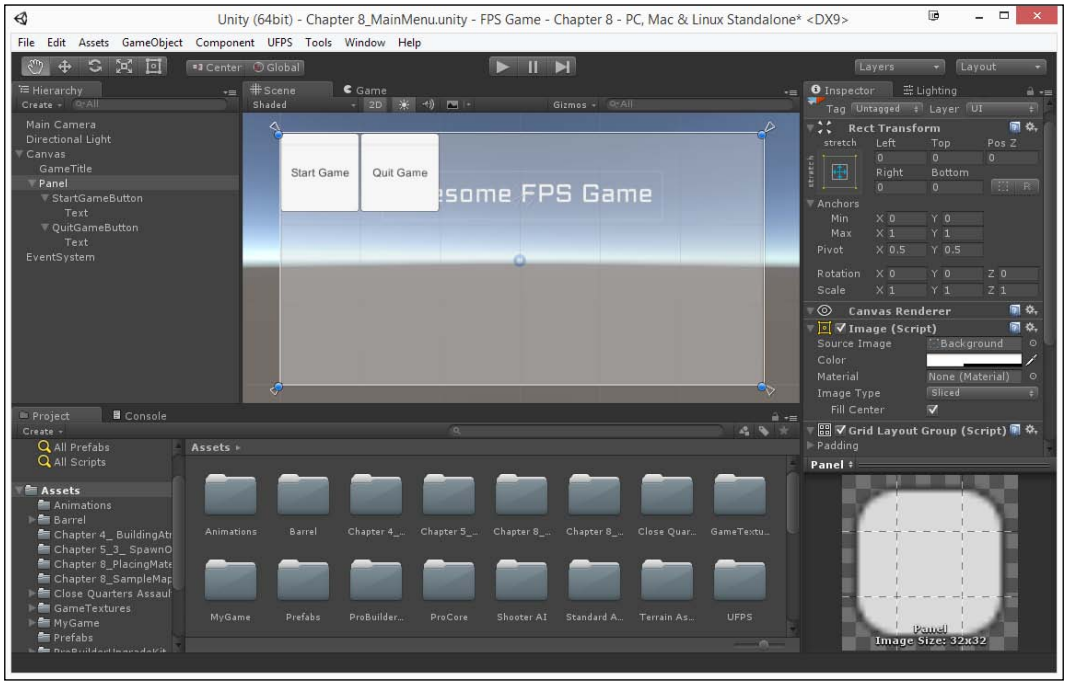
The text is displayed on top of the button because the Canvas renders objects from the top down; thus, the Button is drawn first, then the Text on top of it. You can change the order by dragging objects as needed

3. Select the `Text` object and in the **Text** component set the **Text** variable to `Start Game`. Rename the `Button` object to `StartGameButton`.
4. After that, let's add this button as a child of the panel by dragging and dropping buttons on top of the `Panel` object.
5. Now let's add a second button to the screen by selecting the button and pressing *Ctrl* + *D*. Rename this newly created object to `QuitGameButton` and drag it away from the original object. Under its `Text` object change the **Text** component's **Text** variable to **Quit Game**.



Now we could hand-place these buttons to fit on our screen nicely but, rather than do that, let's explore how we can use **Layout Groups** to make our job easier.

6. From the **Hierarchy** tab, select the `Panel` object and then from the **Inspector** go all the way down and select **Add Component**. From there type in `Grid` and select **Grid Layout Group**.



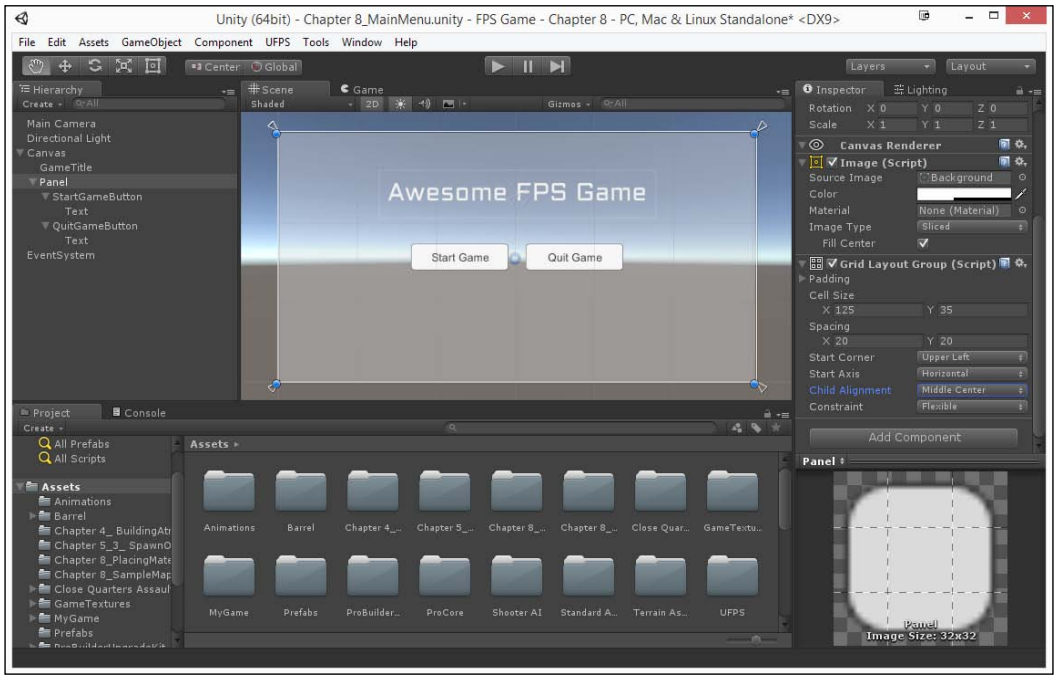
As you can see, now that we've added a **Layout Group** to our parent object, all of the children now have their **Rect Transform** component values set by this grid.



For more info on different kinds of Layouts, check out `http://docs.unity3d.com/Manual/comp-UIAutoLayout.html`.

7. Let's make this look a little nicer. In the **Inspector** tab, go to the **Grid Layout Group** component and change the **Cell Size** to (`125, 35`); you'll notice that the elements are changing at the same time.
8. Then set the **Spacing** in the **X** and **Y** fields to `20` and you'll notice that the objects are moving away from each other.

Next change **Child Alignment** to **Middle Center** to center the objects.



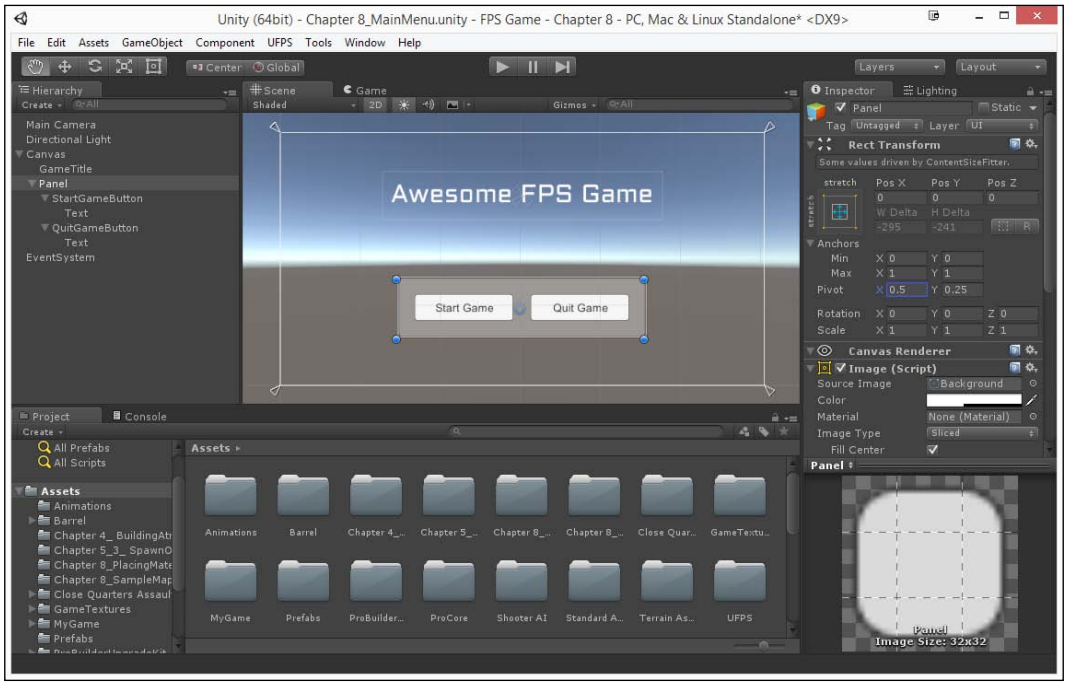
9. We don't want the Panel to fill the screen, but it may be nice as a holder for all of the buttons in our menu. Select the `Panel` object, click the **Add Component** button, and add a **Content Size Fitter** component to the object. Next, inside our newly created component, change the **Horizontal Fit** to **Preferred Size** and **Vertical Fit** to **Min Size**.

   You'll see that now the panel has been resized to fit the contents of its children, and no more. This is great because, if we create a number of other buttons, the object will resize as necessary, making it easy to expand. However, I think it looks a little strange being so tightly bound, so let's add some padding to the sides.

10. Next, in the **Grid Layout Group** expand the **Padding** variable and set the **Left**, **Right**, **Top**, and **Bottom** values to `20`.

    And with that, we're almost there but it looks a little awkward in the center, so I'm going to lower it down just a little bit.

11. In the Panel's **Rect Transform** component, change **Y** under **Pivot** to `.25` and set **Pos Y** to `0`.



We now have the visuals for the buttons created, and have a good understanding as to how the buttons are placed.

Creating a main menu: part 3 – button functionality

Now that we have the buttons in the scene, let's make it such that, when we click on them, they will both start and quit the games, respectively.

Our buttons need to have some functionality on clicking them, but we need to provide a function for them to call to go to another level. Let's create a new object with a new component that we will use to do this.

1. Go to **GameObject** | **Create Empty**. Rename our newly created object to `MainMenuEvents`.
2. From the **Project** tab, open up the `MyGame/Scripts` folder and then select **Create** | **C# Script** and call it `MainMenuEvents`.

3. Double-click on the script to bring it into MonoDevelop. Once opened, replace what is there with the following code:

```
using UnityEngine;
using System.Collections;

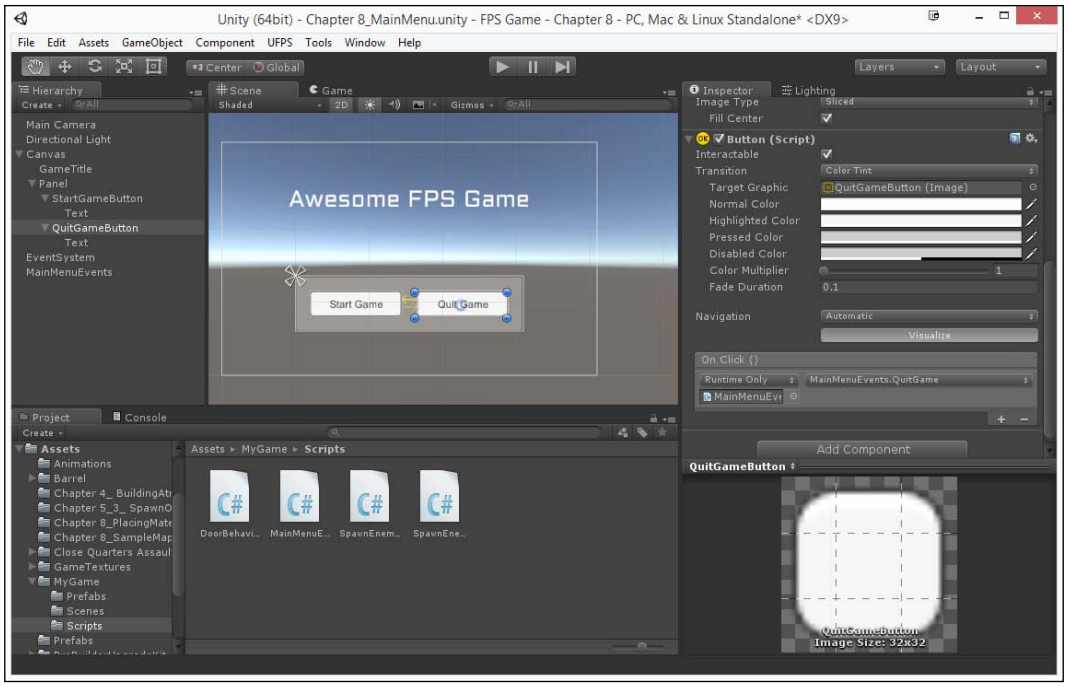
public class MainMenuEvents : MonoBehaviour
{

  public void LoadGameLevel(string level)
  {
    Application.LoadLevel(level);
  }

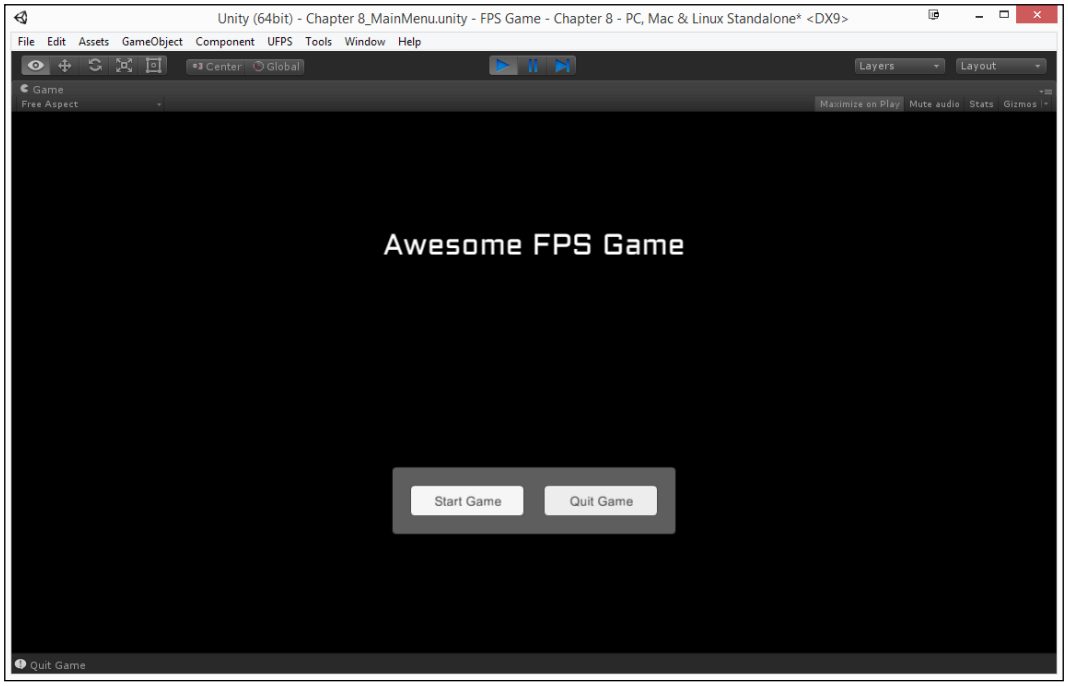
  public void QuitGame()
  {
    print ("Quit Game");
    Application.Quit();
  }
}
```

   This class contains two functions for us to work with: `QuitGame`, which will have the application quit once it's called, and `LoadGameLevel`, which will take in a string and open up the level with that name (if it's part of the build).

4. Attach the **MainMenuEvents** component to the `MainMenuEvents` game object.
5. Now let's add this to our buttons. Select the `QuitGameButton` and from the **Inspector** tab move down to the bottom of the **Button** component. From there click the **+** button at the bottom of the **On Click ()** list.
6. Drag the `MainMenuEvents` object from the **Hierarchy** tab to the slot to the left of the newly added item in the list.
7. Then, from the drop-down that currently says `No Function`, click and select **Main Menu Events** | **Quit Game**.



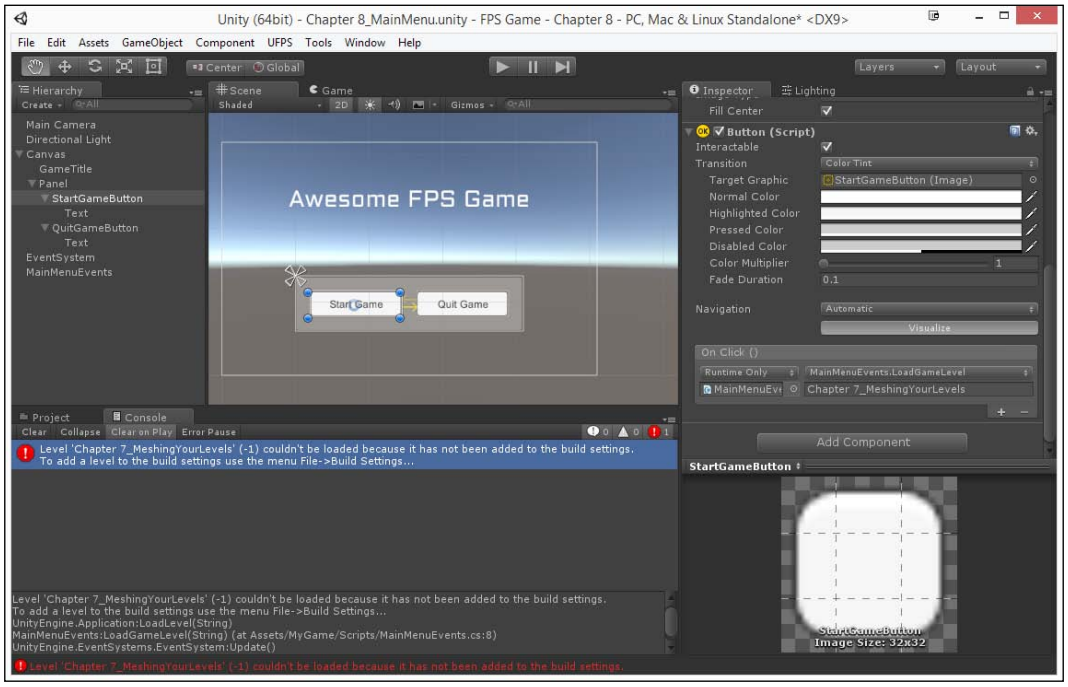
8. Save the scene and play the game!



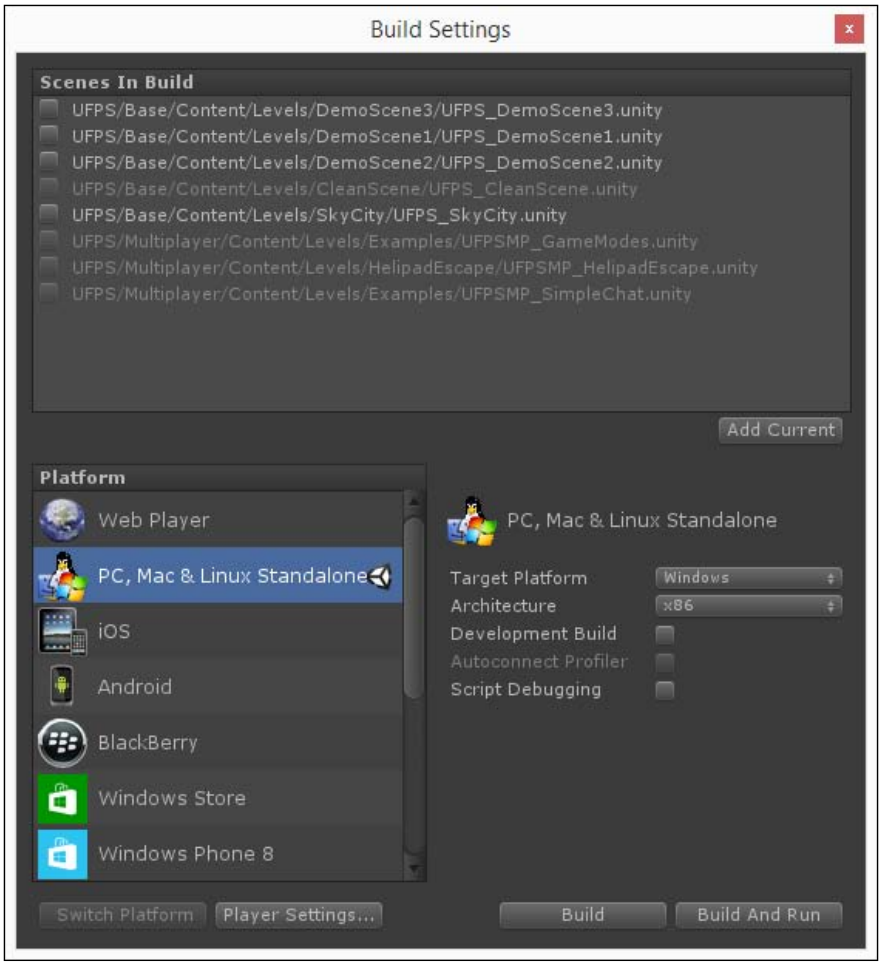
If you'll notice now, when we click the **Quit Game** button, the words Quit Game are displayed on the bottom left where **Console** outputs are displayed! As `Application.Quit` doesn't do anything in the editor, this shows us that the function is being called correctly.

9. Now let's do the same thing for the **Start Game** button. Select the object and then click on the **+** below the **OnClick ()** list.
10. Once created, drag-and-drop the `MainMenuEvents` object in and then, from the drop-down, select **MainMenuEvents | LoadGameLevel**. You'll notice that unlike the last time there's now a new slot placed below the function's name. This is where we will put the game level we want to load (in this case I am going to put `Chapter 7_MeshingYourLevels` because that's the name of my final created game level).

    However if you were to play the game, you would notice that nothing happens when we click the **Start Game** button; or rather we'd see the Console giving us an error.



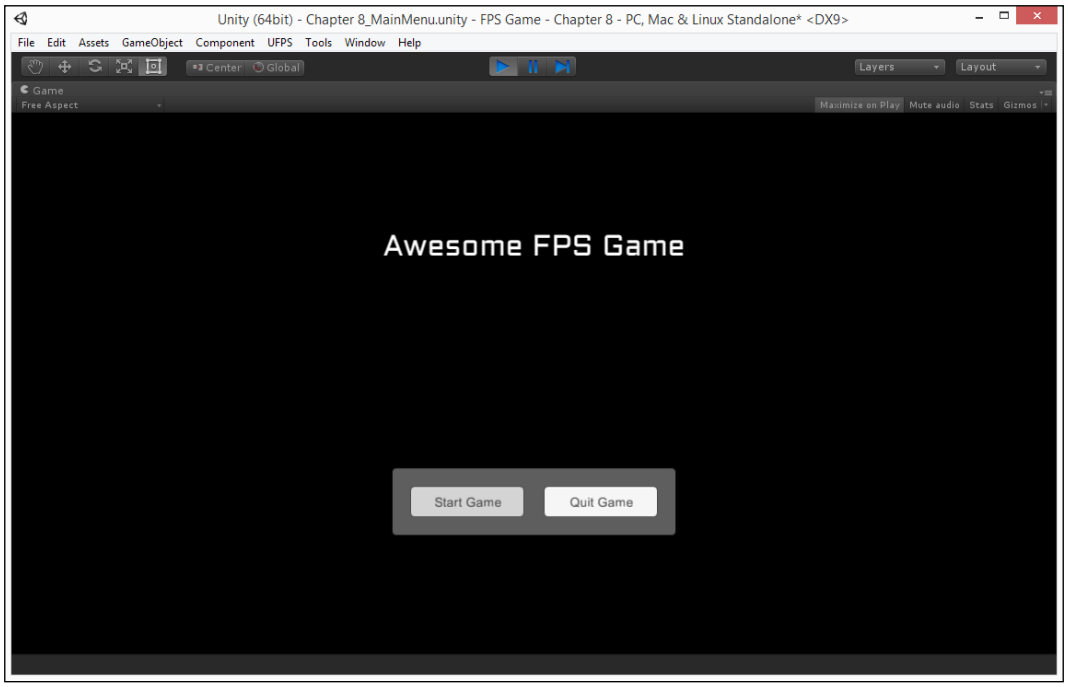
11. Go into **File | Build Settings**. You'll notice there are already a number of levels included, and this is because of defaults in UFPS.



12. Select all of the levels currently in the **Scenes in Build** property and delete them by pressing the *Del* key. Next add both the Main Menu and the Game levels with the Main Menu being at the top (the top is the level that will start if you export the game). If it is not there, feel free to drag it up there via the mouse.

    You can add levels by either pressing **Add Current** after loading each level or by dragging-and-dropping the levels from the **Project** tab.

13. Once finished, save your work and start the game again!



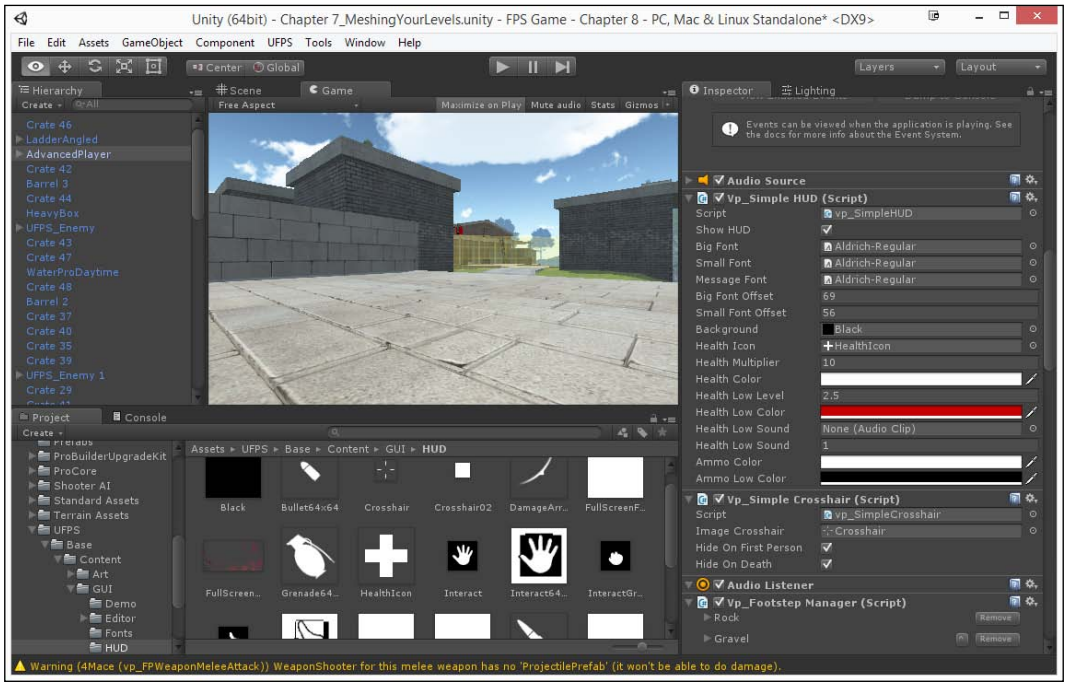
At this point, whenever we click on the **Start Game** button it will load our level correctly!

Replacing the default UFPS HUD

We now have a Main menu, but our regular game level is showing a default UI. While it's fine for testing purposes, we may want to make something a little more custom for our finished product.

1. Start off by opening up any of the levels you've created that have the `AdvancedPlayer` prefab inside it (I chose `Chapter7_MeshingYourLevel`).

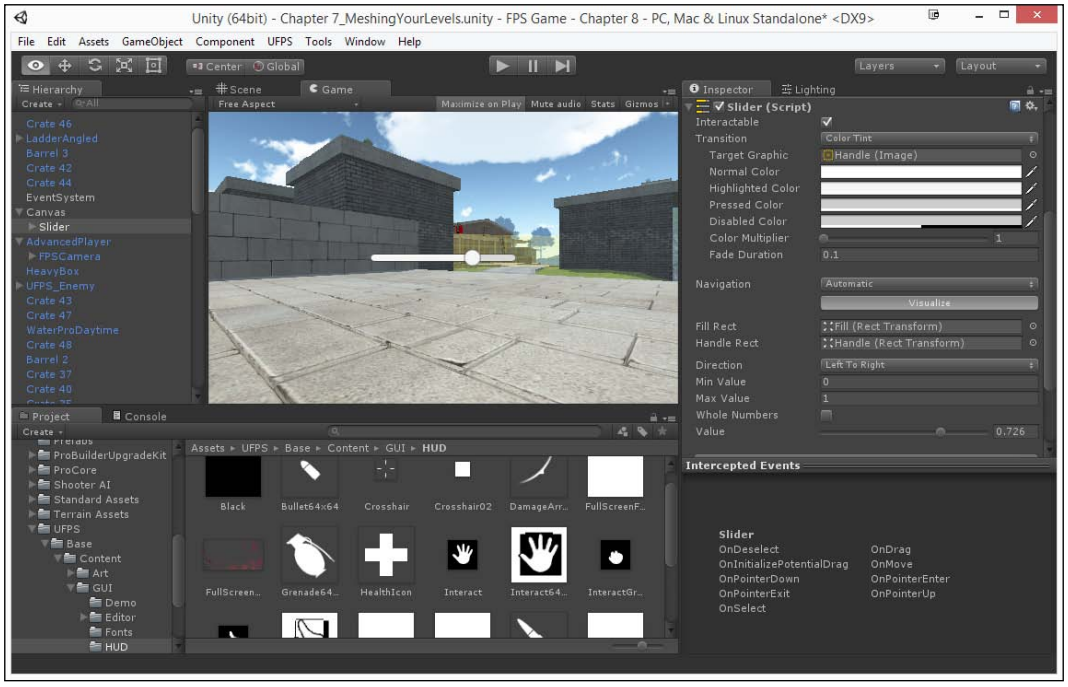
2. From there select your `AdvancedPlayer` and look at it from the Inspector. One of the components you'll see is the `vp_SimpleHUD` component.



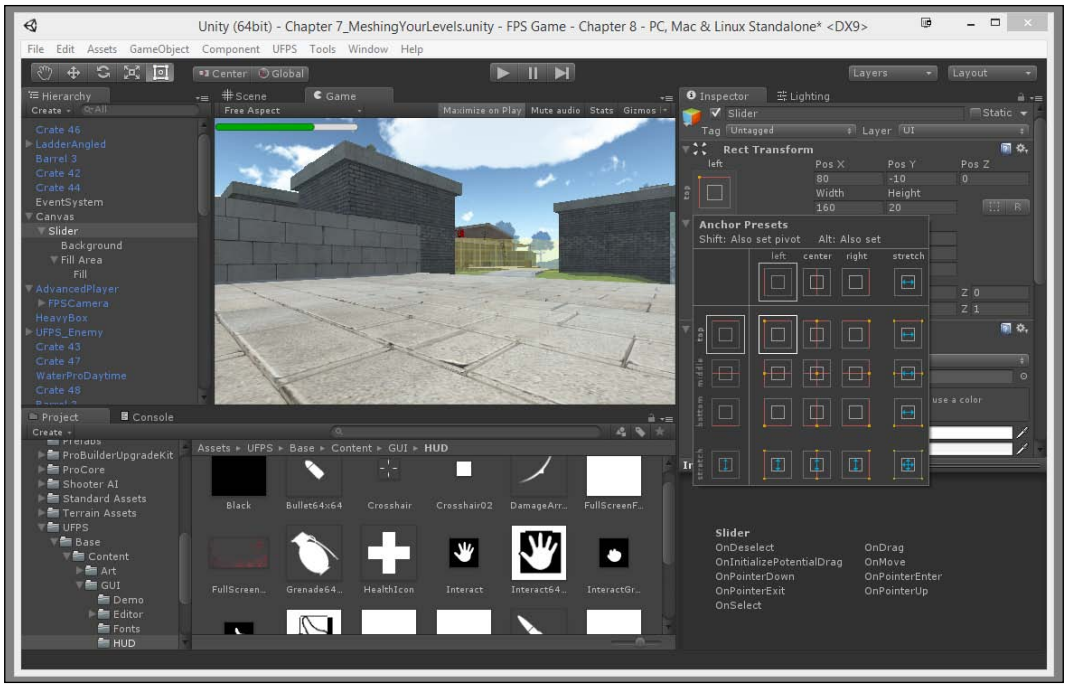
This is what draws the screen in the default way and can be a good reference when it comes to accessing variables. However, we do not want to use it in this case.

3. Right-click on the **vp_SimpleHUD** component and select **Remove Component**.
4. Now that we've removed the previous HUD, it's time to add in our own. Create a Canvas by selecting **GameObject** | **UI** | **Canvas**.
5. For our purposes we're going to be using **Slider** as the basis of our lifebar. To do that with the Canvas selected, go to **GameObject** | **UI** | **Slider**.

A slider is a graphic with a handle that the user can drag to change a value between a min and max value. In this instance we are using a slider for display purposes rather than the interaction that it offers but let's first see what it does. If you go into the Inspector and scroll down to the **Slider** component you'll see a value called **Value**; when you change it you'll see the slider move as well.



6. Next let's customize how this looks so it fits the roll of a healthbar better. Expand the Slider object in the Hierarchy to see the **Fill Area** and **Handle Slide Area** objects. Delete **Handle Slide Area** as we will not be needing it.
7. After that, from the Hierarchy tab inside the Slider object expand the **Fill Area** object to see its children and from there select the **Fill** object. From there set the **Color** on the **Image** component to green to make it look more like a health bar.
8. Next let's position this to the top left of the screen. Select the `Slider` object and then click on the anchor helper button, hold down the *Alt* key, and then press the top left option.



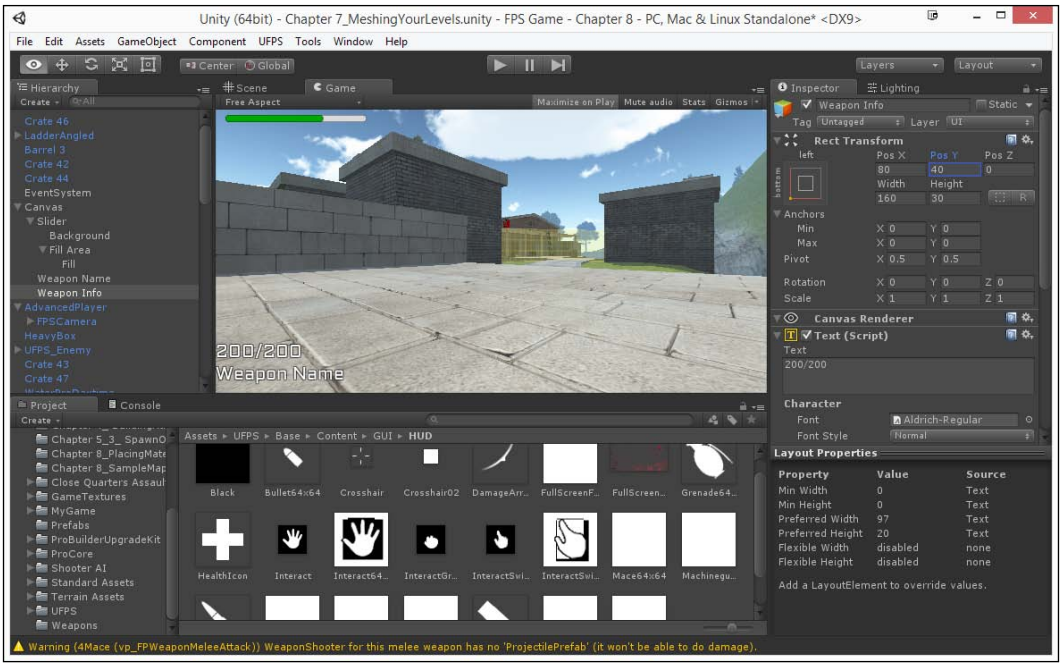
When you hold down the *Alt* key and click on the button, you're also setting the position of the object in addition to setting the anchor.

9. You'll notice now that it's placed in the correct position, but **Pos X** is `80` when we'd rather it be `0`; so we need to change **X** in the **Pivot** section to `0` and **Y** to `1`. Then set **Pos X** to `10` to give some spacing from the edge of the screen.

   Next, we need to add and display some information about our weapon: its name and the ammo we have.

10. Go to **GameObject** | **UI** | **Text**. This time put the object in the bottom left corner.
11. Change the **Text** inside the object to `WeaponName` and rename the object to `Weapon Name`.
12. Change the **Font Size** to `20` and the **Color** to white.
13. After that, to make it easier to read, let's add an **Outline** component by selecting the object and then clicking **Add Component** and selecting **Outline** after typing it in.

14. Then select the `Weapon Name` object and duplicate it (*Ctrl* + *D*). Move the duplicated object above the current one and rename it to `Weapon Info`. Then change **Pos Y** to `40` and **Text** to `200/200`: this will be where we will tell the player how much ammo they have in the currently selected weapon.



Now that we have the base UI in, let's add in some code to display the correct data for these UI elements.

15. In the **Project** tab go into the `MyGame/Scripts` folder and create a new **C# Script** called `GameHUD`.
16. Double-click on the script to open up `MonoBehaviour` and use the following code:

```
using UnityEngine;
using UnityEngine.UI; // Text, Slider

/// <summary>
/// Updates and provides details for the in-game HUD.
/// </summary>
public class GameHUD : MonoBehaviour
{
  // Will show Object References at the top of these
    //variables
  [Header("Object References")]
```

```
    public Slider slider;
    public Text weaponName;
    public Text weaponInfo;
    public Transform advancedPlayer;

    // Contains all of the info we need for the HUD
    protected vp_FPPlayerEventHandler m_Player = null;

    void OnLevelWasLoaded ()
    {
      m_Player = advancedPlayer.
                 GetComponent<vp_FPPlayerEventHandler>();
    }

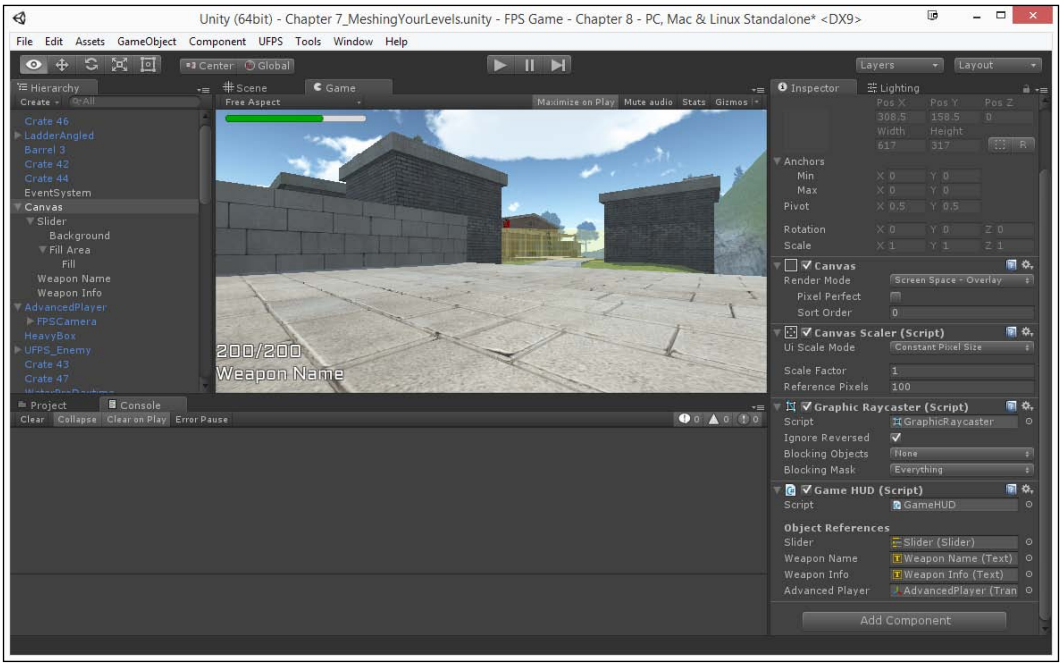
    // Update is called once per frame
    void Update ()
    {
      // Only do the actions if m_Player has a value other
        // than null
      if(m_Player)
      {
        // First we'll set the health
        slider.value = m_Player.Health.Get() /
                       m_Player.MaxHealth.Get();

        // Next, we want to display the ammo
        int currentAmmo = m_Player
                          .CurrentWeaponAmmoCount.Get();
        int maxAmmo = m_Player
                      .CurrentWeaponMaxAmmoCount.Get();

        // If we have ammo, display the details
        if((maxAmmo > 0) && (currentAmmo > 0))
        {
          weaponInfo.text = currentAmmo.ToString() + "/" +
                            maxAmmo.ToString();
        }
        else
        {
          //If we don't, just be blank
          weaponInfo.text = "";
        }

        // Lastly, let's display what weapon we're using
        weaponName.text = m_Player.CurrentWeaponName.Get();
      }
    }
}
```

17. After that, attach the newly created script to the Canvas object and assign the variables to the objects with the same name.

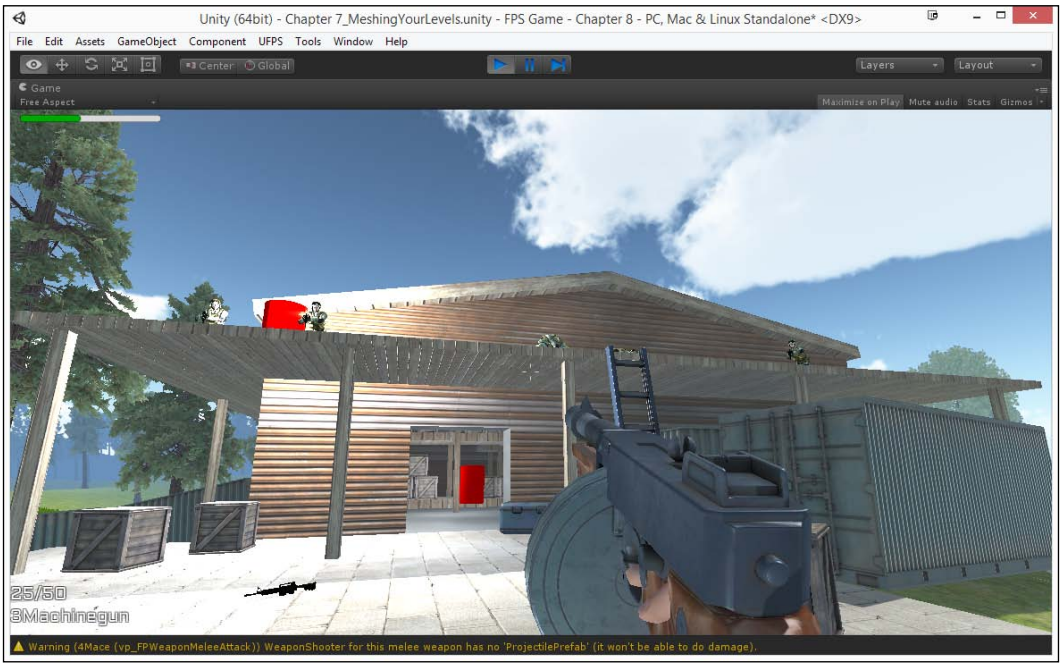


You'll notice that we now have a header on top of the object references. This is due to the decorator drawer that we used. These are actually one of my favorite parts of the engine as they allow you to customize how variables are shown and their layout.



For more info on these and other kinds of drawers, check out: `http://blogs.unity3d.com/2012/09/07/property-drawers-in-unity-4/`.

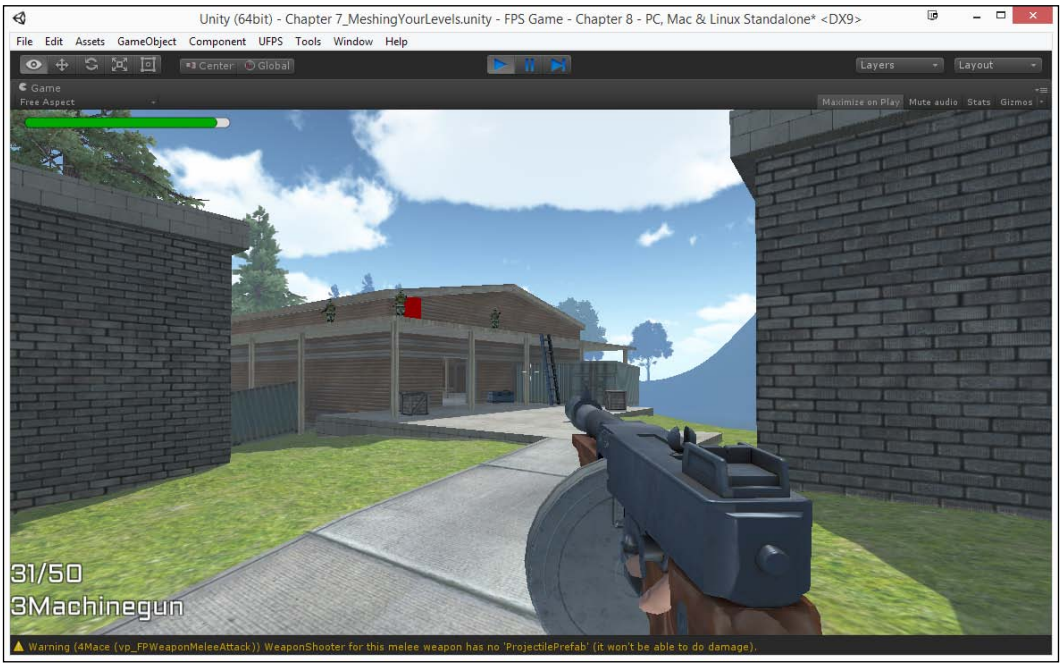
18. Save your game, and play!



And, as you can see, we're almost there. The information is being displayed properly using the names of the objects and the healthbar is working correctly; however, it's very small compared to how it was used in our editor window. We can make this HUD scale with the size of the screen by making use of the **Canvas Scalar** property. Let's do that now:

19. Select your `Canvas` object in the **Hierarchy** and from the **Inspector** tab go to the **Canvas Scalar** component. Under **UI Scale Mode** change the value to `Scale with Screen Size` and leave the other values at their defaults for now.

20. Save the game and play it!



As you can see, things are much clearer now!



For more information on making your UI fit with multiple different resolutions, check out: `http://docs.unity3d.com/Manual/HOWTO-UIMultiResolution.html`.

Summary

With this, you have learned how you can get data from UFPS about your characters and display them in a HUD of your own design. Specifically, we created a main menu and added text, added buttons and their functionalities, and replaced the default UFPS HUD.

In the next and last chapter, you will learn about how to get your game out into the world; in other words, publishing the project and getting it seen!

9
Finalizing Our Project

Once you finish your game project, it's important to take the time to get your projects out in the correct way.

Here is the outline of our tasks:

- Building the game in Unity
- Building an installer for Windows

Prerequisites

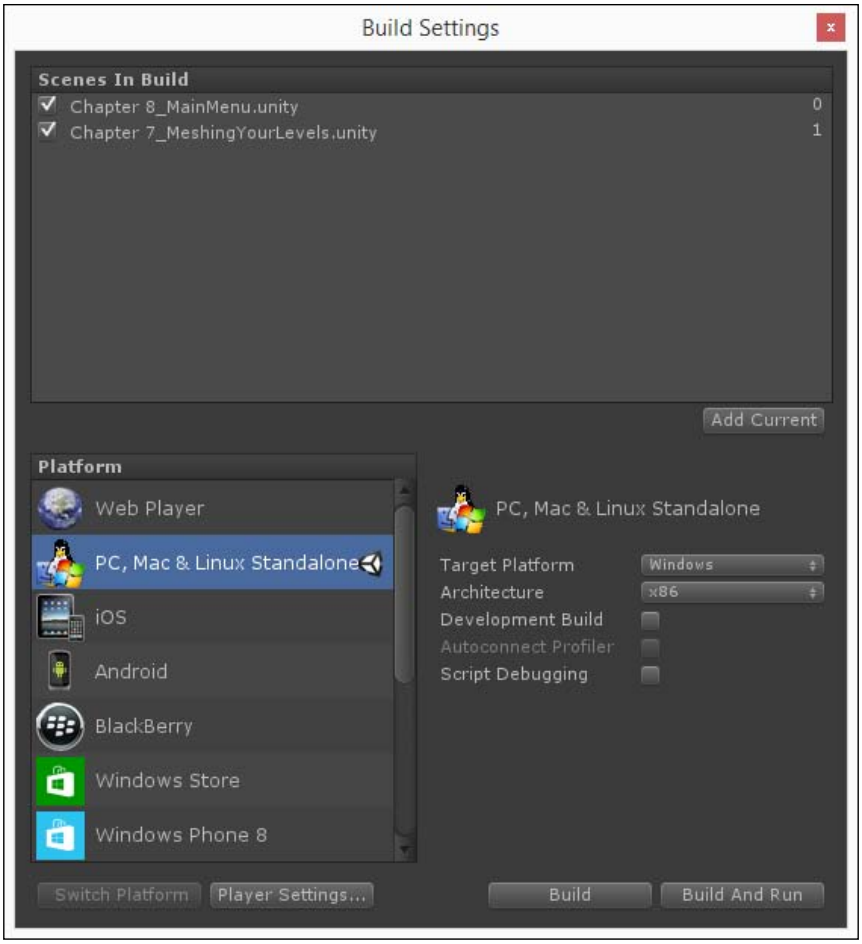
Before you start, you need to have your project completed just the way that you want it to be when you send it out into the world.

Building the game in Unity

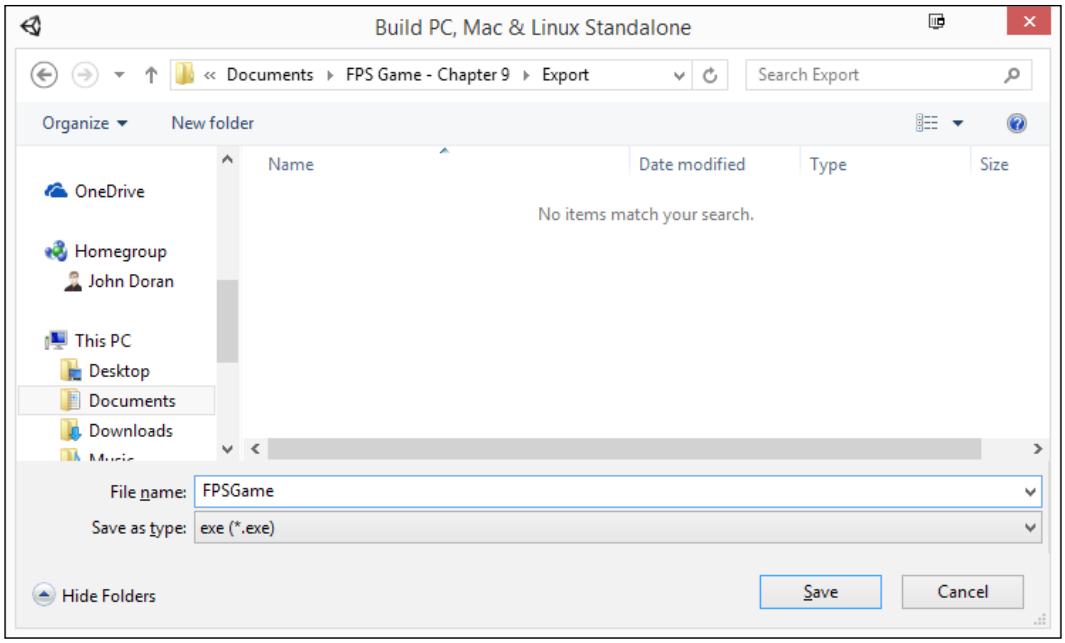
There are many times during development you may want to see how your game will appear if you build it outside the editor. It can give you a sense of accomplishment; I know, I felt that way the first time I pushed a build to a console development kit.

No matter what platform we wish to create out game for in order to build it we need to go to the **Build Settings** menu.

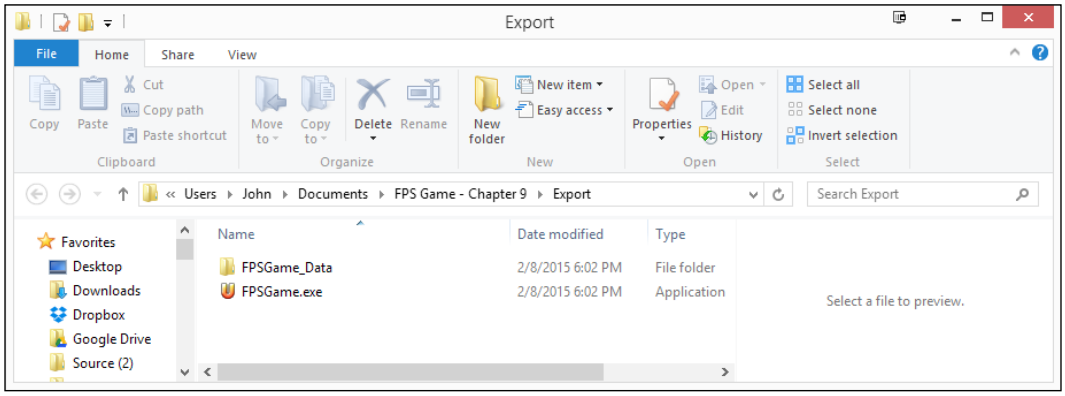
1. In order to access **Build Settings**, we will need to go to **File | Build Settings** from the top menu (or by pressing *Ctrl* + *Shift* + *B*).



2. Once you're ready, select **Platform** from the bottom-left menu. The Unity logo will show the one you're currently compiling for. We're going to compile for Windows now, so if it is currently not set to **PC, Mac & Linux Standalone**, select it and press the **Switch Platform** button.
3. Once you have all of this set up, press the **Build** button. It will ask you for a name and a location to place the game. I'm going to name it `FPSGame` and put it in an `Export` folder located in the same directory as the `Assets` and `Library` folders. Afterward, hit **Save**.



4. You may need to wait a bit, but as soon as it finishes, it will open up the folder with your new game.



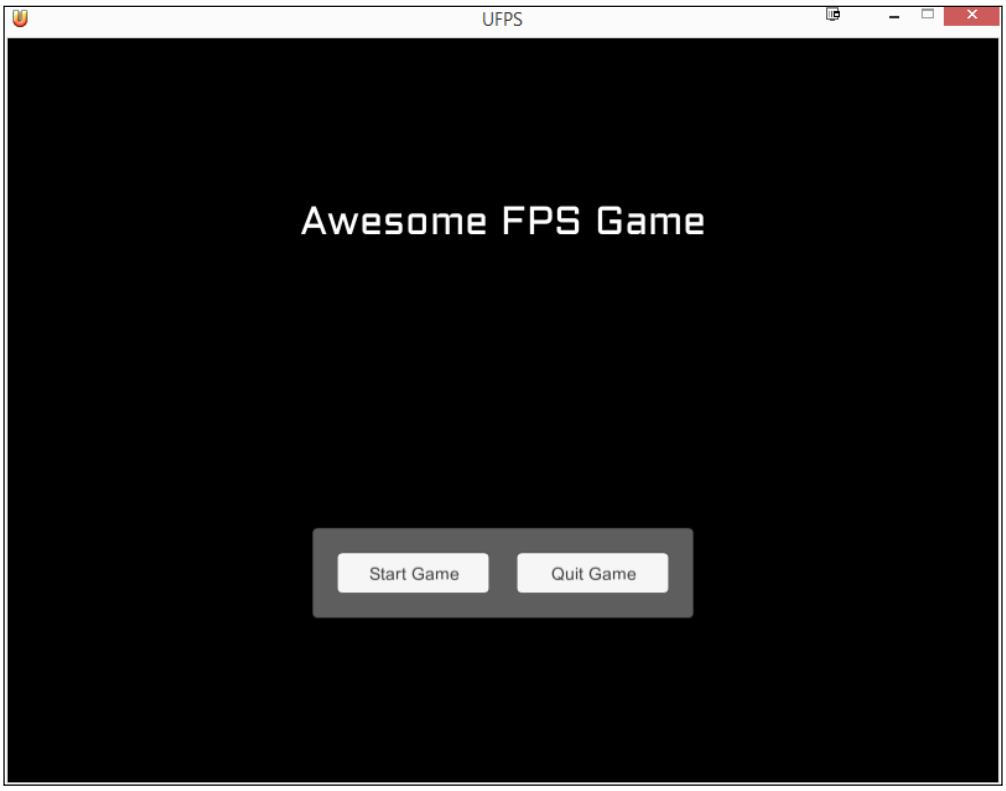
While building for Windows, you should get something like the preceding screenshot. We have the executable, but we also have a `Data` folder that contains all of the assets for our application (right now called `FPSGame_Data`). You must include the `Data` folder with your game or it will not run. This is troublesome; but later on in this chapter, will create an installer that will put it on a computer without any hassle.

If you build for Mac, it will bundle the app and data together. So, once you export it, all you need to give people is the application.



If you are interested in submitting your Mac game to the Mac App Store, there is a nice tutorial on how to do it at `http://www.conlanrios.com/2013/12/signing-unity-game-for-mac-app-store.html`.

If you double-click on the `.exe` file in order to run the game, you'll see the following screenshot:



As you can see, the game pops up and we can play it.

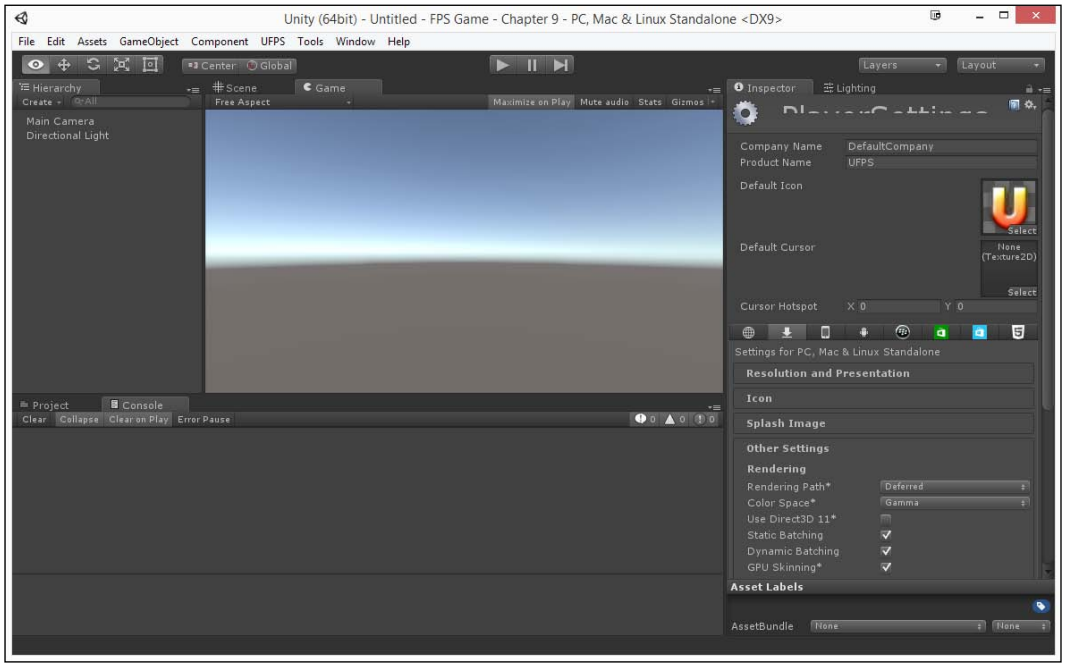


For more information on publishing and the specific things to look out for, check out `http://docs.unity3d.com/Manual/PublishingBuilds.html`.

Building an installer for Windows

Now that we know what happens by default, let's take some time to customize the project to make it look as nice as possible. **PlayerSettings** is where we can define different parameters for each platform that we want to put the game onto.

1. To open **PlayerSettings**, you can either click on the **PlayerSettings** button from the **Build Settings** menu or go to **Edit** | **Project Settings** | **Player**.



PlayerSettings is actually shown in **Inspector**. There are some key properties at the top that are cross-platform, which means that they will apply to all platforms (or rather will be the defaults you can override later).

2. From the **Project** tab, create a new folder within the `MyGame` folder and name it `Sprites`.
3. Now, in the `Example Code` folder, you'll find a `cursor_hand` image. Drag and drop it to the `Assets/MyGame/Sprites` folder of the **Project** browser. Once there, select the image and change **Texture Type** to **Cursor**.

   Note that, while the image I created works, you're more than welcome to create your own cursor and put it here to suit your particular game.

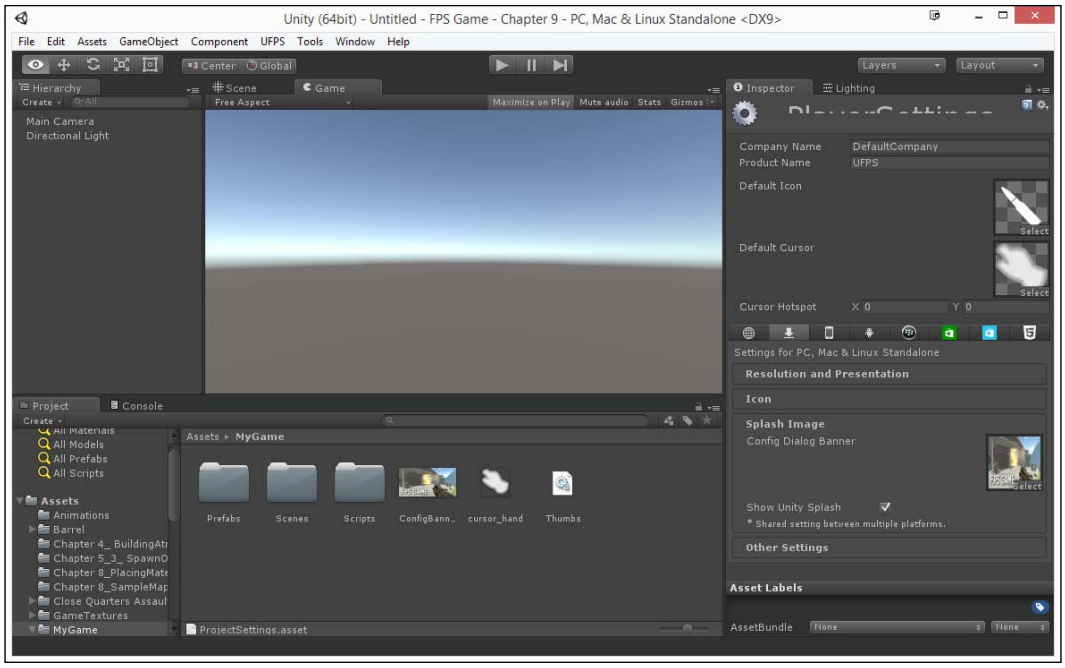
4. Then, in **Player Settings**, drag and drop the cursor image into **Default Cursor** and **Default Icon** to the `MachinegunBullet64x64` image that is located in the `Assets/UFPS/Base/Content/GUI/HUD`.

   If you would like your game to have multiple cursors or change cursors at runtime, the `Cursor.SetCursor` function would be quite helpful. For more information on this, check out `http://docs.unity3d.com/ScriptReference/Cursor.SetCursor.html`.

   However, while dealing with UFPS' certain functionalities (such as the obsolete `Screen.LockCursor` Unity command, which doesn't work with UFPS), if you'd like to have both mouse cursor input and FPS gameplay co-exist in UFPS, check out `http://visionpunk.vanillaforums.com/discussion/comment/243/#Comment_243`.

   For more information on working with mouse input with projects using UFPS, check out `http://www.visionpunk.com/hub/assets/ufps/manual/input`.

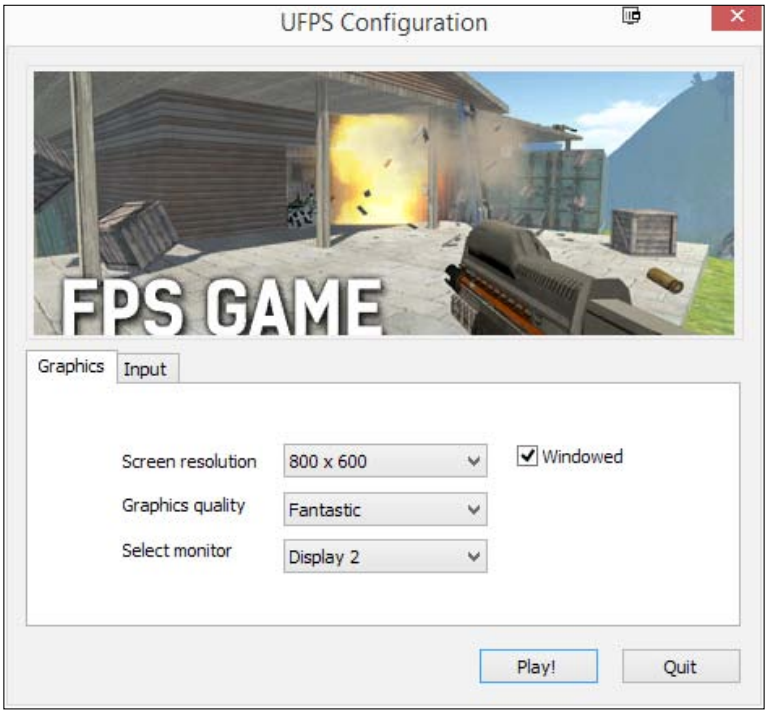
5. On your computer, in the `Example Code` folder, move the `ConfigBanner` image into the `Assets/MyGame` folder. Select the object and under **Texture Type**, change it to **Editor GUI and Legacy GUI** and then click on **Apply**.
6. Then, under the **PlayerSettings** section in **Inspector**, click on the **Splash Image** section to open the **Config Dialog Banner** property in which you should set the newly imported image.



If you want to create a **Config Dialog Banner** property of your own, make sure that you make the image 432 x 200 pixels in size or smaller.

7. Next, you'll need to decide whether you want **Display Resolution Dialog** to be displayed or not. If you want to keep it hidden, skip this step, since it's disabled by one of the assets we've imported. Otherwise, open the **Resolution and Presentation** section and, under **Standalone Player Options**, set **Display Resolution Dialog** to **Enabled**.
8. With this finished, go to **File** | **Save Project** and build the game once more, overwriting the previously created one.

Depending on your choices, you'll either see the game again or you'll see the following menu:



We can see that it has been modified with the image we created and a cursor showing up on the main menu and whenever we're not in the main game.

The game already looks much better and more polished than before. There are a number of other things you can do like restrict the kind of aspect ratios your game runs on or its resolution, or force windowed or full screen. I leave it to you to play around and make your project as nice as possible before you move ahead.



For more information on all of the properties for all of the different platforms that are available, check out `http://docs.unity3d.com/Manual/class-PlayerSettings.html`.

Note that the **Input** tab on this screen will not work with UFPS. However, UFPS has its own input manager in order to be able to support the rebinding of controls through script at runtime. For more information on this, check out `http://www.visionpunk.com/hub/assets/ufps/manual/input`.

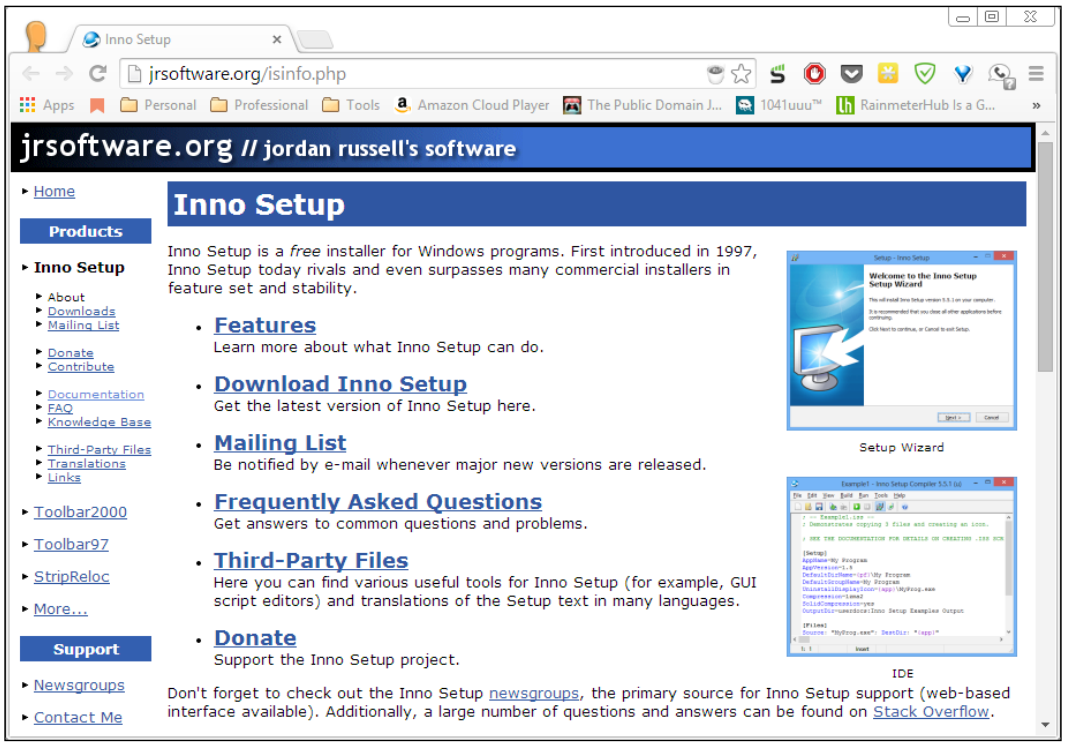
There is also a third-party asset called `cInput` that provides the preceding functionality along with a very nice ingame GUI to rebind the controls.

It exists as an affordable Unity asset and a free bridge script for UFPS, and is available at `https://www.assetstore.unity3d.com/en/#!/content/18471` and `https://www.assetstore.unity3d.com/en/#!/content/25471`.

Building an installer for Windows

Like I mentioned previously, having a separate `Data` folder with `.exe` is somewhat of a pain. Rather than give people a `.zip` file and hope they extract it all and keep everything in the same folder, I'd let the process be automatic and give the person an opportunity to install it just like a professional game. With this in mind, I'm going to go over a free way to create a Windows installer.

1. The first thing we need to do is to get our setup program. For our demonstration, I will be using Jordan Russell's **Inno Setup** installer. Go to `http://jrsoftware.org/isinfo.php` and click on the **Download Inno Setup** link.

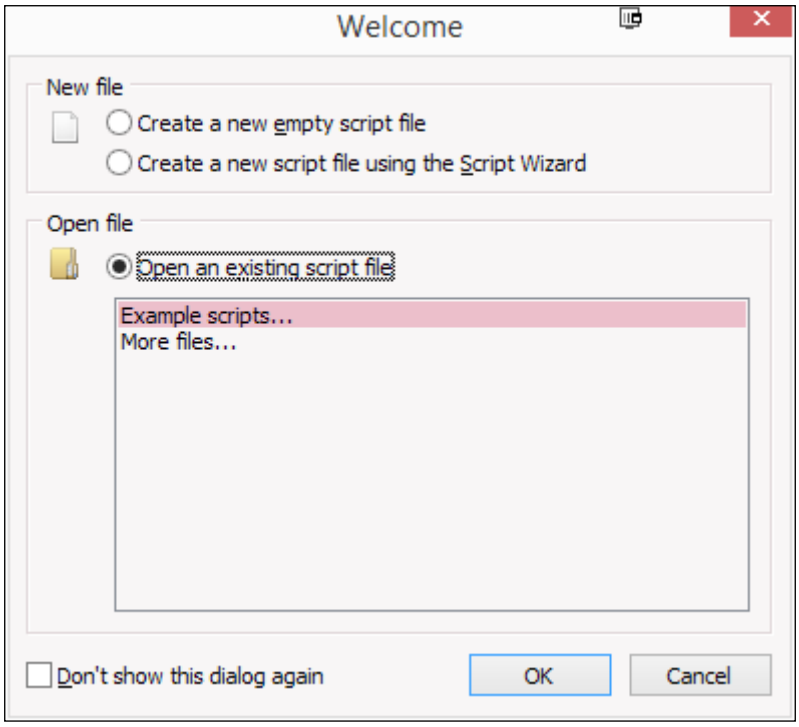


2. Click on the **Stable Release** button and select the `isetup-5.5.6.exe` file. Then, double-click on the executable to open it, click on the **Run** button if it shows a **Security Warning** dialog, and select **Yes** to allow changes.
3. In the **Select Setup Language** window, leave the selected language to **English** and click on **OK**.



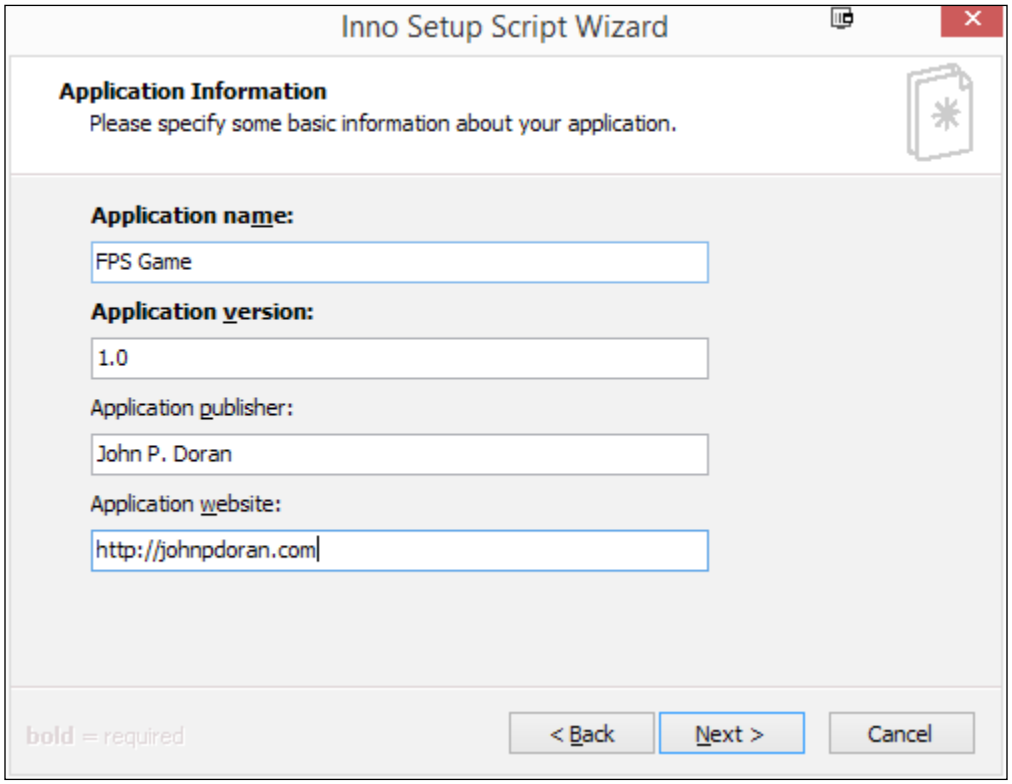
4. Run through the installation, making sure that the **Install Inno Setup Preprocessor** option is unchecked since we won't be using it. Upon finishing, make sure that **Launch Inno Setup** is checked and then press the **Finish** button.

   When you open the program, it will look similar to the following screenshot:

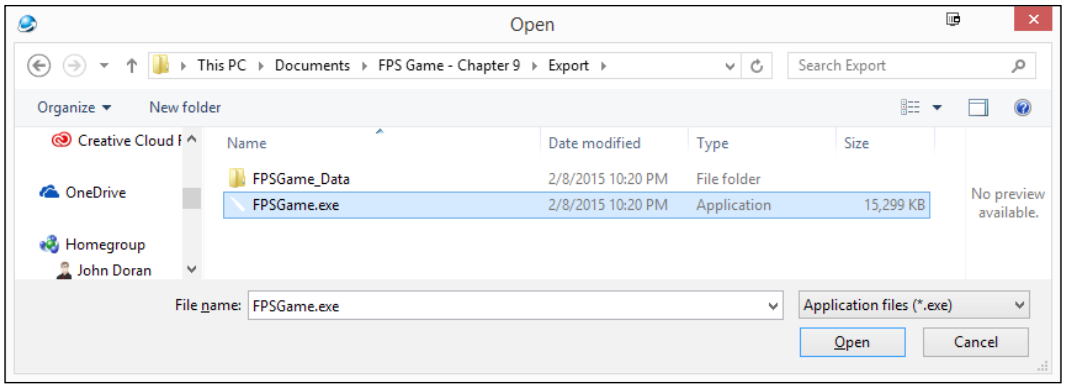


5. Choose **Create a new script file using the Script Wizard** and select **OK**.

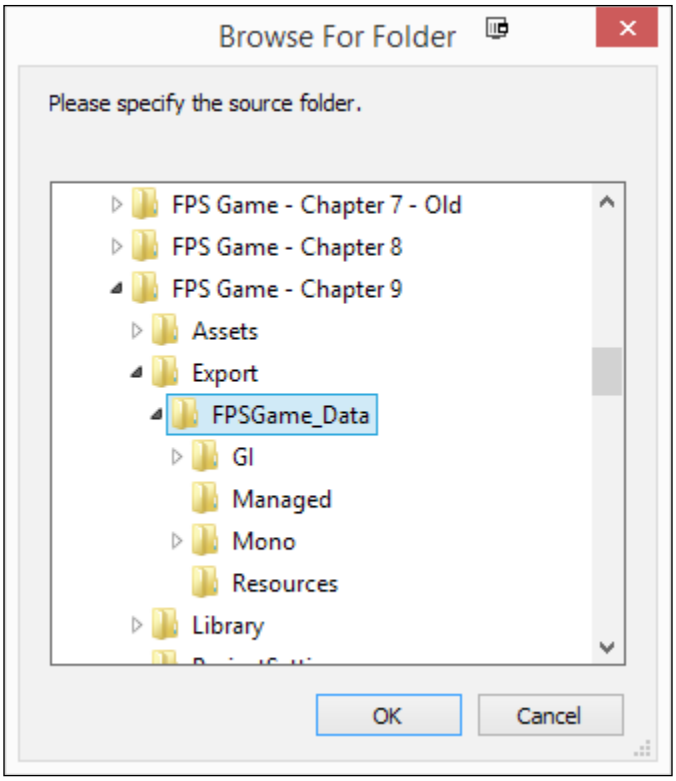
6. Click on the **Next** button and you'll come to the **Application Information** section. Fill in your information and press **Next**.



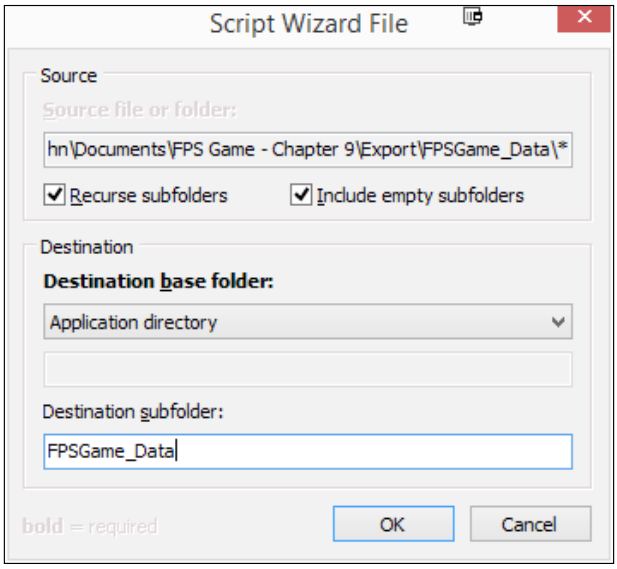
7. Next, you'll see some information on the `Application` folder. In general, you will not want to change this information, so you can go ahead and click on **Next**.
8. You'll be brought to the **Application Files** section, where you need to specify the files you want to install. Under the application main executable file, select **Browse** to go to the location of your **Export** folder, select the `.exe` file, and click on **Open**.



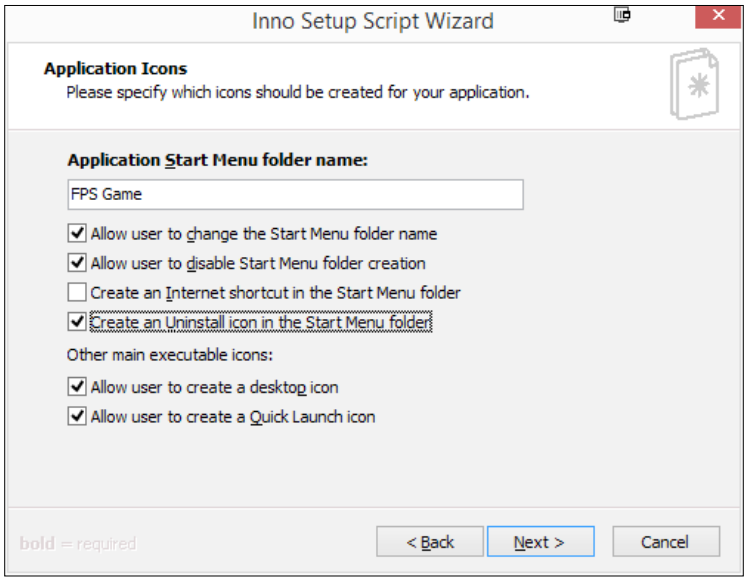
9. Now you need to add the `Data` folder. Click on the **Add Folder…** button, select the `Data` folder, and then press **OK**.



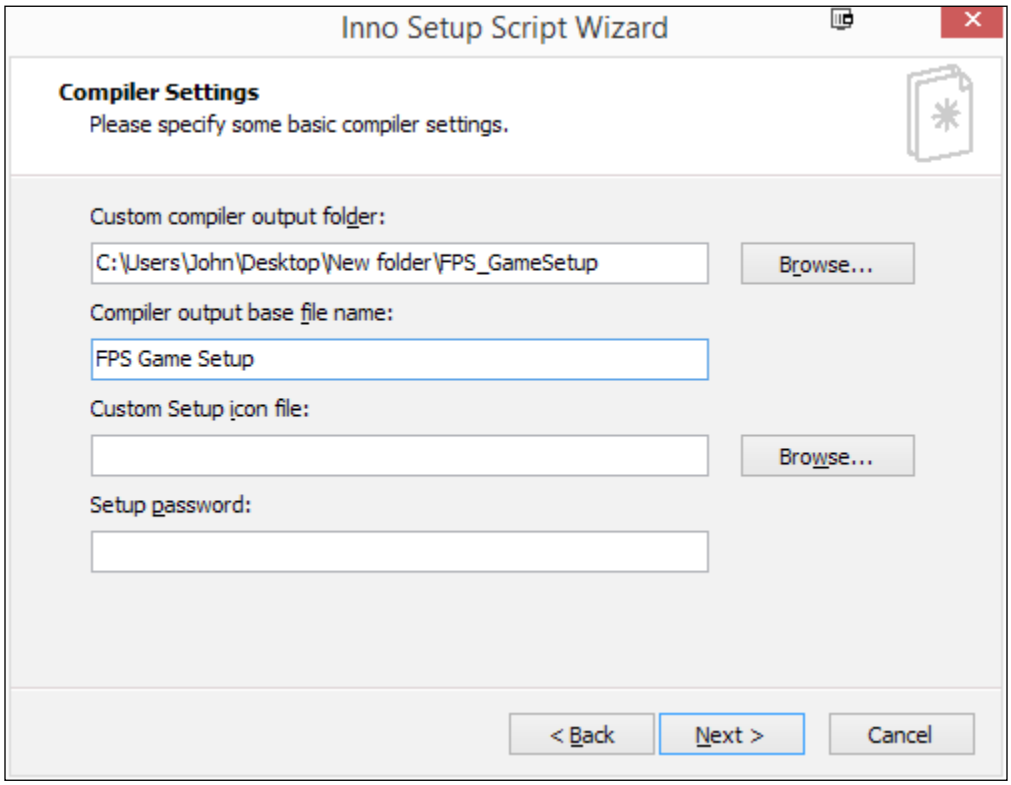
10. It will ask whether the files in the subfolders should be included as well. Select **Yes**. Then, select the folder in the **Other Applications file** section and press the **Edit** button. Set the **Destination subfolder** property to the same name as your `Data` folder, press **OK**, and then press **Next**.



11. In the next menu, check whichever options you'd like and press **Next**.



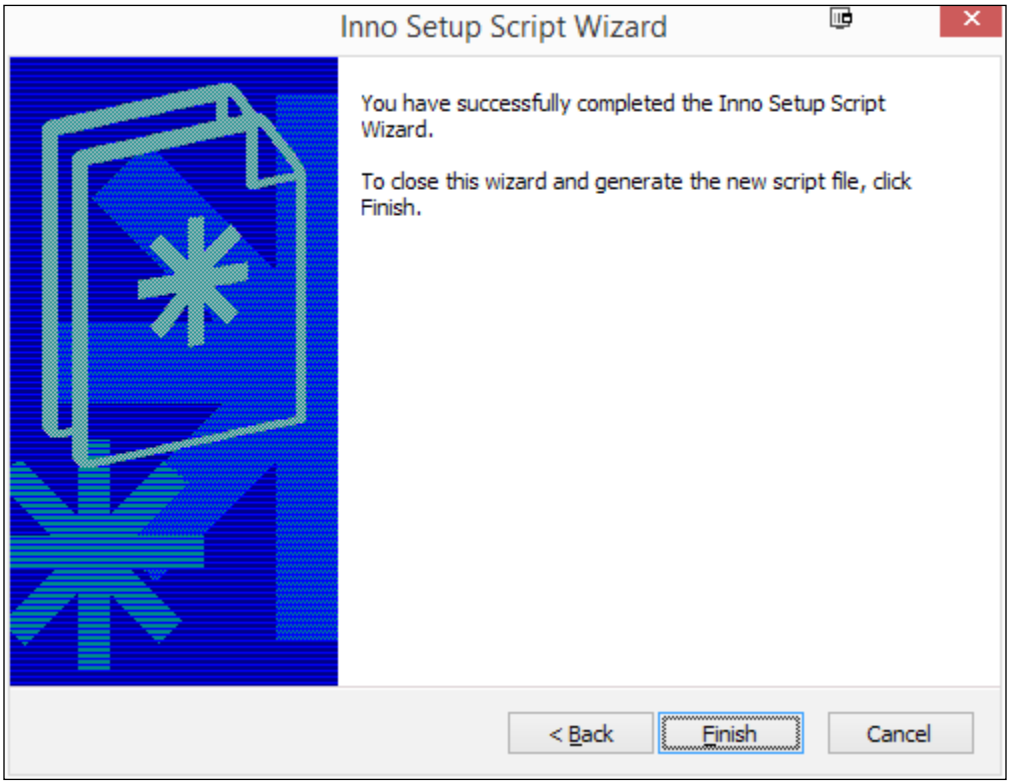
12. Now, you'll have an option to include a license file, such as an EULA or whatever your publisher may require and any personal stuff you want to tell your users before or after installation. The program accepts `.txt` and `.rtf` files. Once you're ready, click on the **Next** button.
13. Next, they'll allow you to specify the languages you want the installation to work for. I'll just go for **English**, but you can add more. Afterward, click on **Next**.
14. Finally, you need to set where you want the setup to be placed as well as the icon for it or a password. I created a new folder on my desktop called `TwinstickSetup` and used it. Then, click on **Next**.



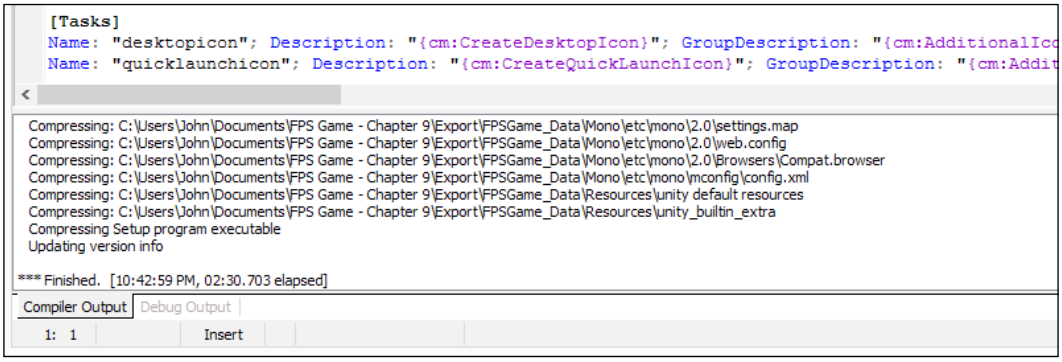


If you want to include a custom icon but don't have an `.ico` file you can use `http://www.icoconverter.com/`

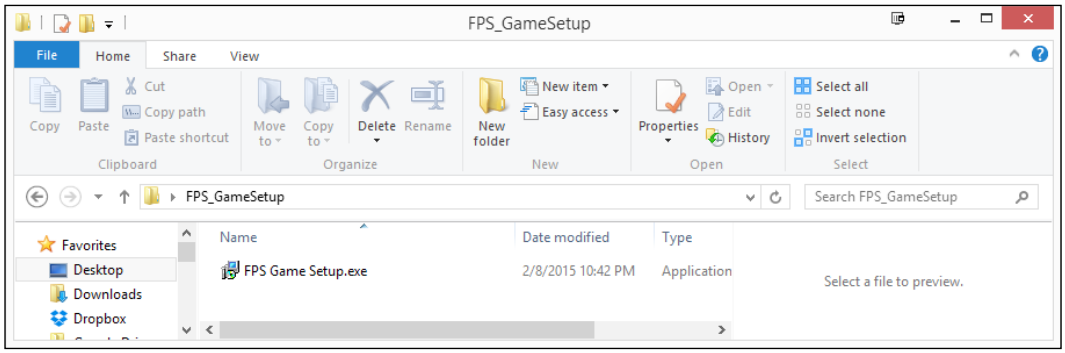
15. Next, you'll be brought to the successfully completed Script Wizard screen. After this, click on **Finish**!



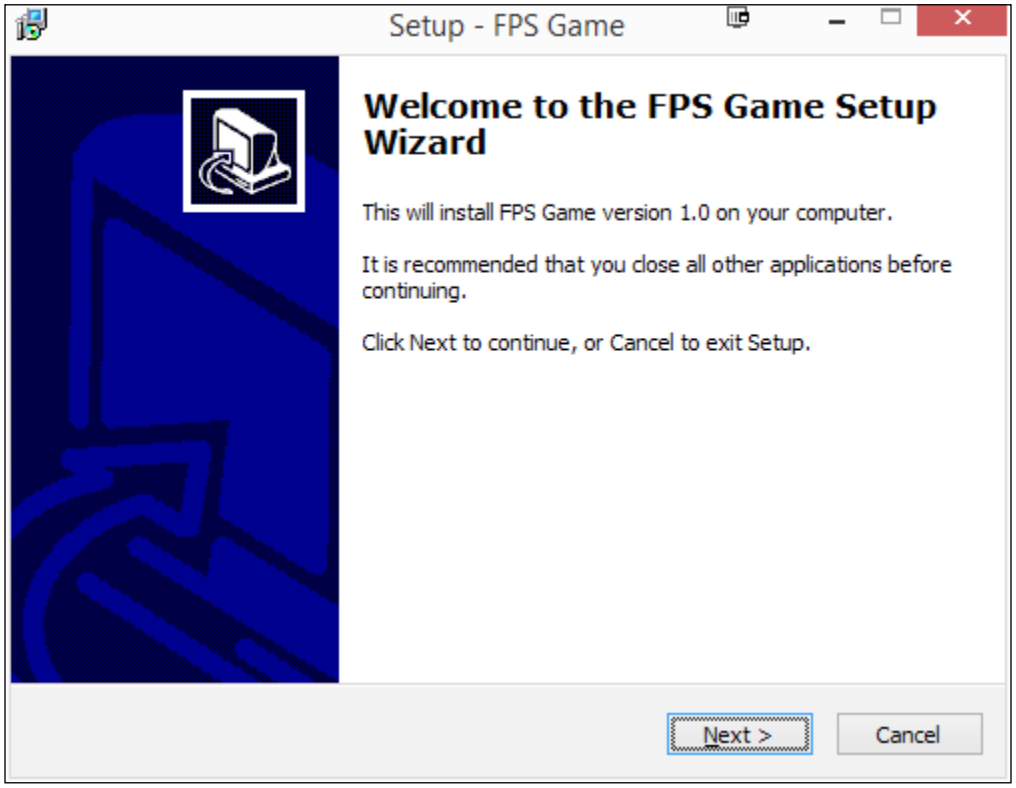
16. Now, it will ask you if you'd like to compile the script. Select **Yes**. It'll also ask if you want to save your script. Select **Yes**; I saved it to the same folder as my exporting. It'll take a minute or two; but as soon as you see **Finished** in the console window, it should be done.



17. If you go to the same place as your export folder, you should see your installer.



18. If you run it, it'll look something like the following screenshot:



With this, we have a working installer for our game.

Summary

We now have our game compiled and running on multiple platforms. Specifically, you learned how to build the game in Unity and build an installer for Windows.

Index

A

aerial perspective
  reference 117
AI Shooter
  using 153-163
Animation tool
  reference link 203
architectural overview, level prototype
  3D modeling software 44
  creating 44
  geometry, constructing with brushes 44
  modular tilesets 45
Artificial intelligence (AI) 129
Asset Store 6-9
atmosphere
  building 115
  Fog 116
  Skybox 115

B

Behaviour Tree Editor
  reference 149
Binary Space Partitioning (BSP) 12

C

canvas
  about 257
  reference 257
collision
  preventing 73-75
colors
  applying to walls 84-86
complimentary colors
  reference 119
Constructive Solid Geometry (CSG) 12, 44
Cursor.SetCursor function
  reference link 284
custom GUI
  button functionality 265-269
  buttons, adding 261-265
  creating 255
  default UFPS HUD, replacing 270-278
  prerequisites 255
  text, adding 256-260
custom health and ammo pickups
  reference 187
custom weapons
  building 19
  creating 30-34
  mesh, creating 24-28
  models, obtaining for 22, 23
  prerequisites 20
  properties, customizing 34-39
  sound effects 22, 23
  testbed, setting up 20-22
  UnitBank, creating 29, 30

D

Damage Handler
  reference link 192
dead AI
  cleaning up 184, 185
doors
  triggers, using for 198-208
doorway
  building 60-67
drawers
  reference 276

E

elevator
creating 208-214
encounters, building
AI Shooter, using 153-163
AI system, integrating 129
dead AI, cleaning up 184, 185
enemies, spawning with trigger 164-179
healthpacks/ammo, placing 186, 187
multiple enemies,
spawning at once 179-184
prerequisites 121
RAIN, using 130-152
simple turret enemy, adding 122-129
environment artist 41
explosive barrel
building 190-197
exterior environments
atmosphere, building 114-119
creating 87
grass, adding 112, 113
prerequisites 87
terrain 88
terrain, creating 91-96
trees, adding 108-111
water, adding 106-108

F

falls
preventing 73, 75
file organization 15
First Person Shooter (FPS)
about 1
prerequisites 2-4
project creation 4
Fog
about 116
for creating atmosphere 117-119
free bridge script, UFPS
reference link 287

G

game
building, in Unity 279-283
game design document (GDD) 43
geometry
creating 46-59
Graphical User Interface (GUI) 255
grass
adding 112, 113

H

hallway
creating 68-72
healthpacks/ammo
placing 186, 187

I

ICO converter
reference link 293
Icon Display controls
reference 182
Inno Setup
reference link 287
input manager, UFPS
reference link 286
installer
building, for Windows 283-295
iteration 181

L

Layouts
reference 263
level designer 41
level prototype
creating 42
prerequisites 42
levels
meshing 249-253

M

Mac App Store
reference link 282
marquee selection 81
material
creating 223-227
placing 227-248
mesher 41, 249
modular tilesets
about 45
mix and match 45
motor 172
mouse cursor
reference link 284
mouse input
reference link 284
multiple enemies
spawning, at once 179-184

N

NavMesh
using 160

P

Paint Details tool 112
PickupHealth 186
PickupHealthLoot 186
PickupSlomo 186
PickupSpeed 186
Position Springs properties
reference 35
prerequisites, level prototype
about 42
architectural overview, creating 44
level design 101 42, 43
ProBuilder
about 45
levels, meshing 249-253
material, creating 223-227
prerequisites 215
reference 45
working with 227-248
ProGrids
about 52
reference 52
properties, for platforms
reference link 286
Prototype
about 42, 45
doorway, building 60-67
geometry, creating 46-59
installing 12-14
reference 15
upgrading, to ProBuilder 216-222
publishing
reference link 283

R

RAIN
using 129-152
Rect Transform component 258
rooms
duplicating 68-72
Rotation Springs properties
reference 36

S

Scene Gizmo
about 80
reference 81
Shell properties
reference 38
simple turret enemy
adding 122-129
Skybox
about 115
modifying 115, 116
reference 116
skydomes 115
Sound properties
reference 39
stairways
adding 76-83
Standard Shader 225

T

terrain
about 88
color, adding 97-105
creating 91, 92
hand sculpting 90
height maps 88, 89
textures 97
Terrain editing tools
Paint details 92
Paint height 92
Paint texture 92
Place trees 92
Raise/Lower height 92
Smooth height 92
Terrain settings 92
Text component
reference 261
tileset 45
trees
adding 108-110
triggers
using, for doors 198-208

U

Ultimate First Person Shooter (UFPS)
about 1
installing 9-12
reference link 214
UnitBank
about 29
creating 29, 30
reference 30
Unit Snapping 52
Unity
about 41
game, building in 279-283
layout, customizing 16
Unity asset
reference link 287
UV Editing and Unwrapping tools
reference 248

V

vp_DamageHandler component 124
vp_Respawner component 125
vp_SecurityCamTurret component 125
vp_Shooter component 124

W

WASD keys 94
water
adding 106-108
Windows
installer, building for 283-295



Thank you for buying
Building an FPS Game with Unity

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`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `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.

[PACKT]
PUBLISHING

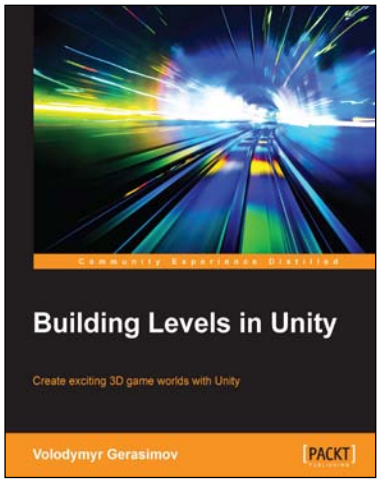


Unity 3D UI Essentials

ISBN: 978-1-78355-361-7 Paperback: 280 pages

Leverage the power of the new and improved UI system for Unity to enhance your games and apps

1. Discover how to build efficient UI layouts coping with multiple resolutions and screen sizes.
2. In-depth overview of all the new UI features that give you creative freedom to drive your game development to new heights.
3. Walk through many different examples of UI layout from simple 2D overlays to in-game 3D implementations.



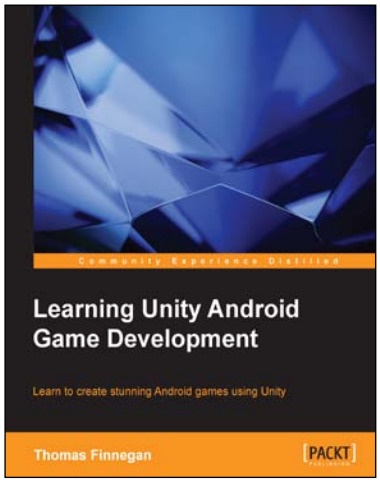
Building Levels in Unity

ISBN: 978-1-78528-284-3 Paperback: 274 pages

Create exciting 3D game worlds with Unity

1. Craft game environments with extreme clarity by adding realism to characters, objects, and props.
2. Import and set up custom assets such as meshes, textures, and normal maps in Unity.
3. A step-by-step guide written in a practical format to take advantage of the many features available in Unity.

[PACKT]
PUBLISHING

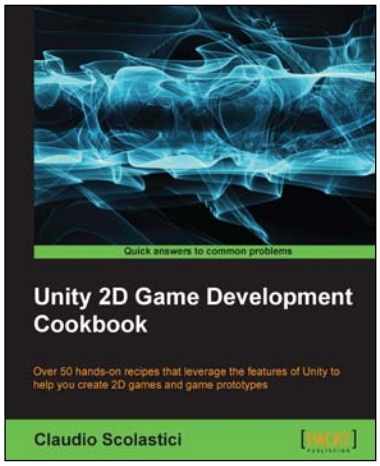


Learning Unity Android Game Development

ISBN: 978-1-78439-469-1 Paperback: 338 pages

Learn to create stunning Android games using Unity

1. Leverage the new features of Unity 5 for the Android mobile market with hands-on projects and real-world examples.
2. Create comprehensive and robust games using various customizations and additions available in Unity such as camera, lighting, and sound effects.
3. Precise instructions to use Unity to create an Android-based mobile game.



Unity 2D Game Development Cookbook

ISBN: 978-1-78355-359-4 Paperback: 256 pages

Over 50 hands-on recipes that leverage the features of Unity to help you create 2D games and game prototypes

1. Create 2D games right from importing assets to setting them up in Unity and adding them to your game scenes.
2. Program the game logic and events as well as the game controls and user interface using the C# scripting language and Monodevelop.
3. A step-by-step guide written in a practical format to take advantage of the many features available in Unity.

Please check **www.PacktPub.com** for information on our titles